Technical Report

UCAM-CL-TR-981
ISSN 1476-2986

Number 981

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The Cerberus C semantics

Kayvan Memarian

May 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 2022 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The C programming language, has since its introduction fifty years ago, become central to our computing infrastructure. It would therefore be desirable to have a precise semantics, that in particular could serve as a reference for implementers of compiler, analysis tools, etc. The ISO standard that notionally defines C suffers from two issues. First, as an inevitable result of being written in prose, it is imprecise. Second, it does not really attempt to precisely define the memory model. These shortcomings leave C's many obscure corners open to differing interpretations, and this is especially apparent when it comes to the memory model. While system programmers often rely on a very concrete view of pointers (even more concrete than what the ISO standard actually offers), compiler implementers take a more abstract view. Some optimisations, in particular ones based on alias analysis, reason about how pointer values are constructed during the program execution instead of only considering their representation, and perform transformations that would not be sound with respect to a concrete view of memory.

In this thesis, we present Cerberus, an executable model for a substantial fragment of C11. The dynamics of C is expressed as a compositional translation to a purpose-built language called Core. With this *semantics by elaboration*, we make the subtleties of C's expressions and statements explicit in the form of syntax in the Core representation. For these aspects of the semantics of C, the existing ISO standard has remained in agreement with de facto practice, and our model follows it. The elaboration allows for a model of the dynamics that is relatable to the ISO prose, and that is tractable despite the complexity of C.

For the memory model, as the de facto standards do not exist as coherent specifications that we could formalise, we opted at the start of this work for an empirical study of the design space for a realistic memory model. We surveyed the mainstream practice in C system programming and the assumptions made by compiler implementers. From this study and through engagement with WG14, the working group authoring the ISO standard, we have designed a family of memory models where pointer values have a provenance. At the time of writing one of these models is being published in collaboration with some members of WG14 as a ISO technical specification to accompany the standard.

We have dedicated significant effort in the executability of the model, both in term of performance and the scope of our frontend, which allows Cerberus to be used on medium scale off-the-self C programs with only limited amount of modification.

With this work we show that by suitably tailoring the target language, a semantics by elaboration produces a tractable definition of a large fragment of C.

Acknowledgements

I am first and foremost grateful to my supervisor, Peter Sewell. His encouragement, patience, and enthusiasm over the years made working on the semantics of C a pleasure. He always made himself available for discussions, both technical and otherwise, and for this, I am deeply grateful.

I thank Francesco Zappa Nardelli for his mentoring during my Masters' internship which gave me a great first start at research in programming languages, and for introducing me to Peter and his team.

I would like to thank my examiners Jeremy Yallop and John Regehr for their careful reading of this dissertation and an interesting discussion during the viva.

I would like to thank all those who contributed to the development of Cerberus. First, Justus Matthiesen, whose original work made Cerberus possible, and with whom I drafted the original design of the Core language. Victor B. F. Gomes for his many contributions and improvements to different parts of Cerberus. Kyndylan Nienhuis and Stella Lau for extending the scope of Cerberus by integrating their models of the C/C++11 concurrency memory model. James Lingard for experimenting with translation validation using an early Cerberus.

I would like to thank those at the Computer Lab who made me feel at home when I moved to Cambridge, in particular Mark Batty, Susmit Sarkar, Kathy Gray, Scott Owens, and Mike Dodds. I am also grateful to all the people who made the Computer Lab and Cambridge such an enriching place through interesting discussions, trolling, board games, and adventurous cooking. Chronologically: Justus Matthiesen, Danel Ahman, Ohad Kammar, Stephen Kell, Jean Pichon-Pharabod, Gabriel Kerneis, Camille Boulay, Kyndylan Nienhuis, Dominic Mulligan, Thomas Tuerk, Jonas Frey, Negar Miralaei, Hannes Mehnert, Ali Sezgin, Kasper Svendsen, Christopher Pulte, David Kaloper Meršinjak, Victor B. F. Gomes, Stella Lau, Thomas Bauereiss, Robin Morisset, Simon Castellan, Raphaël Proust, Ben Simner, Eiko Yoneki, and many others. I thank Jean Pichon-Pharabod for proofreading this dissertation.

I am grateful to the members of WG14 and the memory model study group for their feedback and collaboration on the many N-documents submitted as a result of the work presented in this dissertation, in particular Jens Gustedt and Martin Uecker.

Finally, I am very grateful for the love and support from my parents, my sister, and my niece.

This work has been supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC Advanced Grant *ELVER*, grant agreement No. 789108).

This work has been supported in part by EPSRC Programme Grant EP/K008528/1, *REMS: Rigorous Engineering for Mainstream Systems*.

This work has been supported in part by EPSRC Leadership Fellowship EP/H005633/1, *Semantic Foundations for Real-World Systems*.

Contents

Chapter 1

Introduction

Fifty years after its introduction, C remains central to our computing infrastructure as one of the languages of choice for systems and embedded programming. As an evolution of system programming languages of the time, C was designed to be "close to the machine" and to allow for a simple compilation to the hardware [Rit93]. The language is also characterised by its portability, which helped it to quickly grow in popularity. By the 80s, C was implemented by various compilers on a diversity of platforms.

At the time, it did not have a "formal" definition, the "K&R" book by the authors of the language [KR78] being the closest substitute. This however described an early variant of the language, without features such as `void` and enumeration types, and with only limited support for structures. As an alternative, the main compiler of the time (pcc) was also used as the reference for the semantics of C. Over time, existing practice and the various implementations inevitably drifted from one another, and several dialects appeared. As a result, in 1983 a standard committee was formed which produced an ANSI report in 1989, then turned into an ISO standard (C90). The ISO standard is maintained by the WG14 committee which has published further revisions: C99, C11, and C17.

The committee gave itself the following goal (taken from the rationale for ANSI C [ANSICrationale]):

> "to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments."

The language specified by the standard attempts to solve a challenging problem: to simultaneously allow programmers to write low-level system code, with performance on par with hand-written assembly (keeping with the spirit of the original C); while providing portability between widely different target machine architectures; and finally while supporting increasingly sophisticated compiler optimisations. There is a tension between the first goal, which needs the underlying machine (and in particular its memory model) to be left mostly concrete by the language; and the last two, which can only be achieved by abstracting away some details of the machine. It does so by providing operations both on abstract values (most arithmetic operators are defined over "abstract" integer types, albeit with fixed size and wrapping semantics), and on their underlying concrete representations (e.g. through `unsigned char` pointer introspection). Because these exist as part of a single expression language, they can interact in delicate ways, exposing in the dynamics of the language subtle properties relating to the memory model, type safety,

relaxed concurrency, and so on. Unsurprisingly, these have proven difficult to characterise precisely in the prose specification style of the ISO standard. Even the few people very familiar with the standard often struggle with the subtleties of C, as can be witnessed by the long list of requests for clarification made in the form of defect reports [WG14-DR], and inconclusive discussions of whether compiler anomalies are bugs with respect to the standard. Because of the lack of an executable model that would serve as a test oracle, which would let one simply compute the set of all allowed behaviours of any small test case, discussions often have to rely on the standard committee recalling the original intent of the passage of standard text being discussed.

The obvious semanticist response to this state of affairs is to attempt a mathematical reformulation of the standard. A number of research projects have worked to formalise varying fragments of the language [GH92; Nor98; Pap98; Nor99; Tuc08; TKN07; BL09; Ler09; Win+09; Bat+11; ER12; Kre13; KW13; Kre14a; BBW14; HER15; KW15; Kre15; BBW15; Kan+15]. However, similar to the situation before the creation of the first standard, the shortcomings of having a prose standard for a language as subtle as C have given rise to divergent readings of the standard along system programmers and compiler implementers. Solely focusing on the ISO standard would therefore fail to completely capture C as it exists in practice.

For a more complete picture, we have to consider the behaviours of the various mainstream C compilers (whose choices sometime go beyond the allowance of the standard by defining a particular behaviour for some constructs or practice which the standard makes undefined), the assumptions necessary for the soundness of compiler optimisations, the assumptions that systems programmers make and which are relied on for the correct execution of the large corpus of existing C code, and more recently the assumptions implicit in C analysis tools. Each of these induce mostly unwritten *de facto standards* which all subtly differ from the standard definition, but also from each other. While this could appear as a theoretical concern, disagreements between the assumptions made by compiler implementers (which over time have created more aggressive optimisations, exploiting situations which the standards specifies as undefined behaviour), and the assumptions made by system programmers, have sometime resulted in the introduction of security vulnerabilities by compilers [Wan+12; Wan+13].

In this thesis, we present Cerberus, an executable model for a substantial fragment of C with several distinctive features:

- It formalises the ISO C11 standard for the aspects of the semantics of C where this prose specification is clear and corresponds to the language as it is used in practice. These aspects consist of the statics and dynamics of C's expressions and statements, when abstracting away the memory model and some operators working over pointers. The dynamics of C is expressed as compositional translation from the (slightly sanitised) C abstract syntax into a purpose-built language called Core. Each C operator is mapped into blocks of Core expressions, which can be seen as mathematising the fragment of the prose of the ISO standard defining the dynamics the operator. The aim of this *semantics by elaboration* is to make explicit the inherent complexity of C's dynamics, which is mostly implicit in the syntax, while keeping the semantics readable and tractable. A reader familiar with the semantics of Core and the prose of the ISO standard should be able to easily recognise the formalisation of the latter in Cerberus.

- For the *memory object model* (the semantics of pointers, unspecified values, and the

abstract memory state), where the ISO standard is both unclear and in disagreement with programmers' practice and compiler implementations, we developed a candidate memory model (with several variants) aiming at capturing the de facto standards embodied by practice.

- Regarding the semantics of pointers, our memory model formalises a notion of *pointer provenance*, aiming to capture the assumptions underlying compiler alias analyses, and hinted at in the past by WG14 when responding to a request for clarification. We have engaged with WG14 over several years, presenting papers to investigate the possible designs for a memory model based on provenance. As a result of this, we have produced in collaboration with some other members of WG14 a prose version of the model, which at the time of writing is in the process of being published as a ISO technical specification to be annexed to the standard.

- The model, being executable, can be used as an oracle for exploring the allowed behaviours of a C program (randomly, exhaustively, or interactively), or finding the occurrence of an undefined behaviour. The model is parametric on a subset of implementation-defined behaviour, and can therefore emulate the behaviour of programs on different platforms.

- Substantial work has gone into the frontend to allow Cerberus to operate on reasonably sized C translation units. While the model only covers a fragment of the C language and standard library, the frontend is robust enough to allow off-the-shelf C programs within our supported fragment to be used without modification.

- To satisfy our goal of providing a usable semantic tool for practitioners and the C standards community, the model is equipped with an intuitive web-based user interface, developed by Victor Gomes.

- Because the model is structured as a semantics by elaboration, one can build analysis tools leveraging the C semantics while only having to deal with (the much simpler) Core language.

The parts of the model where we deal with the statics and dynamics of C are implemented in Lem [Mul+14], a language based on a pure fragment of OCaml designed for the development of executable formal models. We use its backend translation into OCaml. As we do not use the logical constructs of Lem, this translation is very lightweight. The remaining parts, such as the parsers and infrastructure wrapping the model into usable tools, are directly written in OCaml. The development is open-source and available online at `https://github.com/rems-project/cerberus`, and an instance of the web interface is available at `https://cerberus.cl.cam.ac.uk/`.

Plan of this thesis For the memory model, because the ISO standard is unclear and the de facto standards do not exist as coherent specifications that we could formalise, in the early phase of this work, we opted for an empirical study of the design space for a realistic memory model. With this aim, we surveyed the mainstream practice in C system programming and the assumptions made by compiler implementers, which we discuss in Chapter 2. In Chapter 3, we motivate our design choice of a semantics by elaboration, and give a high-level presentation of the Core language. We then show in Chapter 4 the key subtleties hidden in the dynamics of the C's expressions and statements which we

make explicit by elaboration, and how they motivated the design of Core. In Chapter 6, we give a formal presentation of the semantics of Core. The memory interface used by Core programs, and implemented by our candidate memory models, is given in Chapter 5 and Appendix A, and we show an overview of the elaboration function in Chapter 7. In Chapters 8 and 9, we explain the design of our provenance-based memory object model, followed by its formal presentation, and a discussion of its validation. In Chapter 10, we discuss two integrations of Cerberus with the C/C++11 concurrency memory model as a result of two collaborations: with Kyndylan Nienhuis et al. [NMS16] integrating a previous version of the model with an operational version of Batty et al. [Bat+11]; and with Stella Lau et al. [Lau+19] on a bounded model checker combining the thread-local semantics of Cerberus, a modern memory object model, and a large class of axiomatic concurrency models. In Chapter 11, we discuss the C frontend, the command-line tool and web-based user interface of Cerberus, along with analysis tools (such as a refinement type system for the verification of system programming idioms) built by others, showcasing the reusability of our model. In Chapter 12, we discuss how we validated the model. Finally, in Chapter 13, we discuss the related work, and conclude in Chapter 14.

Previous publications and joint work The development of Cerberus builds upon the work of Justus Matthiesen in his Part II project dissertation [Mat11]. In particular, the intermediate Ail language originates from his work. The design of the semantics by elaboration and of the early version of Core target language was joint work with Matthiesen, which he presented in his 2011-12 MPhil dissertation [Mat12]. The long development of Cerberus also involved cooperation with Peter Sewell for the study and development of provenance-aware memory models, Victor Gomes (who wrote the web user interface, and several example backends), Kyndylan Nienhuis for the operational C++11 concurrency model [NMS16], and Stella Lau for Cerberus-BMC [Lau+19].
Part of the work presented in this thesis was previously published in two papers:

- At PLDI16 [Mem+16], where we discussed the disagreements between the ISO standard and the de facto standards found in practice, and presented a early state of the Cerberus model:
  "Into the Depths of C: Elaborating the De Facto Standards",
  K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson and P. Sewell. Distinguished paper award.

- At POPL19 [Mem+19], where we presented our provenance-based memory object model, and reported on new developments of the Cerberus model.
  "Exploring C Semantics and Pointer Provenance",
  K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell

As part of our interaction with the WG14 and W21 committees the progressive work regarding the memory object model of C was discussed in a series of technical papers [N2012; N2013; N2014; N2089; N2090; N2091; N2223; N2219; N2220; N2221; N2222; N2263; N2362; N2364; P1796R0; N3005].

Limitations of Cerberus Cerberus covers a substantial fragment of ISO C11, but it is still missing or simplifying some features:

- The frontend does not model the C preprocessor, and instead receives as input preprocessed translation units from GCC or Clang.
- The syntax of compound type initialisers has some restrictions. This is the result of defects in their desugaring, that remain to be fixed.
- The support for floating types is added for convenience, but is informal. Only `float` and `double` are accepted by the frontend, and their semantics within the executable semantics simply makes use of OCaml's `Float` module. We therefore do not capture the underlying state of the abstract machine for floating-point, and the related undefined behaviours are not modelled.
- The `volatile`, `restrict`, and `register` qualifiers are not modelled beyond the syntactic constraint requirements.
- Flexible array members are technically supported, but further validation is needed (in particular regarding the detection of the associated undefined behaviour).
- The following features are not supported at all:
  - the C11 generic selection operator;
  - the types relating to C11's character-set features;
  - bit-fields;
  - variable length arrays, and function parameters of array type with the `static` keyword or *;
  - non-local jumps (<`setjump.h`>), and signal handling (<`signal.h`>).
- The undefined behaviour when an assignment operator has partially overlapping store and read (§6.5.16.1#3) is not modelled.
- Our memory object models do not model support sub-object provenance, the semantics for which is an open problem.
- The base version of Cerberus presented in this work does not model the C/C++11 concurrency memory model. There are two extensions led by collaborators which add support for it, which we discuss in Chapter 10.
- Support for a fragment of the standard library was only added as needed. Apart from our formal modelling of the `printf()` functions, the standard headers we expose make use of an adaptation of the musl libc [musl-libc] source code. The undefined behaviours specified by the standard are therefore not precisely captured. The model will only detect undefined behaviours resulting for the execution of musl libc's code.

Chapter 2

Surveying de facto C

In this chapter, we discuss our investigative work for establishing the design space of a memory model for C. Our goal is to capture C as it is used in practice, in particular in low-level system code. Here, we focus on sequential issues: the nature of pointer values, unspecified values, the semantics of casts operations between pointer and integer values, etc. For clarity, we refer to this as the *memory object model*, as opposed to the *memory concurrency model* that captures the relaxed behaviour of C/C++11. The latter has already been formalised in the work of Batty et al. [Bat+11] and others [Bat+12; Vaf+15; BDW16a; BDW16b; Lah+17]. We refer to Chapter 10 for a discussion of how Cerberus supports concurrency by combining our object model with operational and axiomatic versions of the C/C++11 concurrency model.

Characterising the *de facto* C semantics is not straightforward: unlike the ISO standard, there is no concrete document defining it. It instead exists in multiple forms, mostly unwritten and corresponding to different perspectives, and as a result sometimes conflicting:

- There are the languages actually implemented by mainstream compilers (GCC, Clang, ICC, MSVC, etc.). Each of these have some syntactic extensions, though we do not concern ourselves with these here. Within the common syntax, however, there remain subtle semantic differences between different implementations. Firstly, as allowed by the parametric nature of the ISO standard, they choose behaviour for some aspects of the semantics which are left implementation-defined or unspecified. Secondly, they give more specified behaviour (to some situations which have undefined behaviour according to the ISO standard) which programmers have come to rely upon, while at the same time introducing new undefined behaviour for other situations. The former can for example be triggered by the use of compiler flags (e.g. GCC's `-fno-strict-overflow` ,`-fwrapv`); while the latter may arise, from assumptions about the user code, that compilers make for the soundness of their optimisations (e.g. those relating to pointer aliasing).
- There are the idioms used in the corpus of mainstream systems, especially in specific large-scale systems: Linux, FreeBSD, Xen, Apache, etc.
- There are the beliefs of the systems programmers regarding behaviours they can rely upon.
- There is the behaviour assumed, implicitly or explicitly, by code analysis tools.

It is our assessment that mainstream usage and implementations rely on and implement a significantly different language, or languages, from what is defined by the standard;

this divergence makes the standard less relevant than one might think and leaves practice on an uncertain footing. The situation largely arises from the ISO standard's attempts to define a rather complicated memory object model as part of a large prose document. While the standard succeeds at clearly defining some aspects of the C semantics such as the dynamics of expressions and statements, it is often vague or fails to address key questions regarding the memory. At the same time, the perspective naturally taken by compiler implementers sometimes conflicts with that of system programmers, as we shall see.

To inform our design of a formal memory object model capturing the *de facto* C language, and in particular define an envelope for the design space, we opted to directly probe the unwritten assumptions made by system programmers and compiler writers by surveying them. To the best of our knowledge, this constitutes a novel approach to investigating the de facto semantics of a widely used language.

We designed two surveys which we now present. The responses informed the design of a proposed memory object model which we present in Chapter 9.

2.1 First survey: "The C memory quiz"

In the first half of 2013, we disseminated a web form made of 42 questions, ranging over the semantics of pointers, the representation of objects, and the interaction of the two with other values (e.g. through the use of cast operators). Each question consisted of a prose description of a programming idiom, followed by a concrete C program. Because of our interest in establishing any difference between the language defined by the ISO standard and the de facto use of C, for each of these idioms, the responders were first asked two multiple choice questions: the first regarding what they thought would happen in practice; and the second asking whether the ISO standard allowed the idiom. Here is one of the questions:

Casting of pointers: roundtrip properties

This question asks how generally one can cast a pointer to other pointer types and then back to the original.

CPR.1 (usage) Can one cast a pointer to a series of arbitrary other pointer types and back to the original type to obtain a pointer that is equivalent to the original (i.e., dereferencing it is undefined behaviour if and only if dereferencing the original is, it points to the same object as the original, and it compares equal to the original)?

Example:

```
#include <stdio.h>
int x=1;
int main() {
  int *p = &x;
  float *q1 = (float *) p;
  char **q2 = (char **) q1;
  int *q3 = (int *) q2;
  // are p and q3 now equivalent?
  // For example:
  // - is the following not undefined behaviour?
  int y = *q3;
```

```
  // - does the following compare true?
  int b = (q3==p);
  *q3=2;
  int z = *p;      // - does this give 2?
  printf("y=%i (q3==p)=%s z=%i\n",y,b?"true":"false",z);
}
```

(a) used in practice and supported by compilers

(b) questionable (I would discourage this in a code review)

(c) should not be used; compilers might well not support it

(d) not useful, but compilers do support it

(e) not useful, and compilers do not support it

(f) don't know

(g) other: (write in)

CPR.1 (standard) Is it allowed by the C standard?

(a) allowed by the standard

(b) not allowed by the standard

(c) the standard is unclear, contradictory, or does not address this

(d) don't know

We targeted this survey at a small number of experts, including multiple contributors to the ISO C or C++ standards committees, C analysis tool developers, experts in C formal semantics, compiler writers, and systems programmers. The results were very instructive, but this survey demanded a lot from the respondents; it was best done by discussing the questions with them in person over several hours. As a result, the number of participants was limited to 16. A key takeaway was that despite the expertise of the responders, we observed for some of the questions significant divergence among the responders, both regarding what they reported having seen in practice, and what they believe the ISO standard says. As an example, for the question CPR.1 above, the responses were:

Question regarding practice:

used in practice and supported by compilers	5 (31.3%)
questionable (I would discourage this in a code review)	2 (12.5%)
should not be used; compilers might well not support it	2 (12.5%)
not useful, but compilers do support it	3 (18.8%)
not useful, and compilers do not support it	0
don't know	0
other: (write in)	1
no response	3

Question regarding the ISO standard:

allowed by the standard	3 (18.8%)
not allowed by the standard	3 (18.8%)
the standard is unclear, contradictory, or does not address this	7 (43.8%)
don't know	58 (18%)
no response	3

While the small sample size does not allow to conclude on the desired semantics, the

lack of consensus is of note. An issue came from the question being poorly phrased, as it did not make clear that we intended the responders to assume that the pointer values were well aligned for all pointer types involved (otherwise the ISO standard unambiguously makes the casts undefined).

2.2 Second survey

In early 2015, we made a simpler survey with the aim of targeting a larger audience, reducing the previous one to the 15 most interesting questions. While the first survey inquired about the responders' understanding of the ISO standard in addition to existing mainstream practice, with this new survey we instead focused on the latter: the behaviour that programmers assume they can rely on; the behaviour provided by mainstream compilers; and the idioms used in existing code, in particular systems code. We made this new focus clear in the preamble of the survey, to prevent responders colouring their answer with their knowledge or guesses about the ISO standard. Another important change was the removal of the C code examples. These proved distracting to some of the responders of the first survey, who thought they were intended as realistic examples, as opposed to illustrations of particular semantic questions.

We distributed the survey to the University of Cambridge systems research group, at EuroLLVM 2015, via John Regehr's blog, and via various mailing lists: gcc, llvmdev, cfe-dev, libc-alpha, xorg, a FreeBSD list, xen-devel, a Google C users list, and a Google C compilers list. We also sent it to some Linux and MSVC people, but we did not widely advertise within these communities. A key aim was to target an expert audience, rather than a wider population. The survey ran between 2015/04/10 and 2015/09/29, and received 323 responses. Of those, 223 included a name and/or an email address while 100 were anonymous. At the beginning of the survey, the responders were asked about their expertise by selecting from a fixed list the categories corresponding to them (multiple choices were allowed). Most had expertise in C systems programming and significant numbers reported expertise in compiler internals and in the C standard:

C applications programming	255
C systems programming	230
Linux developer	160
Other OS developer	111
C embedded systems programming	135
C standard	70
C or C++ standards committee member	8
Compiler internals	64
GCC developer	15
Clang developer	26
Other C compiler developer	22
Program analysis tools	44
Formal semantics	18
no response	6
other	18

The data we collected had a few duplicate submissions from non-anonymous responders. For these, the earlier submissions are not included in the numbers we present in this section. There may also be a small number of duplicates from anonymous people. As it

is hard to be certain here about exactly which are duplicates, we left these unchanged in the data. The small number means they should not significantly affect the results. There were also a few responses directly to the mailing lists, which we include in the text discussion that follows, but not in the numbers. In total, the responses include around 100 printed pages of textual comments, which we previously made available [N2015]. These are often more meaningful than the numerical survey results. In the remainder of this section, we go through each question of the survey, show the numerical results, and analyse them using a few representative textual comments. This is based on previously published discussions [N2014].

> **Question 1: How predictable are reads from padding bytes?**
> If you zero all bytes of a struct and then write some of its members, do reads of the padding return zero? (e.g. for a bytewise CAS or hash of the struct, or to know that no security-relevant data has leaked into them.)

Responses

Will that work in normal C compilers?

yes	116 (36%)
only sometimes	95 (29%)
no	21 (6%)
don't know	82 (25%)
I don't know what the question is asking	3 (1%)
no response	6

Do you know of real code that relies on it?

yes	46 (14%)
yes, but it shouldn't	31 (9%)
no, but there might well be	158 (49%)
no, that would be crazy	58 (18%)
don't know	25 (7%)
no response	5

Additionally, responders which did not expect the idiom to always work were asked to check (potentially more than one) reasons from the following list:

you've observed compilers write junk into padding bytes	31
you think compilers will assume that padding bytes contain unspecified values and optimise away those reads	20
no response	150
other	80

Analysis From the responses, it is unclear what behaviour compilers currently provide (or should provide) for this idiom.

We see four main possible semantics, listed in order of decreasing predictability for the programmer and increasing looseness, and hence increasing permissiveness, for optimisers:

(a) Structure copies might copy padding, but structure member writes never touch padding.

(b) Structure member writes might write zeros over subsequent padding.

(c) Structure member writes might write arbitrary values over subsequent padding, with reads seeing stable results.

(d) Padding bytes are regarded as always holding unspecified values, irrespective of any byte writes to them, and so reads of them might return arbitrary and unstable values.

In the responses, one side is arguing for a relatively tight semantics:

- A modest but significant number of respondents say they know real code that relies on this.
- In some circumstances, it seems important to provide systems programmers with a mechanism to ensure that no information is leaked via padding. Rewriting structure definitions to make all padding into explicit fields may not be practicable, especially if one wants to do so in a platform-independent way, and so option (d) is not compatible with this. Option (c) makes it possible but awkward to prevent leakage, as, there, padding must be re-zero'd after member writes.
- In some circumstances, programmers may rely on predictable padding values, at least in the absence of structure member writes, e.g. for `memcmp()`, hashing, or compare-and-swap of struct values. Again, (d) is not compatible with this, and (a) or (b) are preferable. But it is not clear whether any of those usages are common or essential.
- More deterministic semantics is in general desirable for debugging.
- One respondent suggests that the MSVC compiler provides (a).

The other side appears to consider what optimisations compilers actually do, which may force a relatively loose semantics:

- Structure assignments observably sometimes do copy padding.
- Some respondents expect that writes to a single member might overwrite adjacent padding with zeros, in a wide write. But we do not yet have concrete cases on modern mainstream architectures where this or any of the following three actually happen.
- Some respondents expect that writes to a single member might overwrite adjacent padding with arbitrary values, in a wide write.
- Many respondents suggest that padding bytes could be deemed by the compiler as holding unspecified values irrespective of any source-code writes of those bytes, and hence that such writes could be omitted and later reads of the padding bytes be given arbitrary (and unstable) values. But this would mean that there is no way for the programmer to avoid leakage or provide deterministic padding values. It is unclear whether this actually happens at present.
- Joseph Myers, a developer of GCC and member of WG14, suggests for GCC: a *plausible sequence of optimizations is to apply SRA (scalar replacement of aggregates), replacing the* `memset` *with a sequence of member assignments (discarding assignments to padding) in order to do so.* This could require something equivalent to the above to make the existing compiler behaviour admissible, but it is similarly unclear to us whether it actually does at present.
- David Chisnall, at the time a member of the University of Cambridge's systems research group, suggests that by the time the optimisation passes operate, padding has been replaced by explicit fields, so neither over-wide writes nor permanently-undefined-value behaviour will occur.

Question 2: Uninitialised values

Is reading an uninitialised variable or struct member (with a current mainstream compiler):

a) undefined behaviour (meaning that the compiler is free to arbitrarily miscompile the program, with or without a warning)

b) going to make the result of any expression involving that value unpredictable

c) going to give an arbitrary and unstable value (maybe with a different value if you read again)

d) going to give an arbitrary but stable value (with the same value if you read again)

e) don't know

f) I don't know what the question is asking

(This might either be due to a bug or be intentional, e.g. when copying a partially initialised struct, or to output, hash, or set some bits of a value that may have been partially initialised.)

Responses

a)	139 (43%)
b)	42 (13%)
c)	21 (6%)
d)	112 (35%)
e)	3 (0%)
f)	2 (0%)
no response	4

Do you know of real code that relies on it?[1]

yes	27 (11%)
yes, but it shouldn't	52 (22%)
no, but there might well be	63 (27%)
no, that would be crazy	80 (34%)
don't know	10 (4%)
no response	91

Analysis The lack of consensus makes it hard to infer what behaviours is currently provided by compilers, but the responses are dominated by the "undefined behaviour" and "arbitrary but stable" options, with a roughly bimodal distribution. It is not clear whether people are actually depending on the latter, beyond the case of copying a partially initialised struct, which it seems must be supported, and comparing against a partially initialised struct, which it seems is done sometimes. Many respondents mention historical uses to attempt to get entropy, but that seems now widely regarded as a mistake. There is a legitimate general argument that the more determinacy can be provided, the better for debugging. But it seems clear that GCC, Clang, and MSVC do not at present exploit the undefined behaviour specified by the ISO standard in the correctness of optimisations, which could lead to arbitrarily miscompiled code. One respondent however suggested that (at the time of the survey) "LLVM is moving towards treating this as UB in the cases where the standards allow it to do so".
For GCC, Joseph Myers said:

[1]This question was only asked to responders who chose any of the answers b), c) or d).

- *Going to give arbitrary, unstable values (that is, the variable assigned from the uninitialised variable itself acts as uninitialised and having no consistent value). (Quite possibly subsequent transformations will have the effect of undefined behavior.) Inconsistency of observed values is an inevitable consequence of transformations* `PHI (undefined, X) -> X` *(useful in practice for programs that don't actually use uninitialised variables, but where the compiler can't see that).*

For MSVC, one respondent said:

- I am aware of a significant divergence between the LLVM community and MSVC here; in general LLVM uses "undefined behaviour" to mean "we can miscompile the program and get better benchmarks", whereas MSVC regards "undefined behaviour" as "we might have a security vulnerability so this is a compile error / build break". First, there is reading an uninitialized variable (i.e. something which does not necessarily have a memory location); that should always be a compile error. Period. Second, there is reading a partially initialised struct (i.e. reading some memory whose contents are only partly defined). That should give a compile error/warning or static analysis warning if detectable. If not detectable it should give the actual contents of the memory (be stable). I am strongly with the MSVC folks on this one - if the compiler can tell at compile time that anything is undefined then it should error out. Security problems are a real problem for the whole industry and should not be included deliberately by compilers.

It looks as if several compiler writers are saying (b), while a significant number of programmers are relying on (d) (which may also be what MSVC supports).

> **Question 3: Can one use pointer arithmetic between separately allocated C objects?**
>
> If you calculate an offset between two separately allocated C memory objects (e.g. malloc'd regions or global or local variables) by pointer subtraction, can you make a usable pointer to the second by adding the offset to the address of the first?

Responses

Will that work in normal C compilers?

yes	154 (48%)
only sometimes	83 (26%)
no	42 (13%)
d) don't know	36 (11%)
I don't know what the question is asking	3 (0%)
no response	5

Do you know of real code that relies on it?

yes	61 (19%)
yes, but it shouldn't	53 (16%)
no, but there might well be	99 (31%)
no, that would be crazy	73 (23%)
don't know	27 (8%)
no response	10

When asked to clarify, in the case they had answered that the idiom does not always work, 51 responders selected "you know compilers that optimise based on the assumption that that is undefined behaviour"; 51 wrote a custom reason; and 228 did not answer.

Analysis We see that a large number of responders expect this idiom to be supported by compilers, and a non-negligible number report real code that relies on it:

- it is used in both Linux and FreeBSD for per-CPU variables. (Robert Watson, David Chisnall, and Paul McKenney)

- it is used for calculating a fingerprint of bytes in memory, for FIPS validation. The OpenSSL FIPS canister is one example. (Jonathan Lennox)
- *QEMU relies heavily on pointer arithmetic working in the "obvious" way on the set of machines/OSes we target. I know this isn't strictly standards compliant but it would break so much real code to enforce it that I trust that gcc/clang won't do something dumb here. (IIRC there was a research project that tried to enforce no buffer overruns by being strict to the standards text here and they found that an enormous amount of real world code did not work under their setup.)* (Peter Maydell)
- *The MPI Forum (which includes me) recognizes the problems of address arithmetic in C and has utility functions to make it possible to do things that are necessary, but in a portable way (of course, the implementation is platform specific).* (Jeff Hammond)
- *It's undefined behavior, but an implementation is permitted to use undefined behavior in its own code since it ostensibly has control over it. An example of this is the glibc* `strcpy` *source (generic C version) using a* `ptrdiff_t` *between src and dest to create a single offset and then walking through only one pointer.* (Chris Young)
- *I've seen this done in an OS to link system function calls into ELF binaries* (anon)
- *For example, coreboot contains a mechanism to relocate part of its data segment from one base address to another during execution. All accesses to globals in that segment go through a wrapper which after the migration uses arithmetic like this to find the new address (e.g. something like*
  `return !migration_done ? addr : addr - old_base + new_base;`*).* (anon)

While there are still some embedded architectures with distinct address spaces, it is not clear that "mainstream" C (e.g. GCC/Clang on an x86_64 or arm64 architecture at user mode) should be concerned with this. Some responders nonetheless mention a few cases that could be identified as language dialects or implementation-defined choices:

- *On PICs and MCS51s, the two objects could actually be in different data spaces (e.g. RAM vs flash memory). It would be nonsense to do pointer arithmetic on them.* (David Grayson)
- the IBM AS/400
- *This is mostly a problem with hardware architecture like GPUs.* (JF Bastien)
- *"No", because of e.g. segmentation in MS-DOS. MS-DOS lives (unfortunately).* (anon)
- the CHERI architecture [Woo+14].

It is straightforward to define the semantics for either language dialects in which out-of-bound pointer arithmetic is always or never allowed. However, it appears clear that current compilers sometimes do optimise based on an assumption (in a points-to analysis) that this does not occur (see comments from Joseph Myers and Dan Gohman). How could these be reconciled with a permissive dialect?

- One could argue that the use cases should be rewritten, but that seems unlikely to actually happen in practice.

- One could turn off the relevant optimisations (e.g. with `-fno-tree-pta` for GCC).
- The analysis could treat inter-object pointer subtractions as giving integer offsets that have the power to move between objects (though the possibility of occurrences split across compilation units might mean one has to be too pessimistic).
- One could add additional annotated pointer or integer types to identify in the source where this might occur.

Question 4: Is pointer equality sensitive to their original allocation sites?
For two pointers derived from the addresses of two separate allocations, will equality testing (with ==) of them just compare their runtime values, or might it take their original allocations into account and assume that they do not alias, even if they happen to have the same runtime value? (for current mainstream compilers)

Responses

it will just compare the runtime values	141 (44%)
pointers will compare nonequal if formed from pointers to different allocations	20 (6%)
either of the above is possible	101 (31%)
don't know	40 (12%)
I don't know what the question is asking	16 (5%)
no response	5

Do you know of real code that relies on it?

yes	60 (26%)
yes, but it shouldn't	16 (7%)
no, but there might well be	68 (29%)
no, that would be crazy	46 (20%)
don't know	37 (16%)
no response	96

Analysis The responses are roughly bimodal: many believe "it will just compare the runtime values", while a similar number believe that the comparison might take into account how its pointer operands where constructed (as opposed to only looking at their concrete runtime representation). Of the former, 41 "know of real code that relies on it". In its current wording, the ISO standard specifies that (in the base case) two object pointers only compare equal if they are "pointers to the same object". The precise meaning of this phrase is however unclear, in particular if one allows pointer arithmetic to move across memory object boundaries. In practice, we see that GCC does sometimes takes allocation provenance (some ghost state regarding how pointers where constructed) into account, with the result of a comparison (in an one-past case, comparing `&p+1` and `&q`) sometimes varying depending on whether the compiler can see the provenance, e.g. on whether it is done in the same compilation unit as the allocation. We do not see any reason to forbid that, especially as this `n+1` case seems unlikely to arise in practice, though it does complicate the semantics, effectively requiring a nondeterministic choice at each comparison of whether to take provenance into account. But for comparisons between pointers formed by more radical pointer arithmetic from pointers originally from different allocations, as in Question 3, it is not so clear.
The best "mainstream C" semantics here seems to be to make a nondeterministic choice at each comparison of whether to take some provenance information into account, or to

just compare the runtime representation of pointers. This corresponds to the third option given to responders, and, in the vast majority of cases, the two will coincide.

> **Question 5: Can pointer values be copied indirectly?**
> Can you make a usable copy of a pointer by copying its representation bytes with code that indirectly computes the identity function on them, e.g. writing the pointer value to a file and then reading it back, and using compression or encryption on the way?

Responses

Will that work in normal C compilers?

yes	216 (68%)
only sometimes	50 (15%)
no	18 (5%)
don't know	24 (7%)
I don't know what the question is asking	9 (2%)
no response	6

Do you know of real code that relies on it?

yes	101 (33%)
yes, but it shouldn't	24 (7%)
no, but there might well be	100 (33%)
no, that would be crazy	54 (17%)
don't know	23 (7%)
no response	21

The responders are overwhelmingly positive in their expectation that compilers supports this idiom, and they provide many specific use cases in their comments, e.g.:

- Marshalling data between guest and hypervisor. (Jon)
- *You can go much stronger than that. Many security mitigation techniques rely on being able to XOR a pointer with one or more values and recover the pointer later by again XORing with one or more possible different values, (whose total XOR is the same as the original set).* (Richard Black)
- *Windows /GS stack cookies do this all the time to protect the return address. The return address is encrypted on the stack, and decrypted as part of the function epilogue.* (Austin Donnelly)
- *I've written code for a JIT that stores 64-bit virtual ptrs as their hardware based 48-bits. This is a valuable optimisation, even if it's not strictly OK.* (anon)
- *I've also worked on 64-bit ports of 32-bit code that purposefully keep 32-bit pointer-like ints to keep their memory footprint low (with appropriate calls to tell the system exactly where we want our data).* (anon)
- *The current Julia task-scheduler does this, by way of copying a task's stack into a buffer, and copying the buffer back to the stack later.* (Arch D. Robison)
- *BLOSC (http://blosc.org/) does something like this. It compresses data stored in RAM with the goal of reading compressed data from RAM into L1 cache faster than an uncompressed memcpy. If pointer values can't be copied indirectly, then BLOSC users are in trouble.* (Alan Somers)

The responses about current compiler behaviour are clear that in simple cases, with direct data-flow from original to computed pointer, both GCC and Clang support this. But for computation via control-flow, it is not so clear:

- with respect to GCC: *Yes, it is valid to copy any object that way (of course, the original pointer must still be valid at the time it is read back in). It is not, however, valid or safe to manufacture a pointer value out of thin air by, for example, generating random bytes and seeing if the representation happens to compare equal to that of a pointer. See DR#260. Practical safety may depend on whether the compiler can see through how the pointer representation was generated.* (Joseph Myers)

- with respect to Clang: *Pretty sure this is valid behaviour. We go out of our way to support this. Well, okay, it depends how indirectly. If you want to be completely loopy, this won't work in our compiler:*

```
bool isThisIt(uintptrt i) { return i == 0x12341234; }
void *launderpointer()
{
  int stackobj;
  for (uintptr_t i = 0; ; ++i) {
    if (isThisIt(&stackobj + i)) {
      return (void*)(i - 0x12341234);
    }
  }
}
```

  because we may return false for every call to `isThisIt()` *even though I think it's technically valid. We generally forbid guessing the addresses of values where we're allowed to pick the address (ie., we fold* `&stackobj == (void*)rand()` *to false), but we didn't account for the case someone tries the entire address space in a loop. Don't care. Taking the pointer and capturing/escaping it is supported, we assume it may come back in from anywhere in the future, including by being typed in at the console.* (Nick Lewycky)

- Similarly for GCC in the discussion of a bug report https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752, the undefined behaviour for round-trip casts of modified pointers is mentioned (see comment 25) as being exploited by the alias analysis.

- *Some compilers require the computation of the pointer to somehow depend on the original pointer – you can round-trip through a file, but you can't just guess the address, even if you guess right (for instance, if you ask the user to type in a number and assume it's the pointer, and the user gets the number from a debugger, that will not work in practice).* (Richard Smith)

Overall, it appears that a reasonable "mainstream C" semantics should allow indirect pointer copying, but with some restriction to accommodate compiler alias analyses. This could be achieved by requiring a visible data-flow provenance path. It should allow pointers to be marshalled and read back in, and the simplest way of doing that is to allow any pointer value to be read in, with the compiler making no aliasing/provenance assumptions on such value, and with the semantics checking whether the numeric pointer value points to a suitable live object only when and if it is dereferenced.

> **Question 6: Pointer comparison at different types**
> Can one do == comparison between pointers to objects of different types (e.g. pointers to int, float, and different struct types)?

Responses

Will that work in normal C compilers?	
yes	175 (55%)
only sometimes	67 (21%)
no	44 (13%)
don't know	29 (9%)
I don't know what the question is asking	2 (0%)
no response	6

Do you know of real code that relies on it?	
yes	111 (35%)
yes, but it shouldn't	47 (15%)
no, but there might well be	107 (34%)
no, that would be crazy	27 (8%)
don't know	17 (5%)
no response	14

Analysis The phrasing of this question was ambiguous, which affected some of the responses. We were intending to ask about the application of the comparison operator to pointers that have first been cast to a common type (`void*`, or `char*`), but which point to memory objects with different types. Without the casts, there is a constraint violation (type error) according to the ISO standard, though in the absence of the `-pedantic` flag most compilers silently accept such code.

With the casts, the responses seem clear that it should be allowed, save for architectures with segmented memory (which are nowadays unusual) or where the pointer representations are different (again this is not the case for mainstream architectures).

- *There are a lot of examples of this, in particular in libc, or possibly implementations of vtables.* (anon)

Some of the responses suggest that when compiling with the `-fstrict-aliasing` flag, such comparisons may be treated as evaluating to false, e.g.

- *Depends on strict-aliasing flags? I think LLVM TBAA might optimise this sort of check away?* (Chris Smowton)

We suspect these responses are under the assumption that the question was referring to mistyped comparisons (with no casts on the operands).

> **Question 7: Pointer comparison across different allocations**
> Can one do < comparison between pointers to separately allocated objects?

Responses

Will that work in normal C compilers?	
yes	191 (60%)
only sometimes	52 (16%)
no	31 (9%)
don't know	38 (12%)
I don't know what the question is asking	3 (0%)
no response	8

Do you know of real code that relies on it?	
yes	101 (33%)
yes, but it shouldn't	37 (12%)
no, but there might well be	89 (29%)
no, that would be crazy	50 (16%)
don't know	27 (8%)
no response	19

Analysis This idiom seems to be widely used for lock ordering and collection data structures. As for Question 3, there's a potential issue for segmented memory systems (where the implementation might only compare the offset). But we see these as being outside the scope of the "mainstream" C we aim to capture. For recent systems, it is unclear what reason would lead implementations to forbid it. Regarding GCC, Joseph Myers commented:

- *This is likely to work in practice (for e.g. implementing functions like* `memmove`*) although not permitted by ISO C.*

However, for the same question in the first survey, Hans Boehm commented:

- *May produce inconsistent results in practice if* `p` *and* `q` *straddle the exact middle of the address space. We've run into practical problems with this. Cast to* `intptr_t` *first in the rare case you really need it.*

> **Question 8: Pointer values after lifetime end**
>
> Can you inspect (e.g. by comparing with ==) the value of a pointer to an object after the object itself has been free'd or its scope has ended?

Responses

Will that work in normal C compilers?	
yes	209 (66%)
only sometimes	52 (16%)
no	30 (9%)
don't know	23 (7%)
I don't know what the question is asking	1 (0%)
no response	8

Do you know of real code that relies on it?	
yes	43 (14%)
yes, but it shouldn't	55 (18%)
no, but there might well be	102 (33%)
no, that would be crazy	86 (28%)
don't know	18 (5%)
no response	19

The ISO standard specifies that when the lifetime of an object ends, the value of any live pointer referring to it becomes unspecified. Strictly compliant code can therefore not rely on this. However, we can see that the responders largely expect this to work in practice, and they include various use cases:

- *The pointer itself is still valid, and can be compared. Dereferencing the pointer can't.* (Warner Losh)
- *A pointer is a value which does not cease to have a value because you happened to pass that value to a function called free (or any other function annotated with* `_Frees_ptr_`*) but the set of things that it would be reasonable to do with such a pointer would be extremely limited.* (Richard Black)
- *Where I've seen this is code like this:*
  `free(myptr); release_extra_data_keyed_by_pointer(myptr);` (Jorg Brown)
- *A common pattern that relies on this is calling* `realloc` *and "checking whether it moved" to decide whether to update other copies of the pointer.* (Nick Lewycky)
- *As discussed in Q4, the current stable version of ntpd does this.* (Pascal Cuoq)

- *You can't deference the pointer, but the value remains valid. The only good use for it I can think of it to log a debugging message (which would only be useful if one also logged the allocate). In fact, I have logged such messages myself when unloading a loadable kernel driver (because all evidence of what had been at those pages was gone; so, anything faulting referencing the unloaded driver would be a complete mystery).* (Herbie Robinson)

There are debugging environments that will warn of it, however, e.g.:

- *Microsoft PREfast has a warning for use of a pointer after freeing it.* (Austin Donnelly)

And the practice appear to not be guaranteed to work with GCC:

- *Such a comparison may not give meaningful or consistent results (although the consequences are likely to be bounded in practice).* (Joseph Myers)

The "pointer lifetime-end zap" semantics currently mandated by the ISO standard has been and remains the topic of active discussions both at WG14 and WG21, in particular because it is at odds with well established concurrent algorithms [N2369; P1726R4].

Question 9: Pointer arithmetic

Can you (transiently) construct an out-of-bounds pointer value (e.g. before the beginning of an array, or more than one-past its end) by pointer arithmetic, so long as later arithmetic makes it in-bounds before it is used to access memory?

Responses

Will that work in normal C compilers?

yes	230 (73%)
only sometimes	43 (13%)
no	13 (4%)
don't know	27 (8%)
I don't know what the question is asking	2 (0%)
no response	8

Do you know of real code that relies on it?

yes	101 (33%)
yes, but it shouldn't	50 (16%)
no, but there might well be	123 (40%)
no, that would be crazy	18 (5%)
don't know	14 (4%)
no response	17

Analysis This is unambiguously disallowed by the ISO standard, which makes such arithmetic undefined behaviour. However, from the answers, it seems that this is often assumed to work, e.g.:

- *All the time. All the time.* (anon)

- *The Numerical Recipes in C rely on it; that's widely-used code in the physics community with some pretty horrible (and probably illegal) C code. This code explicitly stores and passes out-of-bounds pointers. If I index a multi-dimensional array manually, then I there's a chain of arithmetic like* `p + i * di + j * dj + k * dk` *or so, where p is a pointer and the others are integers, and I don't pay attention to the order in which these are evaluated. This just may temporarily lead to out-of-bounds pointers, depending on the order of evaluation.* (Erik Schnetter)

- *Tcpdump does a bit of this where they create a variable from an array and then check it is in bounds* (Brooks Davis)

- *Yeah, we didn't even bother with this one in clang* `-fsanitize=undefined`. (Nick Lewycky)

On the other hand, compilers may in fact not guarantee this to work:

- *This is not safe; compilers may optimise based on pointers being within bounds. In some cases, it's possible such code might not even link, depending on the offsets allowed in any relocations that get used in the object files.* (Joseph Myers)
- *The situation has not gotten friendlier to old-school pointer manipulations since* https://lwn.net/Articles/278137/ *was written in [This is a case where GCC optimised away a comparison involving an out-of-bounds pointer] The pattern could still be found in code exposed to malicious interlocutors in 2013:* https://access.redhat.com/security/cve/CVE-2013-5607 (Pascal Cuoq)
- *Pretty sure this one I've seen buggy code optimised away by real compilers.* (David Jones)

The answers expose a point of tension between programmers and implementations. The prevalence of transiently out-of-bounds pointer values in real code suggests it is worth seriously asking the cost of disabling whatever compiler optimisation is done based on this, to provide a simple predictable semantics.

> **Question 10: Pointer casts**
>
> Given two structure types that have the same initial members, can you use a pointer of one type to access the initial members of a value of the other?

Responses

Will that work in normal C compilers?

yes	219 (69%)
only sometimes	54 (17%)
no	17 (5%)
don't know	22 (6%)
I don't know what the question is asking	4 (1%)
no response	7

Do you know of real code that relies on it?

yes	157 (50%)
yes, but it shouldn't	54 (17%)
no, but there might well be	59 (19%)
no, that would be crazy	22 (7%)
don't know	18 (5%)
no response	13

Analysis The ISO standard allows this when such structures appear as members of a union (though the *common initial sequence* mechanism). The more general case that this question raises is however made illegal by the effective types rules. From the responses, it is however clear that this is commonly used:

- *LLVM's hand rolled rtti does this!* (JF Bastien)
- *The FreeBSD kernel and many other things do this. Most anything that uses structs to access IPv4 and IPv6 header data.* (Brooks Davis)
- *This is very common. It is often achieved by simply making the first member of the second structure an instance of the first structure, but in some cases (e.g. the Berkeley socket address types) even dissimilar views to the same representation data are used at different times.* (Ethan Blanton)

- *Lots of code uses this type punning.* (Warner Losh)
- *This happens all the time. Not just restricted to initial members, using the* `CONTAINING_RECORD()` *macro.* (Austin Donnelly)
- *Guaranteed by the standard only if the structures are members of the same union (clause 6.5.2.3, structure and union members) but it will normally work for bare structures. Very common for implementing object-oriented polymorphism, e.g. in bytecode interpreters.* (Tony Finch)
- *I can swear I've seen this in both Windows headers and the Linux kernel.* (anon)
- *This is a common idiom in X11 event handling code - you are forced into it by the Xlib API which assumes that you can read the event type from the first member of the XEvent union regardless of which subtype of the union will be used to read the rest of the data.* (Peter Benie)
- *Half of the Win32 API, BSD sockets and most OOP done in C would break.* (anon)
- *This is used so commonly that no compiler would dare to do anything than what you expect.* (anon)
- *This is used all over the place.* (Herbie Robinson)

However, this does not appear to be something one can rely on with GCC:

- *This is something that GCC tends to actually kill in practice (if strict aliasing is on); I've had to fix bugs that were caused by it.* (Jonathan Lennox)
- with respect to GCC: *This is not safe in practice (unless a union is visibly used as described in 6.5.2.3#6).* (Joseph Myers)

The responses suggest that a "mainstream C" semantics should support this, corresponding to what the behaviour of GCC appears to be when using the flag `-no-strict-aliasing`.

> **Question 11: Using unsigned char arrays**
>
> Can an unsigned character array be used (in the same way as a malloc'd region) to hold values of other types?

Responses

Will that work in normal C compilers?

yes	243 (76%)
only sometimes	49 (15%)
no	7 (2%)
don't know	15 (4%)
I don't know what the question is asking	2 (0%)
no response	7

Do you know of real code that relies on it?

yes	201 (65%)
yes, but it shouldn't	30 (9%)
no, but there might well be	55 (17%)
no, that would be crazy	6 (1%)
don't know	16 (5%)
no response	15

Analysis Here again it is clear that it is very often relied on for character arrays arising from identifiers (non-malloc'd), and it should work, with due care about alignment. For example:

- *BSD kernels use the* `caddr_t` *typedef for allocations that will be manipulated as bytes.* (Brooks Davis)
- *Encoder/Decoders do this all the time. They read bytes from a file into an unsigned char buffer, then cast a* `struct *` *on top of it to pick out the relevant fields and move on.* (Austin Donnelly)

This is however disallowed by the ISO standard, and GCC appear to make use of this:

- with respect to GCC: *No, this is not safe (if it's visible to the compiler that the memory in question has unsigned char as its declared type).* (Joseph Myers)

> **Question 12: Null pointers from non-constant expressions**
> Can you make a null pointer by casting from an expression that isn't a constant but that evaluates to `0`?

Responses

Will that work in normal C compilers?

yes	178 (56%)
only sometimes	38 (12%)
no	22 (6%)
don't know	67 (21%)
I don't know what the question is asking	11 (3%)
no response	7

Do you know of real code that relies on it?

yes	56 (18%)
yes, but it shouldn't	21 (6%)
no, but there might well be	113 (37%)
no, that would be crazy	63 (20%)
don't know	50 (16%)
no response	20

Analysis While the ISO standard only provides for the construction of null pointer values from a constant integer expression, the majority of responders expect this to work. The only exception seems to be some (unidentified) embedded systems.

- `NULL` *was until maybe C99 or so only conventionally zero, and on some embedded platforms it in practice had a nonzero value. I have not seen this in a very long time.* (Ethan Blanton)
- *Some embedded compilers use a non-zero null pointer so they can point it at unaddressable memory, when the zero page is addressable.* (Richard Smith)
- with respect to GCC: *In practice this is safe with GCC (as a consequence of casting between pointers and integers working), although not guaranteed by ISO C.* (Joseph Myers)

> **Question 13: Null pointer representations**
> Can null pointers be assumed to be represented with `0`s?

Responses

Will that work in normal C compilers?

yes	201 (63%)
only sometimes	50 (15%)
no	54 (17%)
don't know	7 (2%)
I don't know what the question is asking	4 (1%)
no response	7

Do you know of real code that relies on it?

yes	187 (60%)
yes, but it shouldn't	61 (19%)
no, but there might well be	42 (13%)
no, that would be crazy	7 (2%)
don't know	12 (3%)
no response	14

Analysis The ISO standard leaves this underspecified; strictly compliant code should therefore not rely on any particular representation. We see however that a large majority expect this to work for mainstream systems. In particular, we see that, unlike for the previous question, a majority reports knowing real code relying on it.

- *For all targets supported by GCC, yes.* (Joseph Myers)
- *My understanding is that (1)* `memset`*-ing a pointer to zero is NOT guaranteed by the spec to produce a null pointer, but that (2) it does on all systems that most people care about, and that there is real code that relies on that. Being able to* `memset` *a struct to zero and have all the fields come out null/zero is convenient enough that I kind of wish the spec would change in this regard.* (Matthew Steele)
- *Note that the POSIX committee is currently discussing a requirement that a pointer value with all bits zero be treated as a null pointer (the requirement is specifically that* `memset()` *on a structure containing pointers initialize those pointers to nulls).* (anon)

Architectures with segmented memory remains a potential exception, but we do not consider these relevant for the "mainstream" practice we are aiming at in this work:

- *But some segmented memory systems (IBM AS/400 IIRC) the* `NULL` *pointers isn't actually all-zeros since the pointer bits include a non-zero segment selector, so this break much code as above. I don't know of any current systems where that's actually the case however.* (Austin Donnelly)

Question 14: Overlarge representation reads

Can one read the byte representation of a struct as aligned words without regard for the fact that its extent might not include all of the last word?

Responses

Will that work in normal C compilers?

yes	107 (33%)
only sometimes	81 (25%)
no	44 (13%)
don't know	47 (14%)
I don't know what the question is asking	36 (10%)
no response	8

Do you know of real code that relies on it?

yes	40 (13%)
yes, but it shouldn't	39 (13%)
no, but there might well be	103 (35%)
no, that would be crazy	42 (14%)
don't know	67 (23%)
no response	32

Analysis This is sometimes used in practice and believed to work, with some restrictions regarding alignment and page-boundary alignment. However, some dynamic analysers such as valgrind and MSAN also appear to not support this.

- *The C version of* `strcmp()` *in FreeBSD is a good example* (Brooks Davis)
- *Lots of code assumes that if you can read any part of a word, you can read the full word. It won't always use the bits that aren't valid, but some crazy code does. Often you'd see this expressed as a variation on a theme of using* `bcopy` *where you might see a length computed by* `&a[1] - &a[0]` *rather than* `sizeof(*a)` *or* `sizeof(a[0])`. (Warner Losh)
- *Incidentally, LLVM will do this to stack accesses in its optimizer.* (Nick Lewycky)
- *If nothing else it requires the compiler to support something like GCC's* `__attribute__((__may_alias__));` *otherwise the read is undefined already due to aliasing violations.* (Rich Felker)
- *In practice this is safe with GCC except for possibly generating errors with sanitizers, valgrind etc. (but should be avoided except in special cases such as vectorized string operations).* (Joseph Myers)

A "mainstream C" semantics could either forbid this entirely (slightly limiting the scope of the semantics) or could allow it, for sufficiently aligned cases, if some switch is set.

> **Question 15: Union type punning**
> When is type punning - writing one union member and then reading it as a different member, thereby reinterpreting its representation bytes - guaranteed to work (without confusing the compiler analysis and optimisation passes)?

There is widespread doubt, disagreement and confusion here, e.g.:

- *always* (anon)
- *Never* (anon)
- *According to the standard never; in practice always.* (Chris Smowton)
- *As long as all accesses are via the union, and not, say, by taking separate pointers to the union's fields.* (anon)
- *Type punning always works. The compiler knows very well which fields in a union have what offsets so it knows what writes to one union impact which fields in another member of the union. It should not be confused.* (Richard Black)
- *only when one of the types is a* `char` *type. otherwise, never guaranteed to work.* (David Jones)
- *GCC and Clang try to allow it when it's sufficiently obvious that you're doing type punning (for instance, when you're directly accessing a block-scope union variable). GCC documents this, Clang does not (and only really does it for GCC compatibility).* (Richard Smith)

- *You are allowed to pun the prefixes of structure types when the struct members in the prefix have the same types.* `unsigned char []` *and other types is probably OK. Otherwise, you are getting into strict aliasing problems.* (Tony Finch)

- *Per the standard? Never. The conforming way to do this is with* `memcpy` *to a local, and the compiler is plenty smart enough to not actually emit the* `memcpy` *or the local. GCC's documentation claims that they support this as long as you've declared the union in advance. This is pretty scary because it means lexically in advance. So two identical function bodies before and after an unrelated declaration introducing a union may change the generated code for the two functions. In practice these unions go into header files and come before the rest of your code, so people tend not to notice.* (Nick Lewycky)

2.3 Larger semantics test suite

The code examples part of our first survey were designed to act as probes for the semantics of C. While they proved confusing to the responders, they formed a useful test suite once we started developing our proposed memory object models based on the results from the surveys. From these, we built a larger collection of around 260 tests which explored many aspects of the design space for the memory object model, including the semantics of pointer values with provenance. This helped with the initial debugging of our memory object model. To facilitate the process, the tests can be run using a test harness, `charon`, that generates individual test instances from JSON files describing the tests and tools; `charon` logs all the compilation and execution output (together with the test itself and information about the host) to another JSON file for analysis. Using this infrastructure, we collected the results for the major compilers at various optimisation levels, static analysers, and other formalisation of C [note30]. It is important to note that these tests were designed to be concise illustrations of semantic questions regarding the design of the memory object model. They are not tuned to trigger interesting compiler behaviour, which might only occur in a larger context that permits some analysis or optimisation pass to take effect. As a result, in collected data the absence of observed optimisations is not conclusive, whereas their presence is. We also evaluated our memory object models against these tests against those same tools, as we discuss in Chapter 12.

2.4 Outcome of the surveys

The upshot of all this is complex. The surveys expose many areas where there are real disagreements between what many programmers among the relatively expert audience surveyed expect to work, and what compilers currently support in general. For some of these, the current standard does give a particular answer, while for others the standard is unclear or silent. The responses generally do not identify an uncontroversial way forward, but rather serve as a starting point for discussion:

- For pointer provenance, we have engaged with WG14 and the community, from 2015 to date to develop the model we describe in Chapters 8 and 9. WG14 has a working draft Technical Specification [N3005] showing how that could be integrated into the standard, and a straw poll at the WG14 meeting in February 2022 for the question

"Does WG14 wish to see this (or something similar) in some future version of the standard?" received 21 yes, 0 no, and 1 abstain votes.

- For the semantics of uninitialised reads, many discussions and working papers [N2089; N2221; notes98; cmom0006] have failed to reach a consensus. The Cerberus semantics we present in this thesis implements the "option (b)" we propose in [notes98].
- For the use of pointer values after the end of lifetime of the object they originally pointed to, McKenney and others have presented a series of papers to both WG14 and WG21 [N2369; P1726R4]. These have however not yet reached a consensus within the committees. In Cerberus, we offer a user-selectable switch in the memory object model offering both the "zapping" of pointer values (current ISO behaviour), and the absence of zapping.
- Regarding the interaction of pointer provenance and sub-objects, which also interact with the notion of effective types present in the text of ISO C11, we have explored possible design choices in committee papers with WG14 [note30], and WG21 [P1796R0], but a coherent design remains elusive. In Cerberus, we do not address this issue: provenance relates to the memory footprint of whole objects.

For us, this overall picture re-emphasises the need for a clear mechanised semantics of C that can be used both for discussion and as a test oracle. The divergence of opinion that the survey results and subsequent discussion expose illustrates the limitations of prose standardisations.

Chapter 3

Motivation for the semantics by elaboration, and introduction to Core

3.1 Advantages of a semantics by elaboration

We structure our model of the dynamics of C as an elaboration – a compositional translation – from a typed AST, close to the source of C, into Core, a language we designed specifically to be the target of this elaboration. This design offers the following advantages:

1. It makes syntactically explicit the many subtle behaviours left implicit by the syntax of C's expressions and statements, and disentangles their different facets. To name a few, these include: the implicit type conversions in expressions; the loose evaluation order of operands; and the partiality as a result of undefined behaviours. For these, the ISO standard succeeds in giving a mostly unambiguous specification, but they remain a major source of complexity in the dynamics of C expressions, as the dynamics of a single operator typically tightly combines several of these facets. By designing Core to only have simple, specialised constructs, the elaboration produces a specification where, for a given C operator, each aspect of their dynamics is separated into different syntactic constructs. To illustrate this point, consider C variables `x` and `y` that have been declared with types **`int`** and **`short`**. The elaboration of the expression `x + y` is elaborated into the following Core expression:

```
1   letweak (la : loaded integer, lb : loaded integer) =
2     unseq(load('signed int', x), load('signed short', y)) in
3   pure(case (la, lb) of
4     | (Specified(a : integer), Specified(b : integer)) =>
5       Specified(
6         catch_exceptional_condition('signed int',
7           conv_int('signed int', a) + conv_int('signed int', b)
8         )
9       )
10    | _ : (loaded integer, loaded integer) =>
11      undef(<<UB036_exceptional_condition>>)
12  end)
```

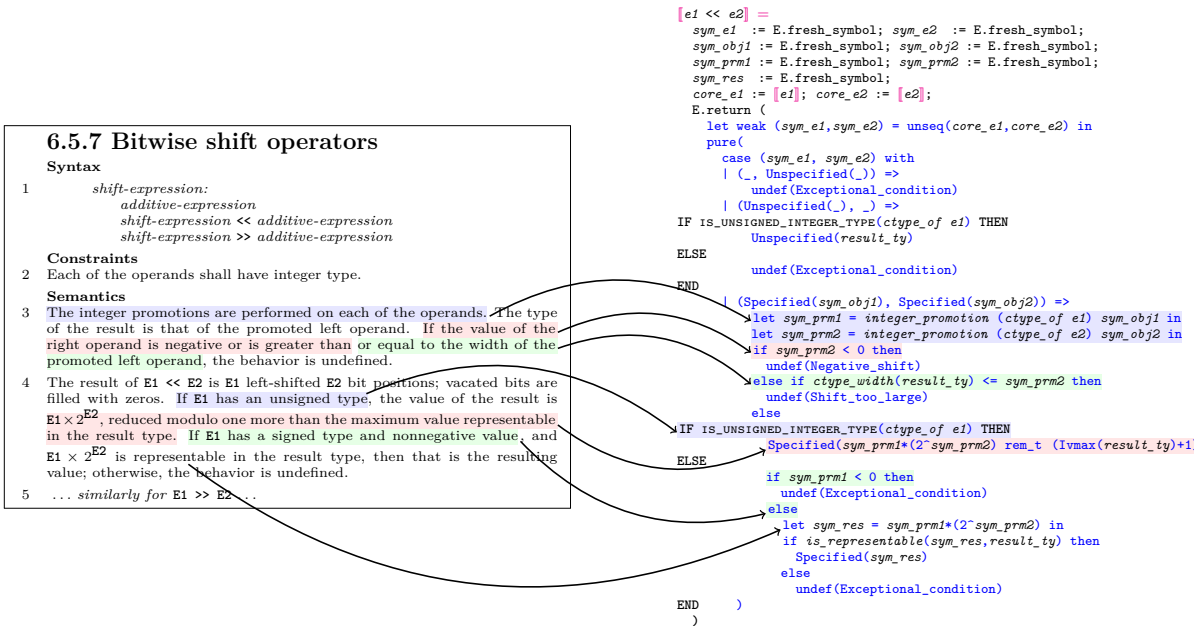
The constructs in lines 1 and 2 model the loose sequencing of the addition operator (i.e. the lack of sequencing between its operands); in line 2, the load access resulting from lvalue conversions on the identifier is explicit, and shows the lvalue types; the remaining lines model the computation of the addition operator while making explicit that no further interaction with the memory state occur; line 7 shows the integer promotion performed on the operands; and at line 11, a potential undefined behaviour is made explicit.

2. While Core technically has more constructs than C, and some may be unusual, they are all simple and directly motivated by some particular aspect of the C language that the elaboration aims to render explicit. Their simplicity means that the dynamics of Core can be written as a mostly straightforward operational semantics. As such, in our formalisation, the subtleties of C's semantics are kept within the definition of the elaboration function.

3. The elaboration is a function defined by induction over the typed AST of C, which effectively maps each C operator into a small Core program fragment. These fragments can be thought of as formal presentations of the part of the ISO standard prose specifying the dynamics C's expressions and statements (i.e. §6.5 and §6.8). A reader familiar with the standard prose only needs to learn the comparatively simple semantics of Core to recognise the correspondence with the prose. To illustrate this point, consider Figure 3.1. On the left is the passage of the ISO C11 standard specifying the syntax, statics, and dynamics for the `e1 << e2` left-shift expression operator. On the right is a typesetting of the clause of the elaboration function for that operator. Quite few features from Core are present here, and we will present them in the second half of this chapter and the next one. The key point here is that each sentence of the standard prose can easily be associated with some part of the Core program, as indicated by the arrows. However, as we show in Figure 3.2, the actual Lem implementation of the elaboration function contains boilerplate code, necessary for the implementation, but making it more challenging to read.

4. By keeping abstract the values over which the Core language operates, we are able to make a large portion of the formalisation (namely the elaboration function, and to some extent the dynamics of the Core language itself) parametric on some implementation-defined behaviours. The same is done for the memory object model: Core programs interact with a small set of abstract operations, whose concrete semantics is mostly irrelevant to the elaboration function.

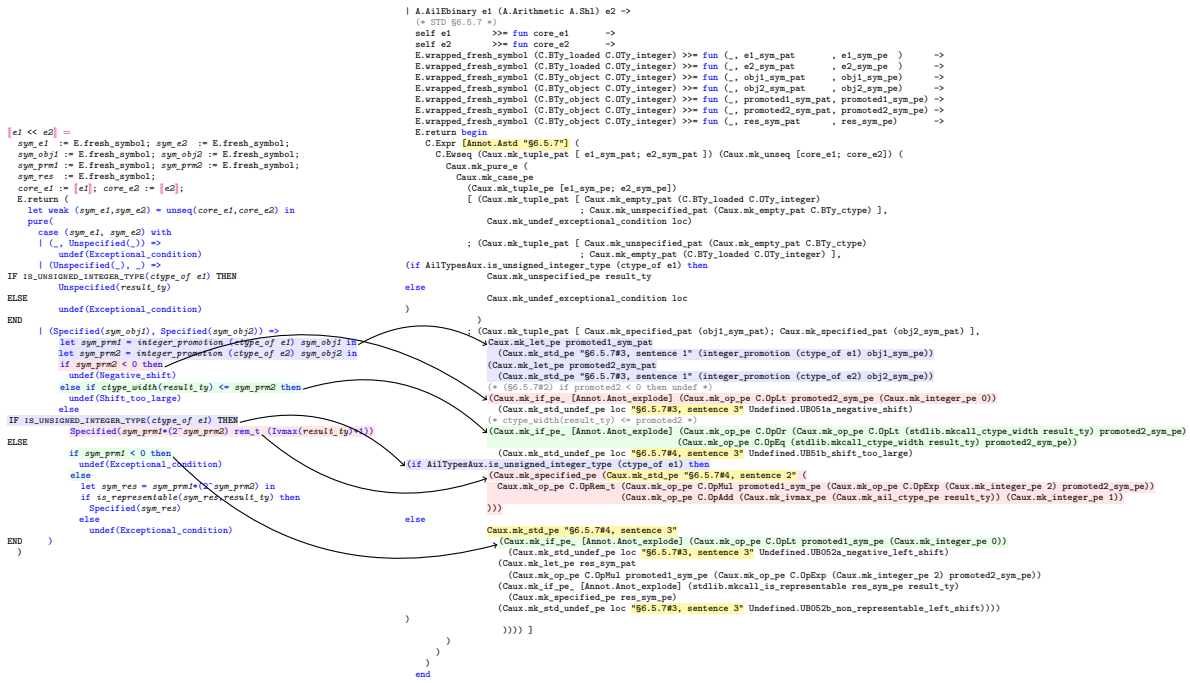
In total, the formalisation of the dynamics of C consist of the combination of: the elaboration from C to Core; the dynamics of the far simpler Core language, which we give as direct small-step operational semantics; and a memory object model. The latter defines the state used by the operational semantics of Core, along with the implementations of integer, pointer, and floating types, and the operators over these. The interaction between the operational semantics and memory object model is performed through an opaque interface. A different memory object model can therefore easily be swapped in without requiring a change in the elaboration function or the operational semantics of Core.



On the left is an excerpt of the ISO standard for bitwise shift operators, and on the right is the corresponding clause of the elaboration function (denoted by the $[\![\cdot]\!]$ operator). The arrows illustrate how the two closely relate to one another by connecting highlighted fragments of prose and their counterparts in the body of the elaboration. On the right, the first 4 lines construct fresh Core identifiers. The next line recursively calculates the elaboration of the sub-expressions *e1* and *e2*. This is done in a state monad for fresh identifiers. The remaining lines then show the construction of the Core expression elaborating a left shift operator. We typeset in lower-case blue the Core constructors. The first arrow shows how the integer promotions on the operands are performed as calls to functions in Core. The second and third arrows shows the dynamic tests on the value of the right operand, which guard potential undefined behaviours `Negative_shift` and `Shift_too_large`.

Note that there are two kinds of conditionals. The first is for those parts of the elaboration function itself (that we typeset in upper-case boldface); these are static conditionals. For example, the fourth arrow is branching on the statically known type of the first operand of shift. The second is for the Core conditionals (in lower-case blue), which are executed at runtime by the Core operational semantics. For example, the second arrow (which relates red regions) shows a conditional on the value of the right operand.

Figure 3.1: Sample extract of the C11 standard and the corresponding pretty-printed clause of the elaboration function



On the right, we show the actual implementation of this clause of the elaboration function in Lem. While it has the same structure as the pretty-printed version, there is additional boilerplate code making it harder to read and relate to the ISO standard. For example, the construction of the Core expressions is done through various auxiliary functions hiding annotation fields part of the Lem implementation of the Core AST. These annotations, e.g. source locations and typing, allow our tools to give user-friendly error messages, but at the same time are orthogonal to semantic content of the elaboration function. Other annotations remain visible here: the code highlighted in yellow annotates the Core expression with the relevant references to the ISO standard. Again this allows for better error messages.

Figure 3.2: The same pretty-printed clause of the elaboration function compared to its actual implementation in Lem

3.2 The Cerberus pipeline

To produce a executable semantics on real C translation units, the elaboration function and runtime of the Core language sit within the Cerberus architecture shown in Figure 3.3. This pipeline takes preprocessed C translation units, and parses them to the Cabs AST; the parser (using Menhir [PR05]) closely follows the ISO standard specification, except for the modification of Jourdan and Pottier [JP17] to deal with the grammar's ambiguity.

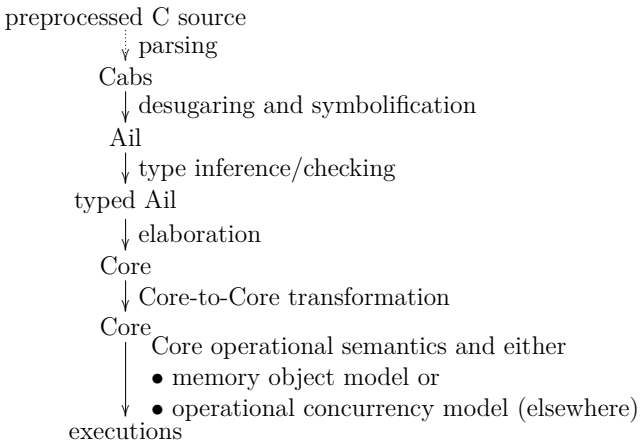


Figure 3.3: Cerberus architecture

The Cabs AST is desugared to the Ail intermediate representation, which remains very close to C. The desugaring involves the following: C operators which are defined by the ISO standard in terms of others are substituted (e.g. the prefix increment operator); coercions to rvalues, array and function decays are made explicit; struct/union and array initialisers are unfolded; the scoping and linkage of identifiers is resolved, and they are turned to symbols; constant expressions are evaluated (by creating a local instance of the whole pipeline down to the Core runtime. See the end of Section 11.1.2 for more details); `for` statements are translated into `while` statements; `continue` and `break` statements are translated into `goto` jumps; and type qualifiers and specifiers are turned into a canonical form. The desugaring stage is also where we detect and report statically checkable undefined behaviours and constraint violations which are not type related.

Next, the statics of C is modelled by typechecking the Ail representation. This is done while remaining agnostic of implementation-defined choices regarding the size and value range of types, and of the ranking relation between integer types. This stage produces a fully type-annotated Ail AST, in which rvalue coercions and array/function decays have been made explicit.

This is used as the input by the next stage: the elaboration function to Core. As discussed above, one of the design goals of the elaboration function is to be easily relatable to the ISO prose. As a result, it is defined as a simple induction over the Ail AST. This often produces Core expressions which are unnecessarily verbose, or that can reduced by partial evaluation. Both to ease the readability of the generated Core, and to optimise execution, several optional semantics-preserving Core to Core transformations can be performed.

The final stage is the Core runtime. Several Core programs, each elaborating a C translation unit, may be linked together. A driver then combines the Core thread-local operational semantics with either our candidate sequential memory object model or (in previous versions of Cerberus) the C11 operational concurrency model. Other tools use various points in this pipeline, as we describe later (Refined C, BMC-Cerberus, CN). Throughout the pipeline, checks for constraint violations and undefined behaviour are annotated with the corresponding clauses of ISO prose.

3.3 Overview of the Core language

We now give an overview of Core. The aim is to familiarise the reader with the main features of the language before Chapter 4, where we illustrate the subtleties of C expressions and statements by explaining them in terms of their elaboration into Core. We delay a complete and more formal presentation of Core's static and dynamics semantics until Chapter 6.

At heart, Core is a call-by-value, strongly typed expression language with recursive functions. There are two levels of expressions:

- **an inner pure language**, pure in the sense that computations cannot perform memory effects, but allowing divergence and "abnormal termination", which is used to model C's undefined behaviour.
- **an outer effectful language**, which consists of *memory actions* and *operators* used to interact with the memory state, and a small calculus of operators to order them. This fragment also features labelled expressions and a goto-like operator, and an operator for spawning threads.

The state in Core corresponds to an abstract representation of the state of a C program, in the spirit of the *abstract machine* alluded to in the ISO standard. Its concrete definition is part of the memory object model, which is external to the Core dynamics. The separation between pure and effectful expressions does not correspond to C's expressions and statements, because C's expressions are themselves stateful. In Core, control-flow manipulation is done using a combination of recursion and the goto-like operators, whereas in C it is mostly done using statements. Unlike statements, effectful expressions have a type and yield a value, derived from the pure expressions they contain.

The dynamics of Core is written as a small-step operational semantics, defined directly over the AST. The outer language semantics simply relates a configuration made of a state, an effectful expression, and a call stack into either another such configuration, or a terminal undefined variant. The semantics of the inner language is given as a big-step semantics.

3.3.1 Pure language

We now discuss the inner "pure" language, where all arithmetic computations in Core are performed. Its base values are: `Unit`; boolean constants; the abstract syntax of C types as values (to which we give the Core type `ctype`); and a class of *object values*. The latter are for values that can be stored in a memory object. They mirror the structure of values found in C, and have the Core types: `integer`; `floating`; `pointer`; arrays whose elements are of a given object type; and, structs and unions (whose members have object types). Then there are two constructors: `Specified`() which takes a object value as an argument, and is used for the values resulting from loading a properly initialised object; and `Unspecified`() which takes a C type as argument, and is used for the values resulting from loading an uninitialised object. These have type `loaded` T, where T is the type of value they hold (in the specified case).

Finally, one may build tuples and lists from the previous values.

In contrast to C and its multitude of sized integer types, all Core integer values belong to a single unbounded integer type. The arithmetic operators (`+`, `-`, `*`, `/`, the truncating and flooring modulos, and exponentiation) are defined with an unbounded

and total semantics (in particular division by zero is defined to zero). As we will show in Section 4.2, the burden of capturing C's bounded arithmetic and implicit conversions between the various integer types is done by the elaboration function. Similarly, the relational and equality operators, which are defined over integers, floating values, pointer values and `ctype`, yield a boolean. The elaboration of C's corresponding operators takes care of the mapping between C's zero/non-zero values and Core's boolean values. Over the boolean type we have the negation (`not`()), conjunction (`/\`) and disjunction (`\/`) operators. Importantly these do not correspond to C's `&&` and `||` operators, because the dynamics of the latter have additional aspects that are made explicit by the elaboration: first, like the relational and equality operators, they evaluate to a zero or non-zero integer; second, their operands are only evaluated lazily. Control is performed with an **`if`** operator (whose controlling expression is a boolean), pattern matching, or a recursive call to a function whose body is pure. Calls to functions are fully applied, and have a simple call-by-value substitution semantics. For convenience, values can be bound using a usual **`let`** binder.

To summarise, the pure language is an unsurprising expression language. For a concrete illustration, consider the following implementation of the factorial function:

```
fun fact (n : integer) : integer :=
  if n <= 1 then 1 else n * fact(n-1)
```

As with integers, Core has a single *floating* type into which all C floating types are mapped. The arithmetic, relational, and equality operators over this type are distinct from the integer variants (though for convenience we overload their notation). One must use explicit conversion between the integer and floating types using primitive functions: `Ivfromfloat`(), taking as its operand a floating value, and evaluating to an integer value; and `Fvfromint`() for the converse. For the integer types, there are additional constructs for building values which we describe shortly.

There are additional primitive functions for building structure and union values, and for lists and tuples, a pointer arithmetic operator, an operator for accessing structure and union members, and an operator to deal with function pointers. We omit these for now, as they are not needed for Chapter 4. In addition to the above, Core has two unusual features that we now discuss.

C types as values C types, as referred by their abstract syntax, are values in Core. They are written as one would in C, inside single quotes, e.g. `'signed int'`, and have the type `ctype`. Only two operations are defined over them: equality and a compatibility test (as defined in the C type system). This is sufficient to allow the elaboration function to be independent of some implementation-defined behaviour, for example the size of C's integer types. To illustrate this, consider the following auxiliary pure function:

```
fun is_representable_integer (n : integer, ty : ctype) : boolean :=
  Ivmin(ty) <= n /\ n <= Ivmax(ty)
```

It takes two parameters: an integer n and a C type ty, and returns a boolean value expressing whether the integer can be represented as a bit pattern in the C type. Its body uses two primitive functions constructing `integer` values: `Ivmin`() and `Ivmax`(), both of which take a C type as an operand, and evaluate respectively to the minimal and maximal integer value of that C type. These constructors are the mechanism by which the

Core function remains agnostic of the implementation-defined sizes and ranges of integer types. For this purpose, there are two other primitive functions of note: `Ivsizeof`() and `Ivalignof`(), both taking a C type as operand and evaluating respectively to the size (in bytes) and alignment constraint of their operand. The concrete definition of these functions is part of the memory object model, which defines them in terms of the implementation-defined choices for integer types, and that one needs to provide to the Core dynamics.

Explicit undefined behaviour Core has an **undef**() operator, which allows the elaboration function to explicitly express the possible occurrence of an undefined behaviour in the dynamics of a construct in C. This operator can have any type, and may therefore appear anywhere in a pure expression. It takes as operand the identifier for the particular undefined behaviour it raises; the set of identifiers is based on the Annex J.2 of the ISO standard. To illustrate its use, consider the following auxiliary pure function, which is used by the elaboration function to model the occurrence of an undefined behaviour in signed arithmetic operations producing out of range values:

```
fun catch_exceptional_condition (ty : ctype, n : integer) : integer :=
  if is_representable_integer(n, ty) then
    n
  else
    undef(<<UB036_exceptional_condition>>)
```

It takes as parameters an integer and a C type, and returns an integer value. Using the function previously described, it checks whether the integer parameter is in the range of the C type. If so, it simply returns the integer; otherwise, it indicates an undefined behaviour. One might worry that this operator gives rise to an effect in the pure fragment, which becomes observable in the presence of nondeterminism (as introduced by the effectful fragment we present next). This is however not the case because of how "severe" the evaluation of this operator is on a program's semantics: if any possible execution of a Core program evaluates an **undef**(), the program is given an "undefined" semantics. As a result, the evaluation order of this operator remains irrelevant; it only matters whether it is reachable or not.

The **undef**() operator allows the elaboration to make most of C's undefined behaviours syntactically visible in the Core programs it produces. The exceptions are unsequenced races, which remain implicit in Core programs, and any undefined behaviour that occurs as part of a memory action. These are respectively detected by the dynamics of Core, and some internal reductions of the memory object model.

The two functions we presented, and indeed anything within the pure fragment of Core, do not involve any interaction with a memory state, or any observable effects in term of the C abstract machine. Typically, the elaboration function uses them to model auxiliary computations that are implicitly part of the dynamics of C's expressions. We believe that the separation between pure and effectful expressions helps in the readability of the semantics, in particular by restricting the concerns about evaluation order to where it is relevant, namely interactions with the memory state and concurrency.

3.3.2 The effectful language

We now look at the constructs for interacting with the memory state, and how they are ordered. These form the "outer" language. Pure expressions are introduced either using the unary **pure**() operator (which is how effectful expression yield values), as an operand of an effectful construct, or as the body of function. As a convention, we call a function whose body is an effectful expression a *procedure*. These can only be called from an effectful expression. The elaborations of C functions are mapped to procedures in Core, while pure functions are used as auxiliaries. We make a stylistic distinction between *actions* which actually access or change the C memory state, and *operations* whose implementation depends on the concrete details regarding the representation of object types, or which need to access ghost state associated with pointer provenance.

Memory objects are allocated using the **allocate_object**() action taking an integer parameter specifying the alignment constraint, and a `ctype` parameter specifying the type for the object being allocated. The result is a pointer value to the object. As for all other memory actions and operators, all the operands are pure expressions. Deallocation is performed using the **kill**() action applied to a pointer value[1]. Accesses to memory objects are performed using the **load**() and **store**() actions, taking as arguments a C type specifying the desired footprint for the access (this corresponds to the type of the lvalue in a C memory access), a pointer value, and, for the latter, the object value being stored. The load evaluates to the loaded object value (which can be either `Specified`() or `Unspecified`()), and the store evaluates to unit. These are complemented by a slightly more convoluted action used to model the non-separable load and store performed by C's postfix expression and compound assignments, and additional actions to deal with C/C++11 relaxed concurrency. Among memory operations are relational and equality operations over pointer values, conversion operations between pointer and integer values, a validity check for pointer values, pointer arithmetic operators, and some others used to model aspects of the C standard library. These are discussed in Chapter 6.

Sequencing calculus Memory actions and operators are the atoms of the effectful language. To combine atoms, Core has a small calculus of ordering constructors which we designed to capture the looseness resulting from the under-specified evaluation order of expressions in C. We discuss the nature of these requirements and their modelling in Core in Section 4.3, and instead for now just introduce the syntax of the sequencing calculus.

The n-ary **unseq**() operator expresses the lack of sequencing between its effectful operands, and its value is the tuple combining their values. For example, for a read and write access that may be performed in any order, we write:

$$\textbf{unseq}(\textbf{load}(ty_1, ptr_1), \textbf{store}(ty_2, ptr_2, n))$$

which will evaluate to a pair holding the loaded object value and unit. The operands of this unsequencing operator can be any effectful expression, and therefore may perform more than one memory action. In these cases, the operator interleaves its operands at the granularity of their memory actions.

[1] There are technically two variants of this action to differentiate between the deallocation of dynamically allocated regions, and objects corresponding to C identifiers. There is also a variant of the create action producing "read only" objects and a "locking" store action to model **const**-qualified identifiers and string literals. Additionally, to model the initialisation of a **const**-qualified pointer to its own address, a store may be marked as making the object it writes to as read-only henceforth. These are not needed for the presentation in Chapter 4; we therefore defer their discussion to Chapter 6.

To introduce sequencing constraints, there are two let binders:

$$\textbf{letweak}\ pat = E_1\ \textbf{in}\ E_2 \qquad \textbf{letstrong}\ pat = E_1\ \textbf{in}\ E_2$$

The pattern *pat* is used to deconstruct tuples and loaded values resulting from the evaluation of the first operand. Both E_1 and E_2 are effectful expressions. The semantics of these operators is to first perform the actions of E_1 (which may contain one or more **unseq**() operators, and therefore have more than one allowed execution) to give a value which is substituted for *pat* into E_2. The **letweak** variant is "weak" in the sense that it does not force the execution of all actions in E_1 before the substitution. This is done by assigning a polarity to memory actions, and having the weak operator only sequence "positive" actions. We explain this mechanism in Chapter 4.

To illustrate the use of these binders, let us modify the previous expression by adding a third access storing to the same object the value that was read by the load, incremented by one:

$$\textbf{letstrong}\ (a_1, \_) = \textbf{unseq}(\textbf{load}(ty_1, ptr_1), \textbf{store}(ty_2, ptr_2, n))\ \textbf{in}\ \textbf{store}(ty_2, ptr_1, a_1 + 1)$$

We have now added an incrementing store action which must be executed last. The first two actions may still be executed in either order.

Because the **unseq**, and in some cases the **letweak**, operators leave some actions unsequenced, it is possible to introduce a "race" between accesses to overlapping memory. This is used to model C's *unsequenced races* and this situation has the same semantics as the evaluation of an **unseq**() operator.

This small sequencing calculus allows the elaboration function to syntactically precisely express the sequenced-before relation of the C expressions it is elaborating.

Calls to C functions and Core procedures While C functions are elaborated to Core procedures, the elaboration of C function calls does not make use of the Core procedure call operator. This is because the operand of C's function call operator can be an arbitrary expression (potentially reading from memory if we are dealing with function pointers). As a result, Core has a dedicated $\textbf{ccall}(e_f, e_1, \cdots, e_n)$ effectful operator, where the first operand e_f can be an arbitrary pure expression evaluating to a C function pointer. We opted to also have a "vanilla" procedure call operator, to allow the elaboration to use auxiliary procedures which do not correspond to any C function from the source being elaborated. Like their pure counterpart, both effectful call operators are fully applied and have a simple substitution semantics, which in the context of the sequencing calculus means that the evaluation of a procedure is atomic with respect to any unsequenced context – matching the C function calls as per the ISO standard.

Goto-like control operator Recursion and branching using the **if** operator are technically sufficiently expressive to allow the elaboration function to encode all of C's control constructs. Such an encoding would however result in some loss of the structure of the original C program in the generated Core, and in some corner cases would introduce significant code duplication in the generated Core. We instead equip the effectful fragment of Core with a goto-like operator. Labels are declared using the **save** operator:

$$\textbf{save}\ l(x_1 : \mathrm{ty}_1\ \texttt{:=}\ e_1, \ldots, x_n : \mathrm{ty}_n\ \texttt{:=}\ e_n)\ \textbf{in}\ E$$

This operator declares a label l, which like its C counterpart, is in scope in the entirety of the enclosing procedure. The declaration associates a continuation to the label, namely $C[E]$ where $C[\cdot]$ is the context in which the **save** operator is occurring.

This operator is also a binder for the variables $x_1, \ldots, x_n$ into the expression E. These variables are associated with pure expressions $e_1, \ldots, e_n$ defining their *default values*. The expression E will either be executed when the flow of the program execution goes through the **save** operator, in which case the x_i are substituted for their default values; or when a corresponding **run** operator is encountered:

$$\textbf{run}\ l(e_1, \ldots, e_n,)$$

In this case, the current continuation becomes the one associated to l, with its bound variables substituted by $e_1, \ldots, e_n$.

In Section 4.5, we show how the elaboration function makes use of these two operators to model C's jumping and iteration statements. In particular, the variable binding is used to model the implicit object creation and destruction that occur in a C program execution when the boundary of a block statement is crossed.

Chapter 4

Elaborating the intricacy of C

The semantics of C's expressions and statements is a part of the ISO standard which succeeds in being precise and unambiguous. This clear specification however, involves many subtleties which are often not well understood by programmers. For example, the evaluation of a simple arithmetic operator may involve memory accesses, with an unusually loose requirement on their ordering. The computation of the value often involves implicit conversions, based on the types of the operands.

As discussed in the previous Chapter, this abundance of implicit behaviour in the dynamics of C motivated our choice for a semantics by elaboration into a simpler Core language, whose design they guided. In this chapter, we present the major subtleties hidden in expressions and statements, starting from their specification in the text of the standard, and then showing how the elaboration function explicitly fleshes them out into our Core language. Save for the notion of unspecified values, which we discuss in Section 4.6, issues relating to the memory object model (e.g. the construction of pointers and how they may alias with one another) are mostly left under-specified by the text of the standard, and are the subject of discussions and differences of opinion between the authors of the standard, implementers, and system programmers. We defer the discussion of the C memory object model and the semantics of pointers to Chapter 8.

4.1 Underspecification in the ISO standard

The ISO standard aims at defining C such that it can be efficiently implemented on a wide range of environments. As a result, some aspects of the semantics are kept under-specified, leaving implementers some freedom. Furthermore, not all syntactically well-formed C programs are given a defined semantics, and determining whether a program is defined is not always decidable. Three levels of underspecification are used:

1. **Unspecified behaviour**, where the standard allows more than one possible behaviour. This affords the most freedom to implementations, because it does not require them to document their choice. For example, details regarding the representation of most types (e.g. the presence of padding bits, and the mapping between values and their memory bit-pattern), are mostly left unspecified. Another significant example is the order in which operands of most expression operators are evaluated. Given that expressions are effectful, from the presence of the assignment operator and function calls, this looseness in the evaluation order allows some expressions to be non-deterministic, or to exhibit situations akin to a concurrency data

race. Implementations are allowed to make inconsistent choices, either across separate translation units, or even between two instances of a same language construct within a single program execution (e.g. the evaluation of an arithmetic operator appearing within the body of a loop may differ from one iteration to another).

A somewhat distinct kind of unspecified behaviour arises from the occurrence of an *unspecified value*, typically from reading uninitialised memory or padding bytes of a structure. Whereas the implication of other instances of unspecified behaviour prescribed by the standard are mostly clear, the text of the standard introducing unspecified values (C11, §3.19.3) is subject to multiple interpretations – and these can have a large impact on the semantics of arithmetic and control operators.

2. **Implementation-defined behaviour**, which is an unspecified behaviour for which implementations are required to provide a documented choice. This requirement effectively makes implementations more consistent in the choice they take. Examples are the encoding and value range of character and integer types, and the result of converting between integer and pointer types.

3. **Undefined behaviour**, which arises from *non-portable or erroneous* situations, and for which the standard gives free rein to implementations. This allows efficient portability of the language; for example, not all hardware behaves the same when integer arithmetic overflows. By making it undefined behaviour, the standard allows implementations to simply use the underlying arithmetic instructions of their target architecture without having to deal with the semantics of the overflowing case, which would typically require adding costly runtime checks. This also enables compilers to perform ever more sophisticated optimisations, without imposing on them the burden of checking for the occurrence of situations where those optimisations are not sound, the complete detection of which is often not possible at compile-time. For example, a division by zero may trap on some hardware. By making this situation undefined, the standard enables compilers to not only use the underlying division instruction provided by the hardware (which usually performs no checks), but also allows optimisations such as loop invariant code motion to simply assume the absence of such an error.

   This particular example of optimisation also illustrates an important and counter-intuitive aspect of undefined behaviour: its occurrence at a certain point during program execution may very well manifest itself much earlier during the execution that one would expect. For example, consider a loop whose body contains a loop-invariant expression that is only executed from the second iteration of loop, and which performs a division by zero. If compiler optimisation hoists the evaluation of that expression before the loop, the execution of the optimised program may, on some hardware, trap before even performing the first iteration of the loop. This makes undefined behaviour a non-local phenomenon: the occurrence of an undefined behaviour at any point during program execution leaves that program without any meaningful semantics. It is worth observing that to remain useful, this "wildcard" notion of undefined behaviour needs to be restricted when considering programs receiving input from I/O. An input might cause an undefined behaviour for some possible value, which is however precluded by an invariant external to the C program. Lastly, the addition of concurrency in C11 introduced a new form of undefined behaviour in the form of data races.

These impact the elaboration to Core in different ways:

Unspecified behaviour To model the unspecified evaluation order of expressions, Core is equipped with a small calculus of sequencing and composition operators. The elaboration function does not fix the order in any way, but instead uses these operators to precisely express in the generated Core the sequencing constraints induced by the syntax of C's expressions. This is discussed in detail in Section 4.3.

Implementation-defined behaviour From the point of view of a formal semantics, the set of implementation-defined behaviour forms a parameter over which the model is abstracted. In the original design of Cerberus, we aimed for a fully implementation-agnostic semantics. We however later relaxed that goal when it became clear that virtually all useful C programs depend on some implementation-defined behaviour. Examples are any program mixing character constants and arithmetic, manipulating the representation of an object with integer type, having constant expressions, or generally interacting in some way with the environment. A fully agnostic semantics would have required symbolic evaluation, with a heavy cost on the runtime performance. Furthermore, while the encoding of integers is for example allowed by the ISO C11 standard to be either of sign-magnitude, ones' complement, or two's complement, mainstream implementations today all use two's complement and the upcoming revision of the ISO standard (C23) will make it the required encoding.

Cerberus however remains to a large extent agnostic, and as a result can still be instantiated to the choices of various implementations to simulate them. This was particularly helpful to validate the model. For example, our implementation of the C type system (over the Ail intermediate language) does not assume anything about how integer types are implemented. This requires some level of abstraction in the type system as a result of implicit type conversions, which are discussed in Section 4.2. We also opted to keep the elaboration function largely generic in implementation choices, by equipping Core with abstract constructors whose concrete definition have to be provided for the particular implementation one wants to model, and by hiding the memory object model and the semantics of pointers behind an opaque interface. There are two classes: abstract data constructors for dealing with the under specification of the representation of C types and their representation; and *implementation constants*, which can either be Core values or function names, for dealing with implementation-defined behaviour not relating to the memory (e.g. the behaviour of some arithmetic operators).

As an example of abstract data constructors, the Core standard library defines the pure Core function is_representable_integer() which, given an integer value and a C type, checks whether the value is within the range of the type, and returns a boolean accordingly. The function is simply implemented as follows:

```
fun is_representable_integer (n : integer, τ : ctype) : boolean :=
  Ivmin(τ) <= n /\ n <= Ivmax(τ)
```

where `Ivmin` and `Ivmax` are opaque data constructors evaluating respectively to the minimal and maximal integer value of the C integer type (when the Core runtime is set to simulate a particular implementation). Similarly, the elaboration to Core of the `sizeof()`

operator[1] uses another data constructor returning the size of a C type:

$$[\![\ \texttt{sizeof(E}^{\tau}\texttt{)}\ ]\!] \triangleq \texttt{Ivsizeof}([\![\ \tau\ ]\!])$$

This allows the elaboration to be independent of implementation details. It is only when a Core program is executed that these need to be provided to the runtime. In our model, these are specified as part of the memory object model, on which the runtime is parametrised. We describe the memory interface in Chapter 5 and implementation in Chapter 9.

The second class is for example used to model the right shift operator, which is implementation-defined when operating over a signed integer types and when the value of its right operand is negative. In the elaboration to Core, assuming τ is a signed integer type, this is expressed as follows (for the sake of clarity we omit orthogonal details):

$$[\![\ \texttt{E}_1\ \texttt{>>}^{\tau}\ \texttt{E}_2\ ]\!] \triangleq \begin{array}{l}\textbf{letstrong}\ (n_1, n_2) = \textbf{unseq}([\![\ \texttt{E}_1\ ]\!], [\![\ \texttt{E}_2\ ]\!])\ \textbf{in}\\ \ldots\\ \textbf{if}\ n_1\ \texttt{>=}\ 0\ \textbf{then}\ \ldots\ \textbf{else}\ \texttt{<SHR\_signed\_negative>}(\tau,\ n_1,\ n_2)\end{array}$$

the name `<SHR_signed_negative>` is an implementation-defined pure function taking the C type of the operator and the evaluated values of the operand, which at runtime will follow the behaviour of whichever implementation has be selected (as defined in an implementation module). For example, when simulating GCC in x86_64 performing a sign extension [GCC-ints], this function is defined as:

$$\begin{array}{l}\textbf{fun}\ \texttt{<SHR\_signed\_negative>}\ (\tau : \texttt{ctype}, n : \texttt{integer}, m : \texttt{integer}) : \texttt{integer}\ \texttt{:=}\\ \quad \textbf{let}\ n : \texttt{integer}\ = \text{encodeTwos}(\tau, n)\ \textbf{in}\\ \quad \text{decodeTwos}(\tau, (n/2\texttt{\^{}}m) + \text{ones\_prefix}(0, m, \text{ctype\_width}(\tau)))\end{array}$$

Undefined behaviour From a formal point of view, where the semantics of C is a mapping relating the syntax of a program to its set of allowed observables, the existence of undefined behaviours makes the mapping partial. In our model, the elaboration function from the C syntax to Core programs is however total. Most undefined behaviours are inherently dynamic; attempting to have the elaboration function detect them would make it potentially diverging. We instead enrich Core with constructs to explicitly mark computations that are undefined.

Undefined behaviours can arise dynamically in two ways: where a primitive C arithmetic operation has undefined behaviour for some argument values, and from memory accesses (unsafe memory accesses, unsequenced races, and data races). For the former, our elaboration simply introduces an explicit test into the generated Core code guarding the use of an **undef**() operator. If the Core operational semantics reaches one of these, it terminates execution and reports which undefined behaviour has been detected (together with the C source location). This is analogous to the insertion of runtime checks for particular undefined behaviours during compilation, as done by many tools, except that (a) it is more closely tied to the standard, and (b) in Cerberus' exhaustive mode, it can detect undefined behaviours on any allowed execution path, not just those of a particular compilation. Undefined behaviours relating to memory accesses are detected by the memory object or concurrency models, using calculated sequenced before and happens-before relations over actions; except for the occurrence of unsequenced races (which occur

[1] Assuming here for the sake of simplicity that τ is not a variable length array type.

when, within a sequential expression, two overlapping accesses are not sequenced with one another), which are detected as part of the dynamics of the sequencing calculus of Core. Note that in the absence of function calls (which as we will see in Section 4.3 introduce defined non-determinism), the non-exhaustive mode of the Core runtime will detect any possible occurrence of an unsequenced race (i.e. even ones dependent on the evaluation order). This is however not the case for other instances of undefined behaviours. For example, because of the potential non-determinism of C expression, one can write a program where a division by zero only occurs for some allowed executions. When considering whether a program has undefined behaviour, one must check whether any allowed execution exhibits an undefined behaviour.

Let us now consider a concrete example of C operator potentially having undefined behaviour: the signed addition operator, whose substantially simplified elaboration looks like:

$$[\![\ \mathsf{E}_1\ +^{\tau}\ \mathsf{E}_2\ ]\!] \triangleq \begin{array}{l}\textbf{letstrong}\ (n_1, n_2) = \textbf{unseq}([\![\ \mathsf{E}_1\ ]\!], [\![\ \mathsf{E}_2\ ]\!])\ \textbf{in} \\ \text{catch\_exceptional\_condition}(\tau, n_1 + n_2)\end{array}$$

The Core expression first computes the value n_1 and n_2 of the operands, and then sums them using Core's addition operator: which operates over mathematical integers (and is therefore total and never overflows). The result of this addition is then applied to an auxiliary function which checks that the result is within the range of the type τ of the addition operator. If it is not, which would correspond to an overflowing computation, the function signals the presence of an undefined behaviour:

```
fun catch_exceptional_condition (τ : ctype, n : integer) : integer :=
  if is_representable_integer(n, τ) then
    n
  else
    undef(<<UB036_exceptional_condition>>)
```

The instances of undefined behaviours resulting from concurrency and memory errors however remain implicit in Core, as they are part of the memory object model which is abstracted. These are discussed in Chapter 9.

4.2 Implicit type conversions and arithmetic operations

There are multiple *arithmetic types* over which arithmetic operators can be used. These are for example, **char**, **signed int**, **unsigned long**; and also floating types. On typical implementations, some of these types have different sizes, signedness, and value ranges. While there is a cast operator that allows expressions to be explicitly converted from one type to another, it is typically not needed within arithmetic expressions. One can, for example, freely add a **char** expression and an **unsigned int** expression; this results in implicit conversions, as part of the semantics of the addition operator.

To allow this transparent mixing of arithmetic expressions of different types, the standard defines a set of *usual arithmetic conversions* (C11, §6.3.1.8) which determine, based on the types of binary operator's operands, a common type for the computation and result of an arithmetic operation. As integer and floating types are both part of the arithmetic

types, some of these conversions turn integer expressions into floating ones. For example, in the expression `1.0f + 2`, the right operand, which has type **`signed int`**, is implicitly converted to **`float`**, the type of the left operand. When dealing with operands whose types have different sizes, the operand with the smaller type is converted to the larger one. For example in `1.0f + 2.0`, the second operand has type **`double`**, which is larger than the **`float`** type of the first operand. The latter is therefore implicitly converted.

Even when exclusively dealing with integer types, the implicit conversions still have to accommodate operands whose types may have different signedness, and different value ranges. To do so, the standard defines, as part of type system of C, a few functions over integer types. Firstly it introduces an *integer conversion rank* function over integer types (C11, §6.3.1.1p1). The concrete definition of that rank is left implementation-defined. This is because implementations are allowed to provide optional *extended integer types* in addition to the standard ones. The standard however specifies a few axioms about the ranking function, forcing all implementations to be consistent about the treatment of standard integer types. These axioms make the rank of a standard integer type reflect how large its value range is[2]:

1. rank(**`signed`** τ_1) $\neq$ rank(**`signed`** τ_2), for two distinct base types τ_1 and τ_2;
2. rank(**`signed`** τ_1) $<$ rank(**`signed`** τ_2), if the precision (the size of the type in bits, minus any padding bits and the potential sign bit) of τ_1 is smaller;
3. rank(**`long long int`**) $>$ rank(**`long int`**) $>$ rank(**`int`**) $>$ rank(**`short int`**) $>$ rank(**`signed char`**);
4. rank(**`signed`** τ) $=$ rank(**`unsigned`** τ), where τ is a base type having both a signed and unsigned variant;
5. rank(τ_2) $<$ rank(τ_1), where τ_1 is a standard type and τ_2 an extended type of same width (the precision of the type, plus the potential sign bit);
6. rank(**`_Bool`**) $<$ rank(τ), where τ is a standard type;
7. rank(τ_1) $<$ rank(τ_3), when rank(τ_1) $<$ rank(τ_2) and rank(τ_2) $<$ rank(τ_3).

Secondly, the standard defines the following transformation over types, called *integer promotion*:

$$\text{promote}(\tau) = \begin{cases} \tau & \text{if } \tau \text{ is not an integer type} \\ \texttt{int} & \text{if } [\min_\tau, \max_\tau] \subseteq [\min_{\texttt{int}}, \max_{\texttt{int}}] \\ \texttt{unsigned int} & \text{otherwise} \end{cases}$$

Using these two, the usual arithmetic conversions (which we write as usual()) between two integer types are then defined as follows:

[2]There are additional axioms regarding extended types that implementations may optionally add to the language, which we omit here.

$$\frac{\tau' = \text{promote}(\tau_1) = \text{promote}(\tau_2)}{\text{usual}(\tau_1, \tau_2) = \tau'} \text{ [A]}$$

$$\frac{\begin{array}{c}\tau_1' = \text{promote}(\tau_1) \quad \tau_2' = \text{promote}(\tau_2) \\ \tau_1' \text{ and } \tau_2' \text{ have same signedness} \\ \text{rank}(\tau_i) < \text{rank}(\tau_j), \text{ for } i, j \in \{1, 2\}\end{array}}{\text{usual}(\tau_1', \tau_2) = \tau_j'} \text{ [B]}$$

$$\frac{\begin{array}{c}\tau_1' = \text{promote}(\tau_1) \qquad \tau_2' = \text{promote}(\tau_2) \\ \tau_i' \text{ is signed, and } \tau_j' \text{ is unsigned, for } i, j \in \{1, 2\} \\ \text{rank}(\tau_i') \leq \text{rank}(\tau_j')\end{array}}{\text{usual}(\tau_1, \tau_2) = \tau_j'} \text{ [C]}$$

$$\frac{\begin{array}{c}\tau_1' = \text{promote}(\tau_1) \qquad \tau_2' = \text{promote}(\tau_2) \\ \tau_i' \text{ is signed, and } \tau_j' \text{ is unsigned, for } i, j \in \{1, 2\} \\ [\min_{\tau_j'}, \max_{\tau_j'}] \subseteq [\min_{\tau_i'}, \max_{\tau_i'}]\end{array}}{\text{usual}(\tau_1, \tau_2) = \tau_i'} \text{ [D]}$$

$$\frac{\begin{array}{c}\tau_1' = \text{promote}(\tau_1) \qquad \tau_2' = \text{promote}(\tau_2) \\ \tau_i' \text{ is signed, and} \tau_j' \text{ is unsigned, for } i, j \in \{1, 2\} \\ [\min_{\tau_j'}, \max_{\tau_j'}] \not\subseteq [\min_{\tau_i'}, \max_{\tau_i'}] \\ \tau = \text{ corresponding unsigned type to } \tau_i'\end{array}}{\text{usual}(\tau_1, \tau_2) = \tau} \text{ [E]}$$

Figure 4.1: Usual arithmetic conversion rules

The multiplicative operators, the additive and relational and equality operators when their operand have integer types, and the binary bitwise operators, all perform the usual arithmetic conversions on their operand. They apply the rules from Figure 4.1 to the types of their operands to find a common type. The values of both of their operands are converted to that type before the computation of the operator is performed. The rules involve applying the integer promotion to both operands; as a result, arithmetic computations in C are never performed on "small" integer types. Consider for example the following program fragment:

```
signed char c = SCHAR_MAX;
c = c + 1;
```

The identifier expression on the left of addition has type **signed char**; because of the integer promotion, it is therefore converted to a **int** (this conversion does not change the value). In C's type system, integer constants are given the "smallest" integer type starting from **int** that can represent the constant. In our example, this gives the type **int** to the right operand of the addition. The usual arithmetic conversion rules do not induce further conversions, and the type over which the addition computation is performed is **int**. The addition is therefore well-defined; it does not result in an undefined signed overflow, as one might have expected from the type of `c`. One additional implicit conversion happens during the evaluation of the assignment operator: the value of the addition, which is

its right operand, is converted to the type (after lvalue conversion) of its left operand. And it is this converted value which is stored by the assignment. In this example, we have a conversion between two signed integer types, for which the standard specifies the following:

(§6.3.1.3p3) Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Here, the "new type" is `signed char`, and the value indeed cannot be represented; the program behaviour of this assignment is therefore implementation-defined. The implementation choice made by GCC and Clang on x86_64 (which is the implementation setting that Cerberus defaults to) is to provide a truncating semantics with no signal being raised.

This illustrates the rationale for the usual arithmetic conversions, which is to reduce the occurrences of arithmetic undefined behaviour, or precision loss in expressions mixing different types; in particular when intermediate values come out of the ranges of some of these types. Consider for example the following program fragment (adapted from [INT02-C]):

```
signed char c1 = 100, c2 = 3, c3 = 4;
c1 = c1 * c2 / c3;
```

As we have just seen, as a result of the usual arithmetic conversions, both the multiplication and the division operators are performed over the type `int`. The fact that the intermediate value produced by the multiplication is out of range of `signed char` does not matter, as the division brings the final value back to range. The final value stored in `c1` is `75`, as one would expect.

On the other hand, the usual arithmetic conversion rules do have some counter-intuitive corner cases, in particular when mixing operands whose types have different signedness. Consider the following integer comparison:

```
-1 < (unsigned int)0
```

The relation operator performs the conversions on its operands; in this instance, the rule D from Figure 4.1 is applied. The value of the left operand is converted to `unsigned int`. This is an instance of a conversion from a signed integer type to an unsigned one, for which the standard specifies:

(§6.3.1.3p2) Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[60)]

The value is therefore converted to the largest value representable by `unsigned int`, and the relation operator evaluates to `0` (i.e. false). If the constant on the left were turned to a `long int` (e.g. by use of a suffix: `-1L`), on implementations where all values of `unsigned int` can be represented in `long int`, the rule E from Figure 4.1 would instead apply, and the relational operator would evaluated to `1`.

The interaction with implementation-defined aspects of integer types can lead to portability issues. Consider the following program:

```
int main(void)
{
  unsigned char c1 = 0xff;
  char c2 = 0xff;
  return c1 == c2;
}
```

The standard leaves implementation-defined whether the type `char` is signed or unsigned. As a result, and in conjunction with the usual arithmetic conversions performed by the equality operator, on implementations where `char` is signed (like those targeting x86_64, with the usual ABI), this program returns `0`; whereas on implementations targeting AArch64, where `char` is unsigned, it returns `1`.

Ail and Core modelling In our model, the treatment of all the implicit arithmetic conversions we just presented occurs in two places. First, the `Ail` program produced by the desugaring is typechecked and annotated. For arithmetic expressions, this involves marking where implicit conversions need to be performed. For example, in the case of the addition operator over arithmetic types:

$$\frac{\vdash E_1 \rightsquigarrow {E'_1}^{\hat{\tau}_1} : \hat{\tau}_1 \qquad \vdash E_2 \rightsquigarrow {E'_2}^{\hat{\tau}_2} : \hat{\tau}_2 \quad \text{is\_arithmetic}(\hat{\tau}_1) \qquad \text{is\_arithmetic}(\hat{\tau}_2) \quad \hat{\tau} = \mathbf{usual}(\mathbf{promote}(\hat{\tau}_1), \mathbf{promote}(\hat{\tau}_2))}{\vdash E_1 \texttt{ + } E_2 \rightsquigarrow {E'_1}^{\hat{\tau}_1} \texttt{ +}^{\hat{\tau}} {E'_2}^{\hat{\tau}_2} : \hat{\tau}}$$

$$\begin{array}{rcl} \hat{\tau}_{\text{integer}} & ::= & \tau_{\text{integer}} \\ & | & \mathbf{unknown}(ns) \\ & | & \mathbf{usual}(\hat{\tau}_{\text{integer}}) \\ & | & \mathbf{promote}(\hat{\tau}_{\text{integer}}, \hat{\tau}_{\text{integer}}) \end{array}$$

Figure 4.2: Typing of the arithmetic case of add operator

The type annotations which are added to the AST (the subscripted and superscripted $\hat{\tau}$) range over an enriched version of C's types, where the integer fragment has two abstract constructors for usual arithmetic conversions and the integer promotion. The type checker does not actually compute the conversions, and can therefore remain agnostic of the implementation details of integer types (e.g. the ranking function). As a drawback, because C constants are typed based on their value and the value ranges of available integer types, the checker needs to delay their typing and annotation until more is known about the implementation. This is where **unknown**() is used.

Second, the elaboration to Core inspects the annotations, and, using a provided implementation of integer types, calculates the results of the usual arithmetic conversion rules. Using them, it places explicit conversions in the generated Core using calls to auxiliary functions. For example, in the elaboration of the arithmetic addition operator:

$$[\![\ \mathsf{E}_1{}^{\hat{\tau}_1} \texttt{ +}^{\hat{\tau}} \mathsf{E}_2{}^{\hat{\tau}_2}\ ]\!] \triangleq \begin{array}{l} \textbf{letstrong}\ (n_1, n_2) = \textbf{unseq}([\![\ \mathsf{E}_1\ ]\!], [\![\ \mathsf{E}_2\ ]\!])\ \textbf{in} \\ \text{catch\_exceptional\_condition}([\![\ \hat{\tau}\ ]\!], \\ \quad \text{conv\_int}([\![\ \hat{\tau}\ ]\!], n_1) + \text{conv\_int}([\![\ \hat{\tau}\ ]\!], n_2) \\ ) \end{array}$$

the values resulting from the evaluation of both operands are converted to the type $\hat{\tau}$ (which by construction of the typing rule in Figure 4.2 is the common type resulting from the usual arithmetic conversions), using a call to pure Core function conv_int().

This function is implemented as part of the standard library of Core. It closely follows the text of the standard specifying conversions between integer types:

(§6.3.1.2p1) When any scalar value is converted to `_Bool`, the result is `0` if the value compares equal to `0`; otherwise, the result is `1`.[59)]

(§6.3.1.3)

1. When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.
2. Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type. [60)]
3. Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

```
1  fun conv_int (τ : ctype, n : integer) : integer :=
2    if τ = '_Bool' then
3      if n = 0 then 0 else 1
4    else if is_representable_integer(n, τ) then
5      n
6    else if is_unsigned(τ) then
7      wrapI(τ, n)
8    else
9      <Integer.conv_nonrepresentable_signed_integer>(τ, n)
```

Figure 4.3: Specification of conversions between integer types in the ISO standard, followed by its Core formalisation

4.3 Sequencing of evaluations

In the C abstract machine, the execution of a program consists of evaluations of *expressions*, structured using *statements* that shape the control flow. Most constructs interacting with the memory state are part of the expression language (for example, the assignment operator, increment and decrement operators, and the pointer indirection operator); the evaluation order of both statements and expressions is therefore potentially observable. This is to be expected for an imperative language, but a peculiarity of C is the extent to which the language definition lets implementations decide how sub-expressions are evaluated.

The standard uses specific terminology when specifying sequencing constraints:

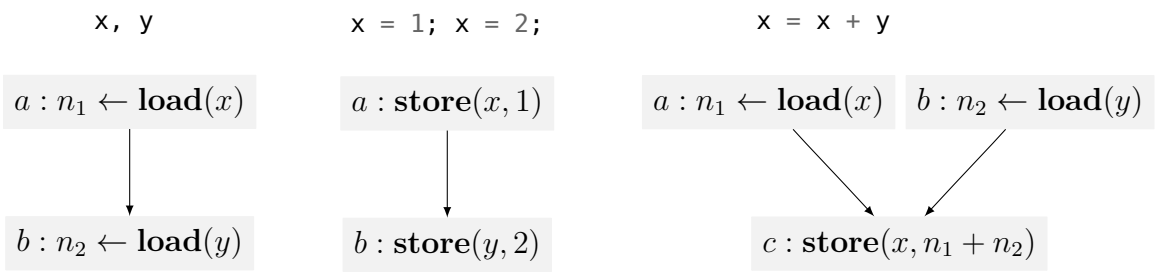
(§5.1.2.3p3) *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is *sequenced after* A.) If A is not sequenced before or after B, then A and B are *unsequenced*. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which.[13)] The presence of a *sequence point* between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the sequence points is given in annex C.)

This paragraph is a relatively new addition to the standard, introduced with the C11 revision. Previously, the sequencing constraints of the language were specified in term of "sequence points", which were points in the program execution where memory accesses arising from previous expressions had to be resolved before the execution could continue. When concurrency features were added to C, the presence of these virtual events made the specification of concurrent accesses difficult, and the sequencing constraints of the language were rephrased more rigorously as the *sequenced before* partial order, which now embodies these constraints. While there are still a few occurrences of the term "sequence points" in the standard, it is now used to specify that all the accesses that are part of an evaluation are sequenced before all the accesses that are part of some other evaluation.

In the phrasing of the standard, "evaluations" are the evaluations of expressions (C11, §6.8p4), and the sequenced before relation is defined as relating them. However during the evaluation of an expression, multiple memory accesses may be performed. Formally, we model the relation as being defined over memory accesses, following the direction taken by formal treatments of C/C++11 concurrency [BA08; Bat+11].

As we will see, in many cases, a sequenced before relation is loose enough to allow multiple execution orders. This looseness is preserved in the elaboration to Core, as the sequenced before relation is modelled directly in the structure of the generated Core expression using the sequencing calculus. The generated Core program models all allowed executions, its evaluation being non-deterministic in the presence of unsequenced or indeterminately accesses. In Chapter 6, we give a traditional small-step operational semantics for Core; however, to guide the presentation of the implicit sequencing constraints of C expressions in this section, we define a graph representation of the sequenced before relation. This representation allows us to describe how to compose evaluations, along with their inner sequencing constraints. Each kind of composition corresponds to an operator in Core. Intuitively, the graph representation defines the envelope of allowed execution that the dynamics of Core models.

sb-graph Consider for example the following expressions and statements, followed by a graphical representation of their sequencing constraints:



Each node is a memory access, and a directed edge between two accesses indicates that they are related by the sequenced before relation.

- The example on the left is a comma operator expression, with two identifiers as its operands. Its evaluation results in two load accesses, performed by the lvalue conversions of the identifiers. These two accesses are strictly ordered by the comma operator, with the access resulting from the evaluation of the left operand sequenced before the other.
- The example in the middle has the same sequencing constraint, but instead involves two assignment expressions sequenced using a compound statement.
- The example on the right illustrates how some accesses can be left unsequenced. It involves an assignment expression applied to a simple identifier as its left operand, and an addition operator as its right operand. The evaluation of the assignment operator results in whatever accesses are performed by the evaluation of its operands, and a store access. As the left operand is a simple lvalue, its evaluation does not result in any memory accesses; it is therefore not visible in the graph. On the other hand, the evaluation of the addition operator results in two load accesses, performed by the lvalue conversions of its identifier operands. The semantics of the addition leaves them unsequenced with one another, as reflected by the lack of an edge between the two corresponding nodes in the graph. They are however both sequenced before the store of the assignment.

Let us consider the sequencing constraints for an evaluation A to be a directed graph associated with some additional information $sb_A = (V_A, E_A, I_A, A_A)$ that we call an *sb-graph*, where:

- V_A is the set of accesses performed by A;
- $E_A \subseteq V_A \times V_A$ is the transitive reduction of the *sequenced before* order[3];
- $I_A \subseteq V_A \times 2^{V_A}$ identifies *indeterminately sequenced* accesses;
- A_A is a set of disjoint subsets of V_A denoting blocks of accesses that must happen atomically with respect to indeterminately sequenced contexts.

From an sb-graph, we can calculate a set of allowed executions, as sequences of memory accesses, such that:

- the order of accesses in any sequence respects the order E_A;

[3]We write E_A^+ for its transitive closure.

- if $(x, Y) \in I_A$, then x is ordered either before all or after all elements of Y;
- for all $(x, \_) \in I_A$ and $X \in A_A$, and for all $y, z \in X$, x does not appear between y and z in any allowed sequence.

Unsequenced evaluations When two evaluations are not related by the sequenced before relation, the standard says that they are *unsequenced*. In term of memory accesses, in the execution of two unsequenced evaluations, the ordering of the accesses of one evaluation is totally unconstrained with respect to the accesses of the other. This looseness in the sequencing constraint is allowed in the specification of most expression operators:

(§6.5p3) The grouping of operators and operands is indicated by the syntax.[85] Except as specified later, side effects and value computations of subexpressions are unsequenced.[86]

For example, in the following expression:

```
(x = 0) + (y + z)
```

the evaluations of the operands of both addition operators are all unsequenced. The two loads on `y` and `z` making up the operands of the inner addition are unsequenced with one another; and they are both also unsequenced with the store on `x` as a result of the semantics of the outer addition. The parentheses on the right do not imply any sequencing.

Accordingly, the sb-graph corresponding to the previous expression has three nodes and no edges:

$$c : \mathbf{load}(z) \qquad b : \mathbf{load}(y) \qquad a : \mathbf{store}(x, 0)$$

An implementation is allowed to order these three accesses in any way, but note that for this particular expression, all orderings lead to the same result.

In term of sb-graphs, forming an evaluation by leaving unsequenced two sub-evaluations A and B results in a graph made from the merge of sb_A and sb_B:

$$(V_A \cup V_B,\ E_A \cup E_B,\ I_{A||B},\ A_A \cup A_B)$$

(we explain the component $I_{A||B}$ in the paragraph about indeterminate sequencing at page 63). Note that formally, this construction needs to be slightly modified: one needs to ensure that if the intersection of V_A and V_B is not empty, their elements are renamed suitably (including in the other components of the sb-graphs) to result in disjoint unions.

Unsequencing and undefined behaviour Given the freedom that unsequenced expressions give to implementations, one might expect that it is possible to write an expression whose result depends on the ordering choice made by the implementation. For example, by leaving unsequenced a store and a load to a same memory object:

```
(x = 0) + x
```

whose result would either be zero, or whatever the value stored in `x` was before the evaluation. But this is akin to having a *data race*, despite the lack of concurrency, and the standard explicitly disallows expressions exhibiting such *unsequenced races*, by giving them undefined behaviour:

(§6.5p2) If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[84)]

This strongly reduces the actual nondeterminism of expressions, which are in fact deterministic when no function calls are used.

Value computations and side effects The standard separates memory accesses within an expression into two kinds:

(§5.1.2.3p2) (...) *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

The first kind forms the *value computation*: it is composed of the memory accesses contributing to the result of evaluating the expression. For example, consider the following expression:

```
*p = x + 2*y
```

Its evaluation performs four memory accesses: the two loads of `x` and `y`, as part of the evaluation of the addition in the right operand of the assignment; a load of the pointer `p` as part of the evaluation of the indirection operator; and a store on the object pointed to by the value read from `p`, performed by the assignment. The result of an assignment operation is defined in C to be the result of its right operand. The loads therefore form the value computation, but the store performed by the assignment does not. It is instead classified as being part of the *side effect* of the expression.

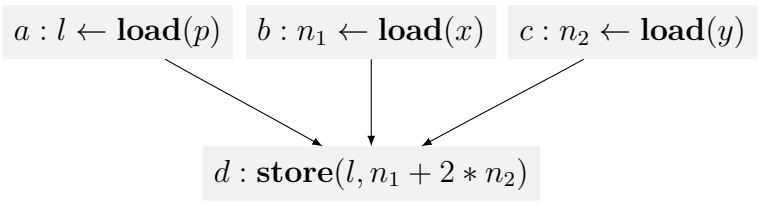
The separation of accesses between these two classes is important because, for many operators, only their value computations are given a sequencing constraint:

(§6.5p1) (...) The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

This is for example the case of the assignment operator:

(§6.5.16p3) (...) The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

With this specification, we get for our last example (`*p = x + 2*y`) the following sb-graph:

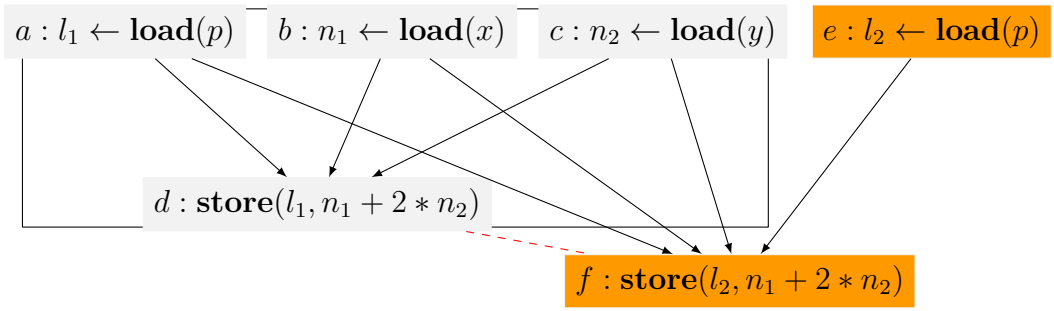


Node a forms the value computation of the left operand `*p`, and nodes b and c form the value computation of the right operand `x + 2*y`. They are all sequenced before node d which is the store of the assignment, as indicated by the edges.

Node d is not part of the value computation of the assignment. However, because this expression is not within a non-trivial context, that has no visible effect on the graph. Let us extend the expression by adding an outer assignment operator through the same pointer `p`:

```
*p = (*p = x + 2*y)
```

We now get the following sb-graph:



The two accesses coming from the new outer assignment are highlighted in orange: node e is the value computation of the left operand, and is unsequenced with all the accesses of the inner assignment; and node f is the updating store, and is sequenced after the value computation of the inner assignment. Since node d is not part of the value computation, the two stores are unsequenced. As they both modify the memory object pointed to by `p`, there is an *unsequenced race* (indicated as a dashed edge), and this expression has undefined behaviour.

In term of sb-graphs, given two evaluations A and B, the sequencing of the value computation of A (noted $\mathrm{val}(A)$) before that of B results in the graph:

$$(V_A \cup V_B,\ (\mathbf{val}(\mathbf{A}) \times \mathbf{V_B}) \cup E_A \cup E_B,\ I_{A;B},\ A_A \cup A_B)$$

Sequence points There are, of course, some C constructs that do introduce strong sequencing between their operands, and this is where *sequence points* come in play. For these, the C11 standard reuses the *sequence point* terminology from its previous revisions. For example, the logical AND operator `&&` is defined with a left-to-right sequencing constraint:

(§6.5.13p4) Unlike the bitwise binary `&` operator, the `&&` operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. (...)

Sequence points order all accesses, both those part of value computations, and those of side effects. As a result, replacing, in the previous example, the addition operator with a logical AND makes the previous unsequenced race disappear:

```
x = (x = 1) && y
```

Among other places where sequence points are present are: before the evaluation of the body of functions when they are called (C11, §6.5.2.2p10), between the operands of the comma operator (C11, §6.5.17p2), and between successive expressions forming a block statement (C11, §6.8p4).

In term of sb-graphs, given two evaluations A and B, placing a sequence point between the two results in the following graph:

$$(V_A \cup V_B,\ (V_A \times V_B) \cup E_A \cup E_B,\ I_{A;B},\ A_A \cup A_B)$$

Indeterminate sequencing and function calls Function calls introduce additional complexity to the sequencing of expressions:

(§6.5.2.2p10) There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[94)]

This passage implies that in the following program:

```
int x;
int f(void) { return x = 1; }
int main(void)
{
  return x + f();
}
```

in the body of `main`, the evaluation of the call to the function `f` is not unsequenced with the load of `x`. This is unlike what one would expect from the sequencing constraints of the addition operator, that we saw previously. The function call is instead *indeterminately sequenced* with the access on `x`. That is, it behaves as if the two accesses on `x` are related in the sequenced before order one way or the other: in the sb-graph, the accesses are related by an edge, the direction of which is left unspecified by the standard. The race that would otherwise exist between the two is therefore eliminated. This program is well-defined; however, note it can either return `1` or `2`. The fact that the order is unspecified means that indeterminately sequenced expressions behave as if an ordering is non-deterministically chosen during their evaluation. In the case of expressions with multiple dynamic occurrences (e.g. for expressions inside loops), the ordering that is chosen may change for each occurrence. For our program, implementations are free to produce an executable returning either value, or, in fact, an executable nondeterministically returning either in multiple executions. They are not required to document their choice, and do not have to be consistent about it.

Note however that the evaluations of the expression arguments to the function call are not part of the indeterminate sequencing. They are sequenced before the call, but may still be unsequenced with the context of the call. For example, if we modify our last example such that the assignment on `x` was performed as part of the function call:

```
return x + f(x = 1);
```

this reintroduces a race, and therefore gives the program undefined behaviour. In addition, the indeterminate sequencing is a property of the whole evaluation of function being called, making it "atomic" with respect to its context (C11, footnote[94)]).

The syntactic scope of indeterminate sequencing is rather limited, since sequence points are present between every *full expression*[4]. However, because C functions can call themselves recursively, it is easy to construct a function with a significant number of allowed execution orders:

```
int x;
int f(int n) {
  x++;
  return n<1 ? 0 : x-n + f(n-1);
}
```

A call to `f` with input `n` is allowed to return any value from the set $\{0, \cdots, \sum_{i=1}^{n} i\}$, and most of these values can by computed using more than one evaluation order.

This can become quite expensive for the model if one wants to detect all undefined behaviours, as the occurrence of undefined behaviours can depend on the evaluation order:

```
int x;
int f(void) { return 1 / x; }
int main(void) {
  return (x = 1) + f();
}
```

Depending on the order in which the addition operator in the body of `main` is evaluated, the program either returns `1`, or performs a division by zero, which is an undefined behaviour. However, as we discussed in the previous section, a program is deemed to have undefined behaviour if any of its allowed evaluations exhibits an undefined behaviour. This shows that, in order to properly detect undefined behaviours, a formal model of C expressions needs to fully explore all the nondeterminism resulting from the indeterminate sequencing of function calls.

In term of sb-graphs, we model indeterminate sequencing in the I component. Having $(x, S) \in I$ means that the memory access x is indeterminately sequenced with all elements of S. Coming back to $I_{A||B}$, which was part of the graph for two unsequenced evaluations, it is constructed as follows:

$$I_{A||B} = I_A \cup I_B \cup (\mathrm{dom}(I_A) \times V_B) \cup (\mathrm{dom}(I_B) \times V_A)$$

where we write $\mathrm{dom}(I_A)$ for the first component of I_A.

For $I_{A;B}$, part of the graph for two sequenced evaluations, the construction is simply $I_{A;B} = I_A \cup I_B$.

Atomicity of some operators The postfix increment and decrement operators (`E++` and `E--`), and compound assignment operators (e.g. `E1 *= E2`) perform two memory accesses during their evaluation: a load and a modifying store. For this class of operators, the standard requires that their two accesses behave as if they were atomic with respect to any indeterminately sequenced function call:

[4] **(§6.8p4)** (...) A full expression is an expression that is not part of another expression or of a declarator. (...)

(§6.5.2.4p2) (...) With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. (...)

(§6.5.16.2p3) (...) with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. (...)

Note that this kind of atomicity only relates to indeterminately sequenced function calls, but not concurrency. When a postfix increment or decrement operators is performed on a (concurrency) atomic type, the standard specifies that there is a unique read-modify-write access, which in that case is atomic with respect any concurrent access. However, for non-atomic types, the two accesses which cannot be separated by the evaluation of an indeterminate function call, can still be interleaved with concurrent accesses.

In term of sb-graphs, this is where the fourth component A is used: the two accesses from a postfix operator are paired into a set, and appended to A.

Core modelling

In order to accurately capture the subtle sequencing semantics of C's expressions in a compositional way, we equip the effectful fragment of Core with a calculus of sequencing operators mirroring the different kinds of sequencing constraints used by the standard. Using these operators, the elaboration of C expressions models the constraints on the sequencing order by making them explicit in the syntax of the Core programs it produces. This allows the elaboration to accurately capture the loose sequencing of C, while remaining compositional.

The elaboration is therefore agnostic of implementation-defined choices regarding sequencing. The Core programs it produces capture all possible evaluation orders a valid C implementation may perform. At the same time, one may sometimes want to explore a specific allowed ordering of a program, instead of the complete envelope. Doing so is simply a matter of fixing an evaluation strategy in the Core evaluation (e.g. consistent left to right evaluation order), or by encoding it through a Core-to-Core transformation. In either case, this is done without having to change anything in the elaboration function.

We now present the sequencing operators of Core. For the sake of clarity, we omit in this section details regarding implicit conversion or undefined behaviours. The following examples only accurately present how the elaboration function models sequencing. We also do not present here the formal semantics of the sequencing operators, but only illustrate how they are used by the elaboration. Their semantics is discussed along with the rest of Core's semantics in Chapter 6.

Strong sequencing In Core, the "evaluations", over which the sequence before relation is defined, correspond to effectful expressions. Sequencing between two Core expressions is expressed using operators modelled after the **let** binder from ML like languages. Let us first consider the *strong sequencing* operator:

$$\textbf{letstrong}\ pat = E_1\ \textbf{in}\ E_2$$

Its dynamics is to: first evaluate its first operand E_1, performing all memory actions that expression contains; then to carry on by evaluating its second operand E_2, where

the pattern *pat* has been substituted for the value resulting from the evaluation of E_1. This effectively models a sequence point. For example, a sequence of C expressions are elaborated as follows:

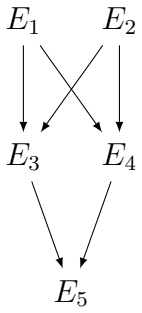
$$[\![\ \mathtt{E}_1\mathtt{;}\ \mathtt{E}_2\mathtt{;}\ \cdots\mathtt{;}\ \mathtt{E}_n\ ]\!] \triangleq \begin{array}{l} \textbf{letstrong}\ \mathtt{Unit} = [\![\ \mathtt{E}_1\ ]\!]\ \textbf{in} \\ \textbf{letstrong}\ \mathtt{Unit} = [\![\ \mathtt{E}_2\ ]\!]\ \textbf{in} \\ \cdots \\ [\![\ \mathtt{E}_n\ ]\!] \end{array}$$

Figure 4.4: Strong sequencing of expression statements

Unsequencing operator To model the absence of sequencing constraints, such as between the two operands of the C addition operator, we equip Core expressions with an n-ary **unseq**() operator:

$$\textbf{unseq}(E_1, \cdots, E_n)$$

Its dynamics allows for any interleaving of the memory-action parts of its E_i operands, and its value is the tuple made of their values. By combining these two simple operators, we are able to model a variety of sequencing graphs:



$$\begin{array}{l} \textbf{letstrong}\ (x_1, x_2) = \textbf{unseq}(E_1, E_2)\ \textbf{in} \\ \textbf{letstrong}\ (y_1, y_2) = \textbf{unseq}(E_3, E_4)\ \textbf{in} \\ E_5 \end{array}$$

Figure 4.5: Sample sequence graph and its corresponding Core expression

Polarised actions and weak sequencing As we discussed earlier, memory accesses performed during the evaluation of a C expression are either classified as being part of the value computation (which contribute to the final value), or as being side effects. For example, in the assignment operator, the store access to the lvalue is classified as a side effect, whereas the load performed on a variable is classified as a value computation.

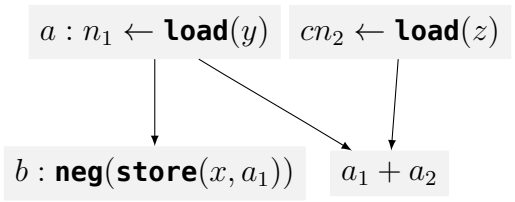
To model the distinction between the *value computation* of a C expression and other side effects, we equip Core memory actions with a polarity. Memory actions that are part of the elaboration of a value computation are positive, the default polarity which needs no syntactic marker. Other memory actions are marked as negative using the **neg**() operator which may only be applied to a memory action whose result type is `unit` (effectively, the elaboration function only applies this operator to stores). The specificity of negative actions is in how they interact with the *weak sequencing* operator:

$$\textbf{letweak}\ pat = E_1\ \textbf{in}\ E_2$$

This operator is similar to **letstrong**, with the distinction that only the positive memory actions part of E_1 are forced to be evaluated before the evaluation carries on to E_2. Since negative actions are constrained to return `unit`, it is always possible to evaluate the value of a Core expression without having to perform its negative actions. Intuitively, one may summarize the dynamics of the **letweak** on a negative action as follow:

$$\textbf{letweak}\ \texttt{Unit} = \textbf{neg}(A)\ \textbf{in}\ E \quad \text{reduces to} \quad \textbf{unseq}(A, E)$$

This allows the modelling of the following sequence graph, corresponding to the C expression `(x = y) + z`:



$$\begin{array}{l}\textbf{letweak}\ (a_1, a_2) = \textbf{unseq}(\\ \quad \textbf{letweak}\ (a_1) = \textbf{load}(ptr_y)\ \textbf{in}\\ \quad \textbf{letweak}\ \texttt{Unit} = \textbf{neg}(\textbf{store}(ptr_x, a_1))\ \textbf{in}\\ \quad \textbf{pure}(a_1)\\ ,\\ \quad \textbf{load}(ptr_z)\\ )\ \textbf{in}\\ pure(a_1 + a_2)\end{array}$$

Detection of unsequenced races In the evaluation of Core expressions corresponding to the elaboration of C expressions, we need to detect unsequenced races and raise an undefined behaviour. For clarity, we choose to allow "racy" memory actions by default in the Core dynamics, a Core expression involving two unsequenced accesses to a same memory location is well defined, albeit nondeterministic. To mark expressions where undefined behaviour is desired for races, the language is equipped with a unary operator **bound**(). In the elaboration function, this operator is used to mark the boundary of an outermost C expression (what the standard calls a *full expression* (C11, §6.8p4)). For example, an *expression statement* (C11, §6.8.3) is elaborated as follows:

$$[\![\ \texttt{E;}\ ]\!] \triangleq \textbf{bound}([\![\ \texttt{E}\ ]\!])$$

The elaboration of all other occurrences of expressions within a statement follows the same pattern, and it is detailed in Section 4.5.

For illustration, let us consider the elaboration of a concrete expression statement with undefined semantics (for clarity, we omit details regarding implicit arithmetic conversions):

$$[\![\ \texttt{x = (x = 1);}\ ]\!] = \begin{array}{l}\textbf{bound}(\\ \quad \textbf{letweak}\ a_1 =\\ \quad\quad \textbf{letweak}\ \texttt{Unit} = \textbf{neg}(\textbf{store}(ptr_x, 1))\ \textbf{in}\\ \quad\quad \textbf{pure}(1)\ \textbf{in}\\ \quad \textbf{neg}(\textbf{store}(ptr_x, a_1))\\ )\end{array}$$

Figure 4.6: Example of unsequenced race

In this example, the store from the inner assignment is not part of the value computation and is therefore elaborated to a negative action. As a result, the weak sequencing operator leaves it unsequenced with respect to the store of the outer assignment. The expression has an unsequenced race, and its evaluation signals an undefined behaviour.

Indeterminate sequencing The **bound**() operator also allows us to model the indeterminate sequencing of function calls. As we explained earlier, the scope of the nondeterminism introduced by an indeterminate sequencing is the boundary of the containing full expression. Note that we not able to instead make use of **letstrong** operators as boundary markers, because they can appear within the elaboration of full expressions; for example in the elaboration of the comma operator, and of logical boolean operators.

C functions are elaborated into Core functions whose body may be effectful, which we call *procedures*. Calls to procedures are not part of the pure subset of the language, but are instead part of the same syntactic category as the sequencing calculus. C has pointers to functions as normal storable values, and a call may take as its first operand an arbitrary expression evaluating to a pointer to the function being called. The elaboration of C calls therefore make use of a dedicated **ccall**() operator, whose first operand is a pointer to a function. This contrasts with the normal call operator for procedures, whose first operand is a name, just like the call operator for pure functions.

The dedicated call operator is additionally used to serve as a marker for the indeterminate sequencing of the elaboration of C calls within the operand of a **bound**() operator. Like in C, the semantics of Core makes the evaluation of procedure calls atomic with respect to the loose thread-local sequencing.

For a concrete example, let us consider the elaboration of a function call within an addition operator (again, implicit conversions are omitted for the sake of clarity):

$$[\![\ \texttt{E}_1\ \texttt{+}\ \texttt{f();}\ ]\!] = \begin{array}{l}\textbf{bound}(\\ \quad \textbf{letweak}\ (a_1, a_2) = \textbf{unseq}([\![\ \texttt{E}_1\ ]\!], \textbf{ccall}(ptr_f, [])) \ \textbf{in}\\ \quad \textbf{pure}(a_1 + a_2)\\ )\end{array}$$

Figure 4.7: Elaboration of an indeterminate function call

Sequential "read-modify-write" The postfix increment and decrement, and compound assignment operators in C perform a load and a store that are specified as "atomic" with respect to indeterminately sequenced function calls. To model this, we equip Core with a rather ad-hoc memory action: $\textbf{seq\_rmw}_b(ptr, z.\, e)$, where the first operand is a pointer value, and the second is a lambda expression specifying an update to perform on the value of the memory object referenced by the pointer. This behaves like an atomic version of:

$$\begin{array}{l}\textbf{letweak}\ x\ = \textbf{load}(ptr)\ \textbf{in}\\ \textbf{store}(ptr, e\{x/z\})\end{array}$$

The flag b indicates whether the value of the action is the value of the read ($b = \text{true}$), or unit. Using this operator, the compound assignment operator is for example elaborated

as follows:

$$[\![\ \mathtt{E}_1\ \mathtt{*=}\ \mathtt{E}_2\ ]\!] \triangleq \begin{array}{l}\mathbf{letweak}\ (a_1, a_2) = \mathbf{unseq}([\![\ \mathtt{E}_1\ ]\!], [\![\ \mathtt{E}_2\ ]\!])\ \mathbf{in} \\ \mathbf{seq\_rmw}_{\mathrm{true}}(a_1, z.\, z * a_2)\end{array}$$

Note that, with this elaboration, we do not model the undefined behaviour relating to partially overlapping accesses (C11, §6.5.16.1#3).

4.4 Lifetime of memory objects

In this section, we discuss the *lifetime* of memory objects, that is, the time interval of a program execution starting from the allocation of an object and ending at its deallocation. Depending on the syntactic construct used to create a new object, one of four possible *storage durations* (C11, §6.2.4) is given to that object, determining its lifetime. Memory objects can be created either through the declaration of an identifier, or by calling a memory management functions (e.g. `malloc()`). Declarations result in objects with one of the following three storage durations:

- **thread storage duration** for any identifier whose declaration contains the storage-class specifier `_Thread_local`. The associated objects have a lifetime corresponding to the span of program execution during which the hosting thread is running.
- **static storage duration** for any other identifier declared outside the body of a function (i.e. global variables), or any identifier whose declaration contains the `static` storage-class specifier. Additionally, compound literals outside of the body of a function and string literals implicitly create objects of this category. The associated lifetime for such objects spans the whole program execution: they are allocated (and if needed initialised) before the execution of the startup function, and are deallocated only after program execution has ended.
- **automatic storage duration** for any other identifier declared inside a block statement (and therefore inside the body, or as a parameter of a function). For the associated objects, their lifetime is the span of program execution spent executing the innermost enclosing block statement: objects are allocated upon entering the block and deallocated upon exiting. The initialisation (if it exists) is however performed at the point of the declaration. Objects implicitly created by compound literals inside the body of a function also have automatic storage duration.

Objects created using a memory management function have their own storage duration:

- **allocated storage duration**: their lifetime starts with the call to the allocating function, and ends with the call to the deallocating function (e.g. `free()`).

For thread storage duration, a single identifier declaration may lead to the creation of multiple memory objects during the program execution: one for each thread instance. Similarly for the automatic storage duration: an identifier declared in a block statement forming the body of a loop leads to the allocation, potential initialisation, and deallocation of a new memory object for each iteration of said loop. On typical implementations, the first three storage durations will correspond to objects stored in the stack or the data section of an executable, while the fourth will correspond to objects stored in the heap. However, these notions are not part of the ISO standard.

Through use of pointers, declarations with the automatic storage duration can lead to undefined behaviours. To illustrate this, consider the C statement of Figure 4.8.

```
1 {
2   int *p;
3   {
4     int x;
5     p = &x;
6   };
7   *p = 1; // attempting to store into x
8 }
```

Figure 4.8: Dereferencing of a pointer referring to an object past its lifetime

At line 2, an object `p` with a pointer type is declared within the outermost block statement. This creates an object with automatic storage duration whose lifetime starts at line 1 and ends at line 8.

At line 4, an object `x` with **`int`** type is declared. Like the previous declaration, this creates an object with automatic storage duration. But because it is done inside a different block statement, the lifetime starts at line 3 and ends at line 6. As a result the lifetime of the **`int`** object ends before that of the pointer.

At line 5, the location of the **`int`** object is stored into `p` and that pointer is then dereferenced at line 7. This effectively attempts to store into the **`int`** object after the end of its lifetime. Therefore, this program snippet has undefined behaviour (C11, §6.2.4p2).

Jump statements introduce additional ways for the program execution to enter and exit a block. For example, consider the following variant of our previous example:

```
1  {
2    int *p;
3    goto l2;
4  l1:
5    {
6      int x = 5;
7      return *p; // dereferencing of a pointer
8                 // to a dead object
9  l2:
10     p = &x;
11     goto l1;
12   }
13 }
```

After the declaration of the pointer `p`, the execution jumps to line 9, bypassing the declaration of `x`. However, as this has caused program execution to enter the block containing that declaration, this effectively starts the lifetime of an object for `x` during the jump, but does not initialise it. In fact, the expression in the initialiser is not evaluated, so any side-effects it might have contained are not performed. Next we store the address of `x` into `p`, and jump back to line 4, just before the beginning of the block. This jump exits the block, and therefore ends the lifetime of `x`. Program execution carries on by re-entering

the block. This starts the lifetime of a new object for `x`; this time, the initialiser is evaluated, and its value is stored into the object. Execution finally reaches line 7, where `p` is dereferenced. However the value of the pointer refers to the previous object whose lifetime has already ended. This is therefore another instance of the undefined behaviour seen in Figure 4.8. It is worth noting that the dependency on control flow makes this kind of undefined behaviour not statically checkable in the general case.

If we change our example further by moving the label `l1` down by one line, into the block statement, and by moving the dereferencing of the pointer before the declaration, we get the C statement of Figure 4.9.

```
1  {
2    int *p;
3    goto l2;
4    {
5  l1:
6      return *p; // return uninitialised memory
7      int x = 5;
8  l2:
9      p = &x;
10     goto l1;
11   }
12 }
```

Figure 4.9: Delayed initialisation in block statements

In this version, the second jump at line 10 does not end the lifetime of `x`, so the dereferencing of the pointer at line 6 performs a load from a live object. However, this object has never been initialised, and the access therefore has undefined behaviour (though reading from uninitialised memory may in some instance be given defined behaviour, as discussed in Section 4.6).

Core modelling

To accurately model the lifetime of memory objects in Core, allocation and destruction events are made syntactically explicit using memory actions. References to memory objects are one of the first-class values. They have type `pointer`, which is an opaque type exposed by the memory interface. We present the complete interface in Appendix A, but discuss here the part relevant to object lifetime. Fresh pointer values are created by performing one of three allocating memory actions, taking as parameters the alignment constraint, and the type or size of the object that needs to be allocated. Their effect is to update the memory state (the details of which is kept opaque and may differ among different memory object models), and yield a pointer value referring to the newly allocated object. This value is then used in the remainder of the Core program to access and ultimately deallocate the object. The allocation actions have the following type signatures:

- **`allocate_object`** : `integer` $\rightarrow$ `ctype` $\rightarrow$ `pointer eff`
- **`allocate_object_readonly`** : `integer` $\rightarrow \tau :$ `ctype` $\rightarrow \alpha_\tau \rightarrow$ `pointer eff`
- **`allocate_region`** : `integer` $\rightarrow$ `integer` $\rightarrow$ `pointer eff`

where α_τ is the Core type used for elaborating the C type τ. The two variants of **allocate_object** are used to elaborate C identifier declarations. Their first parameter is the alignment constraint and their second parameter is the C type of the declaration. For most declarations, the alignment constraint is simply derived from the type. But when **_Alignas** is part of the specifiers, the alignment constraint comes from the type or elaboration of the expression parameter of the specifier.

The second variant is used for the elaboration of **const**-qualified declarations. These always have an initialisation (which, in the case of file scope identifiers, may be left implicit by the syntax), which is performed at the time of the allocation. The third parameter, whose Core type depends on the value of the "C type" parameter, is used to specify the desired initial value. The action also marks as "read-only" the footprint of the newly allocated object. One corner case is not properly handled by this operation: the allocation of a **const**-qualified pointer that is initialised to its own address. To deal with this, we currently use the first variant of the **allocate_object** action, directly followed by a store action marked with a flag that tells it to lock the footprint of the object after it has updated its value. Reading from an (at least partially) locked footprint is then detected as an undefined behaviour. Depending on the memory object model, this elaboration would not be sound in the presence of concurrency if another thread were able to perform an access scheduled between the allocation and the initialising store. However, because of the provenance mechanism we discuss in Chapter 8, this cannot occur. A more satisfactory elaboration would be possible if we enriched the **allocate_object** action by replacing the third operand with a function whose argument holds the pointer value resulting from the allocation.

Finally, **allocate_region** is used in the elaboration of C's memory management functions that make new allocations. The first parameter is the alignment constraint, as for the previous variant. The second parameter is an integer value specifying the number of bytes that need to be allocated. For example, in the elaboration of `malloc()`, this is the size parameter.

Deallocation is performed using a memory action taking a pointer value as argument, and effectfully returning a unit value:

- **kill** : `pointer` → `unit`
- **kill_dynamic** : `pointer` → `unit`

The first variant is used in the elaboration of deallocation events resulting from exiting block statements, and the deallocation at the end of program execution of objects with **static** storage duration. The second is used in the elaboration of memory management functions. A dedicated variant for the latter is required in order to properly model the fact that calling `free()` or `realloc()` on pointers which were not previously returned by an allocating function (e.g. calling them on a pointer to an object with `automatic` storage duration) is undefined behaviour (C11, §7.22.3.3p2).

Like all other memory actions in Core, these are part of the effectful fragment of the language, and may only be used within the sequencing expressions.

Using these memory actions, modelling the various storage durations is simply a matter of placement of the **create** and **kill** actions in the elaboration to Core.

$$[\![\ \texttt{int x = E;}\ ]\!] =$$

```
1 letstrong ptr_x : pointer =
2   allocate_object(Ivalignof('int'), 'int') in
3 letstrong n : loaded integer = [[ E ]] in
4 store('int', ptr_x, n)
```

Figure 4.10: Model elaboration of an identifier declaration with initialiser

The example in Figure 4.10 shows the allocation and initialisation part of the elaboration of a simple identifier declaration, with the elaborated Core expression on the right side. The second line does the allocation; the third evaluates the expression used to initialise `x` (note the recursive call to the elaboration function); and the fourth performs the initialisation using a **`store`** memory action which is applied to the pointer value produced by that allocating action. Each part is sequenced using the **`letstrong`** operator. The allocation is therefore sequenced before whatever memory actions the elaboration of E may contain, and these actions are themselves sequenced before the final store. Since no alignment specifier is present, the first parameter of the allocation action uses `Ivalignof()`, which is an integer value constructor taking a C type and evaluating to an integer value representing the alignment constraint for that type. This is an implementation-defined value. Using an abstract constructor allows the elaboration to be agnostic on the details of implementation.

The placement of the deallocating part of elaboration depends on the context of the declaration in the C program. If it is a file-scoped declaration, the expression from Figure 4.10 is in fact part of the body of a Core global definition, whose value is ptr_x, and bound to a globally visible symbol. A **`kill`** action applied to that symbol is then appended to the end of the elaboration of the C startup function. This effectively models the **static** storage duration.

Let us now consider the elaboration of block-scoped declarations[5], where we assume that the label l is bound somewhere in the outer context:

$$[\![\ \texttt{\{ qs}_1\ \tau_1\ \texttt{x}_1\texttt{; goto l; qs}_2\ \tau_2\ \texttt{x}_2\texttt{ = E; S \}}\ ]\!] =$$

```
letstrong ptr_x1 : pointer =
  allocate_object(Ivalignof([[ τ1 ]]), [[ τ1 ]]) in
letstrong ptr_x2 : pointer =
  allocate_object(Ivalignof([[ τ2 ]]), [[ τ2 ]]) in
kill(ptr_x1) ;
kill(ptr_x2) ;
run l();
letstrong n : α_[[ τ2 ]] = [[ E ]] in
store([[ τ2 ]], ptr_x2, n) ;
[[ S ]] ;
kill(ptr_x1) ;
kill(ptr_x2)
```

where in the C declarations, $\texttt{qs}_1$ and $\texttt{qs}_2$ are type qualifiers. The block statement contains two declarations: the first one is at the entry of the block, and the second one is placed

[5] In Core, E_1`;` E_2 is syntactic sugar for **`letstrong`** _ : `unit` = E_1 **`in`** E_2

after a goto statement. In the corresponding Core elaboration, the allocation actions for both declarations are placed at what corresponds to the entry of the block statement. Accordingly, the deallocating actions are placed at what corresponds to the exit point. However, the store performing the initialisation of the second declaration is sequenced after the elaboration of the statement S. This models the behaviour exhibited by the example in Figure 4.9. Additional deallocating actions are also placed before a **run** operator (a Core construct used in the elaboration of **goto** statements, whose semantics we present in the next section). This models the fact that, as the label l is outside of the block, the two objects declared in the blocks must be deallocated before jumping out to the block. If the label l were located within a block where another object with automatic storage duration is declared, the **run** operator would additionally be preceded by the allocating action for that object. The set of identifiers "visible" from a label are precomputed by a pre-pass on the Ail AST, and given as an argument to the elaboration function.

4.5 Control-flow operators

With the exception of the conditional expression operator (`_ ? _ : _`) and function calls, control operators in C are statements (C11, §6.8). The ISO standard groups these into three categories: *selection statements*, providing conditional branching; *iteration statements* for looping constructs; and *jump statements*, providing labelled branching as well branching within looping constructs. In contrast with expressions, their semantics is mostly straightforward. The semantics of jump statements does however involve some subtleties relating to the lifetime of memory objects with automatic storage classes, which we discussed in Section 4.4.

Selection statements Conditional branching from statements is either done using a **if**, or a **switch** which may have more than two branches:

- `if (E) then S`
- `if (E) then S1 else S2`
- `switch (E) S`

Their first operand, the "controlling expression", is first evaluated (potentially performing side-effects), and its value is used to select which branch the program execution flows to. The evaluation of the expression and the execution of the selected branch are separated by a sequence point. For the **if** statement, the expression does not have not have boolean type, but is instead either of integer, floating, or pointer type. To compute a boolean, its value is compared to zero (or a null pointer, as appropriate). If it compares equal, then `S2` is selected, and `S1` otherwise. The variant of the **if** with no **else**-clause behaves as if rewritten with an added null statement as the *else*-clause:

$$\texttt{if (E) then S} \rightarrow \texttt{if (E) then S else ;}$$

The **switch** statement offers a less structured form of control. The controlling expression has integer type, and the statement operand is inspected for the occurrence of a sub-statement of the form `case n: S` where `n` is equal to the value of the controlling expression. If such a sub-statement exists, the semantics of the **switch** statement is akin

to a **goto** where that particular **case** statement acts as a label. If none is found, but there is instead a sub-statement of the form **default**: S, the flow of execution jumps to S. In the absence of either, the program execution simply skips the statement operand of the switch. The non-structural aspect of the operator comes from the fact that, again like a **goto** statement, after having selected and executed the body of a particular **case** (or the **default**), the program execution will carry on the remainder of the statement operand of the switch, which will typically contain other **case** statements. To prevent this, a **break** statement may be placed at the end of body of a **case**, upon which program execution directly exits the (innermost) **switch** statement.

In Cerberus, both **if** and **switch** statements are elaborated to Core's **if** operator, which, unlike its C counterpart, is controlled by a pure boolean expression. The elaboration therefore makes explicit the sequence point after the evaluation of the controlling expression, and the comparison of the against zero for the **if** statement:

$$\llbracket\ \texttt{if (E) then } \mathsf{S}_1 \texttt{ else } \mathsf{S}_2\ \rrbracket \triangleq \begin{array}{l}\texttt{letstrong } z : \texttt{integer} = \llbracket\ \texttt{E == 0}\ \rrbracket \texttt{ in} \\ \texttt{if not}(z = 1) \texttt{ then } \llbracket\ \mathsf{S}_1\ \rrbracket \texttt{ else } \llbracket\ \mathsf{S}_2\ \rrbracket\end{array}$$

Figure 4.11: Case of the elaboration function for **if** statements

As the pattern of comparing the value of controlling expressions to zero is also used in the specification of iteration statements, we chose to avoid duplicating its modelling in different clauses of the elaboration function, at the cost of making the elaboration function not structurally recursive. The recursive call for the controlling expression models the comparison to zero by reusing the semantics of C's equality operator. This includes the overloading of the constant zero to designate the null pointer when the left operand of the equality has pointer type. In the second line of the Core, the **if** expression is used on a somewhat convoluted check resulting from the dynamics of C's equality operator: it evaluates to the integer value one if true, and to zero otherwise.

The **switch** statement is elaborated into nested **if** operators, one for each **case** statement present in its body, whose **then** branch is a **run** to a label associated to the body of the case. The final **else** branch is a **run** to either a label associated to the body of the **default** statement (if one is present in the body of the **switch**), or to a "break" label pointing to the end of the elaboration of the switch. This block is strongly sequenced before the elaboration of the body of the switch, which is itself strongly sequenced before a **save** operator declaring the "break" label with a body simply made of the unit value. The elaboration of the **switch** statement requires a pass over the statement body to collect the set of integer constants occurring as parameters of a **case** statement.

$$
\begin{array}{l}
[\![\ \texttt{switch (E) S}\ ]\!] \triangleq \\
\quad \texttt{letstrong Unit} : \texttt{unit} = \\
\quad\quad \texttt{letstrong}\ z : \texttt{integer} = [\![\ \texttt{E}\ ]\!]\ \texttt{in} \\
\quad\quad \texttt{let}\ z_{\text{conv}} : \texttt{integer} = \text{conv\_int}(\texttt{'int'}, z)\ \texttt{in} \\
\quad\quad \texttt{if}\ z_{\text{conv}} = n_1\ \texttt{then} \\
\quad\quad\quad \texttt{run}\ l_{\text{case}_1}() \\
\quad\quad \ldots \\
\quad\quad \texttt{else if}\ z_{\text{conv}} = n_k\ \texttt{then} \\
\quad\quad\quad \texttt{run}\ l_{\text{case}_k}() \\
\quad\quad \texttt{else} \\
\quad\quad\quad \texttt{run}\ l_{\text{break}}()\ \texttt{in} \\
\quad \texttt{letstrong}\ () : \texttt{unit} = [\![\ \texttt{S}\ ]\!]\ \texttt{in} \\
\quad \texttt{save}\ l_{\text{break}}()\ \texttt{in} \\
\quad\quad \texttt{pure}(\texttt{Unit})
\end{array}
$$

Figure 4.12: Simplified version of the elaboration function for **switch** statements[6]

Occurrences of **case** and **default** statements (which are only allowed to appear within the body of a **switch** statement) are elaborated into label declarations whose body is the elaboration of the sub-statement operand:

$$[\![\ \texttt{case n: S}\ ]\!] \triangleq \texttt{save}\ l_{\text{case}_n}()\ \texttt{in}\ [\![\ \texttt{S}\ ]\!]$$

$$[\![\ \texttt{default: S}\ ]\!] \triangleq \texttt{save}\ l_{\text{default}}()\ \texttt{in}\ [\![\ \texttt{S}\ ]\!]$$

A **break** statement is simply elaborated into a jump to the "break" label of the inner most switch that contains it:

$$[\![\ \texttt{break;}\ ]\!] \triangleq \texttt{run}\ l_{\text{break}}$$

The Core shown in Figure 4.12, is a substantially simplified version of our actual elaboration function that focuses on the control-flow aspects and, for the sake of clarity, elides the modelling of the lifetime of memory objects. Within the body of a **switch** statement, a **case** or **default** statement may occur within an inner block containing the declaration of local identifiers. In this case, the behaviour of the jump performed by the **switch** continues to mimic **goto** statements, and introduces implicit allocation or deallocation events.

[6]Note that it shows the case of a switch with no **default** statement in its body; otherwise, the last jump would be to a label corresponding to the body of the default.

```
1  {
2    int *p;
3    switch (2) {
4      case 1:
5        ...
6      {
7        int x;
8        case 2:
9          p = &x;
10         break;
11     }
12     ...
13   }
14   *p; // undefined
15 }
```

Figure 4.13: A `switch` statement leaking the pointer of a local variable

For example, in Figure 4.13: from line 3, the program execution jumps to line 8, and, in doing so, allocates a memory object for the identifier `x`, as a result of entering the block on lines 6–11. When the execution exits the `switch` statement at line 10, it also exits the block, and therefore the object is deallocated. As a result, the dereferencing of `p` at line 14 is attempting to read from a dead object, and has undefined behaviour.

To model this, as described at the end of Section 4.4, the elaboration function performs a preliminary pass on the Ail AST to collect the set of "visible" identifiers from every point of the AST that can either be the source or the target of a jump. In the present case, `switch`/`break` statements are source points and `case`/`default` are targets. The Core label associated to a target is defined as taking a pointer argument for each C identifier visible from the statement it is elaborating. In the elaboration case presented in Figure 4.12, all occurrences of Core's **`run`** operator are complemented by a preceding sequence of **`kill`** actions for each C identifier which is visible from the syntactic point from where the jump starts, but not from the syntactic point where is moves the program execution to. Conversely, they are also preceded by **`create`** actions for each C identifier visible from the target, but not the source. The pointer values produced by these allocating actions are passed as arguments to the **`run`** operator.

Iteration statements The three looping constructs in C have similar dynamics, differing only in the order in which their body and controlling expression are evaluated, as suggested by their syntax, and, for `for` statements, whether additional expressions are evaluated between iterations.

- `while (E) S`
- `do S while (E)`
- `for (E1; E2; E3) S // where E1, E2, and E3 are all optional`
- `for (declaration; E2; E3) S // where E2 and E3 are both optional`

The dynamics of the controlling expression follows that of the `if` statement: it has a scalar

type, and is compared to zero (or a null pointer). If it compares equal to zero, program execution exits the statement; otherwise, it loops back.

Within the body of an iteration statement, the control flow can be further modified in two ways: using a **break** statement, the loop can be exited (skipping the remainder of the ongoing iteration); and, using a **continue** statement, program execution can be made to fast forward to the next evaluation of the controlling expression.

In Cerberus, the desugaring from Cabs to Ail transforms these two constructs into corresponding **goto** statements: a **break** statement is turned into a **goto** to an implicit label placed just after the iteration statement, while a **continue** statement is turned into a **goto** to an implicit label placed just before the controlling expression. For example, the **while** statement on the left side of Figure 4.14 is rewritten into the block statement on the right side, where `cont` and `brk` are fresh label identifiers for this particular loop. The new body `S2` is the result of performing the same rewrite to `S` for any inner iteration statement.

```
while ( E ) {
  S;
}
```

```
{
  while ( E ) {
    S2;
cont: ;
  }
brk: ;
}
```

Figure 4.14: Rewriting of a **while** statement

The elaboration of the iteration statements themselves is straightforward, turning both **while** and **do** statements into backward jumps:

$$[\![\ \texttt{while (E) S}\ ]\!] \triangleq \begin{array}{l} \textbf{save}\ l_{\text{loop}}(\dots)\ \textbf{in} \\ \quad \textbf{letstrong}\ z : \texttt{integer} = [\![\ \texttt{E == 0}\ ]\!]\ \textbf{in} \\ \quad \textbf{if}\ \texttt{not}(z = 1)\ \textbf{then} \\ \quad\quad [\![\ \texttt{S}\ ]\!];\ \textbf{run}\ l_{\text{loop}}(\dots) \\ \quad \textbf{else pure}(\texttt{Unit}) \end{array}$$

$$[\![\ \texttt{do S while (E)}\ ]\!] \triangleq \begin{array}{l} \textbf{save}\ l_{\text{loop}}(\dots)\ \textbf{in} \\ \quad [\![\ \texttt{S}\ ]\!]; \\ \quad \textbf{letstrong}\ z : \texttt{integer} = [\![\ \texttt{E == 0}\ ]\!]\ \textbf{in} \\ \quad \textbf{if}\ \texttt{not}(z = 1)\ \textbf{then} \\ \quad\quad \textbf{run}\ l_{\text{loop}}(\dots) \\ \quad \textbf{else pure}(\texttt{Unit}) \end{array}$$

Figure 4.15: Case of the elaboration function for **while** and **do** statements

Note that the Core labels and **run** operators will have parameters to deal with the implicit allocation and deallocation of objects with automatic storage duration. This is done exactly as described for the elaboration of the **switch** statement.

The `for` statement adds the possibility of specifying an expression that is to be evaluated upon initial entry of the iteration statement (`E1`), and an expression to evaluate after each complete iteration (`E3`). In the Cerberus pipeline, these are simply desugared during the Cabs to Ail transform into a `while` statement, blocks, and labelled statements (the elaboration of `for` statements therefore relies on the elaboration of these). A `for` statement like the one on the left side of Figure 4.16 is rewritten into the block statement on the right side, where `cont` is the fresh label to which any `continue` statement within the body `S` jumps, and where `S2` is the result of recursively applying the rewrite to `S`. In the absence of either `E1` or `E3` the rewrite uses a null statement in their place. And in the absence of `E2`, the `while` statement is given the constant `1` as its controlling expression.

```
for ( E1; E2; E3 ) {
  S;
}
```

```
{
  E1;
  while ( E2 ) {
    S2;
cont:
    E3
  }
}
```

Figure 4.16: Rewriting of a `for` statement

Jump statements The `goto` and label statements generalise the `continue` statements we have just discussed, and their elaboration to Core follows the same pattern.

- `l: S`
- `goto l;`
- `return E; // where E is optional`

A labeled statement is elaborated as follows:

$$[\![\ \texttt{l: S}\ ]\!] \triangleq \texttt{save}\ l(ptr_{x_1} : \texttt{pointer := } [\![\ \texttt{x}_1\ ]\!], \ldots, ptr_{x_n} : \texttt{pointer := } [\![\ \texttt{x}_n\ ]\!])\ \texttt{in}\ [\![\ \texttt{S}\ ]\!]$$

Figure 4.17: Case of the elaboration function for labeled statements

where $\{\texttt{x}_1, \ldots \texttt{x}_n\}$ is the set of C variables visible from the scope of the statement S, and where ptr_{x_i} is the Core variable that was generated by the elaboration for the C identifier x_i. This variable will typically have been introduced by the sequencing operator binding the result of an allocating memory action (e.g. in the elaboration of an identifier declaration).

The elaboration of a `goto` is more complex, as allocations and deallocations are made

explicit there:

$$
\begin{array}{l}
[\![\ \texttt{goto l;}\ ]\!] \triangleq \\
\\
\\
\\
\\
\\
\\
\\
\\
\end{array}
\begin{array}{l}
\textbf{kill}(ptr_{x_1})\ ; \\
\ldots \\
\textbf{kill}(ptr_{x_i})\ ; \\
\textbf{letstrong}\ ptr_{y_1} : \texttt{pointer} = \textbf{allocate\_object}(\texttt{Ivalignof}([\![\tau_1]\!]), [\![\tau_1]\!])\ \textbf{in} \\
\ldots \\
\textbf{letstrong}\ ptr_{y_j} : \texttt{pointer} = \textbf{allocate\_object}(\texttt{Ivalignof}([\![\tau_2]\!]), [\![\tau_2]\!])\ \textbf{in} \\
\textbf{run}\ l(ptr_{y_1}, \cdots, ptr_{y_j}, ptr_{z_1}, \cdots, ptr_{z_k})
\end{array}
$$

where $\{\texttt{x}_1, \ldots \texttt{x}_i\}$ is the set of C variables visible from the scope of the **goto**, but out of scope at the target of the jump; $\{\texttt{y}_1, \ldots \texttt{y}_j\}$ is the set of C variables not visible from the scope of the **goto**, but visible from the scope of the target of the jump; and $\{\texttt{z}_1, \ldots \texttt{z}_k\}$ is the set of C variables visible both from the scope of the **goto** and the target of the jump.

The elaboration of the **return** statement makes use of Core's **run** operator to a particular label l_{ret} that the elaboration function creates for each C function, which takes a single parameter which is either of type unit (for C functions with the **void** return type), or the Core type corresponding to the non-**void** type of the C function. When elaborating a complete function definition, a **save** operator for that label l_{ret} is placed at the end of the corresponding Core procedure:

$$
\begin{array}{l}
[\![\ \tau_{ret}\ \texttt{f(...) S}\ ]\!] \triangleq \\
\\
\\
\\
\end{array}
\begin{array}{l}
[\![\ \texttt{S}\ ]\!]\ ; \\
\textbf{proc}\ f(\ldots) : [\![\ \tau_{ret}\ ]\!]\ \texttt{:=} \\
\quad \textbf{save}\ l_{ret}(v : [\![\ \tau_{ret}\ ]\!]\ \texttt{:=}\ \textbf{undef}(\ldots))\ \textbf{in} \\
\quad \textbf{pure}(v)
\end{array}
$$

Figure 4.18: Elaboration of C function with non-**void** return type, other than the startup

For a non-**void** function, the ISO standard states that reaching the end of its body without having seen a **return** statement has undefined behaviour. The elaboration models this by placing the **undef** operator as the default parameter of the label l_{ret}. The startup function is a special case in the ISO standard, such that in the absence of any **return** statement the execution remains defined with value zero. To model this, the elaboration simply substitute the constant zero for the **undef** operator. Functions with the **void** return type also do not require an explicit use of the **return** statement. The elaboration therefore places the unit constant as the default parameter.

In this context, the elaboration of the **return** statement is simply:

$$
\begin{array}{l}
[\![\ \ \texttt{return E;}\ \ ]\!] \triangleq \\
\\
\end{array}
\begin{array}{l}
\textbf{letstrong}\ v : T = [\![\ \texttt{E}\ ]\!]\ \textbf{in} \\
\textbf{run}\ l_{ret}(v)
\end{array}
$$

where T is the Core type corresponding the return type of the enclosing C function.

4.6 Uses of uninitialised memory

The ISO standard introduces situations where a value may not be fully defined. For example, reading from an uninitialised object (e.g. one with automatic storage duration but no initialiser or preceding assignment) results in an *indeterminate value*, which is defined as being either:

- a *trap representation*, as a result of reading an object representation that cannot be interpreted as a value of the desired type; or,
- an *unspecified value* which, while being a valid value of the relevant type, is left under defined by the standard, as it "imposes no requirements on which value is chosen in any instance".

4.6.1 Trap representations

Reading a trap representation through an lvalue with a non-character type has undefined behaviour (C11, §6.2.6.1p5). This mechanism allows implementations to reserve bit-patterns for each type that they can assume never occur during normal program execution. This can either correspond to a hardware trap, for example easing the implementation of pointer types on architecture with segmented memory, or be used by compilers to justify some of their optimisations. This is for example visible in the implementation of the type `_Bool`. Consider the following function[7]:

```
1  int f(_Bool *p)
2  {
3    return *p ? 1 : 0;
4  }
```

When compiled with either GCC or Clang with optimisations turned on, the return statement is translated into a single branch-less load instruction placing the value of dereferencing `p` into the return register of the function. Should the function be called with a pointer referring to an object whose representation corresponds to an integer greater than one at the assembly level, the runtime behaviour of the optimised version of `f` will return neither `0` nor `1`. This is sound because compilers implement the type `_Bool` with only two valid values, namely `0` and `1`, and all bit-patterns are trap representations. The load performed by the dereferencing at line 3 therefore has undefined behaviour, thereby allowing the optimisation.

It is however unclear how much common compilers make use of trap representations for other integer types. In particular, GCC documents that its implementation of integer types is such that "all bit patterns are ordinary values". They are however potentially encountered on floating types as a result of *signalling NaN's*. The Itanium architecture possesses a NaT ("not a thing") flag which is sometimes presented as the rationale for trap representations. This is however inaccurate, as they can only appear in machine registers, whereas the standard (C11, §6.2.6.1p5) describes trap representations as bit patterns storable in memory. Notwithstanding the latter, some WG14 members believe that any type might have trap representations, even if there are no unused bit patterns.

We believe that the standard would be clearer if it equipped each base C type with an implementation-defined set of trap representations. This would allow implementations

[7] adapted from https://trust-in-soft.com/blog/2016/06/16/trap-representations-and-padding-bits/

that do not make use of trap representations to collapse the notion of indeterminate value with that of unspecified value.

4.6.2 Unspecified values

Even in the absence of trap representations, the standard has additional text making a read from an uninitialised object undefined behaviour (C11, §6.3.2.1p2), as long as the address of that object has never been taken (with the exception of reads through an `unsigned char` lvalue). This, however seems, too restrictive as it prevents the member-wise copying of partially initialised structures, which appears to be common practice. While this suggest that the ISO standard should be changed to remove or reduce the scope of this undefined behaviour, a commonly voiced objection within WG14 is that some implementations rely on it to issue warnings useful for detecting bugs resulting from reads of uninitialised objects. In Cerberus, we explore a semantics which gives defined semantics to some uses of uninitialised memory. We assume that only `_Bool` has trap representations (all the representation not corresponding to 0 or 1), and we give a defined semantics for reads of uninitialised objects, involving unspecified values.

The definition of unspecified values in the standard leaves room for several quite different interpretations. For example we could have:

1. at the beginning of the lifetime of objects with automatic storage duration with no apparent initialisation, the semantics chooses nondeterministically a concrete value, and silently initialises the object. Subsequent read accesses to the object see a stable concrete value.

2. each base type could have a symbolic constant representing an abstract unspecified value, which is implicitly stored in objects with no initialisers. This leaves further semantic choices; in particular, read accesses could either nondeterministically choose a concrete value whenever they observe these symbolic constants, or the symbolic constant could be preserved and somehow propagated by arithmetic operations. And control-flow depending on a unspecified value could either be made non-deterministic, or have undefined behaviour.

3. as suggested by Besson et al. [BBW14], at each occurrence of an unspecified value, pick a fresh symbolic value (at the granularity of bits, bytes, or the whole value), and allow some symbolic computation over this.

In Cerberus, we opt for an abstract unspecified value which is propagated by arithmetic in a strict way, with the exception that operations that can have undefined behaviour are *daemonic* when applied to an unspecified value. For example, as the unsigned addition operator over integer types has no undefined behaviour, if either of its operands is an unspecified value, the result is itself the abstract unspecified value for the appropriate type. However, applying an unspecified value as right operand of the division operator has undefined behaviour (because there exists a concrete value, namely zero, for which the operator has undefined behaviour). Similarly, if either operands of a signed addition operator over integer types is an unspecified value, the operator has undefined behaviour (because there exists a concrete value causing an overflow). When the controlling expression of a statement evaluates to an unspecified value, the control-flow will nondeterministically behave as if it evaluated to 0 or 1.

To model this semantics using the elaboration function, we equip Core with an option type `loaded` T, where the parameter T is restricted to object types (e.g. `integer`), and the values are either of two variants: `Specified`(v), where v is a concrete value of type T; and `Unspecified`(τ), which models unspecified values for the C type τ. Memory accesses operate over a loaded type. The return type of a **`load`** action to an object of type **`signed int`** has type `loaded integer`, and the **`store`** action takes an operand with a loaded type for the value being stored. All other operators are directly defined over the object types. For example, Core's addition operator is only applied to operands of type `integer` or `floating`.

The strictness and daemonic semantics of unspecified values is made explicit by the elaboration. For example consider the elaboration of C's addition operator on a signed integer type[8]:

$$
\begin{array}{l}
[\![\ \mathtt{E}_1 + \mathtt{E}_2\ ]\!] \triangleq \\
\quad \mathbf{letweak}\ lv_1 : \mathtt{loaded\ integer} = [\![\ \mathtt{E}_1\ ]\!]\ \mathbf{in} \\
\quad \mathbf{letweak}\ lv_2 : \mathtt{loaded\ integer} = [\![\ \mathtt{E}_2\ ]\!]\ \mathbf{in} \\
\quad \mathbf{pure}(\\
\quad\quad \mathbf{case}\ (lv_1,\ lv_2)\ \mathbf{of} \\
\quad\quad\quad \mid (\mathtt{Specified}(v_1),\ \mathtt{Specified}(v_2)) \Rightarrow \\
\quad\quad\quad\quad \mathtt{Specified}(v_1 + v_2) \\
\quad\quad\quad \mid \_ \Rightarrow \\
\quad\quad\quad\quad \mathbf{undef}(\ldots) \\
\quad\quad \mathbf{end} \\
\quad)
\end{array}
$$

the elaborations of the operands have loaded types: if their values are destructed, and neither are unspecified, Core's addition is used and applied to the `Specified` constructor; otherwise, because of the daemonic behaviour, the **`undef`** operator signalling a signed integer overflow is used.

The nondeterminism of a control operator guarded by an unspecified value is modelled in a similar fashion. For example, consider the elaboration of the **`if`** statement:

$$
\begin{array}{l}
[\![\ \mathbf{if}\ (\mathtt{E})\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2\ ]\!] \triangleq \\
\quad \mathbf{letstrong}\ \_z : \mathtt{loaded\ integer} = [\![\ \mathtt{E}\ \texttt{==}\ \texttt{0}\ ]\!]\ \mathbf{in} \\
\quad \mathbf{letstrong}\ b : \mathtt{boolean} = \\
\quad\quad \mathbf{case}\ \_z\ \mathbf{of} \\
\quad\quad\quad \mid \mathtt{Specified}(z : \mathtt{integer})\ \texttt{=>}\ \mathtt{not}(z = 1) \\
\quad\quad\quad \mid \mathtt{Unspecified}(\_ : \mathtt{ctype})\ \texttt{=>}\ \mathbf{nd}(\mathtt{true}, \mathtt{false}) \\
\quad\quad \mathbf{end}\ \mathbf{in} \\
\quad \mathbf{if}\ b\ \mathbf{then}\ [\![\ \mathtt{S}_1\ ]\!]\ \mathbf{else}\ [\![\ \mathtt{S}_2\ ]\!]
\end{array}
$$

The elaboration of the **`switch`** and iteration statements we presented in Section 4.5 are similarly enriched.

[8] For legibility, we simplify the elaboration of sequencing and omit implicit conversions which are both irrelevant here.

Modelling the subtleties of unspecified values mostly using the elaboration has the advantage of keeping the semantics of the Core simple. But the obvious cost is more verbose Core programs.

Cerberus also supports a semantic switch where a load resulting in an unspecified value directly has undefined behaviour, matching more closely the phrasing of the ISO standard. This switch is implemented by changing the dynamics of the load operator within the definition of the memory object model, with no change to the elaboration function. With such a dynamics, all the pattern-matching produced by the elaboration is unnecessary, as the **`Unspecified`** variant never occurs at runtime. As result, in this switch, the pipeline includes some Core-to-Core partial evaluation, that erases such dead code.

Chapter 5

Overview of the memory interface

A key design choice of Cerberus is the isolation of the implementation of the memory object model from the rest of our modelling of C's dynamics (as embodied in the elaboration and dynamics of Core). This allows us to experiment with different memory object models without reworking the part of the semantics not relating to the memory. The interaction between Core's dynamics and the memory goes through a small interface, which keeps key types opaque. In this chapter, we give a brief overview of this interface. A complete presentation motivating the design choices is given in Appendix A.

The interface declares the following key types:

- *mem_state*, the memory state which it declares opaque, and over which it places no requirements;
- `'a` *memM*, the monad over which memory actions and some operations occur. This type is, in contrast, fixed by the interface, and supports the following features: errors relating to the memory; undefined behaviour; state; and nondeterminism guarded by symbolic constraints.
- *pointer_value*, the type implementing pointer values, which is kept opaque to ensure the dynamics of Core is agnostic to its definition, and in particular to allow the exploration of different semantics of pointers (as we discuss in Chapter 8) without changing the elaboration function.
- *integer_value*, also kept opaque, and which provides the unbounded integers used by Core.
- *mem_value*, the type of memory values used by the load and store memory actions.

Because the types implementing pointers and integer are opaque, the interface also declares pure functions for constructing their values, and a destructor function. For example, it declares the function *null_ptrval* taking a C type and producing a null pointer for that type. It also declares the functions implementing their operators: arithmetic and bitwise operators for integers, pointer arithmetic operators, equality, and relational operators. For the conversions between integer and pointer values, the interface declares two effectful functions (running in the monad *memM*). The effect is necessary because implementations of these may require access to the state of the memory, and some cases have undefined behaviour.

The interface also declares effectful functions for: allocating memory objects and regions (which is the main mean of producing pointer pointers); deallocating memory object

and regions; and, performing memory accesses on them. These are counterparts of Core's memory actions: **allocate_object**(), **allocate_region**(), **kill**(), **load**(), and **store**(), used by the dynamics of Core for their implementation.

In Appendix A, we give the complete list of the types and functions declared by the memory interface, along with their signatures.

Chapter 6

Formal presentation of Core

In this chapter, we expand the overview given in Section 3.3, and give a complete presentation of the Core language's semantics. Its basis is a strongly typed functional language with two syntactically segregated components: a pure fragment resembling a subset of an ML-like language; and an effectful fragment where interaction with the memory state uses memory actions combined using an expressive calculus of sequencing operators.

6.1 The pure fragment

The pure fragment of Core is a first-order functional language with recursive functions, along with constructs motivated by its use as the target of the elaboration of C programs.

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \mathit{<impl\_const>} \\
& | & \mathit{value} \\
& | & \mathbf{undef}(\mathit{ub\_name}) \\
& | & \mathit{ctor}(e_1, \ldots, e_n) \\
& | & \mathbf{array\_shift}(e_1, \tau, e_2) \\
& | & \mathbf{member\_shift}(e, T.z) \\
& | & \mathbf{case}\ e\ \mathbf{of}\ \overline{\mathit{pat}_i\ \texttt{=>}\ e_i}^{i \in I}\ \mathbf{end} \\
& | & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
& | & \mathbf{let}\ \mathit{pat} = e_1\ \mathbf{in}\ e_2 \\
& | & \mathbf{not}(e) \\
& | & e_1 \odot e_2 \\
& | & \mathbf{struct}[T](.z_1 = e_1, \ldots, .z_n = e_n) \\
& | & \mathbf{union}[T](.z = e) \\
& | & \mathbf{cfunction}(e) \\
& | & \mathbf{memberof}[T.z](e) \\
& | & \mathit{nm}(e_1, \ldots, e_n) \\
& | & \mathbf{refine\_ctype}(e_1, e_2, e_3)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{nm} & ::= & \mathit{<impl\_const>} \\
& | & \mathit{fun\_name}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{pat} & ::= & \_ : T \\
& | & x : T \\
& | & \mathit{ctor}(\mathit{pat}_1, \ldots, \mathit{pat}_n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ctor} & ::= & \texttt{Nil}_T \mid \texttt{Cons} \\
& | & \texttt{Tuple} \mid \texttt{Array} \\
& | & \texttt{Ivmin} \mid \texttt{Ivmax} \\
& | & \texttt{Ivsizeof} \mid \texttt{Ivalignof} \\
& | & \texttt{IvCOMPL} \mid \texttt{IvAND} \mid \texttt{IvOR} \mid \texttt{IvXOR} \\
& | & \texttt{Specified} \mid \texttt{Unspecified} \\
& | & \texttt{Ivfromfloat} \mid \texttt{Fvfromint}
\end{array}
$$

$$\odot \in \{+, -, *, /, \texttt{rem\_t}, \texttt{rem\_f}, \hat{}, =, <, >, \leq, \geq, \wedge, \vee\}$$

Figure 6.1: Grammar for Core's pure expressions

While pure in the sense that it allows no memory interaction, this fragment allows one effect: failure as a result of an undefined behaviour. The occurrence of an undefined

behaviour is explicitly signaled using the **undef**() operator. Pure expressions (along with the rest of Core) are strongly typed. Because the **undef**() operator is meant to act as a place-holder for sub-expressions with no defined semantics, its type cannot be determined without a context. While all variables introduced by binders and the definitions of functions are all type annotated, we opted to keep the **undef**() operator unannotated. As a result of this, we write the type system in a bidirectional style [DK21], with the following two judgements for typechecking (blue arrow) and type synthesis (red arrow):

$$\Gamma \vdash e \Leftarrow T \qquad \Gamma \vdash e \Rightarrow T$$

The absence of memory interactions in pure expressions makes sequencing irrelevant, which allows the dynamics to be written as a big-step semantics, where a pure expression either successfully evaluates to a value, or results in undefined behaviour:

$$e \Downarrow \text{DEFINED}(v) \qquad \text{and} \qquad e \Downarrow \text{UNDEF}$$

Both the typing rules and the evaluation rules make use of an environment defining the type signatures, parameters, and bodies of functions, along with the definitions of struct and union types coming from the particular C translation unit being elaborated. As this environment is immutable, we make it mostly implicit in the typing and evaluation rules, to simplify the notation. We write a successful query of the definition of a function as:

$$\text{funmap}(nm) = (x_1 : T_1, \ldots, x_n : T_n).\, e$$

and a successful query of the members associated to a struct or union type with tag T as:

$$\text{members}(T) = (.z_1 : \tau_1, \ldots, .z_n : \tau_n)$$

Some typing rules need to refer to the Core object type corresponding to a C type τ, which we will write $[\![\tau]\!]$. For example, $[\![$`signed int`$]\!] =$ `integer`, while $[\![$`unsigned char*`$]\!] =$ `pointer`.

Values and types Values are structured in three layers. In the innermost layer are *object values*: integers, floating and pointer values, along with arrays, structs, and unions. These correspond to the kinds of values a C program may store to, and load from, its memory. In Core, these accesses are performed using memory actions, which are part of the effectful fragment we present in the next section. These object values correspond to the logical notion of C values, as they appear in the C abstract machine described by the C standard. For example in Core, integer values are, for the most part[1], mathematical integers without fixed size or bounded arithmetic, and the type `integer` holds all integer values, regardless of whether they result from loading a memory object with **signed int** or **unsigned long** C type. The same holds for floating and pointer values.

In Chapter 5, we have seen that the details of *integer_value*, *floating_value* and *pointer_value* are kept abstract by the memory interface. Within Core expressions, they are constructed and manipulated with various data constructors and operators defined in terms of the corresponding constructors and functions that are part of the memory interface.

[1]The extent to which integer values actually satisfy the algebraic properties of mathematical integers differs between different memory object models. This is discussed in Chapter 9.

Because memory objects may be uninitialised, object values are wrapped in a second layer of *loaded values*, which have two variants: `Specified`, which holds an object value; and `Unspecified`, for a given C type. These two layers are mutually recursive: an element of a specified array value, or a member of a specified struct/union value can itself be unspecified.

$$
\begin{array}{lcl}
ov, object\_value & ::= & integer\_value \\
& | & floating\_value \\
& | & pointer\_value \\
& | & \texttt{array}(lv_1, \ldots, lv_n) \\
& | & \texttt{struct}[T](\overline{.x_i = lv_i}^{i \in I}) \\
& | & \texttt{union}[T](.z = lv)
\end{array}
$$

$$
\begin{array}{lcl}
lv, loaded\_value & ::= & \texttt{Specified}(object\_value) \\
& | & \texttt{Unspecified}(\tau)
\end{array}
$$

$$
\begin{array}{lcl}
v, value & ::= & object\_value \\
& | & loaded\_value \\
& | & \texttt{Unit} \\
& | & \texttt{True} \\
& | & \texttt{False} \\
& | & \tau \\
& | & [v_1, \ldots, v_n] \\
& | & (v_1, \ldots, v_n)
\end{array}
$$

Figure 6.2: Grammar of values

Finally, the outermost layer adds values which have no counterpart in C, and cannot be loaded from or stored to the memory state: the unit value, boolean constants (which are not the same as C's `_Bool`, which is an integer type), C types as values (e.g. `'signed int'`), lists, and tuples.

The types in Core follow the structure of values: values from the inner layer have an *object type* (which we write oTy); the `Specified` and `Unspecified` constructors produce values with a *loaded type*.

$$
\begin{array}{lcl}
oTy & ::= & \texttt{integer} \\
& | & \texttt{floating} \\
& | & \texttt{pointer} \\
& | & \texttt{array}(oTy) \\
& | & \texttt{struct}\ T \\
& | & \texttt{union}\ T
\end{array}
$$

$$
\begin{array}{lcl}
T & ::= & oTy \\
& | & \texttt{loaded}\ oTy \\
& | & \texttt{unit} \\
& | & \texttt{boolean} \\
& | & \texttt{ctype}_{kind} \\
& | & [T] \\
& | & (T_1, \ldots, T_n)
\end{array}
$$

$$
\begin{array}{lcl}
kind & ::= & oTy \\
& | & \texttt{wild}
\end{array}
$$

Figure 6.3: Core types

A C type as value has the type $\texttt{ctype}_{oTy}$, where the parameter oTy is the Core object type corresponding to the C type. For example, the value `'signed int'`, has type $\texttt{ctype}_{\texttt{integer}}$. It is possible to construct expressions that may evaluate to C types associated with different object types, e.g. **`if`** e **`then`** `'signed int'` **`else`** `'float'`. To such expressions, we give the type $\texttt{ctype}_{\texttt{wild}}$.

Literals Integer and floating values can be constructed with the usual numeric literals. In addition there is a literal $\texttt{offsetof}(T, .z)$ for constructing the integer value holding the numeric offset of a particular member of a struct type. For pointer values, there is a null pointer literal, $\texttt{Null}(\tau)$ taking as parameter the C type of the reference, and a function designator literal, which takes as parameter a C function name: $\texttt{Cfunction}(f)$.

Implementation-defined symbols As mentioned in Chapter 3, we designed Core so that the elaboration function is mostly agnostic with respect to details which are implementation-defined in C. To a large extent, these details have to do with the representation of integer and floating types. For these, we equipped Core with various data constructors to abstract these away, using the memory interface. We detail this in the next paragraph. Other implementation-defined behaviours do not relate to the memory: e.g. for conversions of values to a signed integer type whose range is too narrow to hold that value as is. For convenience, we want to write the definition of such behaviour for various implementations in Core itself, inside an "implementation file". To refer to them, Core has a distinct predefined set of symbols (written <*impl_const*>), which may appear as a variable in expressions, or as the first operand of a call.

Data constructors Values are also built using various constructors (we write their application $ctor(e_1, \ldots, e_n)$ in Figure 6.1). The first class of constructors build integer values relating to implementation-defined choices. They exist to keep Core expressions agnostic of the choices for the size and alignment constraint of C types, along with the value range of C integer types. They are: `Ivmin` and `Ivmax`, which both take as operand a C integer type expression, and yield the minimal (resp. maximal) value for the type; and `Ivsizeof` and `Ivalignof`, which both take a C type expression, and yield as an integer value the size in bytes (resp. the alignment constraint) for that type.

$$\begin{aligned} \texttt{Ivmin} &: \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \\ \texttt{Ivmax} &: \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \\ \texttt{Ivsizeof} &: \texttt{ctype}_{\texttt{wild}} \to \texttt{integer} \\ \texttt{Ivalignof} &: \texttt{ctype}_{\texttt{wild}} \to \texttt{integer} \end{aligned}$$

Figure 6.4: Constructors for implementation-defined values

The conversions between integer and floating values (which are used for the elaboration of corresponding C casts and conversions) are also implemented using constructors: `Ivfromfloat`, taking as its operand a floating value, and yielding the corresponding integer value; and, conversely, `Fvfromint`, taking as its an integer value, and yielding the corresponding floating value.

$$\begin{aligned} \texttt{Ivfromfloat} &: \texttt{floating} \to \texttt{integer} \\ \texttt{Fvfromint} &: \texttt{integer} \to \texttt{floating} \end{aligned}$$

Figure 6.5: Constructors for conversions between integer and floating types

To allow the elaboration of C's integer bitwise operators, the bitwise complement, bitwise AND, OR, and XOR are provided as constructors. They all take as first operand the C integer type over which the operation is performed (from which it determines the bit width). Note that we originally implemented these constructors directly in Core as auxiliary pure functions, but then opted to internalise them into the memory interface to allow for a significant performance boost in the execution of Core programs (and in a symbolic mode of Cerberus that we describe in Chapter 11).

$$
\begin{array}{rcl}
\texttt{IvCOMPL} & : & \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \to \texttt{integer} \\
\texttt{IvAND} & : & \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \to \texttt{integer} \to \texttt{integer} \\
\texttt{IvOR} & : & \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \to \texttt{integer} \to \texttt{integer} \\
\texttt{IvXOR} & : & \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \to \texttt{integer} \to \texttt{integer}
\end{array}
$$

Figure 6.6: Integer and floating constructors

Their dynamics over Core object types are therefore now defined in term of the corresponding functions declared by the memory interface. Their typing is done by a type synthesis rule, using the signatures we have just presented to typecheck their operands:

$$\frac{ctor : T_1 \to \cdots \to T_n \to T \qquad \Gamma \vdash e_i \Leftarrow T_i}{\Gamma \vdash ctor(e_1, \ldots, e_n) \Rightarrow T} \qquad \frac{ctor : T_1 \to \cdots \to T_n \to T \qquad \Gamma \vdash e_i \Leftarrow T_i}{\Gamma \vdash ctor(e_1, \ldots, e_n) \Leftarrow T}$$

Loaded values have two constructors, corresponding to their two variants: `Specified`, taking an expression with object type, and turning it into a loaded value; and `Unspecified`, taking an expression with type $\texttt{ctype}_{oTy}$ (that is, with a kind parameter other than `wild`), and yielding the unspecified value for that C type.

$$\frac{e \Rightarrow oTy}{\Gamma \vdash \texttt{Specified}(e) \Rightarrow \texttt{loaded}\ oTy} \qquad \frac{e \Leftarrow oTy}{\Gamma \vdash \texttt{Specified}(e) \Leftarrow \texttt{loaded}\ oTy}$$

$$\frac{e \Rightarrow \texttt{ctype}_{oTy}}{\Gamma \vdash \texttt{Unspecified}(e) \Rightarrow \texttt{loaded}\ oTy} \qquad \frac{e \Leftarrow \texttt{ctype}_{oTy}}{\Gamma \vdash \texttt{Unspecified}(e) \Leftarrow \texttt{loaded}\ oTy}$$

The remaining constructors are for lists, tuples, and C arrays, and have the obvious operands and type signatures.

$$
\begin{array}{rcl}
\texttt{Nil}_T & : & [T] \\
\texttt{Cons} & : & T \to [T] \to [T] \\
\texttt{Tuple} & : & T_1 \to \ldots \to T_n \to (T_1, \ldots, T_n) \\
\texttt{Array} & : & \texttt{loaded}\ oTy \to \ldots \to \texttt{loaded}\ oTy \to \texttt{array}(oTy)
\end{array}
$$

Figure 6.7: Other constructors

As some constructors have multiple operands with the same types, the actual type rules are duplicated in the obvious way to satisfy what [DK21] refers to as the "mode-correctness" criterion. We omit these details here.

Undefined behaviour To denote when some execution paths of C program being elaborated result in an undefined behaviour, Core is equipped with an unary **`undef`**() operator whose operand is an identifier for the particular instance of undefined behaviour being signaled. The operator is allowed to take any type as long as its context determines it, as reflected by its typechecking rule (and the absence of an inference rule):

$$\frac{}{\Gamma \vdash \textbf{undef}(ub\_name) \Leftarrow T}$$

The dynamics of the **undef**() operator is simply to stop the evaluation of a pure expression if the **undef**() is ever reached, and to collapse the whole expression to the outcome UNDEF. As expected, the occurrence of an UNDEF operator in a branch of an **if** or **case** control operator that does not get taken has no effect.

$$\frac{}{\textbf{undef}(ub\_name) \Downarrow \text{UNDEF}} \qquad \frac{\exists i \in \{1, \ldots, n\}.\, e_i \Downarrow \text{UNDEF}}{ctor(e_1, \ldots, e_n) \Downarrow \text{UNDEF}}$$

$$\frac{\exists i \in \{1, 2\}.\, e_i \Downarrow \text{UNDEF}}{\textbf{array\_shift}(e_1, \tau, e_2) \Downarrow \text{UNDEF}} \qquad \frac{e \Downarrow \text{UNDEF}}{\textbf{member\_shift}(e, T.z) \Downarrow \text{UNDEF}}$$

$$\frac{e \Downarrow \text{UNDEF}}{\textbf{case}\ e\ \textbf{of}\ \overline{pat_i\ \texttt{=>}\ e_i}^{i \in I}\ \textbf{end} \Downarrow \text{UNDEF}} \qquad \frac{e_1 \Downarrow \text{UNDEF}}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow \text{UNDEF}}$$

$$\frac{e_1 \Downarrow \text{UNDEF}}{\textbf{let}\ pat = e_1\ \textbf{in}\ e_2 \Downarrow \text{UNDEF}} \qquad \frac{e \Downarrow \text{UNDEF}}{\textbf{not}(e) \Downarrow \text{UNDEF}}$$

$$\frac{\exists i \in \{1, 2\}.\, e_i \Downarrow \text{UNDEF}}{e_1 \odot e_2 \Downarrow \text{UNDEF}} \qquad \frac{\exists i \in \{1, \ldots, n\}.\, e_i \Downarrow \text{UNDEF}}{\textbf{struct}[T](.z_1 = e_1, \ldots, .z_n = e_n) \Downarrow \text{UNDEF}}$$

$$\frac{e \Downarrow \text{UNDEF}}{\textbf{union}[T](.z = e) \Downarrow \text{UNDEF}} \qquad \frac{e \Downarrow \text{UNDEF}}{\textbf{cfunction}(e) \Downarrow \text{UNDEF}}$$

$$\frac{\exists i \in \{1, \ldots, n\}.\, e_i \Downarrow \text{UNDEF}}{nm(e_1, \ldots, e_n) \Downarrow \text{UNDEF}} \qquad \frac{\exists i \in \{1, 2\}.\, e_i \Downarrow \text{UNDEF}}{\textbf{refine\_ctype}(e_1, e_2, e_3) \Downarrow \text{UNDEF}}$$

Figure 6.8: Rules for undefined evaluations

Note that in the undefined rule for **refine_ctype**(), we only check the first two operands. We explain the motivation for this when presenting the semantics of the operator.

In the actual implementation of Cerberus, there is an additional **error**() operator, typed and behaving similarly to the **undef**() operator but instead used to signal static errors, e.g. C constraint violations resulting from an C integer constant that could not be typed. This is because, when invoked in its implementation-agnostic mode, Cerberus is not always able to statically detect some of these errors, and instead delays their detection to the Core runtime. As this operator is otherwise exactly like the **undef**() operator, we omit it from the presentation.

Pointer arithmetic operators There are two operators for dealing with pointer arithmetic: **array_shift**(), which is used for the elaboration of C's additive operators when they are applied to one pointer operand and one integer operand (which happens to also

be how C's array subscripting operator is defined, hence the name); and **member_shift**(), for the elaboration of C's member access operators. They both take as first operand the pointer on which the arithmetic is performed. The array operator then takes a C type literal denoting the element type of the array object within which the arithmetic is being performed (the case where the pointer operand refers to a scalar object behaves as if the object is an array of size one, as specified by the ISO standard), and an integer denoting by how much the pointer value is "shifted". The member operator instead takes a struct type name, and a member identifier for that type. The actual semantics of both operators is again hidden by the memory interface, but intuitively, the result of $\mathbf{array\_shift}(p, \tau, n)$ is a pointer value whose numeric address is that of p added to $\texttt{sizeof}(\tau)$ times n, while the result of $\mathbf{array\_shift}(p, T.z)$ is a pointer value whose numeric address is that of p added to $\texttt{offsetof}(T, z)$. Note that unlike pointer arithmetic in ISO C, these two operators are always well-defined; in particular, arithmetic resulting in an out of bounds pointer is defined. These operators are therefore only used by the elaboration function when Cerberus is invoked with semantic switches for which a more permissive semantics for pointer arithmetic is desired. To model the strict ISO semantics, there are two counterparts to these operators as part of the effectful calculus. This is because, in some implementations of the memory interface, the detection of out of bounds arithmetic requires inspecting the memory state.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{pointer} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{integer}}{\Gamma \vdash \mathbf{array\_shift}(e_1, \tau, e_2) \Rightarrow \texttt{pointer}}$$

$$\frac{\Gamma \vdash e \Leftarrow \texttt{pointer} \qquad .z \in \Gamma(T)}{\Gamma \vdash \mathbf{member\_shift}(e, T.z) \Rightarrow \texttt{pointer}}$$

Control operators Within the pure fragment, control is manipulated in a functional style: either using the **if** operator, or **case** for pattern matching, or through recursive function calls for looping. Note that looping in the pure fragment is only used by a few auxiliary functions part of the Core standard library (e.g. functions traversing lists), and does not correspond to how C's iteration statements are elaborated. Instead, those make use of additional control operators of the effectful calculus. The statics and dynamics of the **if** and **case** operators are mostly standard for a typed functional language. The guarding expression of an **if** operator is a boolean expression, as opposed to an integer for C's corresponding construct. The guard of a **case** operator is checked against the type inferred from the patterns.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{boolean} \qquad \Gamma \vdash e_2 \Rightarrow T \qquad \Gamma \vdash e_3 \Rightarrow T}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow T}$$

$$\frac{\Gamma \vdash e \Leftarrow T_1 \qquad is\_exhaustive(\overline{pat_i}^{i \in I}) \qquad \forall i \in I. \vdash pat_i \Rightarrow T_1 \qquad \forall i \in I.\, \Gamma, (pat_i : T_1) \vdash e_i \Rightarrow T}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \overline{pat_i\ \texttt{=>}\ e_i}^{i \in I}\ \mathbf{end} \Rightarrow T}$$

$$\frac{e_1 \Downarrow \text{DEFINED}(\texttt{True}) \qquad e_2 \Downarrow z}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow z}$$

$$\frac{e_1 \Downarrow \text{DEFINED}(\texttt{False}) \qquad e_3 \Downarrow z}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow z}$$

$$\frac{e \Downarrow \text{DEFINED}(v) \qquad match\_pattern(\overline{pat_i}^{i \in I}, v) = \texttt{Some}(k) \qquad \{v/pat_k\}e_k \Downarrow z}{\mathbf{case}\ e\ \mathbf{of}\ \overline{pat_i\ \texttt{=>}\ e_i}^{i \in I}\ \mathbf{end} \Downarrow z}$$

There is a small complication in the typing of both operators, in order to allow expressions such as:

$$\texttt{if}\ e\ \texttt{then}\ \texttt{'signed int'}\ \texttt{else}\ \texttt{'float'}$$

while preserving subject reduction. When branches of either operators have the type `ctype` but with different kinds, the type of the whole **`if`** or **`case`** expression is weakened to $\texttt{ctype}_{\texttt{kind}}$.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{boolean} \qquad \Gamma \vdash e_2 \Rightarrow \texttt{ctype}_{kind_1} \qquad \Gamma \vdash e_3 \Rightarrow \texttt{ctype}_{kind_2} \qquad kind_1 \neq kind_2}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{ctype}_{\texttt{wild}}}$$

$$\frac{\begin{array}{lll}\forall i \in I. \vdash pat_i \Rightarrow T_1 & \Gamma \vdash e \Leftarrow T_1 & is\_exhaustive(\overline{pat_i}^{i \in I}) \\ \forall i \in I. \Gamma, (pat_i : T_1) \vdash e_i \Rightarrow \texttt{ctype}_{kind_i} & & \exists i, j \in I.\, kind_i \neq kind_j\end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \overline{pat_i\ \texttt{=>}\ e_i}^{i \in I}\ \texttt{end} \Rightarrow \texttt{ctype}_{\texttt{wild}}}$$

Let binder and variables Values are bound to variables using a standard let constructor. For convenience, the binder can be an arbitrary pattern; this is mostly used to destruct tuples. As shown in Figure 6.1, variables in patterns are type-annotated, making the typing fully determined by the binder. The dynamics is then given, as is standard, by evaluating the first operand, and substituting in the second operand the variables of the pattern for the value.

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x \Rightarrow T} \qquad \frac{\vdash pat \Rightarrow T_1 \qquad \Gamma \vdash e_1 \Leftarrow T_2 \qquad (pat : T_1), \Gamma \vdash e_2 \Rightarrow T_2}{\Gamma \vdash \texttt{let}\ pat = e_1\ \texttt{in}\ e_2 \Rightarrow T_2}$$

$$\frac{e_1 \Downarrow \textsc{defined}(v) \qquad \{v/pat\}e_2 \Downarrow z}{\texttt{let}\ pat = e_1\ \texttt{in}\ e_2 \Downarrow z}$$

Figure 6.9: Statics and dynamics for the **`let`** binder

Binary operators and boolean not Arithmetic is performed using familiar binary operators $(+, -, *, /, \ \ldots)$ which operate over both the `integer` and `floating` types. Operands with different types cannot be mixed; conversions between integers and floating must instead be done explicitly. Their concrete semantics are again abstracted by the memory interface. For the memory object models we present in Chapter 9 where the `integer` type is simply the type of mathematical integers, they are usual non-wrapping arithmetic operations. All arithmetic operations are fully defined; in particular, division by zero is defined as zero, and similarly for the two 'remainder' operators. The undefined behaviour arising from a division by zero in C is made explicit by the elaboration function

using the **undef**() operator.

$$\frac{\Gamma \vdash e_1 \Leftarrow T \quad \odot \in \{+, -, *, /\} \qquad \Gamma \vdash e_1 \Leftarrow T \quad T \in \{\texttt{integer}, \texttt{floating}\}}{\Gamma \vdash e_1 \odot e_2 \Rightarrow T}$$

Additionally, for the `integer` type there is an exponentiation operator and two remainder operators: `rem_t`, which is truncating, rounding towards zero; and `rem_f`, which is flooring, rounding towards minus infinity. Again their concrete semantics are abstracted from Core by the memory interface, but it is expected their implementations follow what is described in the [ISO94] standard. The truncating variant is used for the elaboration of the modulo operator of C, while the flooring variant is used by the Core function modelling the modulo arithmetic of C's integer types.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{integer} \quad \Gamma \vdash e_1 \Leftarrow \texttt{integer} \quad \odot \in \{\texttt{rem\_t}, \texttt{rem\_f}\}}{\Gamma \vdash e_1 \odot e_2 \Rightarrow \texttt{integer}}$$

The equality operator (=) is available for integers, floats and C types. Unlike their C counterparts, its return type is `boolean`. For C types, the strict syntactic equality is performed.

$$\frac{\Gamma \vdash e_1 \Leftarrow T \quad \Gamma \vdash e_1 \Leftarrow T \quad T \in \{\texttt{integer}, \texttt{floating}, \texttt{ctype}\}}{\Gamma \vdash e_1 = e_2 \Rightarrow \texttt{boolean}}$$

The comparison operators ($<, >, \leq, \geq$) are available over integer and floats: they also return a boolean, and have the usual semantics.

$$\frac{\Gamma \vdash e_1 \Leftarrow T \quad \odot \in \{<, >, \leq, \geq\} \qquad \Gamma \vdash e_1 \Leftarrow T \quad T \in \{\texttt{integer}, \texttt{floating}\}}{\Gamma \vdash e_1 \odot e_2 \Rightarrow \texttt{boolean}}$$

Finally there are the boolean operators: the binary (conjunction $\wedge$), (disjunction $\vee$) and the unary `not`(); these take boolean operands, and do not have the short-circuit semantics of their C counterparts. While their operands cannot perform any memory access, their evaluation may result in an undefined behaviour.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{boolean} \quad \Gamma \vdash e_1 \Leftarrow \texttt{boolean} \quad \odot \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \odot e_2 \Rightarrow \texttt{boolean}} \qquad \frac{\Gamma \vdash e \Leftarrow \texttt{boolean}}{\Gamma \vdash \textbf{not}(e) \Rightarrow \texttt{boolean}}$$

$$\frac{e_1 \Downarrow \text{DEFINED}(v_1) \quad e_2 \Downarrow \text{DEFINED}(v_2) \quad v = \begin{cases} \texttt{True} & \text{if } v_1 = v_2 = \texttt{True} \\ \texttt{False} & \text{otherwise} \end{cases}}{e_1 \wedge e_2 \Downarrow \text{DEFINED}(v)}$$

$$\frac{e_1 \Downarrow \text{DEFINED}(v_1) \quad e_2 \Downarrow \text{DEFINED}(v_2) \quad v = \begin{cases} \texttt{False} & \text{if } v_1 = v_2 = \texttt{False} \\ \texttt{True} & \text{otherwise} \end{cases}}{e_1 \vee e_2 \Downarrow \text{DEFINED}(v)}$$

$$\frac{e \Downarrow \text{DEFINED}(\texttt{True})}{\textbf{not}(e) \Downarrow \text{DEFINED}(\texttt{False})} \qquad \frac{e \Downarrow \text{DEFINED}(\texttt{False})}{\textbf{not}(e) \Downarrow \text{DEFINED}(\texttt{True})}$$

Struct and union constructors The construction of struct and union values is done using the **struct**() and **union**() constructors. Their semantics is straightforward, the only point of interest being that (similar to the C array constructor) their operands are expressions with loaded types. This is to be able to account for specified struct values with unspecified members.

$$\frac{\text{members}(T) = (.z_1 : \tau_1, \ldots, .z_n : \tau_n) \quad \Gamma \vdash e_1 \Leftarrow \texttt{loaded}\ [\![\tau_1]\!] \quad \ldots \quad \Gamma \vdash e_n \Leftarrow \texttt{loaded}\ [\![\tau_n]\!]}{\Gamma \vdash \textbf{struct}[T](.z_1 = e_1, \ldots, .z_n = e_n) \Rightarrow \textbf{struct}\ T}$$

$$\frac{e_1 \Downarrow \text{DEFINED}(lv_1) \quad \ldots \quad e_n \Downarrow \text{DEFINED}(lv_n)}{\textbf{struct}[T](.z_1 = e_1, \ldots, .z_n = e_n) \Downarrow \text{DEFINED}(\textbf{struct}[T](\overline{.z_i = lv_i}^{i \in \{1,\ldots,n\}}))}$$

$$\frac{\text{members}(T) = (.z_1 : \tau_1, \ldots, .z_n : \tau_n) \quad i \in \{1, \ldots, n\} \quad \Gamma \vdash e \Leftarrow \texttt{loaded}\ [\![\tau_i]\!]}{\Gamma \vdash \textbf{union}[T](.z_i = e) \Rightarrow \textbf{union}\ T} \qquad \frac{e \Downarrow \text{DEFINED}(lv)}{\textbf{union}[T](.z = e) \Downarrow \text{DEFINED}(\textbf{union}[T](.z = lv))}$$

Struct and union member operator Similar to the '.' operator in C, a struct or union value can be destructed to access one of its members, using the **memberof**$[T.z]()$ operator. Most uses of the '.' operator in C, appear in lvalues. As a result, they are really instances of pointer arithmetic and are elaborated using the **member_shift**() we have introduced earlier. However they can also appear outside of lvalues (namely when applied to a function call returning a struct). In these instances, they are acting as destructors and we elaborate them with the **memberof** operator.

$$\frac{\text{members}(T) = (.z_1 : \tau_1, \ldots, .z_n : \tau_n) \quad i\ in \{1, \ldots, n\} \quad \Gamma \vdash e \Leftarrow (\texttt{struct}|\texttt{union})\ T}{\Gamma \vdash \textbf{memberof}[T.z_i](e) \Rightarrow \texttt{loaded}\ [\![\tau_i]\!]}$$

$$\frac{e \Downarrow \text{DEFINED}(\textbf{struct}[T](.z_1 = lv_1, \ldots, .z_n = lv_n)) \quad i\ in \{1, \ldots, n\}}{\textbf{memberof}[T.z_i](e) \Downarrow \text{DEFINED}(lv_i)}$$

For the dynamics where the operand is a union, when the member referred by the operator differs from the one held in the value of the operand, type punning needs to be performed. This requires knowledge of the representation of object types which is hidden by the memory interface. This case is therefore defined using one of the functions of the interface.

Function calls The typing and dynamics of calls to Core functions (which have a pure expression as their body) is straightforward. Note that Core is first-order, and function calls are therefore always fully applied.

$$\frac{\mathrm{funmap}(nm) = (x_1 : T_1, \ldots, x_n : T_n).\, e \qquad \Gamma \vdash e_1 \Leftarrow T_1 \quad \ldots \quad \Gamma \vdash e_n \Leftarrow T_n}{\Gamma \vdash nm(e_1, \ldots, e_n) \Rightarrow T}$$

$$\frac{\mathrm{funmap}(nm) = (x_1 : T_1, \ldots, x_n : T_n).\, e \qquad e_1 \Downarrow \mathrm{DEFINED}(v_1) \quad \ldots \quad e_n \Downarrow \mathrm{DEFINED}(v_n)}{nm(e_1, \ldots, e_n) \Downarrow \mathrm{DEFINED}\left(\overline{\{v_i/x_i\}e}^{\,i \in \{1,\ldots,n\}}\right)}$$

C function inspector To allow the elaboration function to express all the checks required by the ISO standard for the dynamics of C's function calls (in particular ones using function pointers), Core is equipped with a **`cfunction`**() operator, which, given a expression evaluating to a specified pointer to a C function, evaluates to a tuple containing: the return type of function; a list holding the types of its parameters; a boolean indicating whether the function is variadic; and a second boolean indicating whether the function had a prototype.

$$\frac{\Gamma \vdash e \Leftarrow \texttt{loaded pointer}}{\Gamma \vdash \textbf{\texttt{cfunction}}(e) \Rightarrow (\texttt{ctype}_{\texttt{wild}}, [\texttt{ctype}_{\texttt{wild}}], \texttt{boolean}, \texttt{boolean})}$$

$$\frac{\mathrm{cdecl}(f) = \tau_{\mathrm{ref}}(\tau_1, \ldots, \tau_n, b_{\mathrm{is\_variadic}}, b_{\mathrm{has\_proto}}) \qquad e \Downarrow \mathrm{DEFINED}(\texttt{Cfunction}(f))}{\textbf{\texttt{cfunction}}(e) \Downarrow \mathrm{DEFINED}((\tau_{\mathrm{ret}}, [\tau_1, \ldots, \tau_n], b_{\mathrm{is\_variadic}}, b_{\mathrm{has\_proto}}))}$$

Refining ctypes As we have seen, expressions with the type $\texttt{ctype}_{\texttt{wild}}$ can only be used as an operand to a restricted set of operators. It is however sometimes necessary to refine the type to a specified kind, which can be done safely by inspecting the C type value at runtime. For example, in the elaboration of C's function call expressions `E1(...)`, the function pointer value resulting from evaluating `E1` is applied to the **`cfunction`**() operator, which, among other things, results in a list of elements of type $\texttt{ctype}_{\texttt{wild}}$ holding the types of the parameters of the function designated by the function pointer. The remainder of the elaboration of C function calls then needs to first check that the type of each argument expression is compatible with the corresponding type in that list; and second, it needs to convert the value of each argument expression to the corresponding C type in that list. Because the Core type system does not feature polymorphism, we implement C's conversions as separate Core functions for each object type. The function used to elaborate conversion between two integer types has the signature:

$$\texttt{conv\_int} : \texttt{ctype}_{\texttt{integer}} \to \texttt{integer} \to \texttt{integer}$$

To express such refinements, we use the **`refine_ctype`**() operator which takes three `ctype` operands. The first and third operands must have a particular known kind oTy, while the second one can have the `wild` kind. The dynamics of **`refine_ctype`**() is to check whether

the values of the first two operands are compatible C types. If they are, the value of the operator is that of its second operator; otherwise, it is that of its third operand.

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{ctype}_{oTy} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{ctype}_{\texttt{wild}} \qquad \Gamma \vdash e_3 \Leftarrow \texttt{ctype}_{oTy}}{\Gamma \vdash \textbf{refine\_ctype}(e_1, e_2, e_3) \Leftarrow \texttt{ctype}_{oTy}}$$

$$\frac{e_1 \Downarrow \tau_1 \qquad e_2 \Downarrow \tau_2 \qquad \mathit{are\_compatible}\ \tau_1\ \tau_2 \qquad e_2 \Downarrow z}{\textbf{refine\_ctype}(e_1, e_2, e_3) \Downarrow z}$$

$$\frac{e_1 \Downarrow \tau_1 \qquad e_2 \Downarrow \tau_2 \qquad \neg \mathit{are\_compatible}\ \tau_1\ \tau_2 \qquad e_3 \Downarrow z}{\textbf{refine\_ctype}(e_1, e_2, e_3) \Downarrow z}$$

6.2 Effectful expressions

We now present the effectful part of Core, where interactions with the memory can be performed, and their various ordering constraints can be precisely stated, using a small calculus of operators. This part of the language also adds a goto-like control operator used in the elaboration of both C's iteration and jump statements, as well as concurrency. As a convention, we call subroutines whose body is a effectful expression *procedures*; these are what the definitions of C functions get elaborated into. We first present the effectful constructs and their typing rules, and then detail their dynamics in Section 6.2.1.

$$
\begin{array}{rcl}
E & ::= & \textbf{pure}(e) \\
& | & \textbf{memop}(\mathit{memop}, e_1, \ldots, e_n) \\
& | & \mathit{action} \\
& | & \textbf{neg}(\mathit{action}) \\
& | & \textbf{case}\ e\ \textbf{of}\ \overline{\mathit{pat}_i\ \texttt{=>}\ E_i}^{i \in I}\ \textbf{end} \\
& | & \textbf{let}\ \mathit{pat} = e\ \textbf{in}\ E \\
& | & \textbf{if}\ e\ \textbf{then}\ E_1\ \textbf{else}\ E_2 \\
& | & \textbf{ccall}_{\mathit{is\_variadic}}(e_f, e_1, \ldots, e_n) \\
& | & \textbf{pcall}(nm, e_1, \ldots, e_n) \\
& | & \textbf{unseq}(E_1, \ldots, E_n) \\
& | & \textbf{letweak}\ \mathit{pat} = E_1\ \textbf{in}\ E_2 \\
& | & \textbf{letstrong}\ \mathit{pat} = E_1\ \textbf{in}\ E_2 \\
& | & \textbf{bound}(E) \\
& | & \textbf{nd}(E_1, \ldots, E_n) \\
& | & \textbf{save}\ l(x_1\ \texttt{:=}\ e_1, \ldots, x_n\ \texttt{:=}\ e_n)\ \textbf{in}\ E \\
& | & \textbf{run}\ l(e_1, \ldots, e_n) \\
& | & \textbf{par}(E_1, \ldots, E_n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{memop} & ::= & \texttt{PtrEq} \mid \texttt{PtrNeq} \\
& | & \texttt{PtrLt} \mid \texttt{PtrGt} \mid \texttt{ptrLe} \mid \texttt{PtrGe} \\
& | & \texttt{Ptrdiff} \\
& | & \texttt{IntFromPtr} \mid \texttt{PtrFromInt} \\
& | & \texttt{PtrValidForDeref} \mid \texttt{PtrWellAligned} \\
& | & \texttt{PtrArrayShift} \\
& | & \texttt{Memcpy} \mid \texttt{Memcmp} \mid \texttt{Realloc} \\
& | & \texttt{Va\_start} \mid \texttt{Va\_copy} \\
& | & \texttt{Va\_arg} \mid \texttt{Va\_end}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{action} & ::= & \textbf{allocate\_object}(e_1, e_2) \\
& | & \textbf{allocate\_object\_readonly}(e_1, e_2, e_3) \\
& | & \textbf{allocate\_region}(e_1, e_2) \\
& | & \textbf{kill}_b(e) \\
& | & \textbf{store}_b(e_1, e_2, e_3, \mathit{mo}) \\
& | & \textbf{load}(e_1, e_2, \mathit{mo}) \\
& | & \textbf{seq\_rmw}_b(e_1, e_2, x.\, e_3)
\end{array}
$$

Figure 6.10: Grammar of Core's effectful expressions

The motivation for the division between Core's pure and effectful expressions is to allow, in Core programs produced by the elaboration from C, for a clear distinction between pure calculations (e.g. the elaboration of C's implicit type conversions, or the bounded arithmetic) and interaction with the memory. This is unrelated to C's division between expression and statements, and in fact both of these are elaborated to effectful

expressions. Pure expressions appear as operands of a few effectful operators, and in particular can be lifted into effectful expressions using the **pure**(e) operator. As a syntactic convention, we use the lowercase e for pure expressions, and the uppercase E for effectful expressions. The effectful expressions yield values ranging over the same types as pure expressions. The style of the type system is the same as before, with the two judgements adapted accordingly.

We first give a high-level presentation of the effectful constructs, along with their typing rules, and then follow with the formal presentation of their dynamics as a small-step operational semantics.

Simple control operators and procedure calls Mirroring their pure counterparts, there are effectful **let**, **if**, and **case** operators. The operand of the **let** and the controlling operand of **if** and **case** are pure expressions. As a result, these operators do not introduce any sequencing, and behave just like their pure versions. Likewise, the **pcall**() operator used for calling procedures takes pure expressions as arguments for the call. It behaves just like calls to pure functions. We omit the typing rules for these operators, as they are exactly the same as that of their pure counterparts.

Memory actions Interactions with the memory state are performed using memory *actions* and *operations*. Their dynamics is defined by the corresponding monadic functions from Chapter 5. Both memory actions and operations take pure expressions as operands, and are therefore atoms within the sequencing calculus.

Actions are used to allocate, deallocate, and access memory objects. Each use of an action is either "positive" (the default case) or "negative" (if the action appears in the syntax as the operand of the **neg**() operator). This polarity impacts on the sequencing constraint that is applied to the action as a result of its context within operators of the sequencing calculus. Intuitively, a Core action is negative when it elaborates what the C standard calls a "side-effect" (as opposed to value computations), that is, memory accesses which are not directly used for producing the value of a C expression. This is for example, the store performed by a postfix increment operator. We delay the discussion of the precise difference between the polarities to the presentation of the sequencing operators.

There are three different actions for allocating a new memory object. The first two, **allocate_object** and **allocate_object_readonly**, are used for the elaboration of the implicit allocation of objects resulting from the declaration of C identifiers. They both take as first operand an integer denoting the alignment constraint, and as their second operand the C type of the identifier (from which the memory object model will in particular derive the size of the allocation). Both actions yield a pointer value referring to the newly allocated object. In the case of **allocate_object**, the object is left uninitialised, whereas **allocate_object_readonly** initialises the object with the value of its third operand and then makes the object read-only. The read-only variant is typically used in the elaboration of C's string literals, which implicitly declare a character array on which an attempt to modify its value has undefined behaviour.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{integer} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{ctype}_{oTy}}{\Gamma \vdash \textbf{allocate\_object}(e_1, e_2) \Rightarrow \texttt{pointer}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{integer} \qquad \Gamma \vdash e_2 \Rightarrow \texttt{ctype}_{oTy} \qquad \Gamma \vdash e_3 \Leftarrow \texttt{loaded}\ oTy}{\Gamma \vdash \textbf{allocate\_object\_readonly}(e_1, e_2, e_3) \Rightarrow \texttt{pointer}}$$

The third action, **allocate_region**, is used in the elaboration of memory management functions (e.g. `malloc()`). Like the two previous actions, its first operand is an integer denoting the alignment constraint. It however differs in its second operand, which is a integer denoting the size of the allocation in bytes.

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{integer} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{integer}}{\Gamma \vdash \textbf{allocate\_region}(e_1, e_2) \Rightarrow \texttt{pointer}}$$

The **store** and **load** actions are used for the elaboration of accesses through C's lvalues (i.e. the read resulting from an lvalue conversion, and the write performed by an assignment operator). They both take as first operand a C type, denoting the type of the lvalue (thereby determining the footprint of the access). Their second operand is the pointer value used for the access. The **store** action takes as third operand the value being stored; it is of a loaded type containing the object type corresponding to the first operand. Both actions take a last non-expression operand denoting the C11 concurrency memory order of the access. Finally, the boolean flag b in the **store** action denotes whether the memory object is made read-only after the access; this is typically used in the elaboration of the initialisation of **const**-qualified objects. While we could have used the **allocate_object_readonly** action for most such initialisations, it is possible in C to initialise a **const**-qualified pointer to its own address, motivating this flag. For simplicity in the elaboration function, we use locking **store** actions for the elaboration of all **const**-qualified initialisations. A **store** action yields the unit value, while a **load** action yields the loaded value read from the memory object. While the details depends of the particular memory object model being used, the `Unspecified` case typically corresponds to reading from uninitialised memory or some padding bytes.

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{ctype}_{oTy} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{pointer} \qquad \Gamma \vdash e_3 \Leftarrow \texttt{loaded}\ oTy}{\Gamma \vdash \textbf{store}_b(e_1, e_2, e_3) \Rightarrow \texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{ctype}_{oTy} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{pointer}}{\Gamma \vdash \textbf{load}(e_1, e_2) \Rightarrow \texttt{loaded}\ oTy}$$

The **kill** action is used for ending the lifetime of a memory object. It takes as operands a pointer value and a boolean flag indicating whether the action is elaborating an implicit end of lifetime (from exiting a block statement), or a call to a memory management function such as `free()`.

$$\frac{\Gamma \vdash e \Leftarrow \texttt{pointer}}{\Gamma \vdash \textbf{kill}_b(e) \Rightarrow \texttt{unit}}$$

The **seq_rmw** action is used in the elaboration of the postfix increment/decrement operators, and compound assignments. It performs, in one atomic step, a read followed by a store. The first operand is the C type of the lvalue being elaborated; the second operand is the pointer value; and, the third operand is the value to be stored where the value that was read is bound to the variable x. The boolean b indicates whether the operator should yield the value that was read, or the one that is stored (i.e. the result of evaluating e_3).

Memory actions may implicitly result in an undefined behaviour (e.g. a store using an out of bound pointer, or past the lifetime of an object). While we designed Core with

the aim of making undefined behaviours visible in the syntax using **undef**(), the desire to keep the memory object model abstract in Core made that impossible for the undefined behaviours relating to memory.

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{ctype}_{oTy} \quad \Gamma \vdash e_2 \Leftarrow \texttt{pointer} \quad \Gamma \vdash e_3 \Leftarrow \texttt{loaded}\ oTy}{\Gamma \vdash \textbf{seq\_rmw}_b(e_1, e_2, x.\, e_3) \Rightarrow \texttt{unit}}$$

Memory operations In addition to actions, there is a separate category of operations involving pointer values, whose evaluation may, for some implementations of the memory interface, make use of the memory state, or have cases deemed undefined behaviour. For example, in the semi-abstract memory object models that we present in Chapter 9, the outcome of the pointer equality operator depends on the lifetime of memory objects, which are part of a ghost state. Sequencing therefore matters for these operations, despite the fact that they do not technically take part in the sequenced-before relation described by the ISO standard.

The operations are:

- the pointer equality and inequality: `PtrEq`, `PtrNe` : `pointer` → `pointer` → `boolean`; and the pointer relational operators `PtrLt`, `PtrGt`, `ptrLe`, `PtrGe`, with the same signatures.
- `Ptrdiff` : `ctype` → `pointer` → `pointer` → `integer`, which is used to elaborate C's subtraction between pointers. Like the C operator, the operator yields (as an integer value) the difference between the "offsets" of the two pointers, when interpreting them as both referring to an array object (whose element type is specified by the first operand).
- the conversion from pointer to integer (`IntFromPtr`), which takes two C types (the type referenced by the input pointer, and the desired integer type), a pointer value, and yields an integer value; and the opposite conversion (`PtrFromInt`).
- two tests, yielding a boolean value: `PtrValidForDeref`, which checks whether a pointer value is valid for use as the operand of C's unary `*` operator; and, `PtrWellAligned` which checks whether a pointer value satisfies the alignment constraint for a given C type.
- `PtrArrayShift`, an analogue of the pure operator **array_shift**. The ISO semantics for pointer arithmetic deems undefined behaviour the formation of (more than one past) out-of-bounds pointers. To model this, one need access to the memory state. By default, Cerberus does not model this undefined behaviour, and the elaboration therefore makes use of the pure operator (as we have seen in Chapter 2, it is not uncommon for C programming to rely on the construction of out-of-bounds pointer). The model can however be forced to follow the strict ISO semantics, in which case the elaboration switches to the effectful operation.
- additional operations used to implement C standard library features (`Memcpy`, `Memcmp`, `Realloc`, `Va_start`, `Va_copy`, `Va_arg`, `Va_end`), the details of which we omit.

Sequencing operators The ISO C11 standard introduced, as part of the concurrency model, the thread-wise *sequenced-before* relation over memory accesses, capturing the ordering constraint between these accesses. If an access A is sequenced-before an access B, then, for any allowed execution, A must be performed before B. Furthermore, if two accesses which are racing (i.e. they have overlapping footprints and at least one of them is a store) are not related by the sequenced-before relation, then there is an undefined behaviour.

To model this, Core is equipped with a small calculus of sequencing operators, where sequencing of an expression E_1 before an expression E_2 is introduced using a variant of the **let** operator:

$$\textbf{letstrong}\ pat = E_1\ \textbf{in}\ E_2$$

The dynamics of **letstrong** is to first fully reduce E_1 to a value v, and then reduce to $\{v/pat\}E_2$.

During the execution of both expressions, multiple memory actions or operations may be performed. From the point of the view of the sequenced-before relation, this operator makes any action or operation performed by E_1 sequenced-before any action or operation performed by E_2. The **letweak** variant of this operator does not force the execution of negative actions occurring in E_1 (and therefore does not make them sequenced-before the actions and operations in E_2).

The absence of sequencing between multiple expressions is expressed using the **unseq**() operator. Its dynamics allows any interleaving of the actions and operators performed by its operands. If two operands race, there is an undefined behaviour; otherwise, once all operand are reduced to values, the operator reduces itself to the tuple combining the values of its operands.

$$\frac{\Gamma \vdash E_1 \Rightarrow T_1 \quad \ldots \quad \Gamma \vdash E_n \Rightarrow T_n}{\Gamma \vdash \textbf{unseq}(E_1, \ldots, E_n) \Rightarrow (T_1, \ldots, T_n)}$$

The **bound**() operator is used to mark the boundary of the elaboration of a C expression. This operator is used to reduce the non-determinism of the operational semantics. When exploring exhaustively a Core expression, the non-determinism of **unseq**() operators and the detection of unsequenced races is restricted to the outermost **bound**() operator. This greatly reduces unnecessary non-determinism in the language.

C function calls The **ccall**() operator is used for the elaboration of C function calls. Because the function call operator in C takes an arbitrary expression as its function designator, one that can for example read from memory a function pointer, the corresponding operator in Core takes as first operand a pure expression yielding a loaded function pointer, as opposed to a name. The remaining operands are the parameters of the function. Apart from this, the dynamics of the operator is identical to that of the call operator for procedures. In particular, C's implicit promotion of the parameters, and the allocation, initialisation and deallocation of temporary objects for the parameters are not part of the semantics of the operator. These aspects are performed explicitly in the Core expressions produced by the elaboration function surrounding any use of the **ccall**() operator. The type system requires the first parameter to be a boolean and the remaining parameters to all be pointer values, with a special case for calls to a variadic function (as indicated by the *is_variadic* flag), where the last parameter must be a list of pairs of a C

type and a pointer. This is because the elaboration function from C models the allocation of the temporary objects from the caller's side. The initial boolean argument is also a result of the elaboration function, which makes use of it to indicate whether the value returned by the function is used by the caller. This is necessary to model the potential undefined behaviour occurring when the execution of a non-**void** function ends without a **return** statement and its value is used (we show the details in Section 7.2). The pointers passed as argument to the Core procedure modelling a C function refer to the temporary objects created for the call.

The dynamics of the **ccall**() operator takes care of the *indeterminate sequencing* of C's function call: because the body of the called procedure is executed in a new continuation pushed onto the execution stack, calls are atomic from the point of view of operators with which the call is unsequenced. This mechanism also prevents the dynamics of the **unseq**() operator from improperly signalling unsequenced accesses to the same memory footprint performed through a C function call as undefined (these are instead defined, and introduce observable non-determinism in the language).

$$\frac{\begin{array}{c}\Gamma \vdash e_f \Rightarrow \texttt{loaded pointer} \\ \Gamma \vdash e_1 \Leftarrow \texttt{boolean} \\ \Gamma \vdash e_2 \Leftarrow \texttt{pointer} \quad \ldots \quad \Gamma \vdash e_{n-1} \Leftarrow \texttt{pointer} \\ \Gamma \vdash e_n \Leftarrow \begin{cases} [(\texttt{ctype}, \texttt{pointer})] & \text{if } \mathit{is\_variadic} \\ [\texttt{pointer}] & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash \textbf{ccall}_{\mathit{is\_variadic}}(e_f, e_1, \ldots, e_n) \Rightarrow [\![\tau]\!]}$$

Labelled continuations All of C's iteration and jumping statements are elaborated into a goto-like operator in Core. Continuations are labelled using the **save** operator, which is similar to labelled statements in C. Jumps are performed using the **run** operator, which is similar to the **goto** statement. The former takes the form:

$$\textbf{save}\ l(x_1\ \texttt{:=}\ e_1, \ldots, x_n\ \texttt{:=}\ e_n)\ \textbf{in}\ E$$

which declares the label l in scope of the whole body of the containing procedure. This label refers to the continuation resulting from composing the context of the operator with its body E. Additionally, the operator binds the variables x_i in E. Each of these variables is associated with a pure expression whose evaluation yields its *default* value. The dynamics of the operator is to reduce to E where the default values have been substituted for their variables x_i.

The latter takes the form:

$$\textbf{run}\ l(e_1, \ldots, e_n)$$

Its dynamics is similar to a call: the current continuation is replaced with the one associated to the label l (remember this is the composition of the context of the corresponding **save** operator with its body E). The values of the pure expression operands are used to substitute the variables declared by the corresponding **save** operator.

The dynamics of these operators do not deal with C's implicit allocation and deallocation of objects from block-scoped variables. They are once again dealt with by the elaboration function. Their parameter operands are, however, used by the elaboration to forward pointer values referring to objects of such variables.

Because the continuation of a **save** operator itself is part of the labeled continuation it defines, without restriction, it would be possible using a forward **run** operator to jump

over the binder of a variable. This would lead to a runtime error. We prevent this by imposing the following syntactic restriction: for any label, for any **run** to that label, the difference between the set of variables in scope of the body associated E and the set of variables in scope of the **run** operator shall be disjoint from the set of free variables in the continuation of the **save** operator.

Nondeterminism operator The **nd**() operator non-deterministically reduces into one of its operands. This constructor is used to model the non-determinism introduced by our treatment of C's unspecified values when they appear in the controlling expression of a statement.

$$\frac{\exists i \in \{1, \ldots, n\} \quad \Gamma \vdash E_i \Leftarrow T \quad \forall j \neq i. \Gamma \vdash E_j \Rightarrow T}{\Gamma \vdash \mathbf{nd}(E_1, \ldots, E_n) \Rightarrow T}$$

Thread creation operator New threads are created using the $\mathbf{par}(E_1, \ldots, E_n)$ operator. For each of its operands, it starts a new thread with empty stack and E_i as its initial continuation. The current thread is blocked until all the new threads have terminated, at which point the operator yields the tuple made of the values returned by all the created threads. This operator is used to elaborate `cppmem`-like thread creations [Bat+11]; they are not meant to model more general constructs, such as POSIX threads.

$$\frac{\Gamma \vdash E_1 \Rightarrow T_1 \quad \ldots \quad \Gamma \vdash E_n \Rightarrow T_n}{\Gamma \vdash \mathbf{par}(E_1, \ldots, E_n) \Rightarrow (T_1, \ldots, T_n)}$$

6.2.1 Operational semantics

We now present the dynamics of effectful expressions, for which we use a small-step operational semantics with reduction contexts. We define three reduction relations:

- *effectless reductions*, which do not require interaction with the memory state, or any change to the current continuation. Because some of the reductions may involve the evaluation of a pure expression, they can result in an undefined behaviour. We reuse the notations from the evaluation of pure expressions for denoting successful reductions, and those resulting in an undefined behaviour:

$$E \rightsquigarrow \text{DEFINED}(E') \qquad \text{and} \qquad E \rightsquigarrow \text{UNDEF}$$

- *thread-local reductions*, that may interact with the memory (e.g. by performing a memory action), or change the current continuation (e.g. jumping to a label, or calling a procedure). These are defined using a relation over tuples $\langle \sigma, C[E], \kappa \rangle$ consisting of: σ, the memory state; the effectful expression which is the current continuation of the procedure being reduced (which we refer to as the "arena" from now on); and κ, the call stack of the thread being reduced. To deal with reductions changing the current continuation and some of the rules dealing with the sequencing calculus, we write the arena in the form $C[E]$: a decomposition of a reduction context C applied to an effectful expression E. We give the grammars for call stacks and reduction contexts in Section 6.2.1.3.

  Some transitions are labelled by the kind of interaction (here noted α) with memory state that they perform. We discuss the detail of these labels in Section 6.2.1.3,

when presenting the reduction rules for memory actions and operations. As for the previous relation, there are two possible outcomes: successful reduction to a new tuple, or an undefined behaviour:

$$\langle \sigma, C[E], \kappa \rangle \xrightarrow{\alpha} \text{DEFINED}(\langle \sigma', C'[E'], \kappa' \rangle) \qquad \text{and} \qquad \langle \sigma, C[E], \kappa \rangle \xrightarrow{\alpha} \text{UNDEF}$$

- *thread reductions*, for the spawning and ending of threads. These relate pairs of the memory state and thread pool:

$$\langle \sigma, T \rangle \xrightarrow{\alpha} \text{DEFINED}(\langle \sigma', T' \rangle) \qquad \text{and} \qquad \langle \sigma, T \rangle \xrightarrow{\alpha} \text{UNDEF}$$

  A thread pool T is a map from thread IDs to thread configuration tuples $\langle t_{\text{opt}}, C[E], \kappa \rangle$, where: t_{opt} is either `None` (for the startup thread), or `Some`(t) to indicate that this thread was created by the thread with ID t; $C[E]$ is the expression that remains to be executed by the thread; and κ is its stack.

  Finally, there is a special judgement for the end of execution of the program:

$$\langle \sigma, T \rangle \rightarrow \text{DONE}(v)$$

6.2.1.1 Footprint annotations

The main point of interest in the dynamics of effectful expressions is the ordering of memory actions, and in particular the detection of *unsequenced races*: the occurrence of a memory write and another memory access to an overlapping memory footprint which are not related by C's sequenced-before relation. As discussed earlier, when elaborating a C expression or statement, the sequencing calculus of Core is used to explicitly express the sequenced-before relation. The lack of any sequencing constraint between two overlapping memory actions in Core corresponds to an unsequenced race in C.

To detect these races, thread-local reductions performing memory accesses progressively add to the arena annotations keeping track of the memory *footprint* which has been touched by the program execution so far. We extend the syntax of effectful expressions to account for these annotations, with the following two variants:

$$\begin{array}{rcl} E & ::= & \cdots \\ & | & {}^{A}E \\ & | & \textbf{exclude}[\mathbf{n}](action) \end{array}$$

The first one denotes that, in the process of reducing the arena to E, the memory footprints contained in annotation A were accessed. The second will perform the memory action it contains, with the additional semantics that the footprint of that action is to be denoted by the natural number $\mathbf{n}$. This identifier allows annotations to refer to the memory action (i.e. to express sequencing with respect to it).

Annotations are sets whose elements are either of the following two variants:

$$\textbf{neg}(\mathbf{n}, \{\mathbf{n}_1, \ldots, \mathbf{n}_n\}, \text{fp}) \mid \textbf{pos}(\{\mathbf{n}_1, \ldots, \mathbf{n}_n\}, \text{fp})$$

The first variant is used to keep track of the footprint resulting from performing negative memory actions, while the second is used for positive memory actions. Within an annotation A, the annotation elements need not be of the same variant. They only differ in the natural number taken as first operand by the negative variant (the remaining operands in the negative variant are the same as in the positive one). The natural number plays the

same role as the one found in the **exclude**() operator: it uniquely identifies the annotation element, allowing it to be referenced by other annotations. The scope of these identifiers is not limited to the containing set, but instead extends to the whole execution. The next operand of both variants holds a set of natural numbers, which denotes that this annotation element is not to be regarded as racing with the annotation elements identified by the elements of the set. The last operand holds the memory footprint of the action remembered by the annotation element. Its type is declared by the memory interface, and is kept abstract. The only available operator over footprints is overlapping(), which takes two footprints, and returns a boolean denoting whether its operands interfere with one another. The load and store functions from the memory interface are the only way to produce a footprint, and they are used in the dynamics of Core's memory actions.

6.2.1.2 Effectless reductions

Most of the reduction rules for the control operators are within the scope of the effectless reduction relation. They are unsurprising, simply involving the evaluation of a controlling pure expression:

$$\text{PURE}\frac{e \Downarrow \text{DEFINED}(v) \qquad \neg(\text{is\_value } e)}{\textbf{pure}(e) \rightsquigarrow \text{DEFINED}(\textbf{pure}(v))}$$

$$\text{IF-TRUE}\frac{e \Downarrow \text{DEFINED}(\texttt{true})}{\textbf{if } e \textbf{ then } E_1 \textbf{ else } E_2 \rightsquigarrow \text{DEFINED}(E_1)}$$

$$\text{IF-FALSE}\frac{e \Downarrow \text{DEFINED}(\texttt{false})}{\textbf{if } e \textbf{ then } E_1 \textbf{ else } E_2 \rightsquigarrow \text{DEFINED}(E_2)}$$

$$\text{LET}\frac{e \Downarrow \text{DEFINED}(v)}{\textbf{let } \text{pat} = e \textbf{ in } E \rightsquigarrow \text{DEFINED}(\{v/\text{pat}\}E)}$$

$$\text{CASE}\frac{e \Downarrow \text{DEFINED}(v) \qquad \mathit{match\_pattern}(\overline{\text{pat}}^{\,i}, v) = \texttt{Some}(k)}{\textbf{case } e \textbf{ of } \overline{\text{pat}_i \texttt{ => } E_i}^{\,i} \textbf{ end} \rightsquigarrow \text{DEFINED}(\{v/\text{pat}_k\}E_k)}$$

The function *match_pattern*() is the same as the one used in the evaluation of the pure variant of the **case** operator. Here again, the program reduction will get stuck if the function fails to match the controlling value to any pattern.

The trivial cases of the two sequencing operators, **letstrong** and **letweak**, are also dealt with as effectless reductions: when their first operand is a pure expression, nothing needs to be sequenced, and the reduction simply follows that of a normal **let** binder.

$$\text{LETS-PURE}\frac{}{\textbf{letstrong } \mathit{pat} = \textbf{pure}(v) \textbf{ in } E \rightsquigarrow \text{DEFINED}(\{v/\mathit{pat}\}E)}$$

$$\text{LETW-PURE}\frac{}{\textbf{letweak } \mathit{pat} = \textbf{pure}(v) \textbf{ in } E \rightsquigarrow \text{DEFINED}(\{v/\mathit{pat}\}E)}$$

An **unseq** operator that is only applied to non-annotated fully evaluated pure expressions effectlessly reduces to a tuple:

$$\text{UNSEQ-PURE}\ \frac{}{\mathbf{unseq}(\mathbf{pure}(v_1), \ldots, \mathbf{pure}(v_n)) \rightsquigarrow \text{DEFINED}((v_1, \ldots, v_n))}$$

The traversal of a **save** operator by the program execution (as opposed to when the execution arrives from a jump using a **run** operator) simply substitutes its variables to their default values, with no change to the current continuation:

$$\text{SAVE}\ \frac{\forall i.\, e_i \Downarrow \text{DEFINED}(v_i)}{\mathbf{save}\ l(x_1\ \mathbf{:=}\ e_1, \ldots, x_n\ \mathbf{:=}\ e_n)\ \mathbf{in}\ E \rightsquigarrow \text{DEFINED}\left(\overline{\{v_i/x_i\}E}^{\,i\in\{1,\ldots,n\}}\right)}$$

The reduction of the nondeterministic choice operator is also straightforwardly expressed using the present relation:

$$\text{ND}\ \frac{i \in \{1, \ldots, n\}}{\mathbf{nd}(E_1, \ldots, E_n) \rightsquigarrow \text{DEFINED}(E_i)}$$

A few of the rules we have presented involve the evaluation of pure expressions, and therefore have counterparts for the cases where an undefined behaviour is raised by one of these evaluations:

$$\text{PURE-UNDEF}\ \frac{e \Downarrow \text{UNDEF}}{\mathbf{pure}(e) \rightsquigarrow \text{UNDEF}}$$

$$\text{IF-UNDEF}\ \frac{e \Downarrow \text{UNDEF}}{\mathbf{if}\ e\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \rightsquigarrow \text{UNDEF}} \qquad \text{LET-UNDEF}\ \frac{e \Downarrow \text{UNDEF}}{\mathbf{let}\ pat = e\ \mathbf{in}\ E \rightsquigarrow \text{UNDEF}}$$

$$\text{CASE-UNDEF}\ \frac{e \Downarrow \text{UNDEF}}{\mathbf{case}\ e\ \mathbf{of}\ \overline{pat_i\ \mathbf{=>}\ E_i}^{\,i}\ \mathbf{end} \rightsquigarrow \text{UNDEF}}$$

$$\text{SAVE-UNDEF}\ \frac{\exists i \in \{1, \ldots, n\}.\quad e_i \Downarrow \text{UNDEF}}{\mathbf{save}\ l(x_1\ \mathbf{:=}\ e_1, \ldots, x_n\ \mathbf{:=}\ e_n)\ \mathbf{in}\ E \rightsquigarrow \text{UNDEF}}$$

Thread-local reductions will progressively add annotations, as they perform memory actions. Annotations added to an already annotated expression are combined as the union of their elements.

$$\text{ANNOTS}\ \frac{}{{}^{A_1}\left({}^{A_2}E\right) \rightsquigarrow \text{DEFINED}({}^{A_1 \cup A_2}E)}$$

Annotations introduce a variant to the trivial cases of the two sequencing operators, where the reduced form of the first operand is annotated. In this case the annotation is preserved after the substitution of the bound variable inside the second operand.

$$\text{LETW-ANNOT}\ \frac{}{\mathbf{letweak}\ pat = {}^{A}\mathbf{pure}(v)\ \mathbf{in}\ E_2 \rightsquigarrow \text{DEFINED}({}^{A}\{v/pat\}E_2)}$$

$$\text{LETS-ANNOT}\ \frac{}{\mathbf{letstrong}\ pat = {}^{A}\mathbf{pure}(v)\ \mathbf{in}\ E_2 \rightsquigarrow \text{DEFINED}({}^{A}\{v/pat\}E_2)}$$

Finally, when reducing an **unseq** operator with at least two annotated operands, a check is performed to detect whether an unsequenced race has occurred. We use the notation ${}^{A^?}\mathbf{pure}(e)$ to denote that the expression may or may not have an annotation.

$$\text{UNSEQ-RACE}\frac{\exists i \neq j.\, \text{do\_race}(A_i, A_j)}{\mathbf{unseq}({}^{A_1^?}\mathbf{pure}(v_1), \ldots, {}^{A_n^?}\mathbf{pure}(v_n)) \rightsquigarrow \text{UNDEF}}$$

where do_race(A_1, A_2) if and only if there exists $a_1 \in A_1$, and $a_2 \in A_2$ with any of the following conditions satisfied:

- $a_1 = \mathbf{neg}(\mathbf{n}_1, \mathbf{ns}_1, \text{fp}_1)$ and $a_2 = \mathbf{neg}(\mathbf{n}_2, \mathbf{ns}_2, \text{fp}_2)$, with $n_1 \notin \mathbf{ns}_2$, $n_2 \notin \mathbf{ns}_1$, and overlapping$(\text{fp}_1, \text{fp}_2)$;
- $a_1 = \mathbf{neg}(\mathbf{n}_1, \_, \text{fp}_1)$ and $a_2 = \mathbf{pos}(\mathbf{ns}_2, \text{fp}_2)$, with $n_1 \notin \mathbf{ns}_2$, and overlapping$(\text{fp}_1, \text{fp}_2)$;
- $a_1 = \mathbf{pos}(\mathbf{ns}_1, \text{fp}_1)$ and $a_2 = \mathbf{neg}(\mathbf{n}_2, \_, \text{fp}_2)$, with $n_2 \notin \mathbf{ns}_1$, and overlapping$(\text{fp}_1, \text{fp}_2)$;
- $a_1 = \mathbf{pos}(\_, \text{fp}_1)$ and $a_2 = \mathbf{pos}(\_, \text{fp}_2)$, with overlapping$(\text{fp}_1, \text{fp}_2)$.

The overlapping() function is part of the memory interface presented in Chapter 5. In the absence of any overlapping unsequenced footprints, the annotations are simply combined.

$$\text{UNSEQ-ANNOT}\frac{\forall i \neq j.\, \neg\text{do\_race}(A_i, A_j)}{\mathbf{unseq}({}^{A_1}\mathbf{pure}(v_1), \ldots, {}^{A_n}\mathbf{pure}(v_n)) \rightsquigarrow \text{DEFINED}({}^{A_1 \cdots \cup \cdots A_n}(v_1, \ldots, v_n))}$$

6.2.1.3 Thread-local reductions

We now move on to thread-local reductions, which either perform a memory action, or change the current continuation. As mentioned earlier, for reductions, the arena is written as the application of a context C to an expression in focus E, where contexts are defined as follows:

$$\begin{array}{rcl} C & ::= & \bullet \\ & | & \mathbf{unseq}(E_1, \ldots, E_{k-1}, C, E_{k+1}, \ldots, E_n) \\ & | & \mathbf{letweak}\ pat = C\ \mathbf{in}\ E_2 \\ & | & \mathbf{letstrong}\ pat = C\ \mathbf{in}\ E_2 \\ & | & \mathbf{bound}(C) \\ & | & {}^{A}C \end{array}$$

The first variant denotes a hole, while the other variants correspond to the steering of the program execution. Except for the last variant annotating a context, only the sequencing calculus appears as variants. As we have shown while presenting the previous relation, the reduction rules of the other constructs of the effectful fragments (e.g. the **if** operator) only involve evaluating pure expressions, with no interaction with the memory state, and therefore are not concerned with sequencing.

The call stack of the thread is defined as a list of contexts, corresponding to the continuations of the calling procedures whose execution will resume once the current procedure is done:

$$\kappa ::= \varepsilon \mid C \cdot \kappa$$

Reductions from the first relation are lifted under contexts in the usual way:

$$\text{TAU}\ \frac{E \rightsquigarrow \text{DEFINED}(E')}{\langle \sigma, C[E], \kappa s\rangle \xrightarrow{\tau} \text{DEFINED}(\langle \sigma, C[E'], \kappa\rangle)} \qquad \text{UNDEF}\ \frac{E \rightsquigarrow \text{UNDEF}}{\langle \sigma, C[E], \kappa\rangle \xrightarrow{\tau} \text{UNDEF}}$$

Note that because of the context variant corresponding to the **unseq**() operator, a Core expression may have multiple decompositions into pairs of a context and an inner expression. The TAU rule therefore expresses the interleaving non-determinism for the unsequencing operator.

Performing memory actions Annotations are added when a memory action is performed. Their actual dynamics is abstracted by the memory interface, and the reductions make use of the corresponding functions it declares. We use the following notation: $\sigma \xrightarrow{act} (\sigma', v, fp)$, to denote that performing the action act from memory state σ yields the value v with the new state σ' and that the memory footprint fp was touched. We focus on the load and store actions in this presentation; the allocating and deallocation actions behave similarly to the load action and are omitted.

Performing a positive action is straightforward: their pure operands are evaluated; and the memory interface is used to obtain the state update, value and footprint resulting from the actions; and finally the value is annotated with footprint is placed in the reduction context where the action was. For example, store actions get reduced as follows:

$$\text{POS-STORE}\ \frac{e_1 \Downarrow \text{DEFINED}(\tau) \quad e_2 \Downarrow \text{DEFINED}(ptr) \quad e_3 \Downarrow \text{DEFINED}(v) \quad \sigma \xrightarrow{\texttt{store}(\tau,ptr,v)} \text{DEFINED}\ (\sigma', \texttt{Unit}, fp)}{\langle \sigma, C[\texttt{store}(e_1, e_2, e_3)], \kappa\rangle \xrightarrow{\texttt{store}(\tau,ptr,v)} \text{DEFINED}(\langle \sigma', C[^{\texttt{pos}(\{\},fp)}\texttt{pure}(\texttt{Unit})], \kappa\rangle)}$$

and load actions as follows:

$$\text{POS-LOAD}\ \frac{e_1 \Downarrow \text{DEFINED}(\tau) \quad e_2 \Downarrow \text{DEFINED}(ptr) \quad \sigma \xrightarrow{\texttt{load}(\tau,ptr)} \text{DEFINED}\ (\sigma', v, fp)}{\langle \sigma, C[\texttt{load}(e_1, e_2)], \kappa\rangle \xrightarrow{v=\texttt{load}(\tau,ptr)} \text{DEFINED}(\langle \sigma', C[^{\texttt{pos}(\{\},fp)}\texttt{pure}(v)], \kappa\rangle)}$$

The memory object model may decide that performing a memory action results in an undefined behaviour (e.g. when the pointer value is out of bounds of any live object). As a result, the function used to perform actions is partial, and we need additional rules to lift the undefined behaviour. For example, an invalid load action reduces as follows:

$$\text{MERR-POS-LOAD}\ \frac{e_1 \Downarrow \text{DEFINED}(\tau) \quad e_2 \Downarrow \text{DEFINED}(ptr) \quad \sigma \xrightarrow{\texttt{load}(\tau,ptr)} \text{UNDEF}}{\langle \sigma, C[\texttt{load}(e_1, e_2)], \kappa\rangle \xrightarrow{\tau} \text{UNDEF}}$$

Additionally the evaluation of one or more of the pure operands of an action may result in an undefined behaviour. For these cases, we also need rules lifting the undefined behaviour. For example, still with a load action:

$$\text{UNDEF-POS-LOAD}\ \frac{\exists i \in \{1, 2\}. \quad e_i \Downarrow \text{UNDEF}}{\langle \sigma, C[\texttt{load}(e_1, e_2)], \kappa\rangle \xrightarrow{\tau} \text{UNDEF}}$$

The dynamics of negative actions is more subtle, as they are not sequenced by the **letweak** operator. The POS_LOAD, POS_STORE rules would (in conjunction with the effectless reductions of the weak sequencing operator) add too many annotations. Intuitively, within an arena being executed, we need to hoist negative actions up to the innermost strong sequencing operator present in their context, and make them unsequenced with anything else. In C all *full expressions* (i.e. the ones not part of a larger expression) have their evaluation separated by a sequence point. As a result, the nondeterminism that may arise from mixing function calls and unsequenced evaluation only needs to be explored within the boundary of a full expression. This is the motivation for the **bound**() operator, which is used by the elaboration function to mark, in the generated Core, the boundary of the elaboration of C's full expressions. The hoisting of negative actions therefore only needs to be performed inside (and up to) a **bound**() operator.

Depending on whether it also contains an inner **letstrong** operator, any context C containing a **bound**() operator can be uniquely rewritten either as:

$$C \equiv C_1[\mathbf{bound}(C_2[\mathbf{letstrong}\ pat = C_3\ \mathbf{in}\ E_2])]$$

where neither C_2 nor C_3 contain a **bound**() operator, and C_3 does not contain a **letstrong** operator, or as:

$$C \equiv C_1[\mathbf{bound}(C_3)]$$

where C_3 does not contain a **bound**() operator, and neither C_1 nor C_3 contain a **letstrong** operator.

Having an arena where a negative action is in the focus of the execution means having either form of C_3 directly applied to that negative action: $C_3[\mathbf{neg}(act)]$. Allowing for the action to be delayed with respect to any memory actions within C_3 is done by simply rewriting the arena as[2]:

$$C_3[\mathbf{unseq}(\mathbf{neg}(act), C_3[\mathbf{Unit}])$$

This transformation alone is however not sufficient, as we need to remember that the negative action is not racing with the actions that were already sequenced. The record of these actions is kept within the context C_3 in the form of annotations. We therefore update it as follows: a fresh $\mathbf{n}$ is picked, and, for any annotation context ${}^{A}C$ within C_3, that number is added to the exclusion set of each annotation element (respectively, the second and first component of a negative and positive annotation element). We write this update $mark\_exclusion(C_3, \mathbf{n})$. Additionally, we bind the fresh number to the hoisted negative action by turning $\mathbf{neg}(act)$ into $\mathbf{exclude}[\mathbf{n}](act)$.

[2]This relies on the fact that the value of a negative action is always discarded and replaced by unit.

$$\text{DEFINED}\ \frac{\begin{array}{c} C \equiv C_1[\textbf{bound}(C_2[\textbf{letstrong}\ pat = C_3\ \textbf{in}\ E_2])] \\ \textbf{bound} \notin C_2 \qquad \textbf{letstrong} \notin C_3 \\ \text{fresh}\ \mathbf{n} \qquad mark\_exclusion(C_3, \mathbf{n}) = C_3' \end{array}}{\begin{array}{c} \langle \sigma, C[\textbf{neg}(act)], \kappa \rangle \\ \xrightarrow{\tau} \\ \left\langle \sigma, C_1 \left[\textbf{bound} \left(C_2 \left[\begin{array}{l} \textbf{letstrong}\ (\_, pat) = \\ \quad \textbf{unseq}(\textbf{exclude}[\mathbf{n}](act), C_3'[\texttt{Unit}])\ \textbf{in} \\ E_2 \end{array} \right] \right) \right], \kappa \right\rangle \end{array}}$$

$$\text{DELAY-NO-SSEQ}\ \frac{\begin{array}{c} C \equiv C_1[\textbf{bound}(C_2)] \quad \textbf{bound} \notin C_2 \quad \textbf{letstrong} \notin C_2 \\ \text{fresh}\ \mathbf{n} \qquad mark\_exclusion(C_2, \mathbf{n}) = C_2' \end{array}}{\begin{array}{c} \langle \sigma, C[\textbf{neg}(act)], \kappa \rangle \\ \xrightarrow{\tau} \\ \text{DEFINED}(\langle \sigma, C_1[\textbf{bound}(\textbf{unseq}(\textbf{exclude}[\mathbf{n}](act), C_2'[\texttt{Unit}]))], \kappa \rangle) \end{array}}$$

Figure 6.11: Reductions for delaying negative actions

The reduction rules of the **exclude**() operator are then similar to that of positive memory actions, except that they place the negative variant of annotation elements in the reduced arena. For example, the reduction rule for a store action is:

$$\text{EXCLUDE-STORE}\ \frac{\begin{array}{c} e_1 \Downarrow \text{DEFINED}(\tau) \quad e_2 \Downarrow \text{DEFINED}(ptr) \quad e_3 \Downarrow \text{DEFINED}(v) \\ \sigma \xrightarrow{\textbf{store}(\tau, ptr, v)} (\sigma', \texttt{Unit}, fp) \end{array}}{\begin{array}{c} \langle \sigma, C[\textbf{exclude}[\mathbf{n}](act)], \kappa \rangle \\ \xrightarrow{\textbf{store}(\tau, ptr, v)} \\ \text{DEFINED}(\langle \sigma', C[\textbf{pure}(\texttt{Unit})], \kappa \rangle) \end{array}}$$

Memory operations The reduction of memory operations is more straightforward, as they have no polarity (i.e. they behave like positive memory actions).

$$\text{MEMOP}\ \frac{\begin{array}{c} i \in \{1, \ldots, n\} \quad e_i \Downarrow \text{DEFINED}(v_i) \\ \sigma \xrightarrow{memop(v_1, \ldots, v_n)} \text{DEFINED}\ (\sigma', v) \end{array}}{\langle \sigma, C[\textbf{memop}(memop, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{DEFINED}(\langle \sigma', C[\textbf{pure}(v)], \kappa \rangle)}$$

$$\text{MERR-MEMOP}\ \frac{\begin{array}{c} i \in \{1, \ldots, n\} \quad e_i \Downarrow \text{DEFINED}(v_i) \\ \sigma \xrightarrow{memop(v_1, \ldots, v_n)} \text{UNDEF} \end{array}}{\langle \sigma, C[\textbf{memop}(memop, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{UNDEF}}$$

$$\text{UNDEF-MEMOP}\ \frac{\exists i \in \{1, 2\}. \quad e_i \Downarrow \text{UNDEF}}{\langle \sigma, C[\textbf{memop}(memop, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{UNDEF}}$$

Removal of annotations The **bound** operator limits the scope of annotations. By construction of the elaboration function, this corresponds to the boundary of a C expression as statement.

$$\text{REMOVE-BOUND}\ \frac{}{\left\langle \sigma, C[\mathbf{bound}(^{A?}\mathbf{pure}(v))], \kappa \right\rangle \xrightarrow{\tau} \text{DEFINED}(\langle \sigma, C[\mathbf{pure}(v)], \kappa \rangle)}$$

When reaching the end of the execution of a procedure, any annotation left on the value is discarded:

$$\text{REMOVE-ANNOT}\ \frac{}{\left\langle \sigma, {}^{A}\mathbf{pure}(v), \kappa \right\rangle \xrightarrow{\tau} \text{DEFINED}(\langle \sigma, \mathbf{pure}(v), \kappa \rangle)}$$

Procedure and C calls A call to a procedure evaluates its pure arguments, and fetches the definition of the procedure (which consist of an effectful body and the declaration of parameters bound in the body). The context C of the call is pushed to the top of the call stack, and the body of the procedure (after substitution of the parameters) is placed in the arena with an empty context.

$$\text{PCALL}\ \frac{\begin{array}{c}\text{funmap}(nm) = (x_1 : T_1, \ldots, x_n : T_n).\ E \\ e_1 \Downarrow \text{DEFINED}(v_1) \quad \ldots \quad e_n \Downarrow \text{DEFINED}(v_n)\end{array}}{\langle \sigma, C[\mathbf{pcall}(nm, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{DEFINED}\left(\left\langle \sigma, \overline{\{v_i/x_i\}E}^{i \in \{1,\ldots,n\}}, C \cdot \kappa \right\rangle\right)}$$

A call using **ccall**() (used for procedures elaborating C functions) is similar, only differing in how the name nm of procedure that needs to be called is found. The first operand e_f evaluates to a specified value holding a pointer value that should refer to a function. Using the function case_funsym_opt() provided by the memory interface, the pointer value is inspected to find the procedure name. If this is successful, the remainder of the dynamics is like that of **pcall**().

$$\text{CCALL}\ \frac{\begin{array}{c}e_f \Downarrow \text{DEFINED}(\mathsf{Specified}(ptr_f)) \\ \text{case\_funsym\_opt}(\sigma, ptr_f) = \mathsf{Some}(nm) \quad \text{funmap}(nm) = (x_1 : T_1, \ldots, x_n : T_n).\ E \\ e_1 \Downarrow \text{DEFINED}(v_1) \quad \ldots \quad e_n \Downarrow \text{DEFINED}(v_n)\end{array}}{\langle \sigma, C[\mathbf{ccall}_{is\_variadic}(e_f, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{DEFINED}\left(\left\langle \sigma, \overline{\{v_i/x_i\}E}^{i \in \{1,\ldots,n\}}, C \cdot \kappa \right\rangle\right)}$$

The form of e_f introduces two means of failure, which lead to the indication of a undefined behaviour: it may evaluate to an unspecified value, or it may evaluate to a specified pointer value that does not designate a function (either because it is null, or a pointer to an object).

$$\text{CCALL-UNSPEC}\ \frac{e_f \Downarrow \text{DEFINED}(\mathsf{Unspecified}(\tau))}{\langle \sigma, C[\mathbf{ccall}_{is\_variadic}(e_f, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{UNDEF}}$$

$$\text{CCALL-INVALID}\ \frac{\begin{array}{c}e_f \Downarrow \text{DEFINED}(\mathsf{Specified}(ptr_f)) \\ \text{case\_funsym\_opt}(\sigma, ptr_f) = \mathsf{None}\end{array}}{\langle \sigma, C[\mathbf{ccall}_{is\_variadic}(e_f, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{UNDEF}}$$

$$\text{UNDEF-CCALL}\ \frac{e_f \Downarrow \text{UNDEF} \ \vee \ \exists i \in \{1, \ldots, n\}. \quad e_i \Downarrow \text{UNDEF}}{\langle \sigma, C[\mathbf{ccall}_{is\_variadic}(e_f, e_1, \ldots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{UNDEF}}$$

When the arena is reduced to a **pure**() operator holding a value, the execution of the current procedure has ended. The return to the caller is done by removing the current continuation of the caller from the top of the stack C, and applying it to **pure**(v) in the arena.

$$\text{RETURN}\ \frac{}{\langle \sigma, \mathbf{pure}(v), C \cdot \kappa \rangle \xrightarrow{\tau} \text{DEFINED}(\langle \sigma, C[\mathbf{pure}(v)], \kappa \rangle)}$$

Labelled continuations A jump to a labelled continuation with the **run** operator has reductions similar to that of a procedure call: its pure arguments are evaluated; the definition of the labelled continuation is fetched from a read-only environment. Unlike procedure calls, the definition contains both the new continuation C_l and the body of the labelled of continuation E_l; this reduction leaves the stack untouched.

$$\text{RUN}\ \frac{e_1 \Downarrow \text{DEFINED}(v_1) \quad \dots \quad e_n \Downarrow \text{DEFINED}(v_n) \qquad \mathit{labelmap}(l) = (x_1 \dots x_n).\, C_l[E_l]}{\langle \sigma, C[\mathbf{run}\ l(e_1, \dots, e_n)], \kappa \rangle \xrightarrow{\tau} \text{DEFINED}\left(\left\langle \sigma, C_l\left[\overline{\{v_i/x_i\}E_l}^{\,i \in \{1,\dots,n\}}\right], \kappa \right\rangle\right)}$$

6.2.1.4 Thread reductions

The creation of threads using the **par**() operator selects fresh thread-ids, and, for each thread, adds to the thread pool an initial configuration consisting of a reference to the parent thread, an execution arena made from its corresponding operands E_i, and an empty stack. The arena of the parent thread is updated with an expression that blocks its execution until all its children threads have completed their execution. We reuse the **unseq**() operator here, because the **wait**() operator is blocking, no unsequencing is actually introduced. The operator will instead deterministically reduce to a tuple once the termination of all the children threads has removed all the **wait**() (as shown in the next reduction rule).

$$\text{SPAWN}\ \frac{\forall i \in \{1, \dots, n\}.\, t_i \notin \mathrm{dom}(T) \qquad \mathit{distinct}(t_1, \dots, t_n)}{\begin{array}{c}\langle \sigma, T\,[t_{\text{parent}} \mapsto \langle t_{\text{opt}}, C[\mathbf{par}(E_1, \dots, E_n)], \kappa \rangle] \rangle \\ \xrightarrow{\tau} \\ \text{DEFINED}\left(\left\langle \sigma, T\left[\begin{array}{lcl} t_{\text{parent}} & \mapsto & \langle t_{\text{opt}}, C[\mathbf{unseq}(\mathbf{wait}(t_1), \dots, \mathbf{wait}(t_n))], \kappa \rangle \\ t_1 & \mapsto & \langle \mathsf{Some}(t_{\text{parent}}), E_1, \varepsilon \rangle \\ & \dots & \\ t_n & \mapsto & \langle \mathsf{Some}(t_{\text{parent}}), E_n, \varepsilon \rangle \end{array}\right] \right\rangle\right)\end{array}}$$

When the arena of a thread reduces to a value, its execution ends, and its configuration is removed from the thread pool. If this is not the startup thread (and therefore has a reference to a parent in its configuration), the **wait**() operator that was created when the thread was spawned in the arena of the parent thread is replaced by the value.

$$\text{THREAD-DONE}\ \frac{}{\begin{array}{c}\left\langle \sigma, T\left[\begin{array}{lcl} t_{\text{done}} & \mapsto & \langle \mathsf{Some}(t_{\text{parent}}), \mathbf{pure}(v), \varepsilon \rangle \\ t_{\text{parent}} & \mapsto & \langle t_{\text{opt}}, E, \kappa \rangle \end{array}\right] \right\rangle \\ \xrightarrow{\tau} \\ \text{DEFINED}\left(\left\langle \sigma, T\left[\begin{array}{lcl} t_{\text{done}} & \mapsto & \\ t_{\text{parent}} & \mapsto & \langle t_{\text{opt}}, \{\mathbf{pure}(v)/\mathbf{wait}(t_{\text{done}})\}E, \kappa \rangle \end{array}\right] \right\rangle\right)\end{array}}$$

The execution of the Core program has ended when the arena of the startup thread is reduced to a **pure**() operator holding a value, and the stack is empty:

$$\text{PROGRAM-DONE}\ \frac{}{\langle \sigma, T\,[t \mapsto \langle \mathsf{None}, \mathbf{pure}(v), \varepsilon \rangle] \rangle \xrightarrow{\tau} \text{DONE}(v)}$$

Chapter 7

The elaboration function

In this chapter, we discuss the structure of the elaboration function and its components. We show snippets of the definitions, edited for presentation, as figures with a grey background. The definitions are written in Lem; an automatically typeset version of the full Lem module defining the elaboration is given in Appendix B. We focus on a few representative clauses, as the whole definition is quite long. There is quite a bit of detail made explicit by the elaboration, but this is simply following what the ISO standard says is the semantics of C. To ease reading of the different language levels, we use the following typesetting convention: control operators in Lem are in black uppercase, e.g. **LET**; calls in Lem to operators of the monad used by the elaboration functions are in red, e.g. FRESH_SYMBOL(), and so are calls to auxiliary Lem functions also using the monad; calls in Lem to pure functions are in black small caps (e.g. IS_SIGNED()); and constructors of the Core expressions being constructed by the elaboration are in blue, e.g. **pure**(). We write $[\![\cdot]\!]$ for recursive calls to the elaboration of Ail expressions and statements (mapped to Core effectful expressions) and C types (mapped to Core object types). Example of Core snippets are given in yellow boxes.

7.1 Elaboration of Ail statements and expressions

Most of the work is done by the two functions that elaborate expressions and statements. They are both total functions, defined by structural induction over the Ail AST. They use a state monad for the following purposes:

- Generating fresh symbols used when building Core expressions. This is performed with a call to FRESH_SYMBOL(T), where T is the Core type to be associated to the symbol.
- In the elaboration of expressions, the C string literals are collected. This is necessary to model the static storage duration of memory objects associated with these. The top-level elaboration function creates, for each literal, a Core global of type `pointer`, whose initialisation expression performs the allocation and initialisation of the corresponding character array memory object.
- In the elaboration of statements, when passing through a block statement, we keep track of any block-scoped declaration[1], so that when elaborating the inside of the

[1] In the Ail representation, the declarations of a block are attached to the block AST node, as opposed to statements within the block (as is the case in the C AST).

block, we can know what the set of "visible" C identifiers is at any point (along with their types). This information is used in the elaboration of label declarations and `goto` statements, to properly model the implicit allocation and deallocation of the associated block-scoped objects.

Additionally, these functions take an environment holding the following components:

- If we are translating the initialiser of a global object, there is a Core symbol to be used for the elaboration of lvalues referring to the global itself.
- If we are translating a statement or an expression part of the body of a variadic function, there are two Core symbols: the first refers to the last named function parameter; the second refer to the additional trailing parameter in the Core procedure elaborating the variadic function (which is a list holding the unnamed parameters). These two symbols are used in the elaboration of the `va_start()` macro, which, in the grammar of Ail, is an expression constructor.
- A collection of Core symbols referring to various auxiliary Core functions and procedures (which form a small standard library).
- The member definitions of the struct and union types in the translation unit being elaborated.
- When elaborating a statement, the symbol of the enclosing function; its return type; whether it is specified as `_Noreturn`; and the Core label symbol that will be used as the targets of the Core jumps elaborating `return` statements.

These components are left unchanged throughout the execution of the elaboration functions.

Finally, there are the following components, which do evolve throughout the traversal of the Ail AST:

- When elaborating a statement within a `switch`, there are Core label symbols used to refer to the target of the Core jumps elaborating any `default` statement, and the targets of the Core jumps corresponding to each `case` statement.
- The set of Ail identifiers currently in scope (and their types). This set evolves when going through an Ail block statement, and is used for the elaboration of the implicit lifetime of objects associated to block-scoped identifiers.

7.1.1 Example: elaboration of the division operator

We now go through the definition of the clause of the elaboration of Ail's division operator, which operates over integer and floating types. In the Lem definition in Appendix B, this corresponds to the auxiliary function `translate_div_mod_operator` (which also implements the modulo operator, as its elaboration is nearly identical to the division), which is called in the recursion over the Ail expression AST for the two corresponding constructors. An Ail division expression is of the form `E1 / E2`, and we write τ_1 and τ_2 for the types of the operands, and τ_{res} for the result type of the division itself.

The first six lines create fresh Core symbols that will be used in the construction of binders.

```
[| E1^τ1 /^τres E2^τ2 |] ≜
  1  e1 := FRESH_SYMBOL(loaded [|τ1|]);
  2  e2 := FRESH_SYMBOL(loaded [|τ2|]);
  3  obj1 := FRESH_SYMBOL([|τ1|]);
  4  obj2 := FRESH_SYMBOL([|τ2|]);
  5  conv1 := FRESH_SYMBOL([|τ1|]);
  6  conv2 := FRESH_SYMBOL([|τ2|]);
```

The first two, e_1 and e_2, will be used to refer to the values of the operands of the division (which may be unspecified); obj_1 and obj_2 will be used for the concrete values when they are specified; and $conv_1$ and $conv_2$ will be used to refer to these concrete values after the usual arithmetic conversions have been applied. Because the representation of the Core AST has type annotations at every binder, the symbol generation function takes a Core type as an argument.

Next are four Lem **LET**s, constructing small Core pure expressions which are used several times. In particular, the second one constructs two pure expressions performing the usual arithmetic conversions on the results of evaluating the operands.

```
  7  LET ZERO = { 0    IS_INTEGER(τret)
               { 0.0  IS_FLOATING(τret)   IN
  8  LET (PROMOTED1, PROMOTED2) =
  9    USUAL_ARITHMETIC_CONVERSION(τ1, τ2, obj1, obj2) IN
 10  LET UB = undef(<<UB045a_division_by_zero>>) IN
 11  LET DIV = conv1 / conv2 IN
```

These are typically either of the form:

```
conv_int('τ', obj1)
```

which is a call to an auxiliary Core function converting obj_1 to be within the range of the C type τ (we showed its definition at the end of Section 4.2); or a conversion between an integer and a floating value (when the operands have mixed types). For example, if `E1` has type **`signed int`**, and `E2` has type **`float`**, we have a floating division, and the left operand is converted as:

```
Fvfromint('float', obj1)
```

The type to which the operands are converted is the *common real type* of the two, as calculated by following the rules given by the ISO standard in (§6.3.1.8). As the typing of C expressions is static, this is implemented in Lem and resolved at the time of the elaboration.

We now reach the construction of the Core expression giving the dynamics of the division operator:

```
 12  letweak (e1, e2) = unseq([| E1 |], [| E2 |]) in
 13  pure(
 14    case (e1, e2) of
```

At line 12, we recursively elaborate the operand expressions, making them unsequenced, and binding their values to e_1 and e_2. Here we make use of the weak sequencing operator, because, as for most binary operators, the ISO standard does not specify a sequence point

here. From line 13 until the end, we construct a pure expression; the only sequencing operation is the previous **letweak**; and memory operators may only appear within the elaboration of one of the operands. At line 14, we case split on whether the evaluation of the operands yield concrete values.

```
15      | (Unspecified(_), _) =>
16  IF IS_SIGNED_INTEGER(τ_res) THEN
17          undef(<<UB036_exceptional_condition>>)
18  ELSE
19          Unspecified(τ_res)
20  FI
```

If the first operand is unspecified, depending on whether the result type of the division is signed, we either indicate an undefined behaviour (this is the daemonic instance of a possible integer overflow), or we simply make the division evaluate to the unspecified value of the result type. Control from the pattern matching at line 14 happens at Core's runtime, whereas the branching from line 16 (and all other instances of **IF**) occurs in the evaluation of the elaboration.

```
21      | (_, Unspecified(_)) =>
22          UB
```

If the second operand is unspecified, we indicate the undefined behaviour signalling a division by zero (this is again the daemonic instance), using the pure expression constructed at line 10.

```
23      | (Specified(obj_1), Specified(obj_2)) =>
24          let conv_1 = PROMOTED_1 in
25          let conv_2 = PROMOTED_2 in
26          if conv_2 = ZERO then
27            UB
28          else if is_representable(τ_res, DIV) then
29            DIV
30          else
31            undef(<<UB045c_quotient_not_representable>>)
32    end
33  )
```

Finally, from line 23, we deal with the case where both operands have concrete values. Lines 26 and 27 model the undefined behaviour for when the right operand (after conversion) is equal to zero. Then, at line 28, we check whether the result of the division is in range of the result type. If not, line 31 indicates the appropriate undefined behaviour.

7.1.2 Example: elaboration of equality expressions

Let us now consider the elaboration of the equality operator ==, which, as we will later see, is indirectly used for the elaboration of other Ail constructs (in particular statements with controlling expressions). Like in the previous example, the Lem development dealing with this case is in an auxiliary function: `translate_equality_operator` (which, in its unedited form given in Appendix B, also implements the !=, as its dynamics is again very similar to ==). As before, we write τ_1 and τ_2 for the types of the operands, and τ_{res} for the result type (which, by the C typing rules, is always **signed int**).

There are four cases, depending on the type and form of the operands, which, as for the implicit conversions in the division, we can resolve at the time of the elaboration. The first two are symmetric variants, where one operand is a null pointer constant (i.e. the constant `0` cast to a pointer type, or the macro `NULL`), and the other operand is an arbitrary pointer expression.

```
⟦ E1^τ1 ==^τres E2^τ2 ⟧ ≜
 1  IF IS_NULL_POINTER_CONSTANT(E1) ∧ IS_POINTER(τ2) THEN
 2  z := FRESH_SYMBOL(boolean);
 3  e2 := FRESH_SYMBOL(loaded pointer);
 4  obj := FRESH_SYMBOL(pointer);
 5  letweak e2 = ⟦ E2 ⟧ in
 6  case e of
 7    | Specified(obj) =>
 8        letweak z = memop(PtrEq, obj, Null(τref)) in
 9        pure(if z then Specified(1) else Specified(0))
10    | _ =>
11        pure(undef(<<UB_unspecified>>))
12  end
13  ELSEIF IS_NULL_POINTER_CONSTANT(E2) ∧ IS_POINTER(τ1) THEN
      ... (symmetric of the previous)
```

As in the previous example, we first create some fresh Core symbols: z will be used to refer to the result of the equality test, which in Core yields a boolean; e_2 will refer to the result of evaluating the (non-constant) operand, which as before may yield an unspecified value; obj will be used for the case where that operand has a concrete value. At line 5, the non-constant operand $E2$ is recursively elaborated and bound to e_2 with the weak sequencing operator. We then destruct the value. If it is concrete, we use the Core pointer equality operator against a null pointer value of the referenced type inside τ_2. As we have seen in Chapter 6, this operator is not pure, because we want to allow some memory object models to access their ghost state in their implementation (for example in the provenance-based memory object models we later present, the equality between two pointers depends on the state of abstract memory objects at the time of the test). As a result, the boolean result is also bound with the weak sequencing operator. Finally, at line 9, we turn the Core boolean into an integer, as one would expect from C dynamics. Lines 10 and 11 model the case where the value of the non-constant operand is unspecified, for which we indicate an undefined behaviour. We omit the symmetric case.

In the other two cases, we know that the operands either both have arithmetic types, or both have pointer types. We start similarly with the creation of fresh symbols for the values of the operands, and construction of conversions (used in the arithmetic case). The constructed Core starts by recursively elaborating the operands, leaving them unsequenced, and binding their result using the weak sequencing operator.

```
14  ELSE (the operands both have arithmetic or pointer types)
15  e1 := FRESH_SYMBOL(loaded ⟦τ1⟧);
16  e2 := FRESH_SYMBOL(loaded ⟦τ2⟧);
17  obj1 := FRESH_SYMBOL(⟦τ1⟧);
18  obj2 := FRESH_SYMBOL(⟦τ2⟧);
19  LET (PROMOTED1, PROMOTED2) =
20    USUAL_ARITHMETIC_CONVERSION(τ1, τ2, obj1, obj2) IN
21  letweak (e1, e2) = unseq(⟦ E1 ⟧, ⟦ E2 ⟧) in
```

In the case where both operands have arithmetic types, the remainder of the elaboration is pure, as it simply involves an integer or floating equality test (depending of the implicit conversions resulting from USUAL_ARITHMETIC_OPERATOR). Its boolean result is, as in the previous cases, turned into an integer, as expected. Note that in the case that either operand evaluates to an unspecified value, we simply produce the unspecified value for the result type (i.e. **signed int**).

```
22 IF IS_ARITHMETIC(τ1) ∧ IS_ARITHMETIC(τ2) THEN
23 pure(
24   case (e1, e2) of
25     | (Specified(obj1), Specified(obj2)) =>
26         if PROMOTED1 = PROMOTED2 then Specified(1) else Specified(0)
27     | _ =>
28         Unspecified(τres)
29   end
30 )
```

In the last case, where operands have pointer types, the elaboration is similar to the first case, but where the null pointer constant has been replaced by the concrete value of the second non-constant operand.

```
31 ELSE (the operands both have pointer types)
32 z := FRESH_SYMBOL(boolean);
33 case (e1, e2) of
34   | (Specified(obj1), Specified(obj2)) =>
35       letweak z = memop(PtrEq, obj1, obj2) in
36       pure(if z then Specified(1) else Specified(0))
37   | _ =>
38       pure(undef(<<UB_unspecified>>))
39 end
40 FI
41 FI
```

Factorisation of comparison against `0` In several instances, the ISO standard makes use of the phrase "*compares equal to `0`*" (and its negation) when specifying the dynamics of some expression operators and statements, namely: the logical boolean operators, the conditional operator, and statements involving a controlling expression. This phrase refers to the dynamics of the == operator, which we have just looked at. To avoid duplicating the formalisation of this operator, we define the Lem function MKTESTEXPR(op, E) where op is either == or !=, E is an Ail expression, and which returns the following Ail expression:

$$\begin{cases} \texttt{E op 0} & \text{if E is an integer expression} \\ \texttt{E op 0.0} & \text{if E is a floating expression} \\ \texttt{E op NULL} & \text{if it is a pointer} \end{cases}$$

To elaborate the logical boolean operators, we then rewrite their Ail AST as follows:

- ⟦ E1 && E2 ⟧ ≜ ⟦ MKTESTEXPR(==, E1) ? 0 : MKTESTEXPR(!=, E2) ⟧
- ⟦ E1 || E2 ⟧ ≜ ⟦ MKTESTEXPR(==, E1) ? MKTESTEXPR(!=, E2) : 0 ⟧

Note that the left-to-right evaluation order and sequence point specified by the standard in (§6.5.13#4) and (§6.5.14#4) are inherited from the dynamics of the conditional operator. In the clauses defining the elaboration the conditional operator and statements with controlling expressions, we apply MKTESTEXPR() to the controlling expression before recursively elaborating it.

7.1.3 Example: elaboration of `while` statements

Let us now consider the elaboration of **while** statements, which will illustrate the last point of the previous paragraph, and the access to the set of visible identifiers from the environment. Remember that, as described in Section 4.5, by the time the Ail AST reaches the elaboration function, any **continue** and **break** statements have been turned into **goto** statements, with the necessary label declarations placed around all looping statements. The present clause therefore only needs to deal with the elaboration of the controlling expression, and the looping jump.

```
⟦ while (E) S ⟧ ≜
 1  do_loop := FRESH_SYMBOL(boolean);
 2  test := FRESH_SYMBOL(loaded integer);
 3  obj := FRESH_SYMBOL(integer);
 4  {x_1, ..., x_n} := GET_VISIBLE_SYMS;
 5  l := FRESH_LABEL;
 6  save l(x_1 := x_1, ..., x_n := x_n) in
 7    letstrong test = ⟦ MKTESTEXPR(==, E) ⟧ in
 8    letstrong do_loop =
 9      case test of
10        | Specified(obj) =>
11            if obj = 1 then True else False
12        | Unspecified(_) =>
13            nd(True, False)
14      end in
15    if do_loop then
16      letstrong Unit = ⟦ S ⟧ in
17      run l(x_1, ..., x_n)
18    else
19      pure(Unit)
```

The first three lines create Core symbols as before: *do_loop* will refer to the boolean result of comparing the value of the controlling expression with zero; *test* will refer to the potentially unspecified value of the controlling expression; and *obj* will refer to its concrete value in the case it is specified. At line 4, we query the environment for the set of Ail identifiers in scope from our current position in the AST. In the Lem development, this uses the same datatype as for Core symbols. There is therefore no need to create further symbols. At line 6, we use a **save** constructor to declare the label l pointing to the body of the loop. Its arguments use the visible symbols both in their binders and in their default pure expressions. At line 7, we recursively elaborate the controlling expression with the call to the MKTESTEXPR function. Note that this is strongly sequenced using the **letstrong** operator, as the ISO standard specifies a sequence point between the evaluation of the controlling expression and execution of the body of the loop. Because this is really the elaboration of an Ail == operator, the result has type `loaded integer`. From lines 8

to 14, we therefore convert it into a boolean. As a result of our modelling of unspecified values, if the controlling expression evaluates to an unspecified value, our dynamics is to nondeterministically loop or not, which is modelled using Core's nondeterministic choice operator, **nd**(). This is the reason why these lines are effectful, and we therefore bind *do_loop* with a sequencing operator. At line 15, we finally have the branching depending on the result of the modified controlling expression using an effectful Core **if**. In the taken branch, we have the recursive elaboration of the body statement, strongly sequenced before a jump back the beginning, whereas the else branch yields a unit value.

7.1.4 Example: elaboration of function calls

As a last example, we look at the elaboration of function calls. In the simple case, there are no arguments:

$[\![\ \mathtt{E}_f{}^{\tau_{ret}(*)()}\ \mathtt{()}\ ]\!] \triangleq$

```
1  funptr := FRESH_SYMBOL(loaded pointer);
2  ret_type := FRESH_SYMBOL(ctype);
3  param_types := FRESH_SYMBOL([ctype]);
4  letstrong funptr = [[ E_f ]] in
5  let (ret_type, param_types, _, _) = Cfunction(funptr_) in
6  if params_length(param_types) = 0 then
7    if are_compatible([[τ_ret]], ret_type) then
8      ccall([[τ_ret]], funptr, IS_USED)
9    else
10     pure(undef(<<UB041_function_not_compatible>>))
11 else
12   pure(undef(<<UB038_number_of_args>>))
```

Three Core symbols are created: *funptr* which is bound at line 4 to the elaboration of E_f, the function designator of the call, and will hold the function pointer value resulting from the evaluation of the designator; and, *ret_type* and *param_types* which are respectively bound to the return C type and the list of parameter C types of function designated by the function pointer. The last two are obtained by applying at line 5 the `Cfunction`() operator on the function pointer. It is possible in C to write a function designator such that the function it designates has a signature that is not compatible with the function type advertised by designator. Calls using such designators are undefined behaviour, which is modelled explicitly in the Core elaboration: at line 6, we check that the designated function indeed does not expect any parameters, and at line 7 we check that the return type of the designated function and of the one advertised by the designator are compatible. If both tests succeed, the actual function call is performed using Core's **ccall**() operator at line 8. Otherwise, the corresponding undefined behaviour is indicated. Note that the elaboration of E_f at line 4 is strongly sequenced before the rest of the elaboration, in particular the **ccall**() operator. This models the sequence point specified by the ISO standard. The Core call takes a single argument IS_USED, which is a boolean constant indicating whether the value returned by the function is used by the caller (e.g. if the function call appears as the operand of an arithmetic operation). This constant is constructed by the elaboration function based on the syntactic context in which the Ail function call being elaborated appears.

Calls with arguments In the general case, the picture is more complicated: adding arguments introduces temporary objects that need to be allocated, initialised to the values of the argument (after some potential conversions), and then deallocated. The occurrence of variadic arguments adds further complications. We will go through the definition of the clause of the elaboration for a call with argument $\mathtt{E}_f{}^{\tau_{funptr}}(\mathtt{E}_1^{\tau_{\mathrm{arg}_1}}, \ldots, \mathtt{E}_n^{\tau_{\mathrm{arg}_n}})$, breaking it down to smaller pieces. We write τ_{ret} for the return type of the referenced function type in τ_{funptr}, and $\tau_1, \ldots, \tau_N$ for the types of its parameters. Note that in the case of a call to a variadic function, the number N of parameters in the function type may be smaller than the number n of arguments. First, a few Core symbols are created in addition to the ones used in the simple case:

$$
\begin{array}{lll}
[\![\, \mathtt{E}_f{}^{\tau_{ret}(*)(\tau_1,\ldots,\tau_N)}(\mathtt{E}_1^{\tau_{\mathrm{arg}_1}}, \ldots, \mathtt{E}_n^{\tau_{\mathrm{arg}_n}}) \,]\!] \triangleq & & \\
1 \quad \mathit{funptr} & := \mathrm{FRESH\_SYMBOL}(\mathtt{loaded\ pointer}); & \\
2 \quad \mathit{ret\_type} & := \mathrm{FRESH\_SYMBOL}(\mathtt{ctype}); & \\
3 \quad \mathit{param\_types} & := \mathrm{FRESH\_SYMBOL}([\mathtt{ctype}]); & \\
4 \quad \mathit{is\_variadic} & := \mathrm{FRESH\_SYMBOL}(\mathtt{boolean}); & \\
5 \quad \mathit{call\_ret} & := \mathrm{FRESH\_SYMBOL}\left(\begin{cases} \mathtt{unit} & \text{if } \tau_{\mathrm{ret}} = \mathtt{void} \\ \mathtt{loaded}\ [\![\tau_{\mathrm{ret}}]\!] & \text{otherwise} \end{cases}\right); & \\
6 \quad \mathit{arg\_ptr}_i & := \mathrm{FRESH\_SYMBOL}(\mathtt{loaded\ pointer}); & \forall i \in \{1, \ldots, n\} \\
7 \quad \mathit{arg}_i & := \mathrm{FRESH\_SYMBOL}(\mathtt{loaded}\ [\![\tau_{\mathrm{arg}_i}]\!]); & \forall i \in \{1, \ldots, n\}
\end{array}
$$

The symbol *is_variadic* will be bound to a boolean returned by the `Cfunction`() operator indicating whether the function designated by the designator is variadic. The symbol *call_ret*, will be bound to the result of the **`ccall`**() operator. For each argument, the symbol $\mathit{arg\_ptr}_i$ will be bound to a pointer value referring to the temporary memory object allocated to pass that argument to the function, and the symbol arg_i will be bound to the result of evaluating the argument expression E_i.

We then build the Core expression. First, the designator and argument expressions are recursively elaborated[2]at lines 11 and 12, and left unsequenced with one another, but are strongly sequenced with the rest of elaboration of the call:

```
 8  letstrong (funptr, (ret_type, param_types, is_variadic, has_proto),
 9        arg_1, ..., arg_n) =
10    unseq(
11      letstrong funptr = [| E_f |] in
12      pure(funptr, Cfunction(funptr_)), [| E_1 |], ..., [| E_n |]) in
```

Then, there is a conditional in the elaboration function on whether the type of the function referenced by the type of E_f is variadic. We first look at the variadic case:

[2]For the sake of presentation, we omit here the potential conversion present when a pointer argument is applied where a **`_Bool`** is expected.

```
13  IF IS_PTR_TO_VARIADIC(τ_funptr) THEN
14  if not(params_length(param_types) <= ⟦ n ⟧) then
15    pure(undef(<<UB038_number_of_args>>))
16  else if not(is_variadic) ∨ not(are_compatible(⟦τ_ret⟧, ret_type)) then
17    pure(undef(<<UB041_function_not_compatible>>))
18  else
19    letstrong arg_ptr_1 = CREATE_ARG_1 in
...
20    letstrong arg_ptr_N = CREATE_ARG_N in
21    letstrong arg_ptr_{N+1} = CREATE_VARG_{N+1} in
...
22    letstrong arg_ptr_n = CREATE_VARG_n in
23    letstrong call_ret = ccall(⟦τ_funptr⟧, funptr, IS_USED, arg_ptr_1, ..., arg_ptr_N,
24        [(τ_conv_{N+1}, arg_ptr_{N+1}), ..., (τ_conv_n, arg_ptr_n)]) in
25    letstrong (Unit, ..., Unit) =
26      unseq(kill_static(arg_ptr_1), ..., kill_static(arg_ptr_n)) in
27    pure(call_ret)
```

Lines 13 to 17 perform the compatibility check similar to what we had in the elaboration of function calls with no arguments. Here, however, instead of requiring an empty list, the length of *param_types* (which tells us the number of parameters expected by the function pointer being used by the call) is expected to be smaller or equal to the number of arguments. The function pointer is also required to refer to a variadic function. From lines 19 and 22, the arg_ptr_i variables are bound to Core expressions denoted by Lem variables CREATE_ARG_i and CREATE_VARG_i. These model the allocation and initialisation of the temporary memory objects used to pass the arguments to the function, and their result is the pointer values to these objects. For the first N arguments, for which the function type of the function designator declares a type, we use the Core expressions CREATE_ARG_i; whereas for the remaining arguments, which corresponding to the unnamed arguments of a variadic call, we use the Core expressions CREATE_VARG_i. Details of their construction is given in Figures 7.1 and 7.2. At line 23, the actual call finally occurs using the **ccall**() operator. Because we are dealing with a call to a variadic function (which is evident from the first operand which holds the type of the function pointer), the last operand is a list of pair of C type and pointers to the temporary objects holding the unnamed arguments. Note that the C types used here are the result of performing "the default argument promotions" on the types of the arguments τ_{arg_i}. At line 26, the temporary objects are deallocated, and finally the result of the **ccall**() is returned at line 27.

The case where the type of the referenced function is not variadic is very similar, only differing in the following points: at line 31 in the compatibility check, the negation of the *is_variadic* boolean variable is dropped; on lines 34-35, the allocation and initialisation of the temporary memory objects only involve the non-variadic CREATE_ARG_i blocks; and at line 36, the **ccall**() operator is only passed the IS_USED boolean constant and the "normal" parameters (the list of pairs of C types and pointers from the variadic case is absent).

```
28  ELSE
29  if not(params_length(params_types) = ⟦ n ⟧) then
30    pure(undef(<<UB038_number_of_args>>))
31  else if is_variadic ∨ not(are_compatible(⟦τret⟧, ret_type)) then
32    pure(undef(<<UB041_function_not_compatible>>))
33  else
34    letstrong arg_ptr_1 = CREATE_ARG_1 in
...
35    letstrong arg_ptr_N = CREATE_ARG_N in
36    letstrong call_ret = ccall(⟦τfunptr⟧, funptr, IS_USED, arg_ptr_1, ..., arg_ptr_n) in
37    letstrong (Unit, ..., Unit) =
38      unseq(kill_static(arg_ptr_1), ..., kill_static(arg_ptr_n)) in
39    pure(call_ret)
40  FI
```

```
LET CREATE_ARG_i =
1   param_ty := FRESH_SYMBOL(ctype);
2   arg_ptr := FRESH_SYMBOL(pointer);
3   let param_ty = params_nth(params_types, ⟦ i − 1 ⟧) in
4   if not(are_compatible(⟦τi⟧, param_ty)) then
5     pure(undef(<<UB041_function_not_compatible>>))
6   else
7   LET CONV_VALUE =
8     IF IS_ARITHMETIC(τi) THEN
9       IF IS_INTEGER(τarg_i) THEN
10        conv_loaded_int(param_ty, arg_i)
11      ELSE
12        loaded_ivfromfloat(param_ty, arg_i)
13    ELSE IF IS_FLOATING(τi) THEN
14      IF IS_INTEGER(τarg_i) THEN
15        loaded_fvfromint(param_ty, arg_i)
16      ELSE
17        arg_i
18    ELSE IF IS_POINTER(τi) ∧ IS_NULL_POINTER_CONSTANT(Ei) THEN
19      NULL(τi)
20    ELSE
21      arg_i IN
22  LET MO = { seq_cst   IS_ATOMIC(τi)
           { na        otherwise
23  letweak arg_ptr = create(Ivalignof(param_ty), param_ty) in
24  letweak Unit = store(param_ty, arg_ptr, CONV_VALUE, MO) in
25  pure(arg_ptr)
```

Figure 7.1: Elaboration fragment for an argument $\mathtt{E}_i^{\tau_{\mathrm{arg}_i}}$ with an associated parameter type τ_i

```
LET CREATE_VARG_i =
  1  arg_ptr := FRESH_SYMBOL(pointer);
  2  LET (CONV_TY, CONV_VALUE) =
  3    IF IS_ARITHMETIC(τ_arg_i) THEN
  4      LET PROM_TY = PROMOTE(τ_arg_i) IN
  5      ([[ PROM_TY ]], conv_loaded_int([[ PROM_TY ]], arg_i))
  6    ELSE IF IS_FLOATING(τ_arg_i) THEN
  7      ('double', arg_i)
  8    ELSE
  9      ([[ τ_arg_i ]], arg_i) IN
 10  letweak arg_ptr = create(Ivalignof(CONV_TY), CONV_TY) in
 11  letweak Unit = store(CONV_TY, arg_ptr, CONV_VALUE, na) in
 12  pure(arg_ptr)
```

Figure 7.2: Elaboration fragment for a variadic argument $\mathtt{E}_i^{\tau_{\mathrm{arg}_i}}$

7.2 Top-level elaboration function

For a given C translation unit, the frontend of Cerberus will have produced a fully type-annotated Ail representation consisting of a record with the following components: the collection of file-scoped object and function declarations, containing their type signatures; the initialisation expressions for objects, when they exist; the statement definitions of functions; and the member definitions for each struct and union type. The top-level elaboration function is defined by folding over the collection of declarations.

Elaboration of Ail objects For each Ail object declaration, the elaboration produces a Core global with type `pointer` whose body[3] consists of the allocation action for the object, followed, when the Ail declaration has an initialiser, by the elaboration of the initialiser expression, whose value is then stored in the object. In either case, the result of the body is the pointer to the freshly allocated object. For example, a file-scoped declaration of the form `int x = E;` is elaborated to the following global:

```
1  glob x : pointer :=
2    letstrong ptr = create(Ivalignof('signed int'), 'signed int') in
3    letstrong z = [[ E ]] in
4    letstrong Unit = store('signed int', ptr, z) in
5    pure(ptr)
```

The symbol x is in scope of the whole Core program, and used in the elaboration of lvalues.

Elaboration of Ail functions For each Ail function definition, the elaboration produces a Core procedure. The first argument is a boolean *is_used* which specifies whether the return value is used at the call site. Counterparts to the C arguments then follow which, similar to the type of the global in the previous case, all have type `pointer`. In C,

[3]The effectful Core expression evaluated at the beginning of the Core program execution. Once evaluated, the symbol of the global is bound to the value of this expression for the rest of the program execution.

when calling a function, an implicit memory object is allocated in the abstract machine for each parameter, and initialised to the result of evaluating the parameter expression. All of this is made explicit in the elaboration of the function call expression operator. This is the reason Core procedures corresponding to Ail functions take pointers as arguments. There is an exception for procedures elaborating variadic functions, where there is an argument holding a list of pairs of a `ctype`, and a `pointer` added to the end. This is used to model the passing of unnamed arguments (e.g. an argument following the string literal when calling `printf()`). In the elaboration of a call expression to a variadic function, the types in the list are used to perform the dynamic typecheck, and indicate the appropriate undefined behaviour if it fails. The construction of the elaboration of Ail function is as follows:

```
1  S' := ERASE_LOOP_CONTROL(S);
2  ret := FRESH_LABEL;
3  LET T = { unit                  τ_ret = void
           { loaded ⟦ τ_ret ⟧      otherwise        IN
4  z := FRESH_SYMBOL(T);
5  LET e_ret = { Specified(0)                                   if f is the startup function
               { undef(<<UB071_noreturn>>)                     if f is _Noreturn
               { Unit                                          if f is a void function
               { if is_used then
               {   undef(<<UB088_reached_end_of_function>>)    otherwise
               { else
               {   Unspecified(τ_ret)
6  IN
7    proc f (is_used : boolean, arg_1 : pointer, ..., arg_n : pointer) : T :=
8      letstrong Unit = ⟦ S' ⟧ in
9      save ret : unit(z : T := e_ret) in
10       pure(z)
```

To construct the body of the corresponding Core procedure, the corresponding Ail statement is elaborated to a Core effectful expression (line 7). As discussed in Section 4.5, it is first transformed such that all looping statements (and their associated **`continue`** and **`break`**) and **`switch`** statements are rewritten as a combination of blocks, label declarations, and **`goto`** statements (line 1). Following this transformation, a preliminary traversal over the resulting statement is performed to collect a map associating each label to the set of Ail identifiers which are in scope of the statement declaring that label. This is used by the elaboration of **`goto`** statements for the modelling of implicit allocation and deallocation of block-scoped variables that we discussed in Section 4.4. At the end of the procedure, we place a **`save`** operator (line 8) which declares the target to which the elaboration of **`return`** statements jumps to. This label takes as argument the return value. Using the default pure expression of the argument, we indicate the undefined behaviour that occurs if a non-**`void`** function ends with no **`return`** statement and has its return value used by the caller; the one that occurs if a **`_Noreturn`** function does reach it end; and finally the fact that the startup function is defined (with return value 0) regardless of whether any **`return`** statement is taken (despite the fact that it is required to be a non-**`void`** function).

Chapter 8

Memory object model: pointer values with provenance

In the previous chapters, the presentation of our model of C has kept the semantics of pointers and memory objects abstract. As we discussed in Chapter 5, the Core language and the elaboration function are by design orthogonal of these issues. The elaboration makes explicit where a program interacts with the memory state, and how it constructs pointer values through a small abstract interface. This is a key design choice of our model, allowing us to explore different memory models. In this chapter, we describe a memory model for C aiming to satisfy most of the current ISO standard requirements, while being amenable to mainstream use of the language, in particular for system programming. In the next chapter, we give a formal presentation of this memory model. Before this work, the reconciliation of the ISO standard and practice involving low-level manipulation of pointers remained a largely unresolved issue.

There is a tension between these two constraints, making it a priori unclear whether a single memory model can satisfy both. One might expect such a memory model to be either of two extremes: (1) a concrete model, where the underlying behaviour of the hardware is mostly exposed, pointer values are numeric addresses behaving like integers, and the memory state is simply a partial map from address to bytes; or, (2) an abstract model with a strong distinction between numeric integers and pointer values, for which only operations such as dereferencing are defined.

C, as it exists in mainstream implementations and the existing corpus of code, is neither of these. Its values are not fully abstract: the language intentionally permits manipulation of their underlying representations, via casts between pointer and integer types, `char`* pointers to access representation bytes, and so on. This aspect of the language is critical to support low-level systems programming. At the same time, in current implementations, pointer values cannot be considered to be simple concrete values: while at runtime they will typically just be machine words, compiler analysis reasons about abstract notions of the provenance of pointers and the definedness of values, and compiler optimisations rely on assumptions about these for soundness.

To understand exactly what is allowed, as a C programmer, compiler or analysis tool writer, or semanticist, one might turn to the ISO language standard produced by WG14 [ISO-C11]. However, while in many respects the ISO standard is clear (e.g. regarding the subtleties we discussed in Chapter 4), when it comes to pointers and the memory model it is not. In a defect report from 2001 [DR260], the UK C Panel suggested that a notion of provenance might be associated to values (and particular pointer

values) to accommodate compiler optimisations, and asked WG14 to clarify the standard accordingly. In its response, the WG14 committee hints at the idea that pointer values do indeed have a provenance by asserting that implementations may "[...] treat pointers based on different origins as distinct even though they are bitwise identical.". The exact meaning of that sentence is however left undefined, and no clarifying text was ever incorporated into the standard text, despite there having been several published versions since the committee response. The precise specification of when two pointer values should be deemed to be equal is particularly important, since some compiler optimisations rely on alias analysis to deduce that two pointer values do not refer to the same object, which in turn relies on assumptions that the program only constructs pointer values in "reasonable" ways (with other programs regarded as having undefined behaviour, UB).

Furthermore, in some respects, there are significant discrepancies between the ISO standard and the de facto standards of C as it is implemented and used in practice. Major C codebases typically rely on particular compiler flags, e.g. `-fno-strict-aliasing` or `-fwrapv`, that substantially affect the semantics, but which the standard does not attempt to describe; and some idioms have undefined behaviour in ISO C, but are widely relied on in practice. For example, performing a comparison on a pointer value referring to an object whose lifetime has ended is clearly not supported by the ISO standard, but, in practice, reasonable code does compare against such pointers, and this is widely expected to work – although in some cases mainstream compilers do assume code does not do this. There is also not a unique de facto standard: in reality, one has to consider the expectations of expert C programmers and compiler writers, the behaviours of specific compilers, and the assumptions about the language implementations that the global C codebase relies upon to work correctly. The surveys of these expectations that we describe in Chapter 2 (published in [Mem+16; N2015]) revealed many discrepancies, with widely conflicting responses to specific questions.

All these issues are exacerbated by the fact that the ISO standard is a prose document, as is typical for industry standards. The lack of mathematical precision, while also typical for industry standards, has surely contributed to the accumulated confusion about C's memory model.

8.1 Basic pointer provenance

In this section, we develop a notion of provenance aiming to capture the intent of DR260, and accommodate the optimisations of mainstream compilers, while allowing the normal idioms of how pointer values are constructed and used in practice. To illustrate how the necessity of pointer provenance arises in order to justify optimisations performed by mainstream compilers, we first consider a classic test program [DR260; N1637; Kre15; N2013; Mem+16]. Note that this, along with most other examples in this chapter, are edge-cases intended to explore the boundaries of what different semantic choices allow, and sometimes what behaviour existing compilers exhibit; they are not all suggested as desirable code idioms.

```
1  #include <stdio.h>
2  #include <string.h>
3  int y=2, x=1;
4  int main() {
5    int *p = &x + 1;
6    int *q = &y;
7    printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
8    if (memcmp(&p, &q, sizeof(p)) == 0) {
9      *p = 11;  // does this have undefined behaviour?
10     printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
11   }
12 }
```

Figure 8.1: provenance_basic_global_yx.c

This program allocates two integer objects `x` and `y`, and then allocates two pointers: the first one, `p`, is initialised to one past the address of `x`, while the second, `q`, is initialised with the address of `y`. Depending on implementation-defined choices, the two integer objects may, in some executions, happen to be allocated adjacent in memory, in which case the representation values of the two pointers will be bitwise identical. In such executions, the call to `memcmp()` (which compares the byte representations of the pointers) evaluates to zero, leading the program execution to enter the **`if`** statement. A memory access is performed by dereferencing `p`, a pointer which, while derived from a pointer to `x`, has, at the same time, been established to have the same representation value as a pointer to a different object, `y`.

The question then arises as to whether that memory access is valid, and, if it is, which of the two integer objects is being modified. Note that the suspicious-looking initialisation at line 5 is not the issue here. The formation of the `&x+1` one-past pointer is explicitly permitted by the ISO standard (C11, §6.5.6p8, sentence 4). Furthermore, because the line 9 store is guarded by the `memcmp()`, we know that only program executions where the two integer objects have been allocated adjacently will be performing that access. The following ISO passage appears to give the access undefined behaviour:

(§6.5.6p8, last sentence) (...) If the result points one past the last element of the array object, it shall not be used as the operand of a unary **`*`** operator that is evaluated.

but, because of the guard, it is unclear whether the pointer value being dereferenced at line 9 is to still be viewed as a one-past-pointer to `x`, rather than a valid pointer to `y`.

In a concrete view of the memory model, where pointers are simply numerical values, we should expect executions entering the **`if`** to output `x=1 y=11 *p=11 *q=11`: the store at line 9 should modify the value stored in `y`, and the loads in the call to `printf()` should both load the new value.

However, running this compiled with GCC 12.2.0 -O2 outputs `x=1 y=2 *p=11 *q=2` on some platforms. This suggests that the compiler is reasoning that `*p` does not alias with `y` or `*q`, and hence (one of its optimisations) can propagate the initial value of `y=2` to the call to `printf()` at line 10. We see something similar when compiling with ICC 19 -O2, which produces an executable outputting `x=1 y=2 *p=11 *q=11`. For this compiler and optimisation level, the order of the allocations of `x` and `y` is reversed, and we therefore consider the variant of the program where the declarations at line 3 are

swapped. In contrast, an executable compiled with Clang 15.0.2 -O2 outputs what the concrete semantics predicts.

Note that adding the flag `-fno-strict-aliasing` does not affect the outcomes observed for GCC and ICC. Both `x` and `y` have integer types, and the referenced types of pointers reflect that, the issue raised by the present program is therefore unrelated to type-based alias analysis.

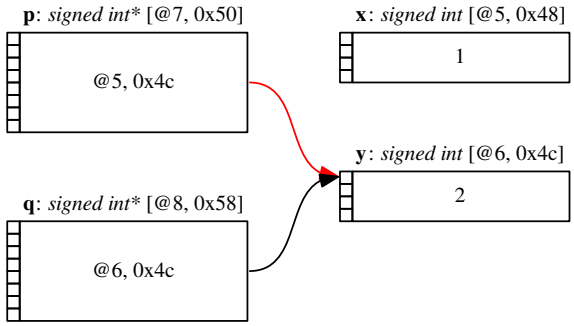
The outcomes for GCC and ICC would make them unsound with respect to the concrete memory model. This brings us back the committee response to DR260, which hints at a notion of provenance associated to pointer values that keeps track of their "origin". Here, the origin of the value of `p` does not involve any valid pointer to `y`, and needs a memory model that uses this fact to disallow the store at line 9 by making it undefined behaviour. In such a provenance-aware model, the optimisations above become sound.

Provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: in the formalisation of C's abstract machine, we equip every pointer value with a *provenance*, identifying the original allocation event the pointer is derived from. In more detail:

- We define a pointer value as a pair (π, a), where a is its concrete numeric address, and π is its associated *provenance*, which can either be $@i$, where i is an identifier for the result of an allocation event, or the *empty* provenance, `@empty`.
- When an object is allocated, the abstract machine nondeterministically chooses a fresh ID i (unique across the entire execution), and the resulting pointer value carries it as its provenance $@i$. Additionally, the abstract machine keeps details of the allocation associated to i (such as its footprint) in ghost state.
- The pointer arithmetic operations, which add or subtract an integer to a pointer value, preserve the provenance of their pointer operand.
- Whenever a pointer value is used to perform a memory access, its numeric address must be consistent with its provenance; otherwise, the access is given undefined behaviour. More precisely:
  - Access via a pointer value which has provenance $@i$ must be within the memory footprint of the object corresponding to i. In particular, this means that the object must still be live.
  - All other accesses, e.g. those with a provenance $@i$ but whose numeric address does not match the footprint, and also any access using a pointer value with empty provenance, have undefined behaviour.[1]

The undefined behaviour for pointer values corresponding to the last bullet allows us to recover soundness for optimisations based on provenance alias analysis. To illustrate how the basic provenance semantics operates on `provenance_basic_global_xy.c`, let us consider a graphical representation of the abstract memory state taken from the Cerberus web interface, and corresponding to when the program execution is about to perform the access at line 9.

[1] In the more precise discussion of our proposed models, we will see that there are some exceptions to this: for example, to allow accesses to memory-mapped devices typically found in embedded programming, we will need to exempt some ranges of numeric addresses.



Each box represents a memory object, with the identifier from the declaration, the type, and the pointer value produced by the allocation of an object written at the top. The current value of the object is written inside the box. We see that the numeric address of both pointers is `0x4c` (the address of `y`), which is represented by the two arrows pointing from each pointer to `y`. However, for `p`, the provenance is @5, which corresponds to the allocation event of `x`. Performing a memory access using this pointer value therefore has undefined behaviour (as indicated by the colour of the arrow leaving `p`).

The provenance mechanism occurs in the *C abstract machine*, used by the ISO standard to specify the C language. It is not meant as a requirement on how compilers *implement pointers at runtime*. The provenance of a pointer value is not required to have a runtime representation, and is therefore not accessible to the programmers. However, for compilers relying on a notion of provenance in their alias analysis and optimisations, this model provides a specification of the assumptions they can soundly make.

While the simple model of pointer values with provenance moves away from a fully concrete view of the memory, it still remains less abstract than the view taken by most previous formal studies of the C memory model, where memory objects could be thought as isolated "islands" within the address space. We discuss the related work on C memory models in Chapter 13.

In a provenance-based model, the numeric component of pointer values allows for a relaxation of the isolation between objects in the model. This raises, even for the basic provenance semantics, some open design questions when defining operations such as pointer arithmetic operations, or the equality and relational operators. We now discuss these questions, and look at some possible choices.

Should the construction of out-of-bounds pointer values be allowed? Consider the example below, where the value of the pointer `q` is transiently out of bounds of the object is it pointing to (by more than one-past), before being brought back into bounds and used for an access.

```
1  #include <stdio.h>
2  int main() {
3    int x[2];
4    int *p = &x[0];
5    //is this free of undefined behaviour?
6    int *q = p + 11;
7    q = q - 10;
8    *q = 1;
9    printf("x[1]=%i  *q=%i\n",x[1],*q);
10 }
```

Figure 8.2: `cheri_03_ii.c`

The ISO standard clearly states that the mere construction of such an out-of-bounds pointer value has undefined behaviour (C11, §6.5.6p8). This is easily captured in the

provenance model, by adding, in the semantics of the pointer arithmetic operations, a check that the numeric address of the resulting pointer remains within the footprint corresponding to the provenance of the pointer value or one-past. This is the same check as the one we described for memory accesses.

The proscription is likely motivated by the needs of implementations where out-of-bounds pointer arithmetic would go wrong, e.g. hardware that does enforce bounds checking, or where pointer arithmetic might wrap at values less than the obvious word size (e.g. "near" or "huge" 8086 pointers). However, these use cases correspond to platforms that are now exotic. Furthermore, the use of transient out-of-bounds pointer construction is observed in commonly used code bases [Chi+15; Dav+19]. It may therefore be desirable to make it implementation-defined whether such pointer construction is allowed. That would continue to permit implementations in which it would go wrong to forbid it, but give a clear way for other implementations to document that they do not exploit this undefined behaviour that may be surprising to programmers. Adapting the basic provenance semantics to this (if desired) is straightforward: the bounds checks are removed from the semantics of the pointer arithmetic operators. In the models implemented by Cerberus, we support both semantics, with a switch. The validity of pointer values remains checked at any access (in the compatibility check between the numeric address and the provenance); hence, the relaxation does not require the dereferencing operators to deal with new cases. The domains of arithmetic and relational operators are however extended, leading to further questions.

Should pointer arithmetic across object boundaries be defined? The example in Figure 8.1 is sensitive to how memory objects are allocated, for which the ISO standard imposes no requirements on implementations. As a result, whether the execution of interest (the one where the `if` is entered by the program execution) is observable for a particular implementation depends on the declaration order of the two integer objects. We can instead write a program involving pointer subtraction to calculate the offset between the numeric addresses of two objects, and attempt to use pointer arithmetic to turn a pointer to one of the object into a pointer to the other:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stddef.h>
4  int x=1, y=2;
5  int main() {
6    int *p = &x;
7    int *q = &y;
8    ptrdiff_t offset = q - p;
9    int *r = p + offset;
10   if (memcmp(&r, &q, sizeof(r)) == 0) {
11     *r = 11; // is this free of UB?
12     printf("y=%d *q=%d *r=%d\n",y,*q,*r);
13   }
14 }
```

Figure 8.3: pointer_offset_from_ptr_subtraction_global_xy.c

As before, we have two integer objects `x` and `y`. This time, however, the pointers `p` and `q` are simply initialised to their addresses, and initialise a new integer object `offset` to the

result of subtracting the two pointers.

The ISO standard deems the subtraction itself to be undefined behaviour, as the operator requires its operands to be pointers to within a same object (C11, §6.5.6p9, sentence 1). This restriction is likely motivated by the desire to allow efficient implementations of C on hardware with segmented memory. However for today's common implementations, this is not necessary from a hardware point of view; one can therefore consider a model where the subtraction operator is totally defined (removing one undefined behaviour from the language). If we consider such a model, the provenance check at line 11 still makes this program undefined. However, permitting this would prevent provenance-based alias analysis (except where a compiler could reason that objects are not accessed via such offsets).

Pointer equality comparison and provenance A priori, we might expect pointer equality comparison (with == or !=) to just compare the numeric addresses of its operand. However, we observe that GCC 12.2.0 with optimisation level -O2 sometimes considers two pointers with the same address but different provenance as non-equal. For example, in the following variant of the Figure 8.1, we see the program outputting `(p==q) = false` despite the print at line 7 showing that `p` and `q` have the same numeric address.

```
1  #include <stdio.h>
2  #include <string.h>
3  int x=1, y=2;
4  int main() {
5    int *p = &x + 1;
6    int *q = &y;
7    printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
8    _Bool b = (p==q);
9    // can this be false even with identical addresses?
10   printf("(p==q) = %s\n", b?"true":"false");
11   return 0;
12 }
```

Figure 8.4: provenance_equality_global_xy.c

Unsurprisingly, this happens in some circumstances, but not others. For example, pulling the equality test into a simple separate function (but still in the same translation unit) inhibits the optimisation.

To allow such compiler behaviour, the pointer equality operator in our memory model should evaluate to false when its operands have different numeric addresses (as expected); but when they do have the same address, it should nondeterministically (at each runtime occurrence) either take provenance into account or not. Alternatively, one could require numeric comparisons, which would be a simpler semantics for programmers, but would make that GCC behaviour unsound. Cerberus supports both options. One might also imagine making it UB to compare pointers that are not strictly within their original storage instance [Kre15], but that would break loops that test against a one-past pointer, which is common practice. One could also require the equality operator to *always* take provenance into account, but that would require implementations to track provenance at runtime.

In its current form, the ISO C18 standard text is too strong here, unless numeric comparison is required: 6.5.9p6 says *"Two pointers compare equal* ***if and only if*** *both*

are [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space", which requires such pointers to compare equal – reasonable pre-DR260 CR, but debatable after it.

Pointer equality should not be confused with the result of alias analysis: we could require == to return true for pointers with the same address but different provenance, while still permitting alias analysis to regard the two as distinct by making accesses via pointers with the wrong provenance UB.

Pointer relational comparison and provenance The ISO standard (6.5.8p5) makes it undefined behaviour to compare (with <, etc.) pointers referring to different memory objects (inter-object). As for the similar inter-object pointer subtraction, there are platforms where this would go wrong, but, again, these are now mostly defunct. On the other hand, there are also substantial bodies of code that rely on the ability to do such comparisons, e.g. for lock orderings, or storage of pointers in ordered structures. It may therefore be desirable to make it implementation-defined whether such pointer construction is allowed. When giving a defined semantics here, in our provenanced model, we only need to make use of the concrete component of pointer values.

8.2 Extending to the rest of C

As a language meant to allow low-level system programming, C provides many other ways to construct and manipulate pointer values. One can:

- cast pointers to integer types and back, possibly with some integer arithmetic in between, e.g. to force alignment, or to store information in unused bits of pointers;
- create copies of a pointer value using certain functions from the standard library, e.g. `memcpy()` and `realloc()`;
- manipulate the representation bytes of a pointer, e.g. in user code making use of `char*` or `unsigned char*` pointers to access these bytes;
- reinterpret the representation bytes of a pointer as an integer value, using type punning;
- perform I/O on pointer values, either using formatting functions such as `fprintf()`/`fscanf()` with the `%p` conversion specifier, or using direct input/output function such as `fwrite()`/`fread()` on the pointer representation bytes; or,
- construct pointer values using additional knowledge about the underlying runtime, such as details about linking or the layout of memory-mapped devices.

All of these break the separation between pointers and integers that one might think the C type system enforces. With provenances attached to pointer values, this raises the design question of whether integer values should also be equipped with a provenance. Similarly, because the representation of pointers can be manipulated by the programmer, we need to specify what are the implications of these manipulations on the provenance of a pointer value.

We define two main alternative provenance-based memory models:

- **PVI** (provenance via integers): in this model, both pointer and integer values are associated a provenance. Conversions from pointers to integers preserve the provenance, and the provenance is tracked throughout integer computations. All integer operations have to be made aware of the provenance of their operands, and make some particular choices whether or not to preserve it. We will see in the next section that adding provenances to integer values is however a significant change to their semantics, and it breaks some of the expected algebraic properties.
- **PNVI** (provenance not via integers): in this model, provenance is restricted to pointer types. The semantics of integer values remains as it is described in the ISO standard. As a result, pointer-to-integer casts erase any provenance; some special attention is therefore required in the semantics of converse casts. Broadly, for these, the model checks whether the numeric address resulting from the conversion points within a live object and, if so, recreates the corresponding provenance for the pointer value produced by the cast. We present three variants of this model in Section 8.4, with varying restrictions on how integer-to-pointer casts can recreate provenance. We will also see that this model is not as damaging to optimisations as one might expect from the apparently less precise tracking of provenance.

8.3 PVI: integer values with provenance

When we started this work on a provenance-based memory model, the documentation and behaviour of GCC and ICC led us to aim for a model where integer values resulting from a pointer-to-integer cast somehow preserve knowledge of the "original" pointer used in the cast. The view was motivated by the following passage from the GCC documentation [GCC-arrays]:

> *"When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 and C11 6.5.6/8."*

Experimentally this can be observed for both GCC and ICC. Consider the following variant of the program in Figure 8.1, where the construction of the problematic pointer has been substituted with integer analogues working over the `uintptr_t` type:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4  #include <inttypes.h>
5  int x=1, y=2;
6  int main() {
7    uintptr_t ux = (uintptr_t)&x;
8    uintptr_t uy = (uintptr_t)&y;
9    uintptr_t offset = 4;
10   ux = ux + offset;
11   int *p = (int *)ux; // does this have UB?
12   int *q = &y;
13   printf("Addresses: &x=%p p=%p &y=%"PRIxPTR\
14          "\n",(void*)&x,(void*)p,uy);
15   if (memcmp(&p, &q, sizeof(p)) == 0) {
16     *p = 11; // does this have undefined behaviour?
17     printf("x=%d  y=%d  *p=%d
   *q=%d\n",x,y,*p,*q);
18   }
19 }
```

Figure 8.5: provenance_basic_using_uintptr_t_global_xy.c

Compiling this with GCC[1] and ICC 19 at optimisation level -O2 (and greater) results in a program with outputs similar to what we observe for the example in Figure 8.1. This behaviour is inconsistent with a concrete numerical view of the addresses.

In the PVI model, this program has undefined behaviour. When casting the two pointers at lines 7 and 8, the resulting integers preserve provenance respectively to `x` and `y`. At line 10, the addition operation is performed with its left operand having a provenance, while the right one (coming from a constant) does not. The result of the operand preserves the provenance of the left operand, namely `x`. As result, the casts to pointer type at line 11 result in a pointer with a provenance referring to `x`, having the same numeric value as the pointer `q`, and therefore is out of bounds for its provenance. The access at line 16 is therefore undefined.

This model is rather straightforward to define, though it requires updating the semantics of all operators over integers. When only one operand has a provenance, or if they both have the same one, it is preserved in the result. Otherwise, the result has no provenance. We make an exception for the subtraction operator, where the result has a provenance only if the left operand is the only one with a provenance.

However, equipping integers with provenance has the undesirable effect of breaking their algebraic properties. Discussions at the 2018 GNU Tools Cauldron suggest instead that at least some key developers regard the result of casts from integer types as potentially broadly aliasing, at least in their GIMPLE IR, and regards such test results as long-standing bugs in the RTL backend.

[1]Tested with various versions of GCC ranging from 4.9 to 12.2. Note that at optimisation level -O1 (and greater) the layout of the two global variables is swapped compared to -O0. The behaviour we describe for the optimised level is therefore for the test variant provenance_basic_using_uintptr_t_global_yx.c which compensates for this.

8.4 PNVI: integers with no provenance

If we deem the behaviour observed for GCC and ICC on the previous example to be erroneous, the need for tracking provenance through integers becomes less attractive. Shifting to a provenance semantics that does not, leads to a substantial simplification, in the definition of the semantics, in how easy it is for people to understand, and in the consequences for existing code (which might otherwise need additional annotations for exotic idioms).

In PNVI, integers have no provenance, and the definitions of their operations are therefore left unchanged. However, in the semantics of integer-to-pointer casts, we need to choose whether the resulting pointer is ever given a provenance when the integer originated (potentially through arithmetic computation) from the result of a pointer-to-integer cast. In a model where no such casts produce a pointer value with provenance, common low-level practices such as storing metadata in the unused bits (because of alignment constraints) of pointers would be undefined. The most permissive semantics for programmers is the following: when performing an integer-to-pointer case, whenever the numeric component of the pointer being constructed is within the footprint of an object which is live at the time of the cast, pick the provenance of that object. From now on, we will refer to this variant as **PNVI-plain**.

Such a model might seem too constraining for implementations, as one might think that it requires them to assume that the result of a integer-to-pointer cast may alias with any other pointer, thereby precluding optimisation opportunities. An obvious refinement to PNVI, which might alleviate this issue, is to restrict integer-to-pointer casts to only recover the provenance of objects that have had their address taken, recording that in the memory state. Perhaps surprisingly, that seems not to make much difference to the set of allowed programs, because the code one might write tends to already be undefined behaviours due to allocation-address nondeterminism, or to already take the address of an object to use it in a guard. We show this in detail in Section 8.5. This refined variant has the conceptual advantage of identifying these undefined behaviours without requiring examination of multiple executions, and might be more relatable to the internals of compilers' alias analyses. It however has the disadvantage of relying on whether an address has been taken, which is a fragile syntactic property, e.g. not preserved by dead code elimination. This could be mitigated by restricting casts to addresses that have in some sense escaped, but precisely defining a particular such sense is complex and somewhat arbitrary.

We first presented the idea of the PNVI model in [Mem+19]. From discussions with WG14, and in particular the C memory object model study group (including Jens Gustedt, Martin Uecker, and others), the following two variants of PNVI-plain emerged:

- **PNVI-ae (PNVI exposed-address)**: a variant of PNVI that allows integer-to-pointer casts to recreate provenance only to objects that have previously been *exposed*. A memory object is deemed exposed if a pointer value referring to that objects is cast to an integer type, or has one of its representation bytes read (with a non-pointer lvalue), or if it is output using `%p`. Furthermore, in this variant, an integer-to-pointer cast of an address one-past an object results in a pointer with the empty provenance (unless this address also happens to be the beginning of another adjacent exposed object). This variant therefore loses the round-trip property of casts guaranteed by the ISO standard.

- **PNVI-ae-udi (PNVI exposed-address with user disambiguation)**: this variant extends the previous one with additional machinery to bring back support for the round-trip property. This is the currently preferred option in the C memory object model study group and the WG14/WG21 committees.

We give a formal presentation of PVI and the three PNVI variants in Chapter 9, but first look what these models impose on implementation.

8.5 Implications of provenance semantics for optimisations

The provenance semantics aims to formalise assumptions underlying alias analyses that are used by existing C compilers when performing some of their optimisations. As a result, it defines an envelope for the allowed outcome of an alias analysis, and this has a direct impact on what optimisations are allowed to do on pointers to be deemed sound with respect to the memory model. Ideally, we would like a model that is consistent with all existing mainstream code usage and compiler behaviour. However, given the long-standing lack of clarity in the ISO standard regarding the implications of the WG14 committee response to DR260, we suspect that programmer practice and implementations have diverged too much to allow a model encompassing them all. In this section, we look at the impact the four variants of our model have on a few assumptions that we expect common optimisations to rely on.

8.5.1 Optimisations based on pointer equality tests

In PVI and all the variants of PNVI, pointer equality `p==q` can hold in cases where `p` and `q` are not interchangeable (e.g. where dereferencing is only well-defined for one of them). As Lee et al. [Lee+18] observe in the LLVM IR context, that may limit optimisations such as GVN (global value numbering) based on pointer equality tests. While restricting the scope of GVN for pointer types might have an acceptable cost, the situation is more serious for PVI. In this model, the same problem is present for comparison operators over integer types, wherever the operands might be the result of casts from pointers and eventually be cast back.

8.5.2 Allowing non-aliasing assumptions across function frames

Non-aliasing of function arguments with local variables Compilers have to assume that a function never receives as an argument a pointer which may alias with one of its local variables, or an integer which when cast to a pointer could alias a local.

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include "charon_address_guesses.h"
4  void f(int *p) {
5    int j=5;
6    if (p==&j)
7      *p=7;
8    printf("j=%d &j=%p\n",j,(void*)&j);
9  }
10 int main() {
11   uintptr_t i = ADDRESS_PFI_1PG;
12   int *p = (int*)i;
13   f(p);
14 }
```

Figure 8.6: pointer_from_integer_1pg.c

Consider for example the program in Figure 8.6, where, in the `main()` function, the address of `f()`'s local variable is somehow guessed (at line 11), and then cast to a pointer which is passed as an argument to `f()`. In the body of `f()`, the pointer is then accessed under a guard checking that it compares equal to a trivial pointer to the local variable `j`. Compiling this with GCC, even at optimisation level -O0, removes the **`if`** and the write `*p=7`. As a result, even for executions where the macro `ADDRESS_PFI_1PG` is suitably defined, the print at line 8 shows 5. To allow this compiler behaviour, the program needs to be deemed to have undefined behaviour. More generally, this indicates that C programs should not normally be able to rely on implementation facts about the allocation addresses of C variables. In all PNVI-* variants, the access at line 7 is deemed to have undefined behaviour: the cast of the guessed address occurs before the beginning of the lifetime of the local variable and therefore produces a pointer with the empty provenance. In these models, compilers are therefore entitled to assume that the test expression of the **`if`** always evaluates to zero. In the PVI model, the undefined behaviour simply comes from the fact that `j` is created with the empty provenance, and hence `p` inherits that.

If we modify the program such that the "guessed" address of the local variable is passed to the function `f` as a integer, the cast to a pointer needs to happen in the body of that function, and therefore after the lifetime of the local variable `j` has started. As a result, in PNVI-plain, the resulting pointer has a valid provenance (to the local variable), and the program has defined semantics. In the PNVI-ae and PNVI-ae-udi models, as the address of the local variable has not been exposed, the cast results in a pointer with the empty provenance, and the access at line 8 still has undefined behaviour. This example is also UB in PVI.

We do observe the same optimisation as for the previous example when compiling with GCC, Clang and ICC (though this time from optimisation level -O2 and higher). The PNVI-plain model is therefore too strong for these compilers as they currently are. Soundness with respect to PNVI-plain would require compilers to be more conservative with integer-to-pointer casts from integers whose source they cannot see.

Nondeterminism of the allocation of objects Both of the previous examples have a test, comparing the pointer `p` with the address of `j`, guarding the potentially undefined access. If we remove this guard (Figures 8.7 and 8.8), all four models give undefined behaviour to the store through `p`.

```c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1P;
  int *p = (int*)i;
  f(p);
}
```

Figure 8.7: pointer_from_integer_1p.c

```c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

Figure 8.8: pointer_from_integer_1i.c

In the PVI model, both programs remain undefined for the same reason as before: the absence of provenance on the guessed integer. Similarly, the PNVI-* models deem undefined the program in Figure 8.7 because the cast happens before the beginning of the lifetime of the local variable.

The program in Figure 8.8 is also deemed undefined, but for that program, the reasoning makes use of the fact that the allocation of addresses is left unspecified by the ISO standard. As discussed in Section 4.1, with respect to the nondeterminism, it is necessary to define a notion of undefined behaviour, such that any occurrence of a undefined behaviour in any execution results in an undefined behaviour for the whole program. In term of compiler optimisations, this is necessary to allow the soundness of code motion of expressions for which the compiler does not establish the absence of undefined behaviour.

Accordingly, our semantics nondeterministically chooses an arbitrary address for each storage instance, subject only to alignment and non-overlap constraints (ultimately, one would also need to build in constraints from programmer linking commands). This is equivalent to noting that the ISO standard does not constrain how implementations choose storage instance addresses in any way (subject to alignment and non-overlap), and hence that programmers of standard-conforming code cannot assume anything about those choices. Then, in PNVI-plain, the example in Figure 8.8 is undefined because, even though there is one execution in which the guess is correct, there is another (in fact many others) in which it is not. In those, the cast gives a pointer with empty provenance, so the access is forbidden — hence the whole program has undefined behaviour, as desired.

In the PNVI-ae and PNVI-ae-udi models, this example is deemed undefined without reasoning about the allocation nondeterminism: instead, because the memory object for the local variable of `f` has not been exposed before the cast (which would involve casting its address to an integer type), the cast results in a pointer with the empty provenance, and the store access through `p` has undefined behaviour in every execution.

However, if we do expose the addresses of local variables, as in the following:

```c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

Figure 8.9: pointer_from_integer_1ie.c

the cast produces, in one execution, a pointer with a valid provenance. The PNVI-ae and PNVI-ae-udi models then deem the program undefined as a result of the allocation nondeterminism.

Non-aliasing of local pointers with locals of a parent function Conversely, it is desirable to allow compilers to assume that functions cannot create a local pointer which is valid for accessing an object local to a parent function. The example in Figure 8.10 is forbidden by PVI, again simply because `p` has the empty provenance, and by PNVI-plain as a result of allocation-address nondeterminism: as there exist abstract-machine executions in which the guessed address is wrong. One cannot guard the access within `f()`, as the address of `j` is not available there. In the PNVI-ae-* models, the example is simply forbidden because the object `j` is never exposed (though even if it were exposed, the example would remain forbidden because of the nondeterminism argument that comes in play for PNVI-plain).

```c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  f();
  printf("j=%d\n",j);
}
```

Figure 8.10: pointer_from_integer_2.c

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include "charon_address_guesses.h"
4  void f() {
5    uintptr_t i=ADDRESS_PFI_2G;
6    int *p = (int*)i;
7    *p=7;
8  }
9  int main() {
10   int j=5;
11   if ((uintptr_t)&j == ADDRESS_PFI_2G)
12     f();
13   printf("j=%d &j=%p\n",j,(void*)&j);
14 }
```

Figure 8.11: pointer_from_integer_2g.c

In the example in Figure 8.11, where the call to `f()` is guarded with a test checking that the guessed address used in the body of `f()` is correct, the undefined execution from allocation-address nondeterminism is lost, and the example is therefore well-defined in PNVI-plain. PNVI-ae-* also allows this example, because the check involves exposing the object `j`. This does clash with behaviour we observe for Clang at optimisation levels -O2 or greater, which prints `j=5` at line 13.

The problem with lost address-takens and escapes The PVI model allows computations that erase the numeric value (and hence a concrete view of the "semantic dependencies") of a pointer, but retain provenance. This makes examples like the one in Figure 8.12[2], in which the code correctly guesses the address of an object (which has the empty provenance) and adds that to a zero-valued quantity (with the correct provenance), allowed in PVI. We emphasise that we do not think it especially desirable to allow such examples; this is just a consequence of choosing a straightforward provenance-via-integer semantics that allows the byte-wise copying and the bitwise manipulation of pointers above. In other words, it is not clear how it could be forbidden simply in PVI.

[2]Personal communication with Richard Smith.

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "charon_address_guesses.h"
int x=1; // assume allocation ID @1, at ADDR_PLE_1
int main() {
  int *p = &x;
  uintptr_t i1 = (intptr_t)p;
// (@1,ADDR_PLE_1)
  uintptr_t i2 = i1 & 0x00000000FFFFFFFF;//
  uintptr_t i3 = i2 & 0xFFFFFFFF00000000;// (@1,0x0)
  uintptr_t i4 = i3 + ADDR_PLE_1;
// (@1,ADDR_PLE_1)
  int *q = (int *)i4;
  printf("Addresses: p=%p\n",(void*)p);
  if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
    *q = 11;  // does this have defined behaviour?
    printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
}
```

Figure 8.12: provenance_lost_escape_1.c

However, during compilation by some implementations, some algebraic optimisations may be done before alias analysis, and those optimisations might erase the `&x`, replacing it, and all the calculation of `i3`, by `0x0` (a similar example would have `i3 = i1-i1`). But then alias analysis would be unable to see that `*q` could access `x`, and so report that it could not, and hence enable subsequent optimisations that are unsound w.r.t. PVI for this case. The basic point is that whether a variable has its address taken or escaped in the source language is not preserved by optimisation. A possible solution, which would need some adaptation for implementations that do track provenance through integers, would be to require those initial optimisation passes to record the set of addresses that have been "taken" involved in computations they erase, so that that could be passed in explicitly to alias analysis. In contrast to the difficulties of preserving dependencies to avoid thin-air concurrency, this does not forbid optimisations that remove dependencies; it merely requires them to describe what they do.

In PNVI-plain, the example is also allowed, but for a simpler reason that is not affected by such integer optimisation: the object exists at the `int*` cast. Implementations that take a conservative view of all pointers formed from integers would automatically be sound w.r.t. this. At present ICC is not, neither at -O2, nor at -O3.

PNVI-ae and PNVI-ae-udi are more like PVI here: they allow the example, but only because the address of `p` is both taken and cast to an integer type. If these semantics were used for alias analysis in an intermediate language after such optimisation, this would likewise require the optimisation passes to record which addresses have been taken and cast to integer (or otherwise exposed) in eliminated code, to be explicitly passed in to alias analysis.

8.6 Missing arithmetic optimisations in PNVI

The erasure of provenance in the PNVI models, when casting to integer types, does fail to justify some compiler behaviour involving arithmetic optimisations. Consider the example in Figure 8.13[3], where the main function again follows from that of Figure 8.1. When the local objects are allocated in the appropriate order, the integer values `a` and `b` compare equal.

```
1  #include <stdint.h>
2  #include <stdio.h>
3  intptr_t foo(intptr_t a, intptr_t b) {
4    return (a==b)?b:a;
5  }
6
7  int main(void) {
8    int x=0, y=0;
9    intptr_t a = (intptr_t)(&x+1);
10   intptr_t b = (intptr_t)&y;
11   if (a==b) {
12     intptr_t c=foo(a,b);  // a
13     int *r = (int*)c;
14     *r = 42;
15     printf("y=%d\n",y);
16   }
17 }
```

Figure 8.13: pointer_from_integer_gil_1.c

Compiling with GCC at optimisation level -O2 gives a program that outputs `y=0`. Presumably, in the body of `foo`, the occurrence of `b` in the "then" branch of the conditional operator is replaced by `a`, based on the equality. Then, because the two branches of the conditional operator are now the same, the conditional itself is removed, leaving only `a` as the body of the function. These are integer optimisations, which are perfectly sound in that domain. This is then inlined at line 12, and the pointer resulting from the cast at line 13 is deemed to not alias with `y`, presumably because it is derived from an integer value constructed from the address of `x`. We conjecture that it would be a reasonable restriction on GCC to forbid making such non-aliasing assumptions for pointers resulting from an integer-to-pointer cast.

[3]Discussion with Chung-Kil Hur.

Chapter 9

Memory object model: detailed semantics

In the previous chapter, we sketched four memory object models based on a notion of provenance: PVI, where both pointer and integer values have a provenance; and PNVI-plain, PNVI-ae, and PNVI-ae-udi; where only pointer values do. We now give a formal presentation of these models. This chapter is based on [N2364], a working paper first presented to WG14, which in turn contains manually typeset mathematics describing the implementation of the memory models in Cerberus.

The definitions of the four models share much of their structure. In particular, the PNVI-ae and PNVI-ae-udi variants mostly extend the base definitions of PNVI-plain. To avoid needless repetition, we give a fusion of the definitions of the four models, and colour-code the parts only present in some: we write the common base in black font; the parts present only for PNVI-ae and PNVI-ae-udi in blue; the parts present only for PNVI-ae-udi in purple; and when constructing integer values, we write the provenance component which is only present for PVI in orange.

In Chapter 5, we presented the memory interface used by the dynamics of Core, which declares the types of some of its values (integer, floating, pointer and memory values), and the dynamics of its memory actions. The memory models we present here are all implementations of that interface.

9.1 Implementation of pointer, integer and memory values

Recall that the memory interface keeps the types of integer, pointer, and memory values abstract. The motivation for this design choice is that the "ghost information" carried by provenances for pointer values (and integer values in PVI) is not meant to be observable outside of the memory model. Giving the values abstract types prevents us from burdening the dynamics of Core.

A **provenance** π is either: `@empty`, corresponding to the lack of provenance; $@i$, referring to a particular memory allocation by its ID i; or, for PNVI-ae-udi only, a symbolic ι. This variant is used to deal with the ambiguity that may arise when building a pointer value by casting from a integer value whose numeric value can both be interpreted as being one-past an object, and pointing to the beginning of a second, adjacent object.

A **pointer value** p is then either: a `null` pointer; a pair (π, a), where a is a concrete numeric *address* in $\mathbb{Z}$; or `funptr`(*ident*), a pointer to a function.

In the PVI model, an **integer value** is similarly a pair (π, n) where $n \in \mathbb{Z}$, while in the PNVI-* models, it is simply the numeric component.

The addresses of pointers are modelled as unbounded integers; the semantics of operations constructing pointer values enforces the bounded arithmetic arising from the implementation-defined size of pointer type of the particular C implementation given as a parameter to Cerberus. Likewise, the integer values presented here are unbounded integers used in the semantics of Core, and do not directly correspond to their C counterparts. It is the elaboration from C to Core which deals with modelling the bounded arithmetic and the various sizes of C's integer values.

Finally, the type of **memory values** closely follows the structure of C types, a value v is either: an unspecified value `unspecified`(τ) of type τ; an integer, floating, or pointer value; an array value `array`$(v_1, \ldots, v_n)$; a struct value (`struct` T)$\{.x_1 = v_1, \ldots, x_n = v_n\}$; or a union value (`union` T)$\{.x = v\}$.

9.2 The memory state

All four models share the same structure for their state: a tuple (A, S, M), where the first component embodies the abstract view of the state, tracking the collection of *allocations*, the second component is only used by the PNVI-ae-udi model, and the third component holds a concrete representation of state as an array of bytes. An allocation corresponds to either an object in ISO standard parlance (arising from an identifier declaration in the C source), or a region (the result of allocating using a memory management function, e.g. `malloc()`).

Abstract state The component A is a partial map relating allocation IDs to a tuple holding the parameters of the allocation:

$$A : \mathit{allocation\_id} \rightharpoonup \mathit{allocation}$$

Allocations are tuples $(n, \tau_{\text{opt}}, a, l, f, k, t)$ whose components are the following:

- $n \in \mathbb{N}$ is the size of the allocation in bytes.
- The optional τ is a C type. For objects, this the type of the corresponding C identifier declaration, for regions there is no such type information, and this component instead holds `none`.
- a is the numeric base address of the allocation.
- $l \in \{\texttt{alive}, \texttt{killed}\}$ indicates whether the allocation is alive (has not reached the end of its lifetime).
- $f \in \{\texttt{readWrite}, \texttt{readOnly}\}$ is the access permission. Most objects or regions are mutable and have the first variant, but for example `const`-qualified objects have `readOnly` once initialised.

- $k \in \{\texttt{object}, \texttt{region}\}$ is the *kind* of the allocation, indicating whether it resulted from a C identifier declaration or a memory management function.
- The last component only exists for the PNVI-ae and PNVI-ae-udi models; it is the *taint flag* $t \in \{\texttt{unexposed}, \texttt{exposed}\}$ indicating whether the allocation is to be considered when an integer-to-pointer cast is attempting to recover a provenance.

In PNVI-ae-udi, the component S is a partial map relating symbolic provenances ι to sets of one or two allocation IDs.

Concrete state The component M is a partial map from numeric addresses to abstract bytes, which are triples made of:

- a provenance π;
- either a concrete byte b (an 8-bit numeric value), or `unspec`, indicating the abstract byte has not yet been initialised or that it is a padding byte;
- and an optional integer index j (we write `none` in its absence).

The last component is only present in the PNVI-* models, as it is used to indicate that a value of the abstract byte holds a representation byte from a pointer value (j indicates the index of the byte within the representation bytes of the pointer). As we shall see, this is used by the models to distinguish between loads of pointer values that were written as whole-pointer writes vs. those that were written byte-wise or in some other way. Note that addresses that M does not map to anything are those that are not currently reserved by the allocator.

9.2.1 Relating abstract values to their concrete representation

Recall from Chapter 5 that the memory interface operates over abstract memory values, leaving their representations opaque to Core. Here, however, the implementation of memory actions (e.g. `store`() and `load`()) operates over the concrete side of the memory state, and will need to relate values and sequences of abstract bytes. For this purpose, we define two functions: $\text{repr}(v)$, mapping a memory value into its concrete representation as a sequence of abstract bytes; and $\text{abst}(A, S, \tau, bs)$, which is a partial function attempting to interpret part of the sequence bs as a value of C type τ.

Combining provenances When a sequence of abstract bytes is interpreted by abst() as a pointer value, we need to produce a provenance for the value from the provenances of the individual bytes. If all the bytes have the `@empty` provenance, so does the value; if two bytes have different provenances of the form $@i$ (or ι for the PNVI-ae-udi model), the value has the `@empty` provenance. In all other cases, all the bytes with a non-empty provenance have the same provenance, which is the one we use for the value.

$$\text{combine\_prov}(\pi_1, \ldots, \pi_n) = \begin{cases} \pi & \exists k \in \{1, \ldots, n\}.\, \pi_k = \pi \\ & \text{and, } \forall k \in \{1, \ldots, n\}.\, \pi_k = \pi \ \vee \ \pi_k = \texttt{@empty} \\ \texttt{@empty} & \text{otherwise} \end{cases}$$

Intuitively, this allows C programs to manipulate the representation bytes of pointers using an integer type (e.g. through a `unsigned char`*), as long as at least one byte is left

unchanged (so the original provenance is not totally lost) and that representations of two different pointers are never mixed.

Representation function The repr() function is defined by induction over the structure of v, as follows:

- The base cases are specified values with scalar type (e.g. integer, floating, and pointer values), and unspecified values. When applied to an unspecified value, the resulting sequence is exclusively made of abstract types of the form (@`empty`, `unspec`, `none`). In PNVI-* models, specified integer values are mapped to sequences composed of similar abstract bytes except for their second components, which instead contain the two's complement concrete representation of the integer. For the PVI model, where integer values have a provenance, the first components of the abstract bytes hold the provenance of the pointer value the integer values are constructed from. Null pointer values are mapped to sequences composed of abstract bytes (@`empty`, 0, `none`). Other pointer values are mapped to sequences where the first components hold their provenance, the second components hold the two's complement encoding of their concrete address, and the third components hold the index of the enclosing abstract byte within the sequence (hence they go from 0 to $\text{sizeof}(\tau*) - 1$, where τ is the referenced type of the pointer). In all models, floating values are mapped to sequences of bytes similar to those of integer values in PNVI-* models, save for the second components, where the two's complement-encoded bytes are replaced by the appropriate encoding.

- For array values, the resulting sequence of bytes is the concatenation of inductively applying the function to its elements.

- For struct/union values, it is similar, with the addition that, as required by the ABI being modelled, the sequences of their members may be separated by sequences of padding bytes. We encode padding bytes like the bytes of unspecified values.

Because the integer values defined by the memory interface are unbounded, the repr() function is only partially defined. It is by construction of the elaboration function that we ensure that no Core program execution leads to repr() being applied to a non-representable value.

Abstraction function The *abst*() function takes four parameters: the abstract state components A and S, a C type τ with known size (i.e. not **`void`**, nor an array type of unknown size, nor a function type); and a sequence of abstract bytes bs. The function is only defined when the length of the sequence is no smaller than $\text{sizeof}(\tau)$. In these cases, we have $bs = bs_1 \cdot bs_2$, where $|bs_1| = \text{sizeof}(\tau)$, and the function returns a tuple of a memory value (the result of interpreting bs_1 as a value of C type τ), a set of allocation IDs, a potentially updated version of S, and the unconsumed sequence bs_2. In the PVI and PNVI-plain models, the set of allocation IDs is always empty. It is only in PNVI-ae-udi that that function may return an updated version of S. It is defined by induction over the structure of τ, as follows:

- As for the previous function, the base cases are the scalar types for which a base value (as opposed to an aggregate one) is constructed:

  - If any of the abstract bytes being consumed have `unspec` as their second component, the first component of the returned tuple is `unspecified`(τ).
  - Otherwise, depending on whether τ is an integer, floating, or pointer type, the numeric part of the returned value is computed from the second components of the abstract bytes (i.e. applied to the two's complement encoding for integers). In the PVI model, the provenance of the returned value is the result of applying combine_prov() to the provenances of the abstract bytes. In the PNVI-ae-* models, when producing an integer value, the second component of the returned tuple is the set of allocation IDs found in abstract bytes with a non-empty provenance.
  - For pointers, if the numeric address is zero, the resulting value is a null pointer and has the @`empty` provenance.
  - If the numeric address corresponds to the address of a function, the resulting value is a function pointer to that function.
  - For pointers with a non-zero address the PVI and PNVI models differ. In PVI, the first component of the returned tuple is a pointer value whose provenance is constructed in the same way as for integer values. In the PNVI-* models, if the third components of the bytes all carry the appropriate index (namely $0, \ldots, \text{sizeof}(\tau) - 1$), then the provenance of the returned pointer value is the provenance of the first byte (as a result of the semantics of the `store`(), for implementations where all pointer types have the same size, all the bytes have the same non-empty provenance). Otherwise, the abstract component of the memory state is examined to find whether a live (and exposed in the case of the PNVI-ae-* models) allocation exists with a footprint containing the pointer value that is being constructed. If there is such an allocation, its ID is used for the provenance of the pointer value; otherwise, the empty provenance is used. Additionally, in the PNVI-ae-* models, if the abstract component of the state has to be examined, the only allocations which are considered are the ones which are marked as exposed. Furthermore, in the PNVI-ae-udi model, if there are two live and exposed allocations whose footprints contain the address of of the pointer value being constructed, then a fresh symbolic provenance is created and used for the resulting value. The third component of the result records this fresh symbolic provenance and its mapping to the IDs of the two allocations by updating S.

- In the case where τ is an array type of size n, $abst()$ progressively consumes the sequenced bs by calling itself n times with the element type of the array as its first argument. It then returns the array value combining the value component of its recursive calls, and the unconsumed sequence for its last call. In the PNVI-ae-* models, the returned set of allocation IDs is the union of the sets returned by the recursive calls.

- The case where τ is a struct type is similar to the previous one, with recursive calls for each members.

9.3 Dynamics of memory actions and operations

In Chapter 5, we presented the set of memory actions and operations declared by the memory interface to Core. The various models we now present, being implementations of this interface, provide definitions for these actions and operations. Formally, we express their dynamics as a labelled transition system relating a memory state to either the updated memory state (for successful transitions), or UNDEF (for transitions deemed as having undefined behaviour, e.g. a memory action attempted with an out-of-bound pointer value):

$$(A, S, M) \xrightarrow{\text{LABEL}} (A', S', M') \qquad \text{and} \qquad (A, S, M) \xrightarrow{\text{LABEL}} \text{UNDEF}$$

The labels hold the name, arguments, and (in the case of successful transitions) potential value of the action or operation performed by a transition. For example, the successful transition for a load action with type τ, on a pointer value p, reading a value v while resulting in an update of the state, is written:

$$(A, S, M) \xrightarrow{\text{load}(\tau,p)=v} (A', S', M')$$

For clarity, as the preconditions to some of the transitions are quite large, we present the semantics of actions as rules of the form:

$$\frac{[\text{LABEL} : \textbf{action}(arg_1, \ldots, arg_n) = v] \quad precond_1 \ldots precond_m}{(A, S, M) \to (A', S', M')}$$

We write the update of a map A, mapping i to a new result u, as:

$$A[i \mapsto u] \triangleq \lambda z. \begin{cases} u & \text{if } i = z \\ A(z) & \text{otherwise} \end{cases}$$

We overload this notation in the usual way for multiple updates. For updates to the bytemap M, we also overload it as follows to write updates of ranges of addresses:

$$M[a_1, \ldots, a_n \mapsto bs] \triangleq \lambda z. \begin{cases} bs[i] & \text{if } z = a_i \\ M(a) & \text{otherwise} \end{cases}$$

We introduce the notation $\text{select}(i, S, z.\, X)$ to denote $\exists z \in S.\, (i = z \wedge X)$ (where z may occur in X). If we require an unique existential quantifier, we write select_uniq() instead.

We now present the successful transitions of actions and operations, followed by the undefined cases. In the rules for successful transitions, we annotate in red the preconditions whose failure would result in an undefined behaviour. We refer to these annotations when listing the undefined behaviours in the second subsection.

9.3.1 Defined reductions

Allocating an object or a region Recall from Chapter 5, that the memory interface declares two actions for creating new allocations:

- **allocate_object**(), modelling the beginning of the lifetime of the memory objects associated with the declaration of a C identifier;

- **allocate_region**(), modelling a call to a memory management function from the standard library, on e.g. `malloc()`.

The former, depending its third argument, may also initialise the object it allocates; this is used in the elaboration of string literals (which are not modifiable).

The semantics of the two actions are broadly the same: they choose a fresh allocation ID i; then choose an address a for the new allocation which satisfies the alignment constraint given to both actions as their first argument al, and such that a region starting from a and of size n (as derived from their second argument) does not contain the address 0 (reserved for the null pointer), nor overlaps with any of the live allocations already present in A[1].

$$\text{newAlloc}(A, al, n) = \left\{ a \;\middle|\; \begin{array}{l} a \equiv 0 \pmod{al} \wedge \\ 0 \notin [a \mathinner{..} a + n - 1] \wedge \\ \left(\begin{array}{l} \forall i\ n'\ a'.\ (A(i) = (n', \_, a', \_, \_, \_)\ \Rightarrow \\ \quad [a \mathinner{..} a + n - 1] \cap [a' \mathinner{..} a' + n' - 1] = \emptyset) \end{array} \right) \end{array} \right\}$$

This definition allows for the new object or region to not have a valid one-past pointer.

The pointer value resulting from the allocation action is then $(@i, a)$. Both actions update the abstract component of the memory A with a new mapping for i to an allocation tuple, which has the kind component (`object` or `region`) set as appropriate, and for the PNVI-ae-* models, has the taint component set to `unexposed`.

For **allocate_object**(), the allocation tuple has: its size component, calculated from the representation size of the C type τ it is given as second argument; its type component set to τ; and its permission component set according to the third argument of the action. If the third argument is `readOnly`(v), the concrete component of the memory is updated to contain the representation of the value v in the bytes of new allocation. Otherwise, it is `readWrite`, and the concrete component of the memory is updated to contain a sequence of (`@empty`, `unspec`, `none`) bytes. The allocate_region() action only differs by setting its size component directly from its second argument n, and always setting its permission component to `readWrite`.

$$[\text{LABEL} : \textbf{allocate\_object}(al, \tau, \texttt{readWrite}) = p]$$

$$\text{ALLOC-RW}\ \frac{\begin{array}{c} n = \mathsf{sizeof}(\tau) \qquad i \notin \text{dom}(A) \qquad {}^{(1)}a \in \text{newAlloc}(A, al, n) \\ p = (@i, a) \qquad M' = M[a \mathinner{..} a + n - 1 \mapsto (@\texttt{empty}, \texttt{unspec}, \texttt{none})] \end{array}}{(A, S, M) \to (A[i \mapsto (n, \tau, a, \texttt{alive}, \texttt{readWrite}, \texttt{object}, \texttt{unexposed})], S, M')}$$

$$[\text{LABEL} : \textbf{allocate\_object}(al, \tau, \texttt{readOnly}(v)) = p]$$

$$\text{ALLOC-RO}\ \frac{\begin{array}{c} n = \mathsf{sizeof}(\tau) \qquad i \notin \text{dom}(A) \qquad {}^{(1)}a \in \text{newAlloc}(A, al, n) \\ p = (@i, a) \qquad \texttt{Some}(bs) = \text{repr}(v) \\ M' = (M[a \mathinner{..} a + n - 1] \mapsto bs) \end{array}}{(A, S, M) \to (A[i \mapsto (n, \tau, a, \texttt{alive}, \texttt{readOnly}, \texttt{object}, \texttt{unexposed})], S, M')}$$

$$[\text{LABEL} : \textbf{allocate\_region}(al, n) = p]$$

$$\text{ALLOC-REGION}\ \frac{\begin{array}{c} i \notin \text{dom}(A) \qquad {}^{(1)}a \in \text{newAlloc}(A, al, n) \\ p = (@i, a) \qquad M' = M[a \mathinner{..} a + n - 1 \mapsto (@\texttt{empty}, \texttt{unspec}, \texttt{none})] \end{array}}{(A, S, M) \to (A[i \mapsto (n, \texttt{none}, a, \texttt{alive}, \texttt{readWrite}, \texttt{region}, \texttt{unexposed})], S, M')}$$

[1] Note that this non-deterministic choice of the address makes the semantics of these actions admit any implementation allocation scheme.

Ending a lifetime Calling the **kill**() action results in a successful deallocation, if its pointer value operand holds a valid provenance to a live allocation, and such that its numeric component is the starting address of the allocation. The abstract component of the state is updated so that corresponding allocation has `killed` as its fourth component. Recall that the **kill**() action takes a second argument denoting whether we are dealing with the deallocation of an object created by the declaration of a C identifier (i.e. the action was produced by the elaboration function for the end or exit of a C block), or with the deallocation of a heap allocated region (i.e. the action was produced by the elaboration of a call to a memory management function such as `free()`). As a sanity check, we also require that this second argument match the kind component of the allocation being allocated. By construction of the elaboration function, Cerberus should never perform a kill invalidating this check.

The constraint requiring the numeric component of the pointer to coincide with the start of the object or region being deallocated comes from a requirement in the specification of the `free()` function in the ISO standard (C11, §7.22.3.3p2). For deallocation of objects, the construction of the elaboration function also guarantees that this holds true.

The ISO standard also requires calls to the `free()` function on a null pointer to succeed (with no effect on the memory state). Again, by construction of the elaboration function, Cerberus should not perform such calls to the **kill**() action when the kind argument is set to `object`. As a sanity check we disallow these cases.

$$\frac{[\text{LABEL} : \mathbf{kill}_k(p)] \qquad \overset{(1)}{p = (@i, a)} \quad \overset{(2)}{A(i) = (n, \tau_{\text{opt}}, a_i, \texttt{alive}, f, k_i, t)} \quad \overset{(3)}{a = a_i} \quad k = k_i}{(A, S, M) \to (A[i \mapsto (n, \tau_{\text{opt}}, a_i, \texttt{killed}, f, k_i, t)], S, M)}\ \text{KILL}$$

$$\frac{[\text{LABEL} : \mathbf{kill}_{\texttt{region}}(p)] \qquad \overset{(1)}{p = \texttt{null}}}{(A, S, M) \to (A, S, M)}\ \text{KILL-NULL}$$

For the PNVI-ae-udi model, when the pointer value operand has a symbolic provenance, because the definition of the newAlloc() function prevents allocations from overlapping, in the set of allocation IDs mapped to the symbolic provenance, at most one of them can have the base address of its allocation equal to that of the pointer. In this case, the transition of the **kill**() updates the abstract component of the memory state by collapsing the mapping of the symbolic provenance to that single allocation ID, and marks that allocation as killed.

$$\frac{[\text{LABEL} : \mathbf{kill}_k(p)] \qquad \overset{(1)}{p = (\iota, a)} \quad \overset{(4)}{\text{select\_uniq}}\left(i, S(\iota), z.\ \begin{array}{l} A(z) = (n, \tau, a_i, \texttt{alive}, f, k_i, t) \wedge \\ a = a_i \ \wedge \ k = k_i \end{array}\right)}{(A, S, M) \to (A[i \mapsto (n, \tau, a_i, \texttt{killed}, f, k', t)], S[\iota \mapsto \{i\}], M)}\ \text{KILL-IOTA}$$

Finally, the ISO standard specifies that when the lifetime of an object ends, all pointers that point to that object have their value made indeterminate (C11, §6.2.4p2, sentence 4). As discussed in Chapter 2, this does not match some well established practice, for

example some concurrent algorithms do not work in such a "zapping" semantics [N2369; P1726R4]. By default, our models do not implement this aspect of the standard, and we instead have a `zap_dead_pointers` switch to enable it when desired. When enabled, the semantics of the kill() simply changes with an additional update of M, the concrete component of the state to zap_pointers(A, S, S', M, i)[2], where:

$$\text{zap\_pointers}(A, S, M, i) = \lambda a. \begin{cases} (@\texttt{empty}, \texttt{unspec}, \texttt{none}) & \text{if } \exists j. \; A(j) = (n, \tau, a', \texttt{alive}, \_, \texttt{object}, \_) \wedge \\ & \quad a \in [a' \,..\, a' + n - 1] \wedge \\ & \quad (@k, \_, S', \_) = \text{abst}(A, S, \tau, M[a' \,..\, a' + n - 1]) \wedge \\ & \quad i = k \\ M(a) & \text{otherwise} \end{cases}$$

and where S is replaced by S'.

Loading In the PVI and PNVI-plain models, a load of type τ succeeds if: its pointer value as a non-empty provenance(1) referring to a live allocation(2a); its footprint, as deriving from the pointer address a and sizeof(τ), is within that of the allocation(2b); and, all the bytes in M within the footprint of the load can be successfully interpreted as a value v of type τ using the abstraction function(3). In these models, the transition resulting from a successful load does not change the memory state.

$$\text{bounds\_check}_{\text{load}}(a, n, i, A) \triangleq {}^{(2a)}A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \wedge {}^{(2b)}[a \,..\, a + n - 1] \subseteq [a_i \,..\, a_i + n_i - 1]$$

$$\text{expose}(A, I) \triangleq \lambda i. \begin{cases} (n, \tau, a, \texttt{alive}, f, k, \texttt{exposed}) & i \in I \wedge A(i) = (n, \tau, a, \texttt{alive}, f, k, \_) \\ A(i) & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{l} [\text{LABEL} : \mathbf{load}(\tau, p) = (v, fp)] \\ {}^{(1)}p = (@i, a) \qquad {}^{(2)}\text{bounds\_check}_{\text{load}}(a, \text{sizeof}(\tau), i, A) \\ {}^{(3)}(v, I_{tainted}, S', []) = \text{abst}(A, S, \tau, M[a \,..\, a + \text{sizeof}(\tau) - 1]) \\ fp = \mathbf{R}(a, \text{sizeof}(\tau)) \qquad A' = \begin{cases} \text{expose}(A, I_{tainted}) & \text{is\_integer}(\tau) \\ A & \text{otherwise} \end{cases} \end{array}}{(A, S, M) \rightarrow (A', S', M)}$$

In the PNVI-ae-* models, the abstraction function collects a set $I_{tainted}$ of the allocation IDs referred in the provenances of the abstract bytes it is given to interpret. For loads where τ is an integer type, if this set is not empty, we are dealing with a load which is reading the representation of a pointer from a non-pointer type, e.g. the load comes from dereferencing a `char*` pointer on the address of an object, or a type punning using a union having a `uintptr_t` and a pointer member. In this case, the allocations whose IDs are in the set must now be considered *exposed*, and the abstract component of the memory state is updated accordingly using the expose() function.

[2]Note that as Cerberus only models implementations with no trap representations (except for the type `_Bool`), the notion of indeterminate value collapses to that of unspecified value.

In the PNVI-ae-udi model, the pointer value may have a symbolic provenance (which is mapped in the abstract state to one or two allocation IDs). In this case, the load succeeds if one of the allocation IDs satisfies the bounds check(2). Furthermore, the abstract state is updated such that the mapping of the symbolic provenance is now collapsed to the satisfying allocation ID. Note that because the definition of newAlloc() ensures that allocations do not overlap, in the case where a symbolic provenance is mapped to two allocation IDs, at most one of them will satisfy the bounds check.

$$\frac{\begin{array}{c}[\text{LABEL} : \mathbf{load}(\tau, p) = (v, fp)] \\ {}^{(1)}p = (\iota, a) \\ {}^{(2)}\text{select}\ (i, S(\iota), z.\ \text{bounds\_check}_{\text{load}}(a, \text{sizeof}(\tau), z, A)) \\ {}^{(3)}(v, I_{tainted}, S', []) = \text{abst}(A, S, \tau, M[a\ ..\ a + \text{sizeof}(\tau) - 1]) \\ fp = \mathbf{R}(a, \text{sizeof}(\tau)) \qquad A' = \begin{cases} \text{expose}(A, I_{tainted}) & \text{is\_integer}(\tau) \\ A & \text{otherwise} \end{cases}\end{array}}{(A, S, M) \to (A', S'[\iota \mapsto \{i\}], M)}$$

Storing The preconditions for successful stores are the same as for successful loads, with the addition that the allocation being accessed must be writable(2b). The concrete state is updated such that the bytes within the footprint of the store now hold the representation of the value being stored. If the store is indicated as locking (which is the case for the initialisation of a `const`-qualified object) by the flag b, the mapping of the abstract state for the allocation referred to by the pointer value is updated such that its access permission field is set to `readOnly` (in the rules we write such an update for allocation i as lock_alloc(i, A)).

$$\text{bounds\_check}_{\text{store}}(a, n, i, A) \triangleq {}^{(a)}A(i) = (n_i, \_, a_i, \texttt{alive}, {}^{(b)}\texttt{readWrite}, \_, \_) \wedge {}^{(c)}[a\ ..\ a + n - 1] \subseteq [a_i\ ..\ a_i + n_i - 1]$$

$$\frac{\begin{array}{c}[\text{LABEL} : \mathbf{store}_b(\tau, p, v) = fp] \\ {}^{(1)}p = (@i, a) \qquad {}^{(2)}\text{bounds\_check}_{\text{store}}(a, \text{sizeof}(\tau), i, A) \\ fp = \mathbf{W}(a, \text{sizeof}(\tau)) \qquad A' = \begin{cases} \text{lock\_alloc}(i, A) & \text{if } b = \texttt{is\_locking} \\ A & \text{otherwise} \end{cases} \\ \texttt{Some}(bs) = \text{repr}(v)\end{array}}{(A, S, M) \to (A', S, M[a\ ..\ a + \text{sizeof}(\tau) - 1] \mapsto bs)}$$

In the PNVI-ae-udi model, the case where the pointer value has a symbolic provenance also follows from the corresponding case for load accesses.

$$\frac{\begin{array}{c}[\text{LABEL} : \mathbf{store}_b(\tau, p, v) = fp] \\ {}^{(1)}p = (\iota, a) \qquad {}^{(2)}\text{select}\ (i, S(\iota), z.\ \text{bounds\_check}_{\text{store}}(a, \text{sizeof}(\tau), z, A)) \\ fp = \mathbf{W}(a, \text{sizeof}(\tau)) \qquad A' = \begin{cases} \text{lock\_alloc}(i, A) & \text{if } b = \texttt{is\_locking} \\ A & \text{otherwise} \end{cases} \\ \texttt{Some}(bs) = \text{repr}(v)\end{array}}{(A, S.M) \to (A, S[\iota \mapsto \{i\}], M[a\ ..\ a + \text{sizeof}(\tau) - 1] \mapsto bs)}$$

Pointer subtraction Following the ISO standard (C11, §6.5.6p9), the subtraction operator between two pointer values is defined when both pointers point within the same object, and is undefined otherwise. Its result is an integer value counting the difference between the indices of the elements of the array object being pointed to by the two pointers[3].

In our provenance-based models, the defined case corresponds to the subtraction of two pointers of the form $(@i_1, a_1)$ and $(@i_2, a_2)$ with equal allocation IDs ($i_1 = i_2$) such that this ID is mapped in the abstract state to a live allocation, and the addresses of both pointers are within or one-past the footprint of the allocation. Note that in the strict ISO semantics, the pointer arithmetic operators are undefined as soon as they would result in out-of-bound pointers. As as result, the last condition is always satisfied. However as the survey we discuss in Chapter 2 shows (see Question 9), it is debatable whether such a strict semantics matches real world practice. In our models we therefore opted to be more permissive by making the arithmetic operators fully defined. As a result, a check is necessary here to ensure that a common object is pointed to by both pointers.

The result is the numerical difference between the address of the two pointers divided by sizeof(dearray(τ)), where:

$$\text{dearray}(\tau) = \begin{cases} \tau_{\text{elem}} & \tau = \tau_{\text{elem}}[\_] \\ \tau & \text{otherwise} \end{cases}$$

In the PNVI-* models, the returned value is simply that pure integer, and for PVI the returned integer value has @`empty` provenance.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{diff\_ptrval}(\tau, p_1, p_2) = x] \\ {}^{(1)}p_1 = (@i_1, a_1) \qquad {}^{(2)}p_2 = (@i_2, a_2) \qquad {}^{(3)}i_1 = i_2 \\ {}^{(4)}A(i_1) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \qquad {}^{(5)}a_1, a_2 \in [a_i \mathinner{..} a_i + n_i] \\ x = (@\texttt{empty}, (a_1 - a_2)/\text{sizeof}(\text{dearray}(\tau)))\end{array}}{(A, S, M) \to (A, S, M)}$$

In PNVI-ae-udi, the symbolic variant of provenances gives rise to four additional cases where the subtraction is also defined. In the first two cases, one of the pointers is like before of the form $(@i, a_1)$, while the other has a symbolic provenance (ι, a_2)[4]. We then require i to be within the set of provenances associated to ι in the abstract state. The remaining requirements on the existence of a live allocation associated to i and on the addresses are the same as before. However unlike the previous which was transitioning into an unchanged state, here if the ι is associated to a set of two allocation IDs in the original abstract state (that is, (ι, a_2) is an ambiguous pointer), in the resulting abstract state that set is now collapsed to the singleton $\{i\}$. Intuitively, by finding a well defined condition for the subtraction, the memory model now resolves the previous ambiguity.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{diff\_ptrval}(\tau, p_1, p_2) = x] \\ {}^{(1)}p_1 = (@i, a_1) \qquad {}^{(2)}p_2 = (\iota, a_2) \qquad {}^{(3)}i \in S(\iota) \\ {}^{(4)}A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \qquad {}^{(5)}a_1, a_2 \in [a_i \mathinner{..} a_i + n_i] \\ x = (a_1 - a_2)/\text{sizeof}(\text{dearray}(\tau))\end{array}}{(A, S, M) \to (A, S[\iota \mapsto \{i\}], M)}$$

[3]The ISO standard (C11, §6.5.6p7) specifies that the in context of "additive operators" (of which the pointer subtraction is part of), non-array objects are to be seen as arrays of size one.

[4]We omit the rule for the symmetric variant to this case where p_1 has a symbolic provenance, and p_2 the concrete one.

In the third case, the two pointers are ambiguous: they both have symbolic provenance, to which the abstract state associates a set of two allocation IDs. Here we require these two sets to be equal, and the requirements on the existence of a live allocation and on the addresses of the pointers must hold for both allocation IDs (note that this can only happen when $a_1 = a_2$, in which case $x = 0$). In this case no ambiguity is being resolved, and the abstract state is left unchanged by the transition.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{diff\_ptrval}(\tau, p_1, p_2) = x] \\ \begin{array}{lll} {}^{(1)}p_1 = (\iota_1, a_1) & {}^{(2)}p_2 = (\iota_2, a_2) & {}^{(3)}S(\iota_1) = S(\iota_2) = \{i_1, i_2\} \\ {}^{(4_1)}A(i_1) = (n_{i_1}, \_, a_{i_1}, \texttt{alive}, \_, \_, \_) & & {}^{(5_1)}a_1, a_2 \in [a_{i_1} \mathinner{..} a_{i_1} + n_{i_1}] \\ {}^{(4_2)}A(i_2) = (n_{i_2}, \_, a_{i_2}, \texttt{alive}, \_, \_, \_) & & {}^{(5_2)}a_1, a_2 \in [a_{i_2} \mathinner{..} a_{i_2} + ni_2] \end{array} \\ x = 0 \end{array}}{(A, S, M) \to (A, S, M)}$$

In the fourth and final defined case, the two pointers have symbolic provenance but this time the abstract state associates their ι to sets intersecting to a singleton. This is similar to the first two cases, but this time the ambiguity is resolved for the two pointers.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{diff\_ptrval}(\tau, p_1, p_2) = x] \\ {}^{(1)}p_1 = (\iota_1, a_1) \qquad {}^{(2)}p_2 = (\iota_2, a_2) \\ {}^{(3)}\left\{ j \;\middle|\; \begin{array}{c} S(\iota_1) \cap S(\iota_2) \;\wedge \\ {}^{(4)}A(j) = (n_j, \_, a_j, \texttt{alive}, \_, \_, \_) \;\wedge\; {}^{(5)}a_1, a_2 \in [a_j \mathinner{..} a_j + n_j] \end{array} \right\} = \{i\} \\ x = (a_1 - a_2)/\text{sizeof}(\text{dearray}(\tau)) \end{array}}{(A, S, M) \to (A[\iota_1, \iota_2 \mapsto \{i\}], S, M)}$$

Pointer relational operators The memory operator dealing with pointer relational operations has two variants. The first variant has the same strict requirements on the pointers as the pointer subtraction operator: the two pointers must have the same non-empty provenance referring to a live allocation whose footprint contains the addresses of both pointers. When these requirements are satisfied, the result is the boolean resulting from applying the relational operator to the numeric addresses of the two pointers.

$$\text{STRCT\_RELOP}\ \frac{\begin{array}{c}[\text{LABEL} : \textbf{relop\_ptrval}(\odot, p_1, p_2) = b] \\ \begin{array}{ccc} {}^{(1)}p_1 = (@i_1, a_1) & {}^{(2)}p_2 = (@i_2, a_2) & {}^{(3)}i_1 = i_2 \\ {}^{(4)}A(i_1) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) & & {}^{(5)}a_1, a_2 \in [a_i \mathinner{..} a_i + n_i] \\ b = a_1 \odot a_2 & & \odot \in \{<, \leq, >, \geq\} \end{array} \end{array}}{(A, S, M) \to (A, S, M)}$$

In the PNVI-ae-udi model, the cases where either pointer has a symbolic provenance is handled similarly to the subtraction operator.

Pointer equality operators Both pointer equality operators are totally defined. Two pointer values compare equal if they are: both null pointers; pointers to the same function; or pointers to objects with the same provenance and address. If two pointer values to objects with the same address have different provenances, they non-deterministically compare equal or unequal. In all other cases, two pointer values compare unequal.

Additionally in the PNVI-ae-udi model, two pointers values with the same address compare equal if the two have symbolic provenances which are both mapped in the abstract state to the same singleton. All other pairs of pointer values such that one has a

symbolic provenance, non-deterministically compare equal or unequal. In all cases, the abstract state is left unchanged by the transition (no symbolic provenance gets resolved).

$$\frac{[\text{LABEL} : \textbf{eq\_\_ptrval}(p_1, p_2) = b] \quad \begin{cases} b = \texttt{true} & \text{if } p_1 = p_2 = \texttt{null} \\ b = (ident_1 = ident_2) & \text{if } p_1 = \texttt{funptr}(ident_1) \ \wedge \ p_2 = \texttt{funptr}(ident_2) \\ b = (a_1 =_{\mathbb{Z}} a_2) & \text{if } \left(\begin{array}{l} (p_1 = (\iota_1, a_1) \ \wedge \ p_2 = (\iota_2, a_2) \ \wedge \ S(\iota_1) = S(\iota_2) = \{i\}) \ \vee \\ (p_1 = (@i, a_1) \ \wedge \ p_2 = (@j, a_2) \ \wedge i = j) \end{array} \right) \\ b \in \{(a_1 =_{\mathbb{Z}} a_2), \texttt{false}\} & \text{if } \left(\begin{array}{l} \left(\begin{array}{l} p_1 = (\iota_1, a_1) \ \wedge \ p_2 = (\iota_2, a_2) \ \wedge \\ \neg(S(\iota_1) = S(\iota_2) = \{i\}) \end{array} \right) \ \vee \\ (p_1 = (\pi_1, a_1) \ \wedge \ p_2 = (\pi_2, a_2) \ \wedge \ \pi_1 \neq \pi_2) \end{array} \right) \\ b = \texttt{false} & \text{otherwise} \end{cases}}{(A, S, M) \rightarrow (A, S, M)}$$

As discussed in Chapter 8, the above non-determinism is necessary to account for the observable behaviour of current compilers. One might arguably opt for a simpler semantics (at the cost of some optimisation opportunities for compilers) where only the numeric component is compared.

$$\frac{[\text{LABEL} : \textbf{eq\_\_ptrval}(p_1, p_2) = b] \quad \begin{cases} b = \texttt{true} & \text{if } p_1 = p_2 = \texttt{null} \\ b = (ident_1 = ident_2) & \text{if } p_1 = \texttt{funptr}(ident_1) \ \wedge \ p_2 = \texttt{funptr}(ident_2) \\ b = (a_1 =_{\mathbb{Z}} a_2) & p_1 = (\_, a_1) \ \wedge \ p_2 = (\_, a_2) \\ b = \texttt{false} & \text{otherwise} \end{cases}}{(A, S, M) \rightarrow (A, S, M)}$$

Both models are implemented by Cerberus, and can be selected by respectively setting the switch STRICT_POINTER_EQUALITY to false or true.

Pointer array offset The array offset operator (used in the elaboration of pointer versus integer additive operators) has two variants. The first one follows the ISO standard, it is therefore potentially undefined and makes use of the abstract state to determine whether that is the case. It is defined when its pointer value operand is not `null`, has a non-empty provenance(1) referring to a live allocation(2), and is such that adding sizeof() of its C type operand to the address of the pointer value remains within the footprint of the allocation(3). The resulting pointer value preserves the provenance and has the address resulting from addition[5].

$$\text{ISO\_ARRAY} \ \frac{[\text{LABEL} : \textbf{iso\_\_array\_\_offset}(p, \tau, n) = p'] \quad \begin{array}{c} (1) p = (@i, a) \qquad (2) A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \\ a' = a + n * \text{sizeof}(\tau) \qquad (3) a' \in [a_i \ .. \ a_i + n_i] \\ p' = (@i, a') \end{array}}{(A, S, M) \rightarrow (A, S, M)}$$

In the PNVI-ae-udi model, there is again the case where the pointer value operand has a symbolic provenance. If the integer operand of the offset operator is different from

[5]Note that only the address of the resulting pointer is bounds checks (as opposed to its footprint) and therefore may be a one-past pointer to the allocation.

zero, the same bounds check as before is performed with respect to the allocation IDs associated to the symbolic provenance in the abstract state. If there are two allocation IDs, the bounds check must only succeed for one of them, and the abstract state is updated such that the symbolic provenance now only maps to the ID satisfying the check.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{iso\_array\_offset}(p, \tau, n) = p'] \\ n \neq 0 \\ {}^{(1)}p = (\iota, a) \qquad a' = a + n * \text{sizeof}(\tau) \\ {}^{(4)}\text{select\_uniq}\left(i, S(\iota), z.\ \begin{array}{l} A(z) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \wedge \\ a' \in [a_i \mathinner{..} a_i + n_i] \end{array}\right) \\ p' = (@i, a')\end{array}}{(A, S, M) \to (A, S[\iota \mapsto \{i\}], M)}\ \text{ISO\_ARRAY\_IOTA}$$

If the integer operand is zero and the symbolic provenance is mapped in the abstract state to two allocation IDs (hence the pointer value is ambiguous), then one of these IDs must refer to a live allocation such that the address of the pointer operand is within or one past that allocation. In this case, the result is the pointer operand left unchanged, and the abstract state is also not changed (even when both allocation IDs satisfy the bounds check).

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{iso\_array\_offset}(p, \tau, n) = p'] \\ {}^{(1)}p = (\iota, a) \qquad n = 0 \\ {}^{(4)}\exists i \in S(\iota).\ A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \ \wedge\ a \in [a_i \mathinner{..} a_i + n_i] \\ p' = p\end{array}}{(A, S, M) \to (A, S, M)}\ \text{ISO\_ARRAY\_IOTA\_ZERO}$$

The second variant of the operator is more permissive, and omits the bounds check.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{permissive\_array\_offset}(p, \tau, n) = p'] \\ {}^{(1)}p = (@i, a) \qquad p' = a + n * \text{sizeof}(\tau)\end{array}}{(A, S, M) \to (A, S, M)}\ \text{ARRAY}$$

In the PNVI-ae-udi model, it remains however that the operator is only defined if the pointer value operand is not ambiguous.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{permissive\_array\_offset}(p, \tau, n) = p'] \\ {}^{(1)}p = (\iota, a) \qquad {}^{(2)}|S(\iota)| < 2 \\ p' = a + n * \text{sizeof}(\tau)\end{array}}{(A, S, M) \to (A, S, M)}\ \text{ARRAY\_IOTA}$$

Pointer struct/union member offset This member offset operator is similar to the previous, but instead of adding some multiple of the size of a type, it adds to its pointer operand the offset associated to a particular member of a struct or union type. Again there are two variants. The first variant models the strict requirements of the ISO standard: the pointer value must have a non-empty provenance to a live allocation whose footprint contains the resulting pointer.

$$\frac{\begin{array}{c}[\text{LABEL} : \textbf{iso\_member\_offset}(p, \texttt{T}, .\texttt{x}) = p'] \\ {}^{(1)}p = (@i, a) \qquad {}^{(2)}A(i) = (n_i, \_, a_i, \texttt{alive}, \_, \_, \_) \\ {}^{(3)}a + \text{offsetof}(\texttt{T}, .\texttt{x}) \in [a_i \mathinner{..} a_i + n_i] \\ p' = (@i, a + \text{offsetof}(\texttt{T}, .\texttt{x}))\end{array}}{(A, S, M) \to (A, S, M)}$$

In the PNVI-ae-udi model, the case where the pointer value has a symbolic provenance is dealt like for the array offset operator.

The second variant is more permissive by removing the requirements on the pointer value. Furthermore, in the case where the pointer is null, the operator is defined and results in a pointer with empty provenance whose address is the offset of the member.

$$\frac{[\text{LABEL} : \textbf{permissive\_member\_offset}(p, \texttt{T}, \texttt{.x}) = p'] \quad {}^{(1)}p = (\pi, a) \qquad p' = (\pi, a + \text{offsetof}(\texttt{T}, \texttt{.x}))}{(A, S, M) \rightarrow (A, S, M)}$$

$$\frac{[\text{LABEL} : \textbf{permissive\_member\_offset}(p, \texttt{T}, \texttt{.x}) = p'] \quad {}^{(1)}p = \texttt{null} \qquad p' = (\texttt{@empty}, \text{offsetof}(\texttt{T}, \texttt{.x}))}{(A, S, M) \rightarrow (A, S, M)}$$

Casts between integer and pointer types In the PVI model, the pointer-to-integer cast operator simply converts null pointers to zero, and non-null pointers to an integer value holding the numeric address of the pointer (when that is within the range of the values representable in the integer type to which the cast is performed, otherwise flagging an undefined behaviour, as required by the ISO standard (C11, §6.3.2.3p6, sentence 2)). The provenance of the pointer is carried into the integer value.

$$\text{cast\_ptrval\_to\_ival}_{\texttt{PVI}}(\tau, p) = \begin{cases} (\texttt{@empty}, 0) & \text{if } p = \texttt{null} \\ (\texttt{@empty}, \text{addr\_of}(\mathit{ident})) & \text{if } p = \texttt{funptr}(\mathit{ident}) \\ (\pi, a) & \text{if } p = (\pi, a) \ \wedge \ a \in \text{range}(\tau) \\ \text{UNDEF} & \text{otherwise} \end{cases}$$

The integer-to-pointer cast operator simply does the reverse.

$$\text{cast\_ival\_to\_ptrval}_{\texttt{PVI}}(\tau, x) = \begin{cases} \texttt{null} & \text{if } x = (\texttt{@empty}, 0) \\ \texttt{funptr}(\mathit{ident}) & \text{if } x = (\texttt{@empty}, a) \ \wedge \ a = \text{addr\_of}(\mathit{ident}) \\ (\pi, a) & \text{if } x = (\pi, a) \end{cases}$$

In PNVI-plain, the pointer-to-integer cast operator is the same as in PVI, but this time the provenance of the pointer is discarded. The integer-to-pointer cast however needs to produce a provenance for the pointer value it returns. It does so by looking in the abstract component of the state for a live allocation whose footprint contains the address of the pointer value being constructed, taking its allocation ID for the provenance of the pointer value. In the PNVI-ae-* models, it is additionally required that the allocation be marked as exposed. If no such allocation exists, the pointer value has the empty provenance.

$$\frac{[\text{LABEL} : \textbf{cast\_ival\_to\_ptrval}_{\texttt{PNVI}}(\tau, x) = p] \quad x = 0 \qquad p = \texttt{null}}{(A, S, M) \rightarrow (A, S, M)}$$

$$\frac{[\text{LABEL} : \textbf{cast\_ival\_to\_ptrval}_{\texttt{PNVI}}(\tau, x) = p] \quad x = \text{addr\_of}(\mathit{ident}) \qquad p = \texttt{funptr}(\mathit{ident})}{(A, S, M) \rightarrow (A, S, M)}$$

$$\frac{[\text{LABEL} : \textbf{cast\_ival\_to\_ptrval}_{\texttt{PNVI}}(\tau, x) = p] \quad \begin{array}{c}\text{select}\,(i, \text{dom}(A), z.\, A(z) = (n, \_, a, \texttt{alive}, \_, \_, \texttt{exposed})) \\ x \in [a \,..\, a+n] \qquad\qquad p = (@i, a)\end{array}}{(A, S, M) \to (A, S, M)}$$

$$\frac{[\text{LABEL} : \textbf{cast\_ival\_to\_ptrval}_{\texttt{PNVI}}(\tau, x) = p] \quad \begin{array}{c}x \neq 0 \qquad\qquad (@\texttt{empty}, x) \\ \neg\,(\exists i.\, A(i) = (n, \_, a, \texttt{alive}, \_, \_, \texttt{exposed}) \;\wedge\; x \in [a \,..\, a+n-1]) \\ \neg\,(\exists \mathit{ident}.\, x = \text{addr\_of}(\mathit{ident}))\end{array}}{(A, S, M) \to (A, S, M)}$$

Additionally, in the PNVI-ae-* models, when the pointer-to-integer cast operator is called on a pointer value with a non-empty provenance referring to a live allocation, the abstract component of the memory state is updated to reflect that the allocation is now exposed.

$$\frac{[\text{LABEL} : \textbf{cast\_ptrval\_to\_ival}_{\texttt{PVNI}}(\tau, p) = x] \quad \begin{array}{c}p = (@i, a) \qquad x = a \qquad A(i) = (n_i, \tau_{\text{opt}}, a_i, f_i, k_{i,}) \\ a \in \text{range}(\tau)\end{array}}{(A, S, M) \to (A[i \mapsto (n_i, \tau_{\text{opt}}, a_i, \texttt{alive}, f_i, k_i, \texttt{exposed})], S, M)}$$

Finally, in the PNVI-ae-udi model, the bounds check on the live and exposed allocations differs slightly by allowing one-past pointers. And, as in the definition of the repr() function, if two allocations satisfy the bounds constraint, a fresh symbolic allocation is created and given to the returned pointer value. In the abstract component of the memory state, the allocation IDs of the two allocations are mapped to the symbolic provenance. Note that this can only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second.

$$\frac{[\text{LABEL} : \textbf{cast\_ival\_to\_ptrval}(\tau, x) = p] \quad \begin{array}{c}\text{select}\left(i_1, \text{dom}(A), z.\; \begin{array}{l}A(z) = (n_1, \_, a_1, \texttt{alive}, \_, \_, \texttt{exposed}) \;\wedge \\ x \in [a_1 \,..\, a_1 + n_1]\end{array}\right) \\ \text{select}\left(i_2, \text{dom}(A), z.\; \begin{array}{l}A(z) = (n_2, \_, a_2, \texttt{alive}, \_, \_, \texttt{exposed}) \;\wedge \\ x \in [a_2 \,..\, a_2 + n_2]\end{array}\right) \\ i_1 \neq i_2 \qquad \iota \notin \text{dom}(S) \qquad p = (\iota, x)\end{array}}{(A, S, M) \to (A, S[\iota \mapsto \{i_1, i_2\}], M)}$$

Operators over integers As a result of the type of integer values being kept abstract by the memory interface, the interface must also declare the arithmetic, relational, and equality operators over integers. For the PNVI-* models, their implementations are the expected operators over mathematical integers.

However for the PVI model, we need to define how provenances propagate over these operators. The relational and equality operators simply ignore the provenance of their operands:

$$\begin{array}{rcl}(\pi, n) =_{\text{ival}} (\pi', m) & = & (n = m) \\ (\pi, n) <_{\text{ival}} (\pi', m) & = & (n < m) \\ (\pi, n) \leq_{\text{ival}} (\pi', m) & = & (n \leq m)\end{array}$$

The subtraction operator results in a value with the empty provenance, except in the case where only its first operand operand has a provenance of the form $@i$. In this case that

provenance is propagated to the result:

$$(\pi, n) -_{\text{ival}} (\pi', m) \quad = \quad \begin{cases} (\texttt{@empty}, n - m), & \text{if } \pi = @i \text{ and } \pi' = @i', \\ & \text{regardless of whether } i = i'; \\ (@i, n - m), & \text{if } \pi = @i \text{ and } \pi' = \texttt{@empty}; \\ (\texttt{@empty}, n - m), & \text{if } \pi = \texttt{@empty}. \end{cases}$$

For all other integer operators in C, the resulting value has a non-empty provenance if and only if only one of the operands does, and it is the provenance of that operand which is used. For the operators having direct counterparts in Core (and therefore the memory interface), that is the additive and multiplicative operators, this is directly reflected in Core's operators:

$$\begin{aligned} \pi \oplus \pi' \quad &= \quad \begin{cases} \pi, & \text{if } \pi = \pi' \text{ or } \pi' = \texttt{@empty}; \\ \pi', & \text{if } \pi = \texttt{@empty}; \\ \texttt{@empty}, & \text{otherwise.} \end{cases} \\ (\pi, n) \odot_{\text{ival}} (\pi', m) \quad &= \quad (\pi \oplus \pi', n \odot m), \text{ where } \odot \in \{+, *, /, \text{rem\_t}, \text{rem\_r}\} \end{aligned}$$

For the bitwise shift operators, which are elaborated into Core using multiple operators, that property comes from the details of the elaboration. In particular, Core's exponentiation operator, which is used in their elaboration and has no counterpart in C, always results in a value with the empty provenance:

$$(\pi, n) \wedge_{\text{ival}} (\pi', m) \quad = \quad (\texttt{@empty}, n \wedge m)$$

9.3.2 Undefined reductions

The following undefined behaviours occur when preconditions from the reductions presented in the previous subsection fail:

- **allocate_object**() and **allocate_region**():
  - if in any (1) fails the call to newAlloc() returns an empty set, there is an "out of memory" error.
- **kill**():
  - if no instance of (1) is satisfied, the pointer is either null or function pointer.
  - if either (2) or (3) fail, by construction of the elaboration function this must be happening for a pointer to a region. If it is the former, there is a double `free()` undefined behaviour, otherwise the pointer being freed was not the result of an earlier call to a memory management function. Both of these cases are made undefined by (C11, §7.22.3.3p2).
  - if (4) fails in rule KILL_IOTA, one of the undefined behaviours from the previous bullet occurred.
- **load**():
  - if no instance of (1) is satisfied, the pointer is either null, a function pointer, or has empty provenance.

  - if (2a) fails, the pointer refers to a dead allocation.
  - if (2b) fails, the pointer is out of bounds.

- **store**():
  - if no instance of (1) is satisfied, the pointer is either null, a function pointer, or has empty provenance.
  - if (2a) fails, the pointer refers to a dead allocation.
  - if (2b) fails, the pointer refer to a non-modifiable object (e.g. one that resulted from a `const`-qualified declaration).
  - if (2c) fails, the pointer is out of bounds.

- **diff_ptrval**(), and **relop_ptrval**():
  - if no instance of (1), is satisfied, the left pointer is either null, function pointer or has empty provenance.
  - if no instance of (2), is satisfied, the right pointer is either null, function pointer or has empty provenance.
  - if no instance of (3), is satisfied, the two pointers refer to unrelated allocations.
  - if no instance of (4), is satisfied, at least one pointer refers to a dead allocation.
  - if no instance of (5), is satisfied, at least one pointer is out of bounds.

- **iso_array_offset**():
  - if no instance of (1) is satisfied, the pointer is either null, a function pointer, or has empty provenance.
  - if (2) fails, the pointer refers to a dead allocation.
  - if (3) fails, the resulting pointer is out of bounds.
  - if (4) fails, the resulting pointer is either out of bounds or ambiguous.

- **permissive_array_offset**():
  - if no instance of (1) is satisfied, the pointer is either null, a function pointer, or has empty provenance.
  - if (2) fails, the resulting pointer is ambiguous.

- **iso_member_offset**():
  - if no instance of (1) is satisfied, the pointer is either null, a function pointer, or has empty provenance.
  - if (2) fails, the pointer refers to a dead allocation.
  - if (3) fails, the resulting pointer is out of bounds.

- **permissive_member_offset**():
  - if no instance of (1) is satisfied, the pointer is a function pointer.

Chapter 10

Integration with C11 concurrency

There are two extensions adding support for C/C++11 concurrency to the default Cerberus pipeline. These are work led by collaborators, for which we provided support by adapting Cerberus as was needed.

Operational C/C++11 concurrency model This work by Kyndylan Nienhuis et al. [NMS16] equipped a past version of Cerberus with an operational version of the C/C++11 concurrency memory model. In the driver of the Core runtime, the steps of an operational semantics for the C11 concurrency model are allowed to interleave with the steps of the operational semantics of Core. The Core side provides the C thread-local semantics and incrementally builds a *candidate execution* (a graph of memory actions with various edges C11 defines over them, inducing various ordering such as sequenced-before) that the concurrency side checks for consistency. Critically, the two are allowed to progress independently from one another. This is necessary because the Core operational semantics follows the program order arising from the sequenced-before relation, whereas the concurrency model follows a "commitment order". A notable difficulty arises from this structure: when the Core semantics performs a load, the concurrency model may be unable to provide Core with a concrete value for that accesses (for example as a result of load buffering). Instead, the concurrency model returns a symbolic value, which it makes concrete at a later point in the execution. To allow the Core execution to continue, we extended the implementation of the Core dynamics to deal with the reduction of symbolic expressions. Note that as a result of the changes and improvements made to the development of Cerberus, this work is not operational in the current source.

Cerberus-BMC This work by Stella Lau et al. [Lau+19] takes the result of the elaboration, translates the Core representation into an SMT problem, and combines it with an axiomatic concurrency model (either C11, or one provided by the user, such as RC11 or the Linux kernel memory model), and a memory object model based on PNVI-plain. The result is a bounded model-checker and exploration tool for concurrent and sequential C11.

Chapter 11

Implementation of Cerberus and tools

In this chapter, we discuss the implementation of the pipeline we presented in Section 3.2, the implementation of our memory object models we presented in Chapter 9, and how they combine in the larger development of Cerberus. We refer to their source locations with reference to the public repository, which can be found at `https://github.com/rems-project/cerberus`.

Previous and joint work The development of Cerberus builds upon the work of Justus Matthiesen in his Part II dissertation [Mat11]. In particular, we inherit from his work the structure of our frontend, with its Cabs and Ail languages, and the separate desugaring phase (performing the "semantic analysis") from Cabs to Ail, followed by a typechecking phrase performed on Ail. Our Ail language and its typechecker is based on the version found in Matthiesen's MPhil thesis [Mat12] His original Part II thesis supported a considerably smaller fragment of C, excluding features such as: struct, union, and enumeration types; proper treatment of type qualifiers; some expression operators; and goto statements. The syntax of declarations and initialisers was also greatly simplified compared to ISO C.

While we have kept their structure, we have largely reimplemented both Cabs and Ail when extending them to the substantial fragment of C11 they now support. The Cabs parser has been completely rewritten and now supports all of C11, as has the Cabs to Ail transformation (which now supports a large fragment of C11). The Ail typechecker extends the original development.

Victor Gomes also contributed significantly to the development of Cerberus, this included: the integration of the C2X attribute syntax, the web interface, the integration of SibylFS, support for user-defined variadic functions, and work on the infrastructure in general. In Section 11.9, we discuss tools making use the Cerberus infrastructure whose development was led by others, but for which we provided support.

11.1 Structure of the development

As we discussed in Section 3.2, the structure of the Cerberus pipeline is reminiscent of that of a compiler.

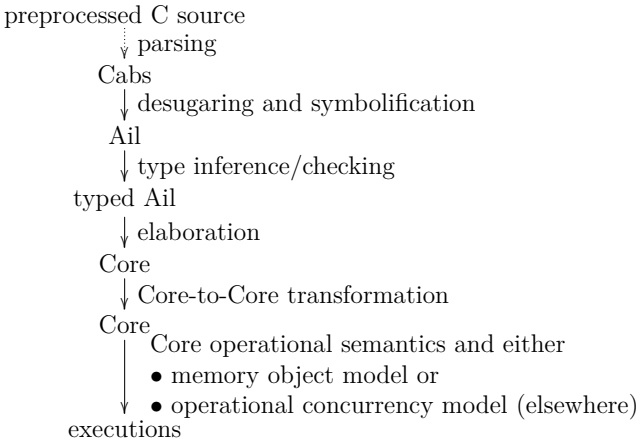


Figure 11.1: Cerberus architecture

Most of the development is written in Lem [Mul+14], which closely corresponds to the pure fragment of OCaml, and is automatically transliterated to OCaml for execution. This includes the definitions of all intermediate languages, and the stages of the pipeline starting from desugaring to the Core runtime. All of these are located in the `frontend/model/` directory. The rest of the development is directly written in OCaml. This includes: the C and Core parsers, pretty-printing and error reporting modules, the implementation of the PVI and PNVI memory object models, and the top-level infrastructure which instantiates the pipeline into our various executables. In the Lem modules, effects are written in monadic style. For readability of the development as a specification, we try to use the simplest monad for each component. In the OCaml modules, we keep a purely functional style whenever possible. We originally planned to write all modules (other than those constituting the infrastructure) in Lem, with the aim of using the Lem backends to theorem provers. As the project developed, we retargetted our efforts to building a robust tool, that could handle as input translation units from realistic C systems without requiring unreasonable adaptation by the user. The style of the Lem code is as a result sometimes not well-suited for extracting theorem prover definitions that would be convenient to reason about. More recent components, such as the PNVI memory object models, were written in OCaml, for greater convenience of development.

The group of Lem and OCaml modules corresponding to the pipeline are parametric on the memory interface, and can therefore be built with different memory object models with no modification. The various executables we developed using the pipeline are outside that abstraction, and all use fixed instantiations of the pipeline.

11.1.1 C11 parser

We chose to write our own C11 frontend after considering using a pre-existing one, such as CIL [Nec+02]. This was motivated by our desire to minimise any implicit deviation from the specification of the ISO standard, and to allow us to track where our model makes specific choices regarding implementation-defined aspects of the ISO specification. As observed by Matthiesen [Mat11, p. 3.1.1], the CIL frontend performs transformations over the source program, some of which change its semantics, or require fixing some implementation-defined behaviour (e.g. the size and representation of integer types). For example, expressions containing side effects are broken down into multiple statements.

This transformation loses the subtle unsequencing of some arithmetic expressions that we want to precisely capture, replacing it with a fixed evaluation order. In particular, this would make it impossible to detect unsequenced races. Another problematic transformation is the evaluation of constant expressions, which effectively involve modelling the dynamics of a large fragment of C11's expression language. While we perform the same transformation during our desugaring phrase, we will see in the next section how we do so without duplicating our modelling of the dynamics of expressions.

Our definition of the C abstract syntax (Cabs) directly follows the grammar of the ISO C11 standard (C11, Annex A). For convenience, however, we do extend expressions and statements with some common compiler extensions and a few standard library constructs.

- In expressions, we add: the `assert()` function; the `offsetof()` macro; the functions and macros for writing user-defined variadic functions (`va_start()`, `va_copy()`, `va_arg()`, and `va_end()`); and the GCC statement-as-expression extension. Integrating the standard library constructs into the AST allows for better error messages, and eases the implementation of the typechecking in Ail.

- We also add two unary expression operators specific to Cerberus: `__cerb_printtype()`, which is transparent with respect to the dynamics, but causes the frontend to print the inferred type of its operand; and `__BMC_ASSUME()`, for writing annotations when using the Cerberus-BMC tool.

- In statements, we add the lightweight thread creation construct of cppmem [Bat+11] (`{-{ S1 ||| ... ||| SN }-}`) and GCC's inline assembly extension. We added support for these features for the benefit of users of the frontend that do not use the rest of the pipeline (discussed in Section 11.9). As a result, it is currently only carried by the pipeline up to the Ail representation, and we do not model the dynamics of inlined assembly at all.

- In C type specifiers, we add the GCC `typeof()` extension.

The type definitions for Cabs are located in the Lem module `frontend/model/cabs.lem`, while the lexer and parser are written in OCaml, and are located in the `parsers/c/` directory. The parser makes uses of the Menhir parser generator [PR05]. To deal with the ambiguity of the C11 grammar, we implement the elegant solution from Jourdan and Pottier [JP17]. We also incorporate the modifications to the grammar proposed in [N2335], bringing support for attributes throughout the syntax. This feature is being incorporated in the upcoming C2X standard revision. Preprocessing directives (C11, §6.10) are not handled; the lexer assumes that an external preprocessor (with an output compatible with that of `gcc -E`) has first been called.

11.1.2 Desugaring from Cabs to Ail

From the Cabs representation, the pipeline proceeds by desugaring it into an unannotated Ail representation. This representation remains very close to Cabs when it comes to the structure of expressions and statements, but differs on a few points, in particular in the structure of declarations, making processing the Ail representation easier than the Cabs representation:

- in expressions, the operators which are explicitly described by the ISO standard as equivalent to encodings using other operators are removed (e.g. the `!` operator, and the prefix `++` and `--` operators);

- calls to memory atomic generic functions appear as separate constructs, as their types differ from normal function calls by being somewhat polymorphic;
- integer constant expressions are replaced by the result of their evaluation;
- C types and qualifiers have an inductive form, and the "multiset" aspect of the C syntax for type specifiers is normalised;
- occurrences of `typedef` and enumeration types are substituted out, though the necessary metadata are preserved;
- `for` statements are replaced by a corresponding encoding using `while` and block statements;
- `continue` statements are replaced by corresponding `goto` statements;
- struct/union and array initializers are replaced by corresponding fully normalised expressions;
- the form of declarations of object and functions is simplified: potential multiple declarations are collapsed; function signatures and the types of objects, along with their attributes, are made explicit (as opposed to being nested deep in the abstract syntax); the storage durations of objects are made explicit, replacing storage-class specifiers, and the declarations of objects with static duration deep within expressions are hoisted; the same is done for the declarations and definitions of struct and union types;
- all occurrences of static assertions are hoisted;
- all identifiers in Ail are symbols as opposed to strings, and, by construction of the desugaring phase, the pipeline only sees well-scoped uses of identifiers.

The Lem module `frontend/model/ail/AilSyntax.lem` gives the type definitions for the Ail syntax, and `frontend/model/ctype.lem` gives the definitions for Ail types.

The desugaring is implemented in the modules `frontend/model/cabs_to_ail.lem`, and `frontend/model/cabs_to_ail_effect.lem`. It takes care of detecting and reporting all constraint violations relating to the syntax (and the few statically-checkable undefined behaviours) other than typing errors. In particular, this includes checking the scope of identifiers, the determination of their linkage and storage durations, and the occurrence of incompatible duplicate declarations. Where a Cabs construct holds a constant expression (e.g. the file scope initializer or the size of an array type), the corresponding Ail construct instead holds the result of evaluating the expression. When desugaring such constructs, we create instances of the pipeline (with the environment of declarations seen so far) down to the Core runtime (using the module `frontend/model/mini_pipeline.lem`). By doing so, we avoid duplicating the specification of the dynamics of expressions, and the risk of inconsistencies. Because constant expressions are syntactically restricted to a "pure" subset, no memory accesses ever occur during the execution of corresponding Core expressions. The desugaring makes use of a state and exception monad. The state is used to keep track of: the scopes of identifiers and their declarations; the definitions of struct/union names, and enumeration types; and the definitions of global objects, functions, and of static assertions. The exception component is needed for the reporting of constraint violations and undefined behaviours.

11.1.3 Typechecking Ail

We model C11's statics by implementing a typechecker over the Ail representation. This stage produces a fully type-annotated variant of the Ail representation, where three new constructs may be inserted in the AST of expressions to make explicit some of the important conversions left implicit by the syntax of C:

- `rvalue()`, which must be applied to an lvalue, and yields the value the result of reading from the object designated by the lvalue (this models *lvalue conversions* (C11, §6.3.2.1#2));
- `array_decay()`, which must be applied to an expression with array type, and yields a pointer to the first element of the array object (this models (C11, §6.3.2.1#3));
- `function_decay()`, which must be applied to an expression with a function type, and yields a pointer to the corresponding function (this models (C11, §6.3.2.1#4))

A distinctive feature of the typechecker is that it operates without fixing a particular choice for the implementation-defined details regarding integer types. For example, for an addition operation between operand having different integer types, the type of the addition is inferred as being a symbolic *usual arithmetic conversion* for the two types. The types **`size_t`** and **`ptrdiff_t`** are treated as built-in types (instead of being macro-expanded to a particular implementation). We make one exception for this: the typing of constants, which by default requires the implementation details of integer types. Originally, we tried to capture a fully agnostic typechecker, by using the minimal constraint of type ranges when typing constants. However this proved inappropriate, as C programs routinely use constants outside these ranges. The typechecker can however be forced to stay fully agnostic (in the executables we discuss shortly how this is made available to the user using a command-line option). This stage is implemented in the Lem module `frontend/model/ail/genTyping.lem`, along with a few auxiliary modules in the same directory.

11.1.4 Elaboration to Core

Next, the type-annotated representation is elaborated into Core. We already discussed the structure of the elaboration function in Chapter 7. This is implemented in the module `frontend/model/ail/translation.lem`, along with `translation_effect.lem` defining its monad, and the auxiliary `translation_aux.lem`, both in the same directory. Only a state monad is used, for keeping track of string literals (which appear inside Ail expressions, but are hoisted into Core globals), and the visibility of C object identifiers during the elaboration of their lifetime inside statements. Because this stage is only called on well-typed Ail programs, for which the elaboration is fully defined, there is no need for exceptions.

11.1.5 The Core runtime

The implementation of the dynamics of Core is split between the big-step evaluation of pure expressions (in `frontend/model/core_eval.lem`), and the small-step semantics of effectful expressions (in `frontend/model/core_reduction.lem`). The latter does not actually perform the reduction of Core programs, but calculates the set of allowed transitions. The actual reduction is performed by `frontend/model/driver.lem`, which, using

the calculated transitions, orchestrates the interactions between the memory object model and the concurrency model. To accommodate the various execution modes, and a symbolic memory object model that we discuss in Section 11.6, the driver makes use of a more complex monad (defined in `frontend/model/nondeterminism.lem`), which incorporates a state, error, and non-determinism with support for branching under potentially symbolic constraints.

11.1.6 Miscellaneous

All interaction with the memory object model is done through the opaque interface described in Chapter 5, which is defined as an OCaml signature in `ocaml_frontend/memory_model.ml`, and exposed to the Lem side of the development in `frontend/model/mem.lem`. For an optional boost in readability of the generated Core and of the runtime performance, we implement a few simple semantics preserving Core to Core transformations (in `frontend/model/core_rewrite.ml`), which can be applied before the Core execution. We also implement an optional non-semantics-preserving transformation (in `frontend/model/core_sequentialise.ml`) that turns all **`unseq()`** operators into sequences. When activated, this greatly reduces spurious non-determinism in the Core execution, which improves performance when one does not care about the loose ordering of expressions.

11.2 Fragment of the C standard library

To allow our tools to operate on realistic translation units without requiring too much adaption, we provide support for some headers of the C standard library. The supported fragment has grown over time, as needed; we have not aimed at a complete implementation. Save for the `printf()`, `memcpy()`, `memcmp()`, and `realloc()` functions, we are not giving a formal treatment of the standard library. We instead use the code of the `musl` C standard library [musl-libc], which we cut down to our supported fragment. In particular, this means that calls to a standard library function which are specified as having undefined behaviour by the ISO standard are not precisely reported as such by our tools. Instead, such calls will typically give rise to language-level undefined behaviour in the corresponding code of `musl`, which will be reported by the Core evaluator. The headers that we currently (at least partially) support are: `<ctype.c>`, `<math.h>`, `<signal.h>`, `<stdatomic.h>`, `<stdio.h>`, and `<stdlib.h>`. In addition, we support the most commonly used headers that only define types and constants (e.g. `<stdint.h>`, `<limits.h>`, ...). The assertion function (from `<assert.h>`) and the macros for user-defined variadic functions (from `<stdarg.h>`) are also supported, but are implemented directly by the Core runtime and memory object model, as opposed to using `musl`.

11.2.1 Integration with SibylFS

The code of `musl` relies on an underlying implementation of POSIX for its implementation of filesystem-related functions. For this purpose, we make use of the SibylFS formalisation of the POSIX filesystem [Rid+15], which is implemented in Lem. We do so by detecting during the calculation of Core effectful steps the relevant Core procedure, and turning them into dedicated transitions. The driver in the `frontend/model/driver.lem` then takes care of orchestrating with the operational semantics of SibylFS.

11.2.2 Implementation of `printf()`

Perhaps surprisingly, the `printf()` formatting function (and its variants) is implemented by a Lem module (`frontend/model/formatted.lem`), not by using the musl C code. The benefit is the significant reduction in memory accesses, within the semantic trace of program execution. The tests we use to validate our implementation of the memory object models (see Chapter 12) frequently call `printf()`, and avoiding polluting the trace there was desirable. The implementation in Lem also allows for the accurate reporting of errors during the parsing of the formatting string, which are undefined behaviours.

The Lem function `Formatted.printf` receives the formatting string as a list of characters (a wrapper written in Core handles the reading of these characters from memory), the additional arguments of the C `printf()` call as a list of pointer values and associated lvalue types, and a callback function for evaluating memory values into Core values. The formatting string is parsed using monadic parser combinators [HM98]. Whenever a conversion specifier is successfully parsed, a pair of a pointer value and lvalue type is consumed from the list of arguments. If the type required by the specifier does not match the lvalue type, the appropriate undefined behaviour is raised, otherwise the appropriate memory read is performed using the pointer value and the result is formatted. Because of these interactions with the memory, the function operates within the monad of the memory object model.

11.2.3 Support for user-defined variadic functions

In our support for user-defined variadic functions (those using functions and macros defined in `<stdarg.h>`), we make two simplifications:

- We model the variable arguments as objects within the memory object state. This is an accurate implementation of the ISO specification, because this potentially allows the user to interact with the variable arguments without the model being able to detect this.
- We do not detect any of the undefined behaviours from improper calls to the functions and macros defined in `<stdarg.h>`.

11.3 Memory object models

The memory object model used by the Core runtime and most C implementation-defined behaviours are fixed at build time, when instantiating the pipeline. As a result of the parametricity of pipeline, it is however easy to build tools with different memory object models and/or implementation choices. Creating a different memory object model simply requires implementing a single OCaml module interface.

In our command line tool (discussed in Section 11.6), and the web interface (discussed in Section 11.7), we use implementation-defined choices mimicking GCC/Clang with the LP64 data model. For the memory model, both tools can be build with either of two variants of our provenance memory object model which offer different treatment of the allocation of objects:

- **concrete allocator** the first (and default) variant uses an implementation of the PVI and PNVI memory object models with a concrete allocation scheme: the first

allocation gets an address towards the center of the address space, while each subsequent allocation gets a smaller address such that it is adjacent to the previous (modulo alignment constraints). This implementation it the most complete, and follows the presentation from Chapter 9.

- **symbolic allocator** the second variant uses an implementation of an early version of PNVI-plain where the allocation of memory objects is fully non-deterministic (only constrained by size and alignment requirements). As a result of this, the implementation of Core integer and pointer values are symbolic, and the Core runtime deals with potential branching depending symbolic comparisons. For this purpose we instantiate the nondeterminism monad used by the driver module to use constraints encoded as SMT problems which we resolve using Z3 [MB08]. The usefulness of this memory object model is somewhat limited due to performance. It however allows the exhaustive executions of small tests exhibiting the envelope of the allowed non-determinism of memory object allocation. For example the program in 11.2 always returns `1`, showing the transitivity the `<` operator over pointer values.

```
int x, y, z;
int main(void)
{
  if ( &x < &y )
    if ( &y < &z )
      return &x < &z; // always returns 1
  return 1;
}
```

Figure 11.2: Small program checking the transitivity of `<` over pointers

11.4 Switches

Some aspects of the C semantics modelled by the Cerberus frontend behaviour can be adjusted using named switches. Depending on the tool, these are either fixed or configurable by the user (e.g. using a `--switches="NAME1,NAME2,..."` option for the command line driver). For tools, using the provenance model with the concrete allocator, the selection among the three variants of PNVI memory object model is done using this mechanism (using the switch names **`PNVI`**, **`PNVI_ae`**, and **`PNVI_ae_udi`**). The other available switches are:

- **`strict_pointer_arith`**: this makes the pointer arithmetic resulting in more then one past out-of-bounds pointer values undefined behaviour, as specified by the ISO standard. When using the PNVI-ae-* variants of the memory object model, this switch is set by default.

- **`permissive_pointer_arith`**: this is an alternative to the previous, which allows out-of-bounds pointer arithmetic. It is set by default when using the PNVI-plain memory object model.

- **`strict_reads`**: when set, a read access that would produce an unspecified value is given undefined behaviour. This replaces the modelling of unspecified values with delayed and daemonic semantics that we presented in Section 4.6.
- **`forbid_nullptr_free`**: when set, calling `free()` on a null pointer is made undefined (stricter semantics than ISO).
- **`zap_dead_pointers`**: when set, the "zapping" of pointer values when the lifetime of the object they refer to ends (specified by ISO at §6.2.4#2) is performed. By default we leave pointer values unchanged.
- **`strict_pointer_equality`**: when set, the equality operator on pointers does not look at the provenances, and simply uses the numeric addresses.
- **`strict_pointer_relationals`**: when set, using relational operators on pointers referring to different memory object is undefined behaviour. This is the behaviour specified by ISO, that we relax by default.

There are three additional command line options that slightly change the behaviour of some parts of the pipeline:

- `--agnostic`: forces the Ail typechecker to be fully agnostic in the implementation of integer types;
- `--defacto`: relaxes in the Ail typechecker the detection of ISO constraint violations, to allow de facto practices;
- `--permissive`: makes the pipeline accept extensions to ISO (by default Cerberus behaves like the pedantic mode of a compliant C compiler).

11.5 Execution modes

Two modes of execution for the Core runtime are supported:

- **random**: which is used by default, explores one possible execution. It does so by choosing a random evaluation order for the operands of the non-determinism operators in Core (i.e. **`unseq`**() and **`nd`**()), but does not exercise the non-determinism from the choice of addresses for the allocation of objects in the memory object model. Having the latter nondeterminism would lead to unstable addresses across executions which we deemed undesirable with respect to usability. The outcome of execution is either *defined*, with the return value of the startup function and strings output in stdout and stderr, or an undefined behaviour. In the former case, note that because only one possible execution has been explored, overall the program could still be deemed undefined by the model as a result of an undefined behaviour only accessible from a different execution path. This limitation does not apply to unsequenced races, which we detect from any possible execution of an expression.
- **exhaustive**: explores all allowed executions of a program. In this mode the driver computes the list of all possible return values and I/O outputs for defined executions, and of undefined behaviour. Note however that when using the concrete variant of the memory object, the non-determinism from allocation is not exercised. This is a

design choice, as this non-determinism would lead to far too many executions. For programs exhibiting non-determinism, this mode gives a better answer regarding their definedness than the random one as it accurately reports undefined behaviour only occurring on some execution path. The drawback is that there can quickly be a combinatorial explosion of possible executions. This is partially due to the absence of optimisation in the Core runtime (we make no attempt to detect confluences), but is also a result of inherent potential for non-determinism in the evaluation of C expressions. The first issue sometimes leads duplicates in the list of outcomes that do not correspond to different execution paths in the source C. This can be mitigated with the `--rewrite`, or more drastically `--sequentialise`. The second is however to be used with caution, as it is obviously not semantics-preserving: it can both remove defined outcomes (for non-deterministic expressions), and give a defined behaviour for programs that should be reported as undefined as a result of an unsequenced races.

11.6 Command line driver

The default executable packages the pipeline into a command line tool. This turns our C semantics into an executable oracle supporting small- to medium-scale translation units, with support for a large fragment of ISO C11 and some common compiler extensions. This is the backend we used for the debugging and validation of the elaboration function, and the validation of our PVI and PNVI memory object models. From the point of view of the user, the tool behaves like a normal C compiler, taking one or more translation units as input. It can either directly execute them, or produce "compiled" Core object files which can later be linked and executed. To ease the use of the tool within pre-existing build setups, we support common C compiler options:

- `-D` and `-U` for adding/removing predefined macros;
- `-I` for adding directories to the search path of the preprocessor.

The optional Core to Core transformations previously mentioned can be activated with two options:

- `--rewrite` performs the semantics-preserving Core to Core transformations aiming to reduce the spurious non-determinism in the Core runtime;
- `--sequentialise` replaces all **`unseq`**() operators with left-to-right strongly sequenced blocks.

11.7 Web interface: Cerberus C explorer

As a second use of the pipeline, Victor Gomes developed a web application for interactive exploration of the static and dynamic semantics of C programs.

This takes of the form of executable acting as a web server, which instantiates two variants of the pipeline: one using the concrete implementation of the PNVI memory object models, and another using the symbolic implementation. The execution of the pipeline is done on the server side, whereas the client side only executes a user interface

written in Typescript. The user is given a text editor, whose content is interactively processed through the pipeline. Reusing the pretty-printing infrastructure developed for the command line driver, this interface allows the user to navigate throughout the Cerberus pipeline by looking at the intermediate representations.

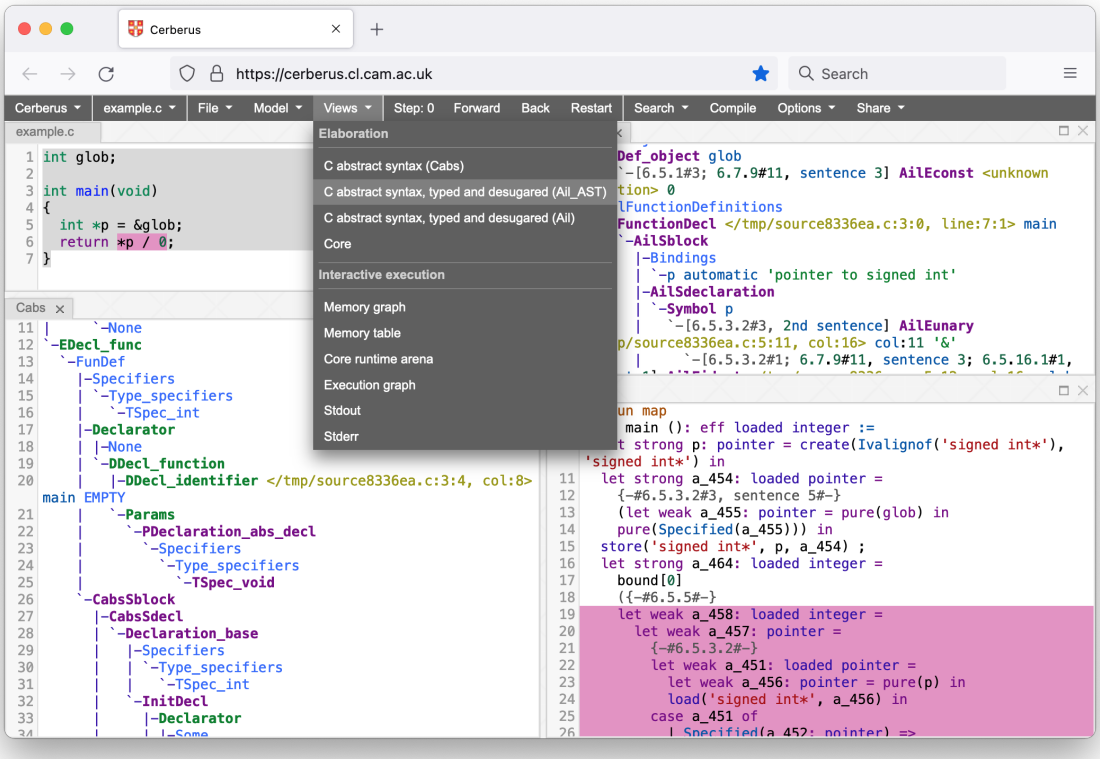


Figure 11.3: Web interface showing the intermediate languages of the pipeline: source C (top-left); Cabs (bottom-left); typed-Ail (top-right); elaborated Core (bottom-right)

The printing of the Core representation can be colour-annotated to help relate each source C construct to the fragment of Core that elaborates it. Furthermore, where the elaboration function has been annotated with the relevant reference to the text of the ISO standard, the interface provides an embedded reader to a snippet of the last available public draft of C11. Like the driver backend, the Core program resulting from the elaboration can be executed either using the random or exhaustive mode, showing the user the allowed values or undefined behaviour.

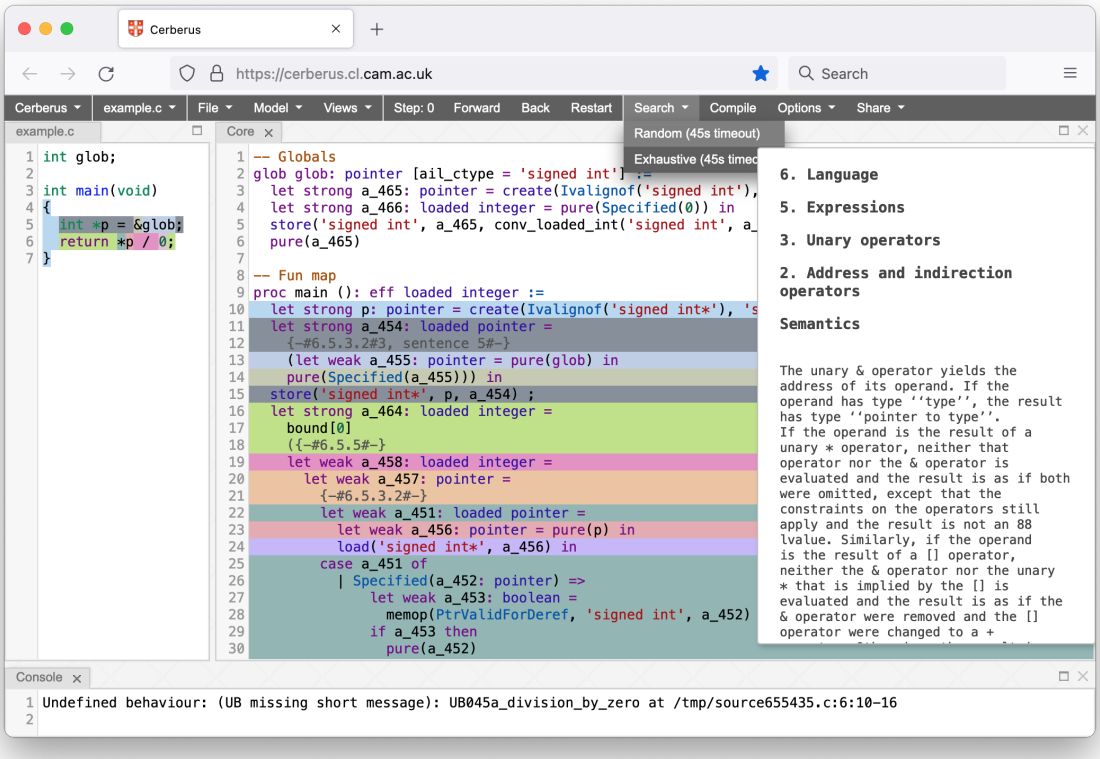


Figure 11.4: Web interface showing the colour-annotated Core, ISO standard quote (right of screen), and the outcome of a random execution (bottom of screen, showing that an undefined division by zero occurs)

The web interface adds a third *interactive mode*, which allows the user to step through the program execution one memory accesses at a time. Coupled with a graphical representation of the memory state (top-left of Figure 11.5), this makes it easy to explore the PNVI memory object models.

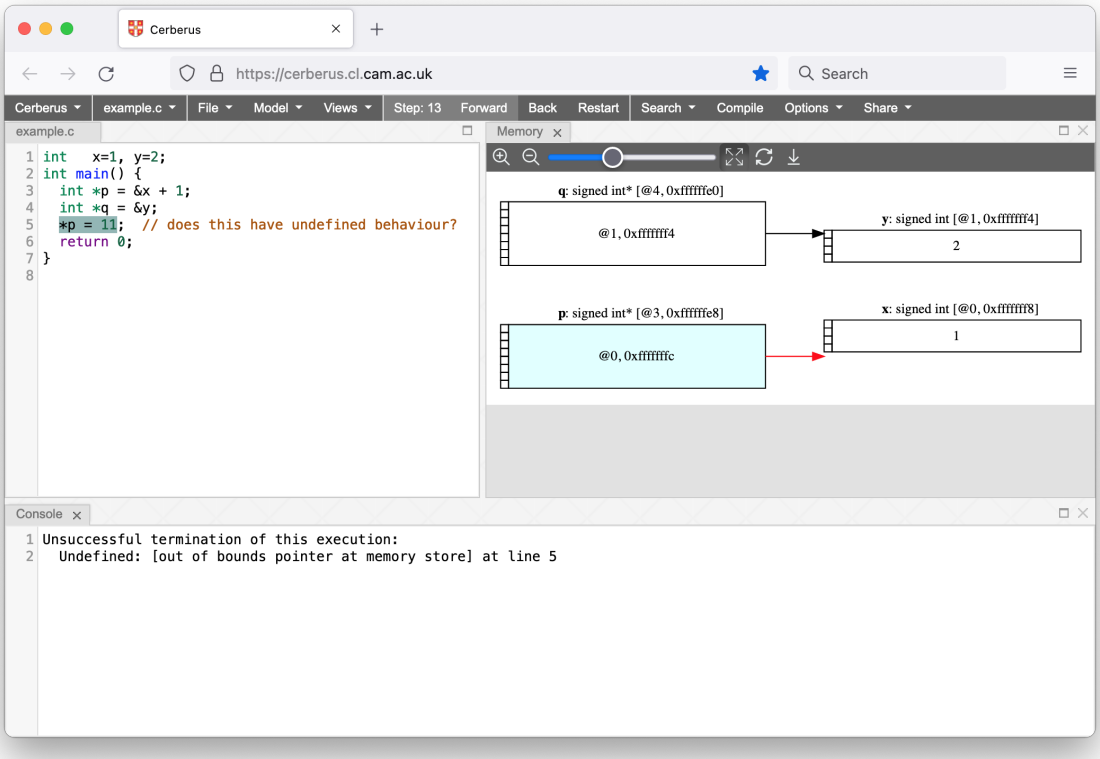


Figure 11.5: Web interface showing the interactive stepping through the execution of a program

Using various menu options, the user can select the desired version and switches of PNVI memory object model. The web interface is accessible at https://cerberus.cl.cam.ac.uk/.

11.8 User friendly error reporting

When producing the AST of our intermediate language, we sometimes annotate nodes with references to the sentences of the ISO standard that justifies why and how that node was created. These annotations are carried down to the Core representation, with the intent of helping users familiar with the ISO standard to understand how the generated Core legitimately elaborates the C source. For example, in Figure 11.4, the address operator at line 5 in the left buffer is elaborated into a Core expression starting from line 13 in the right buffer. The comment at line 12 gives the paragraph in ISO standard defining the dynamics of the address operator.

The following table gives a summary of the lines of code count of Cerberus:

	Lines of code		
	Lem	OCaml	Typescript
Cabs language	332		
Ail language	2,775		
Core language	367		
C parsing		2,675	
C frontend: scope resolution, registration of declarations, normalisation, desugaring, ...	5,182		
Ail typing	1,891		
Elaboration	3,654		
Core typing	2,080		
Core transformations	1,105		
Core dynamics and runtime	5,721		
Memory models	2,684	3,268	
Utilities	5,353	10,702	
Command line driver		3,468	
Web interface	1,529		3,222
Total	56,008		

11.9 Further usage of the Cerberus pipeline

Illustrating the reusability of the pipeline and memory interface, several other executables have been developed by different people (for which we provided varying degrees of assistance):

- **Cerberus-BMC** [Lau+19] (developed by Stella Lau), is a backend providing a bounded model-checker and exploration tool for concurrent and sequential C11, which we discussed in Chapter 10. The tool has a CLI executable and an extension of the web interface we have just discussed which replaces the graphical representation of memory state of the original with a graphical representation of concurrent executions (accessible at `https://cerberus.cl.cam.ac.uk/bmc.html`).

- **CN** [Pul+23] (developed by Pulte et al.), is a backend providing a refinement type system for systems C programs. Like the previous backend, it uses the pipeline down to the Core representation. It performs some partial evaluation and transforms the Core into a A-normalised form. The type system is then implemented as type system for the A-normalised Core representation. The backend makes use of the C2X attribute infrastructures (which the pipeline propagates down to Core), to support user refinement annotations at the level of C source.

- **RefinedC** [Sam+21] (developed by Sammler et al.), is a separate tool making use of the pipeline up to the Ail typechecking as its C frontend. This tool translates the Ail presentation into Coq definitions for an Iris [Jun+18] formalisation of fragment of C11. The support for C2X attributes from our parser is used for the syntax of user refinement-type annotations.

- **VIP memory object model** We have implemented the VIP memory object model of Lepigre et al. [Lep+22] as a third instance of the memory interface. The implementation is partially based on the concrete PNVI memory object model.
- **Core to OCaml** (developed by Victor Gomes) This executable replaces the Core runtime with compilation of the Core language into OCaml, where the generated OCaml makes use of the memory object model OCaml modules. This was implemented at a time where the Core runtime suffered from poor performance (in particular because the reduction of binders were simply implemented by substitution). The performance gain from compiling to OCaml was one to two orders of magnitude. The Core runtime has since been improved, making this executable obsolete.
- **Abstract interpretation via Core** (developed by Victor Gomes) This executable implements as an experiment a simple abstract interpreter for a fragment of C11 by actually performing the analysis of the Core representation. This is done using the APRON numerical abstract domain library [JM09]. This experiment illustrates how one could develop static analysis tools using the Cerberus pipeline without having to deal with a whole C semantics, but only instead the simpler dynamics of Core.
- **Thread-local semantics for operational C/C++11 concurrency model** (Kyndylan Nienhuis et al.), this mode of execution of the pipeline equips a previous version of Cerberus with an operational version of the C/C++11 concurrency memory model [NMS16]. We discussed this work in Chapter 10.

Chapter 12

Validation

In this chapter we discuss how we have established a reasonable level of confidence in our model. As discussed in Chapter 3, a key design goal was to structure the model such that one is able to directly relate clauses of the standard prose to the corresponding component in the model, both for ourselves and for other readers familiar with the ISO standard, to provide initial confidence in the model. To do this, the reader only needs knowledge of simple functional definitions and the syntax and semantics of the Core language (which we believe to be significantly simpler than C). In particular, we have kept the elaboration function as close as feasible to a straightforward transcription of Sections 6.5 to 6.8 of the ISO standard into a mechanically parsed, typechecked, and executable form. We illustrated this point in Figure 3.1, by showing side-by-side an extract of the ISO standard and a typeset version of the corresponding clause in the elaboration function. However as we also illustrate in Figure 3.2, the actual implementation of the elaboration function in Lem remains a conventional hand-written program, with a significant body of definitions (e.g. AST builders, symbol generators, predicates on C types). Like normal code, it is hard to ensure that it is totally free from errors.

Some of the previous related work were developed using theorem provers, and used proofs for some validation. Krebbers [Kre15] formally proved in Coq various meta-theoretical results about his model: some expressing some classic language semantics well-formedness properties; and other relating to the support of compiler optimisations. He also formally showed the correspondence between three semantics written in different styles. Similarly, Norrish used his HOL formalisation to prove a determinism result on the dynamics of expressions in C [Nor99]. We do not consider that would be the best approach to build confidence in our model: firstly the ISO standard does not itself specify properties about the language that we could verify (and C does not have the type safety property that would be the obvious target for much mechanised programming-language proof); secondly, as we have focused the development of our model towards the aim of building a tool that can be used on existing C programs (albeit of modest scale and sometimes with adaptation of the source by the user), the implementation of the model contains a significant amount of boilerplate code in addition to the "semantic part".

We instead opted for a validation via extensive testing, taking advantage of the executability and coverage of the model. We used both differential testing against compilers and previous executable models, using randomly generated programs, established test suites, and new hand-written tests. While this approach cannot prove the absence of errors in the model, it does provide good evidence that the behaviour of our model relates to that of existing mainstream implementations of C. This approach is similar to that

taken by Ellison et al. [ER12]. In the remainder of this section, we present the different test suites we have used.

Differential testing with Csmith During the development of the elaboration function, we used Csmith [Yan+11] to randomly generate small tests which we used to compare the outcome of evaluating the generated Core against executables produced using Clang. This proved a cost-effective way of catching early mistakes such as missing integer promotions and related conversions in the Core expressions. A particularly useful feature of Csmith is the possibility of specifying restrictions in the language features used by the generated tests, along with the size of their expressions and statements. This made it possible to start validating the elaboration function long before it was complete enough to execute pre-existing tests, such as compiler test suites.

Our first set of Csmith tests is restricted to only using small integer expressions (using the flag `--max-expr-complexity` ranging from 1 to 4). There are 1192 tests, ranging from 35 to 991 lines long. For 1186 of them, Cerberus gives the same result as Clang; one of them does not terminate after 5 seconds when compiled with Clang, and therefore we do not test further as it is unlikely to terminate with Cerberus; the remaining 5 tests timeout after 45 seconds.

A second set of tests uses a more relaxed feature restriction to include identifiers with array types, and arithmetic over them. There are 470 tests, ranging from 36 to 1072 lines. For 463 of them, Cerberus gives the same result as Clang; the remaining 7 tests timeout after 45 seconds. In Figure 12.1 we give a cut-down version of a typically medium-sized Csmith-generated test.

While these tests are useful for finding errors in the elaboration of expressions and statements, and in the implementation of Core's dynamics, their scope remain quite limited. In particular they did not stress most of the control-flow of the desugaring from Cabs to Ail. As a result, pre-existing compiler test suites have proven useful in the later stage of our development.

GCC torture tests We use a snapshot of the `execute/` directory of the GCC C torture test suite [GCC-tests] from August 2021. There are 1576 tests, some of which were written with the pre-C89 (K&R style) syntax, which Cerberus does not support. Our frontend is very strict and rejects non-compliant programs that GCC will accept with a warning even when called with the `-pedantic` flag. To allow our frontend to handle non-compliant tests we perform some modifications (these were made manually, with partial automation) on the tests:

- K&R style function declarations and definitions are converted to "ANSI C style" (this is done by hand for the adaptation of the syntax of function parameters).
- We add missing forward declaration of functions where needed (since C99 functions must be declared before being used).
- GNU C allows as an extension "empty initializers" whose behaviour is equivalent to an ISO compliant singleton initializer to zero; we adapt the affected tests accordingly.

In total, 280 tests are adapted. In addition, some tests use attributes with the GNU syntax (`__attribute__`), which our frontend does not support; and, some GNU builtin

```
1  #include "csmith.h"
2  static long __undefined;
3
4  static uint16_t g_2 = 0x901FL;
5  static uint8_t g_8 = 0xEAL;
6  // ... more global declarations
7
8  static uint32_t  func_1(void)
9  {
10     int32_t l_9 = (-1L);
11     uint64_t l_22 = 0x876A47CDEDFC3E8CLL;
12     int32_t l_264 = 0xE5883D10L;
13     int32_t l_276 = (-1L);
14     l_9 = (g_2 || (safe_add_func_int64_t_s_s((~((g_8 = (
15       safe_lshift_func_uint16_t_u_u(g_2, g_2))) <= g_2)), l_9)));
16     l_9 = (65530UL ^ 0x2FBBL);
17     for (g_2 = (-10); (g_2 <= 27); g_2++)
18     {
19         uint8_t l_14 = 0x78L;
20         int32_t l_21 = 0L;
21         int8_t l_23[2];
22         int32_t l_24 = (-6L);
23         int32_t l_25[3];
24         int i;
25         for (i = 0; i < 2; i++)
26             l_23[i] = 0xFAL;
27         for (i = 0; i < 3; i++)
28             l_25[i] = 0x623B26D5L;
29         l_25[1] = ((l_24 &= (l_9 == (safe_mul_func_int16_t_s_s(l_14,
30           ((func_15((safe_mod_func_uint32_t_u_u((l_21 = (l_9 < l_9)),
31             l_22)), l_23[1], l_9)
32           & 4UL) , l_9))))) == 0x6FBFCC013CEF3392LL);
33         g_29 = (g_28 ^= ((l_9 = 1L) >= func_15((((func_15((((
34           safe_mul_func_uint16_t_u_u(g_8, g_2)) ,
35             18446744073709551615UL) == g_8),
36           g_2, g_2) , g_8) < 1L) != l_22), l_25[1], l_22)));
37         ++g_30;
38     }
39     // ... more arithmetic calculations
40 }
41 // ... more function definitions
42
43 int main(void)
44 {
45   // computes hashes of the global variables and outputs the result
46 }
```

Figure 12.1: Anatomy of a small/medium size Csmith-generated test

or ISO standard macros, types, and functions without declaring them or including the necessary headers. To address this, when calling Cerberus on the testsuite, we instruct the C preprocessor to first include a header file `cerberus.h` which contains the required library include directives and macro definitions, and which erases the GNU attributes (or, in the case of alignment attributes, converts them to the corresponding `_Alignas()`). After these modifications, we find 363 tests not supported by Cerberus:

- 154 tests rely on GNU builtins or extensions we do not support (e.g. zero-length arrays, computed labels, nested functions, imaginary constants, statement expressions, ...).
- 146 tests use ISO features not supported by Cerberus (bitfields, `_Complex`, variable length arrays).
- 21 tests use ISO standard library functions that Cerberus does not implement.
- 26 tests use inline assembly.
- 16 tests use non-ISO library functions (`alloca()`, and `mmap()`).

Among the supported tests, Cerberus detects an undefined behaviour in 39 of them, and a constraint violation in 14. We have confirmed these results using the Clang/GCC sanitizers and manual checks (when the sanitizers fail to detect the undefined behaviours). While it is likely that most of these undefined behaviours are intended by the authors of the testsuite to trigger previously misbehaving optimisations in GCC, some might also be unintended defects (e.g. the occurrences of unsequenced races)

Of the remaining 1160 tests, Cerberus currently correctly executes 1148. The 9 failures are the results of known bugs or issues in our frontend:

- the desugaring of struct/union initializers currently fails to deal with some complicated cases (5 tests);
- we currently make an unsound simplification in the elaboration of compound assignments, causing the duplication of any side-effect in the lvalue (2 tests).
- the desugaring has a stack overflow when dealing with huge arrays (2 tests).

Finally, 3 tests take too long to execute with Cerberus, either because they allocate very large objects or perform too many iterations.

Running these tests allowed us to better stress our frontend; finding bugs in corner cases we would have been unlikely to encounter with the shape of C programs we have focused our interest on, in particular in the development of the PNVI memory models. It is however important to observe that when running (and passing) these tests with Cerberus, we are not necessarily exercising their intent, which is often to find semantics-changing misbehaviour in GCC's optimiser.

Toyota ITC benchmark The ITC Toyota benchmark [SMM15] aims to support quantitative comparison of static analysis tools. It consists of 1,268 tests, half with defects and the other half without any defects (meant to detect false positives), in two sets of 50 files each (for a combined total of 40K lines). We exclude some files and tests that use features not supported by Cerberus:

- 9 files in both sets that use the `pthreads` library;
- 1 file in both sets that use the `pow()` library function;
- 7 tests from both sets using bit-fields;
- 43 tests involving calls to `rand()` in the set of tests with defects, and 15 tests in the set without defects. Because we use Cerberus as a dynamic bug finder, the execution either does not terminate for most random values, or fails to trigger the intended defect.

This makes a total of 230 unsupported tests.

Within a file of either set, there is a separate function (sometime with additional global declarations and auxiliary functions) for each test, and a main function which will either execute all tests or only one, depending on a numerical command-line argument. For the "without defects" files, tools are supposed to successfully execute with no errors the main functions where all tests selected, whereas for the "with defects" files, tools should raise an error for every single test function. In the case of Cerberus, as it is a dynamic checker where we treat any constraint violation or undefined behaviour as fatal, we evaluate the "with defects" files with a separate run for each test.

Ideally Cerberus should successfully execute to the end all the "without defects" files, when using the wildcard numerical argument for the main functions. There are two issues preventing this without modification of the test files:

- the 7 tests using bit-fields would cause the frontend of Cerberus to reject their translation unit. We exclude them by modifying the affected files using preprocessor directives detecting when Cerberus is used.
- 11 tests have a constraint violation (ill-typed occurrence of the `?:` operator), or an undefined behaviour (out of range floating to integer conversion, load of an unspecified pointer value, out of bound accesses, unsequenced race, `free()` of an uninitialised pointer) These are all correctly detected by Cerberus, but we believe these are bugs in the testsuite. As a result, we have modified the affected tests to remove these unwanted undefined behaviour when the macro `FIXED` is set.

With these modifications, Cerberus successfully executes all but three of the supported "without defects" files. It timeouts on `memory_allocation_failure.c` and `st_overflow.c`, because some of the tests in these files allocate huge objects. And it non-deterministically diverges when executing `redundant_cond.c`, because some of the tests call the `rand()` function in the controlling expression of loops. Because we use Cerberus as a dynamic bug finder, the execution does not terminate for most random values.

For the set "with defects", Cerberus detects a constraint violation in 2 tests, and an undefined behaviour other than the intended defect in 26 tests. As with the previous set of tests, these are bugs in the testsuite: ill-typed `?:` operator; typos causing undefined behaviours unrelated to the intended defect; used character buffers with improper alignment; typos in variable names; and, unsequenced race. Additionally, 18 tests lack any defect and are therefore deemed defined by Cerberus. These are the result of typos (in variable names or constants), and uses of `calloc()` when the intended defect requires

uninitialised objects. As for the "without defects" set of tests, we have modified the affected tests to exhibit (only) the intended defect when the macro `FIXED` is set. There are however 13 tests for which it was not clear how to do so.

Cerberus successfully detects the intended defects in 351 tests (69.37% of the supported tests – 55.36% overall). For 105 tests (20.75% of the supported tests – 16.56% overall), Cerberus does not report a defect because they are not undefined behaviours but programming errors: unused variables; arithmetic overflow or underflow defined in ISO C; loss of sign in integer arithmetic that is defined in ISO C; redundant conjunctions in `if` and `while` statements, and suchlike. While these are usefully detected by static analysis tools, Cerberus is not intending to detect such errors. Additionally, the 14 buggy tests for which a fix is not clear are either executed with no reported defect, or detect an unrelated undefined behaviour. There are 25 tests for which Cerberus either diverges or timeouts (they involve intended infinite loops, or allocations of huge objects). The 7 tests in `st_overflow.c` contains defects relating to the call stack, which is outside of the scope of the ISO C semantics and therefore of Cerberus. There are 4 tests showing failures of Cerberus: 1 test in `data_overflow.c` involving an out of range floating value triggers a crash; 1 test shows an issue with `memcpy()` on unspecified padding bytes; and the 2 tests in `ow_memcpy.c` involves the undefined behaviour from non-exactly overlapping memory access in assignments, which Cerberus does not currently detect.

Cerberus CI tests Throughout the development of Cerberus, we have accumulated a suite of 228 tests, which are either witnesses of past bugs, or illustrate corner cases of the ISO standard. These include tests meant to successfully execute, tests that have constraint violations, and tests that have undefined behaviour.

TinyCC tests To further stress the model, we also use part of the test suite of the TinyCC [TinyCC] project. We use 70 out of 125 tests (removing those using features we do not support, such as compiler attributes or inline assembly) for a combined total of 2979 lines.

HaCL-star cryptographic functions Finally we use a few cryptographic functions from the HaCL-star [HACL*] project which provide a useful stress test for the evaluation of larger arithmetic expressions and iteration statements.

We use the last three families of tests for continuous integration testing, and all execute as expected.

There are two additional test suites that we believe would be useful for further validation of Cerberus, that we leave for future work:

- the Plum Hall Validation Suite for C, an industrial validation suite for testing the compliance of implementations of C. In particular we expect this test suite would help us in identifying remaining bugs in our frontend; and
- the `example/` directory in the development of the C model of Ellison et al. [KCC18]. This would test the coverage of our detection of undefined behaviours and constraint violations. To use this suite, we would need to manually adapt the "expected output" of each test for Cerberus, currently given as the output of the RV-Match tool.

The source for the tests we have discussed in this section are available at https://github.com/rems-project/cerberus-tests.

Remaining bugs Despite this validation work, there are still known bugs and issues that we have yet to address. These are mostly related to the frontend, and include for example: limitations regarding complicated initialisers (involving non-trivial use of designator mixing struct/union and arrays); the improper desugaring of tentative definitions; and limitations in the compatibility checks for some types across translation units. We also expect new bugs to be found and eventually addressed over time, as we continue improving Cerberus in general. We however believe that the model has reached a sufficient level of robustness to be useful in its current state.

12.1 Validation of the provenance memory models and their implementation

For the validation of our PNVI memory models there was no prose in the ISO standard, nor compiler oracles that we can take as the basis. We instead rely on the test suite derived from our exploration of the design space described in Chapter 2, and from which the examples throughout Chapter 8 were taken. This provides us with a set of properties a desired provenance memory model should satisfy. There are 61 tests grouped in 23 thematic families for which we define the expected outcome, in particular whether there is an undefined behaviour. As a result, it is necessary to use the exhaustive execution mode of Cerberus to take into account undefined behaviour that can only occur in a subset of the allowed executions. It is worth noting that with this test suite, we are testing the formal definition of our PNVI models in the form of their OCaml implementation. We give in the following table a summary of intended and observed behaviour of Cerberus, which as expected coincide.

test family	test	intended behaviour PNVI-plain	intended behaviour PNVI-ae	intended behaviour PNVI-ae-udi	observed behaviour Cerberus (decreasing allocator) PNVI-plain	observed behaviour Cerberus (decreasing allocator) PNVI-ae	observed behaviour Cerberus (decreasing allocator) PNVI-ae-udi
1	provenance_basic_global_xy.c	UB			not triggered		
	provenance_basic_global_yx.c				UB (line 9)		
	provenance_basic_auto_xy.c				not triggered		
	provenance_basic_auto_yx.c				UB (line 9)		
2	**cheri_03_ii.c**	UB			UB (except with *permissive_pointer_arith* switch)		
3	**pointer_offset_from_ptr_subtraction_global_xy.c**	UB (pointer subtraction)			UB (pointer subtraction) Or UB (out-of-bound store with *permissive_pointer_arith* switch)		
	pointer_offset_from_ptr_subtraction_global_yx.c						
	pointer_offset_from_ptr_subtraction_auto_xy.c						
	pointer_offset_from_ptr_subtraction_auto_yx.c						
4	**provenance_equality_global_xy.c**	defined, nondet			not triggered		
	provenance_equality_global_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_auto_xy.c				not triggered		
	provenance_equality_auto_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_global_fn_xy.c				not triggered		
	provenance_equality_global_fn_yx.c				defined (ND except with *strict pointer equality* switch)		
5	**provenance_roundtrip_via_intptr_t.c**	defined			defined		
6	**provenance_basic_using_uintptr_t_global_xy.c**	defined			not triggered		
	provenance_basic_using_uintptr_t_global_yx.c				defined		
	provenance_basic_using_uintptr_t_auto_xy.c				not triggered		
	provenance_basic_using_uintptr_t_auto_yx.c				defined		
7	**pointer_offset_from_int_subtraction_global_xy.c**	defined			defined		
	pointer_offset_from_int_subtraction_global_yx.c				defined		
	pointer_offset_from_int_subtraction_auto_xy.c				defined		
	pointer_offset_from_int_subtraction_auto_yx.c				defined		
8	**pointer_offset_xor_global.c**	defined			defined		
	pointer_offset_xor_auto.c				defined		
9	provenance_tag_bits_via_uintptr_t_1.c	defined			defined		
10	**pointer_arith_algebraic_properties_2_global.c**	defined			defined		
11	**pointer_arith_algebraic_properties_3_global.c**	defined			defined		
12	**pointer_copy_memcpy.c**	defined			defined		
13	**pointer_copy_user_dataflow_direct_bytewise.c**	defined			defined		
13	**provenance_tag_bits_via_repr_byte_1.c**	defined			defined		
15	pointer_copy_user_ctrlflow_bytewise.c	defined			defined		
16	**pointer_copy_user_ctrlflow_bitwise.c**	defined			defined		
17	provenance_equality_uintptr_t_global_xy.c	defined			not triggered		
	provenance_equality_uintptr_t_global_yx.c				defined (true)		
	provenance_equality_uintptr_t_auto_xy.c				not triggered		
	provenance_equality_uintptr_t_auto_yx.c				defined (true)		
18	provenance_union_punning_2_global_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_global_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
	provenance_union_punning_2_auto_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_auto_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
19	**provenance_union_punning_3_global.c**	defined			defined		
20	provenance_via_io_percentp_global.c	**filesystem and scanf() are not currently supported by Cerberus**					
	provenance_via_io_bytewise_global.c						
	provenance_via_io_uintptr_t_global.c						
21	**pointer_from_integer_1pg.c**	UB (line 7)			UB in one exec (line 7)		
	pointer_from_integer_1ig.c	defined (j = 7)	UB (line 8)		defined (j = 7)	UB (line 8)	
	pointer_from_integer_1p.c	UB (line 6)			UB (line 6)		
	pointer_from_integer_1i.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_1ie.c	defined (j = 7)			defined (j = 7)		
	pointer_from_integer_2.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_2g.c	defined (j = 7)			defined (j = 7)		
	provenance_lost_escape_1.c	defined			defined		
22	**provenance_roundtrip_via_intptr_t_onepast.c**	UB (line 10)		defined	UB (line 10)		defined
23	**pointer_from_int_disambiguation_1.c**	defined (y = 11)			defined (y = 11)		
	pointer_from_int_disambiguation_1_xy.c				not triggered		
	pointer_from_int_disambiguation_2.c	UB (line 14)		defined	UB (line 14)		defined (x = 11)
	pointer_from_int_disambiguation_2_xy.c				not triggered		
	pointer_from_int_disambiguation_3.c	UB (line 15)		UB (line 15)	UB (line 15)		
	pointer_from_int_disambiguation_3_xy.c				not triggered		

(**bold** = tests mentioned in the document)

green = Cerberus behaviour matches intent
blue = Cerberus behaviour matches intent (witch *permissive_pointer_arith* switch)
grey = Cerberus' allocator doesn't trigger the interesting behaviour

Simply testing the behaviour of our model is however not enough to establish the extent to which it relates to existing compiler behaviours, and, where it does not, whether divergence is significant. To investigate this, we have executed the test suite with various existing implementations (in particular GCC, Clang, and ICC) using several optimisation levels. Interpreting the result of such runs is more subtle than when running the formal model. Many tests are pathological corner cases, that intentionally do not correspond to real programming practice; but instead exhibit situations where memory accesses should be deemed undefined, or on the contrary allowed by the model. From the perspective of implementations, tests that are deemed defined should not have their behaviour altered by optimisations, on the other hand this is perfectly fine for the tests deemed undefined. Implementations are therefore deemed sound with respect with the PNVI memory model if one only observe semantics-changing optimisations for tests deemed undefined by the memory models. It is however important to observe that the tests were not crafted to provide interesting optimisation opportunities. It is therefore possible that we are failing to detect some existing optimisations.

test family	test	Observed behaviour (compilers), sound w.r.t PNVI-*? (relying on UB or ND?)								
		gcc-8.3			clang-7.0.1			icc-19		
		PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi
1	provenance_basic_global_xy.c	y (n)			y (n)			y (y for O2+)		
	provenance_basic_global_yx.c	y (y for O2+)			not triggered			not triggered		
	provenance_basic_auto_xy.c	y (n)			y (n)			y (y for O2+)		
	provenance_basic_auto_yx.c	y (n)			y (n)			y (y for O2+)		
2	cheri_03_ii.c	y (n)			y (n)			y (n)		
3	pointer_offset_from_ptr_subtraction_global_xy.c	y (n)			y (n)			y (n)		
	pointer_offset_from_ptr_subtraction_global_yx.c							y (n)		
	pointer_offset_from_ptr_subtraction_auto_xy.c							y (y for O2+)		
	pointer_offset_from_ptr_subtraction_auto_yx.c							y (y for O2+)		
4	provenance_equality_global_xy.c	y (n)			y (n)			y (n)		
	provenance_equality_global_yx.c	y (y for O2+)								
	provenance_equality_auto_xy.c	y (y for O2+)								
	provenance_equality_auto_yx.c	y (n)								
	provenance_equality_global_fn_xy.c	y (n)								
	provenance_equality_global_fn_yx.c	y (y for O2+)								
5	provenance_roundtrip_via_intptr_t.c	y (n)			y (n)			y (n)		
6	provenance_basic_using_uintptr_t_global_xy.c	y (n)			y (n)			n (y)		
	provenance_basic_using_uintptr_t_global_yx.c	n (y)			not triggered			not triggered		
	provenance_basic_using_uintptr_t_auto_xy.c	y (n)			not triggered			n (y)		
	provenance_basic_using_uintptr_t_auto_yx.c	y (n)			y (n)			n (y)		
7	pointer_offset_from_int_subtraction_global_xy.c	y (n)			y (n)			y (n)		
	pointer_offset_from_int_subtraction_global_yx.c									
	pointer_offset_from_int_subtraction_auto_xy.c									
	pointer_offset_from_int_subtraction_auto_yx.c									
8	pointer_offset_xor_global.c	y (n)			y (n)			y (n)		
	pointer_offset_xor_auto.c									
9	provenance_tag_bits_via_uintptr_t_1.c	y (n)			y (n)			y (n)		
10	pointer_arith_algebraic_properties_2_global.c	y (n)			y (n)			y (n)		
11	pointer_arith_algebraic_properties_3_global.c	y (n)			y (n)			y (n)		
12	pointer_copy_memcpy.c	y (n)			y (n)			y (n)		
13	pointer_copy_user_dataflow_direct_bytewise.c	y (n)			y (n)			y (n)		
14	provenance_tag_bits_via_repr_byte_1.c	y (n)			y (n)			y (n)		
15	pointer_copy_user_ctrlflow_bytewise.c	y (n)			y (n)			y (n)		
16	pointer_copy_user_ctrlflow_bitwise.c	y (n)			y (n)			y (n)		
17	provenance_equality_uintptr_t_global_xy.c	y (n)			y (n)			y (n)		
	provenance_equality_uintptr_t_global_yx.c									
	provenance_equality_uintptr_t_auto_xy.c									
	provenance_equality_uintptr_t_auto_yx.c									
18	provenance_union_punning_2_global_xy.c	y (n)			y (n)			n (y)	y (y for O2+)	
	provenance_union_punning_2_global_yx.c	n (y)	y (y for O2+)		not triggered			not triggered		
	provenance_union_punning_2_auto_xy.c	y (n)			y (n)			n (y)	y (y for O2+)	
	provenance_union_punning_2_auto_yx.c	y (n)						n (y)	y (y for O2+)	
19	provenance_union_punning_3_global.c	y (n)			y (n)			y (n)		
20	provenance_via_io_percentp_global.c	NO OPT			NO OPT			NO OPT		
	provenance_via_io_bytewise_global.c									
	provenance_via_io_uintptr_t_global.c									
21	pointer_from_integer_1pg.c	y (y for O0+)			y (y for O2+)			y (y for O2+)		
	pointer_from_integer_1ig.c	n (y)	y (y for O2+)		n (y)	y (y for O2+)		n (y)	y (y for O2+)	
	pointer_from_integer_1p.c	can't test with charon								
	pointer_from_integer_1i.c									
	pointer_from_integer_1ie.c									
	pointer_from_integer_2.c									
	pointer_from_integer_2g.c	y (n)			n (y)			y (n)		
	provenance_lost_escape_1.c	y (n)			y (n)			n (y for O2+)		
22	provenance_roundtrip_via_intptr_t_onepast.c	y (n)			y (n)			y (n)		
23	pointer_from_int_disambiguation_1.c	n (y)			not triggered			not triggered		
	pointer_from_int_disambiguation_1_xy.c	not triggered			y (n)			n (y for O2+)		
	pointer_from_int_disambiguation_2.c	y (n)			not triggered			not triggered		
	pointer_from_int_disambiguation_2_xy.c	not triggered			y (n)			y (n)		
	pointer_from_int_disambiguation_3.c	y (n)			not triggered			not triggered		
	pointer_from_int_disambiguation_3_xy.c	not triggered			y (n)			y (y for O2+)		

The previous table shows that current implementations of GCC, Clang, and ICC all exhibit discrepancies with our PNVI-ae-udi memory model (indicated as the red cells). All of the affected tests involve conversions of a pointer value to an integer type, followed by some integer arithmetic, and a conversion back to a pointer which is then used. For GCC, from discussions with GCC developers, it is our understanding that the two affected tests exhibit known long-standing semantic conflicts (which are not expected to be resolved) between optimisations performed by the middle-end and ones performed by the backend.

Whether it is feasible to adapt the mainstream implementations to remove all other discrepancies is an important question, which the specification of PNVI-ae-udi as an in-progress ISO Technical Specification is intended to facilitate.

Chapter 13

Related work

The semantics of C has been studied and formalised by many groups in the past, including Gurevich and Higgens [GH92], Cook and Subramanian [CS94], Lars Ole Anderson [And94], Paul Black [BW96; BW98], and Mark Bofinder [Bof98]. In this chapter we give an overview of more recent works that have focussed on the formalisation of the ISO C, and the study of its memory model.

Norrish (1998) Norrish presents [Nor98] a formal semantics for a large fragment of ISO C90 using the HOL theorem prover. Both the statics and dynamics are modelled, and defined directly over the C abstract syntax. The dynamics is written as a structural operational semantics, with small-steps for expressions, and big-steps for statements. In contrast to previous work, the loose evaluation order of expressions is precisely captured. This aspect of the model is quite different from ours as a result of substantial change in the ISO standard between C90 and C11. The ISO C90, specifies the evaluation order of expressions (and their side-effects) using a notion of *sequence points*, whereas ISO C11 that we use as reference in this thesis has, as a result of the introduction of relaxed concurrency memory model, been rewritten in term of the *sequenced-before* relation. Norrish proves type preservation and type safety for expressions, and that two classes of expressions are deterministic ("syntactically pure expressions", and expressions free of sequence points). He also proved some Floyd-Hoare style rules for statements, derived from the operational semantics. The memory object model is fully concrete, with the state represented in a map from addresses to byte values. This work predates the introduction of effective types in the standard, and the defect reports that suggested the need for provenance in pointer values.

Papaspyrou (1998) Papaspyrou [Pap98] gives a denotational semantics for ISO C90. This covers a larger fragment of the language than Norrish (in particular unstructured statements); only small deviations are made with respect to the standard. He lists them in [Pap98, §2.3], and they mostly relate to equating whole C programs to a single translation unit, with the associated simplification in the dealing of linkage and lifetimes. The model deals with the syntax (and its analysis), the type system, and the dynamics, in three successive layers. The different aspects of computations are represented using monads and monad transformers. These are in particular used to accurately capture the loose sequencing of expressions. Like Norrish, the evaluation order is described using sequence points as specified by the text of the standard at the time. Unlike Norrish's work, no meta-theoretical results are proved about the model. However, the semantics was imple-

mented as an interpreter written in Haskell. This provides an executable form that was used to assess the validity of the model on *"improvised tests and parts of available test suites for C implementations"*. Ellison [Ell12], reports however that the performance of the tool restricts it to small programs. Support for the standard library is omitted, and as a result features such as dynamic memory and variadic functions are not supported.

Blazy and Leroy (2006 onwards) As the basis of the CompCert C verified compiler project, Leroy et al. [Ler09] formalise a semantics for a large fragment of C99 (with some features from C11). The semantics is mechanised in the Coq proof assistant. The dynamics is expressed as a small-step operational semantics, and the formalisation of the statics comes with a proof of type preservation. The parser is formally verified using a validator implemented and proven correct in Coq [JPL12]. The operational semantics is executable in the form of an interpreter for single translation units, and with support of library functions limited to `printf()`, `malloc()`, and `free()`. The interpreter can either choose one random execution, or explore all allowed evaluation orders of the input program. The addition of the interpreter in CompCert 1.9 follows the work by Campbell [Cam12] where he demonstrated how to retrofit the operational semantics of CompCert C.

Because the motivation for the model is to serve as the semantics for the input language of a formally verified compiler, the aim is not to exactly capture the semantics of ISO C; there are therefore some points of difference with ISO C99/C11. In particular it gives defined behaviour to some aspects left undefined by the ISO standard:

- Overflow on signed integer types and applying the >> operator to a negative right operand are defined.
- The lifetime of block-scope variables extends to the whole body of the function containing them (instead of being restricted to the execution of the innermost block).
- Similar to the default behaviour of our memory model, pointer arithmetic going out-of-bounds is defined.
- While the loose ordering of expressions is modelled by the operational semantics, unsequenced races do not raise an undefined behaviour.

Some features of C99 are also not supported: unstructured **`switch`** statements (**`case`** statements cannot appear inside nested **`if`** or iteration statements); functions returning struct or unions are not directly supported (this can be recovered with an unverified desugaring transformation); and, like in Cerberus, variable-length array types. Unlike in Cerberus, bit fields and C11's generic selection operators are supported.

The memory model has seen several revisions over time. An early version is described in [LB08], which presents an axiomatisation and a "concrete model" that satisfies it. The memory state consists of collection of separate blocks of bytes. Each block has a unique identifier, and with a footprint related to the (numerical) address space by two integer bounds. Pointer values are pairs of a block identifier and a byte offset. Compared to the models we propose in Chapter 9, this takes a rather abstract view of memory. The block identifiers in pointer values give a strong provenance semantics, with the semantics of the == relying on their comparison. In this early version of the memory object model, there are notable restriction on manipulation of the representation of pointers:

- pointer values do not contain anything corresponding to the numeric address of a pointer value in a conventional C implementation. They therefore cannot be meaningfully cast to integer types.
- there is no support for manipulation of the representation bytes of values. For the integer and floating-point types that would need a relatively straightforward adaptation of their store function, at least given a fixed implementation-defined representation. But for pointer values, because there is no address information, it would require more radical change.
- there is (correspondingly) no modelling of the layout and padding of C struct and union types.

In [Ler+12], Leroy et al. describe an improved version of the memory object model of CompCert (introduced in version 1.7, and further refined in 1.11). There are two main changes: the support for the manipulation of objects with arithmetic types are at the level of bytes, while preserving the abstract nature of pointers; and the introduction of per-byte permissions in the memory state.

CompCert, and in particular its memory model has been very influential. The work by Ellison, and the work by Krebbers, that we describe shortly build their memory models upon it.

Ševčík et al. (2011, 2013) In their presentation of CompCertTSO, an extension of CompCert 1.5 adding support for TSO relaxed concurrency, Ševčík et al. [Šev+11; Šev+13] discuss a functional characterisation of the threadwise relational semantics for their ClightTSO intermediate language which they have proved equivalent. Using Coq's extraction mechanism to OCaml they produced an interpreter for their dialect of C. The interpreter allowed them to find "subtle errors" in their initial definitions by testing them on small C programs. The memory model also differs on two interesting points from vanilla CompCert 1.5. First, pointer equality is always defined (and as result, the semantics adds supports for the "re-use" of pointers). This was made necessary by the lack of global time arising from the weak concurrency. In this context for pointer equality to only be defined on "in bound" pointers, that operation would need to be effectful. This would lead to the loss of algebraic properties for pointer comparison, complicating the correctness proofs of the compiler, and potentially restricting optimisations. Second, the model supports finite memory, where allocation can fail and "in which pointer values in the running machine-code implementation can be numerically equal to their values in the semantics".

Ellison and Roşu (2012), Hathhorn et al. (2015) Ellison and Roşu [Ell12; ER12] present an executable semantics for C99 (which has since been updated to C18) written in the K framework. The statics and dynamics are expressed directly on the C AST as rewriting rules. The model has been extensively validated by testing against GCC torture tests and other testsuites. The executability of the model yields the tool `kcc` that can be used like a C compiler to produce executables. These executables can be used to explore the allowed behaviour of the input C program, and will report the occurrence of undefined behaviours. The tool has been used as an oracle for the reduction of tests used for findings bugs in compilers [Reg+12]. The memory model is based on that of CompCert, with the state represented as a map from abstract location to blocks of bytes;

and, where pointer values are block-ID/offsets pairs. Hathhorn et al. [HER15] improves this work with better treatment and detection of undefined behaviours. They also extend the memory model with support for the `restrict` qualifiers, and add a treatment of the effective types rules by annotating the byte representation of objects with declaration type informations. Krebbers reports that this approach is less fine-grained than his (involving tree based object representation) and gives more defined behaviours than his model. The treatment of pointer values also "tags" for the detection of undefined behaviours relating to out-of-bounds accesses for sub arrays.

Krebbers (2013 – 2015) Krebbers [Kre15], partly in collaboration with Wiedijk, has developed in Coq a semantics for a substantial fragment of C11, in their CH_2O project. Among the features omitted are: flexible array members; variable-length arrays and bit-fields (that Cerberus also does not support); and floating types (whose uses are directly rejected by the parser). Krebbers accurately captures the implicit lifetime of objects with automatic storage duration, in particular that caused by non-local control flow, using an operational semantics using a zipper data structure [KW13]. He models the subtle sequencing of expressions, and the associated undefined behaviour, using a permission system [Kre14a]. Like previous work, this is modelled in terms of sequence points. For both of these aspects, he gives an operational semantics along with a corresponding separation logic that he proves sound. The basis of the memory model follows CompCert, with the state defined as a partial map from abstract object identifiers to the objects, and pointer values represented as block-ID/offsets. However, it includes a particular interpretation of the ISO standard notion of "effective type" [Kre13]. This departs from the CompCert memory model, by modelling the representation of objects using abstract trees capturing the shape of their C type, and pointer values as pairs of object IDs and paths in these memory trees. The memory model supports accesses to the representation of objects while also supporting most compiler optimisations relying on an abstract view of values. He has developed an executable semantics proven sound and complete with respect to the operational semantics. The resulting interpreter allows the exhaustive execution of programs [KW15]. The scope of the interpreter is limited, as it only deals with single translation units; the frontend lack supports for qualifiers; and the support for the standard library is limited. To allow mechanised reasoning about C programs using his model in Coq, he has developed "a generalization of separation algebras that is well-suited for C verification" [Kre14b; Kre16]. As part of the cross validation of CH_2O with CompCert, he describes [KLW14] two extensions of CompCert that brings it closer to ISO C11 and CH_2O by adding support for one-past pointer values and the byte-wise copy of pointers.

Wiedijk and Krebbers have also interacted with WG14, in particular regarding the under-specification of the semantics of unspecified values. In [DR451; N1747], they observe that that committee response to Defect Report #260 suggests that unspecified values are "unstable" and attempted to find a clarification to the text of the ISO standard. While WG14 reaffirmed their position on Defect Report #260, this effort did not succeed in obtaining clarification from the committee.

Tuch et al. (2005, 2007) In [TK05; TKN07], Tuch et al. present a memory model for C. The design of the memory object model aims to capture low-level idioms involving pointers while also forming "the basis for an expressive implementation of separation logic". This work is implemented in the Isabelle/HOL theorem prover, and was exercised

by verifying the memory allocator of the L4 microkernel.

Besson et al. (2014) In [BBW14], Besson et al. present an extension to the memory model of CompCert which aims at giving defined behaviour to reads of uninitialised variables and programming idioms involving the manipulations of unused bits in pointer values, without "resorting to a concrete representation of the memory". In this model, the content of a memory object is represented as symbolic expressions as long as the concrete value is not needed. The symbolic expression is normalised when a concrete value is needed, for example when control-flow depends on it, with associated memory access given defined behaviour only if the normalisation is unique. The model has been exercised on the Doug Lea's allocator, NaCl crypto, and CompCert benchmarks.

In [BBW15], Besson et al. show that the model is an abstraction of the CompCert memory model, and that the CompCert front-end correctness proof (from CompCert C to Cminor) can be adapted to the new model.

Kang et al. (2015) Kang et al. [Kan+15] present a memory model aimed at supporting roundtrip casts between pointers and integer types (with potential arithmetic over them), in the way a fully concrete model does, while still making a range of compiler optimisations sound and verifiable, in the way that the abstract block-ID/offset models do. They do so by adapting the abstract block-ID/offset model into a "quasi-concrete model": blocks are created as abstract, initially with no associated concrete address. If a pointer referring to a block is ever cast to an integer type, the block is then associated a concrete address chosen non-deterministically. Pointer values are either a concrete integer address or a block-ID/offset pair.

Lee et al. (2018) Lee et al. [Lee+18] present a memory object model for LLVM IR. The problems addressed by this work are close to that of the PNVI models, but because it targets the intermediate language of a compiler, it is subject to different constraints than the C source-language semantics that is the focus of the PNVI models. Pointer values have two forms: *logical pointers*, which are produced by allocation and preserved by pointer arithmetic (these are similar to PNVI pointers with provenance $@i$); and *physical pointers*, which result from integer to pointer casts. The latter are akin to PNVI pointers with a "wildcard" provenance. They are equipped with two additional mechanisms that restrict what objects they can access: a timestamp, used to prevent accesses to local variables; and a set of the past concrete addresses of the pointer, so that bounds checks can be deferred to when dereferencing occurs. They observe that for programs that almost or completely exhaust the allocatable address space, it is possible for code to indirectly learn facts about allocation addresses without explicitly casting them to integers, and that this can make some desirable optimisations unsound. To rule out such programs, they introduce twin allocation: they make one (or more) shadow allocations for each actual allocation, making it easy to reason that, for any example that guesses a concrete address, that there is another similar execution in which the guess is wrong.

Chapter 14

Conclusion

We presented Cerberus, an executable model for a substantial fragment of ISO C11, with the dual emphasis of producing a model that is reasonably relatable to the prose of the ISO standard, while also formalising aspects of the semantics of C that are not clearly addressed in the ISO prose as it stands. This corresponds in particular to the memory object model and the semantics of pointers, where we have aimed to capture their de facto semantics; namely, how they are used by programmers, and implemented by compilers. For other aspects, such as unspecified values, we explored a plausible semantics in the absence of a clear specification by the standard or of a dominating de facto semantics.

A key design choice of our model is its definition by elaboration, where the dynamics of C is given by two disjoint components: an elaboration function from C to the Core language; and the operational semantics for C. With this work, we show that defining the semantics of a real-world systems programming language by such an elaboration into a target language, that we carefully designed to be syntactically explicit, comes with multiple advantages.

- It enabled us to clarify syntactically the subtleties of C's expressions and statements, making the model more approachable:
  - the dynamics of C's expressions and statements takes a form that should be reasonably accessible to C programmers or compiler writers: small programs, instead of formal semantics constructs that might not be familiar to them; and
  - as we showed in Figure 3.1, the elaboration closely follows the structure of the ISO prose (individual sentences of the ISO prose correspond to portions of the elaborating Core programs).
- As a side benefit of the previous point, the close correspondence between the clauses of elaboration function and the ISO prose allows for some immediate form of validation through inspection.
- The elaboration function is to a large extent agnostic on implementation-defined behaviour, for example, through the use, in Core, of abstract constructors for the size of integer types.
- The target language of the elaboration uses carefully selected programming language constructs that are simple and well-understood (with the exception of the sequencing calculus which required some novel constructs). This makes giving it a formal semantics far less challenging than directly for C. As a result, Core is a more

accessible target for the development of static and dynamic analysis tools. This allows the creation of analysis tools for C while focussing on Core, by leveraging the work already done in the elaboration function. In Section 11.9, we list several projects led by third parties illustrating this, most substantially: CerberusBMC, a bounded model checker for C/C++11 concurrency through the translation of Core into SMT problems; and CN, a static verification tool for system C programs through a refinement type system defined on Core.

- One can toggle aspects of the semantics of C, by performing Core to Core transformations as separate stages, without requiring changes to the elaboration function. For example, the CN tool discards the unspecified ordering of C expressions by rewriting occurrences of the **`unseq`**() operator into left-to-right sequenced of **`letstrong`** operators. It also removes the occurrences of C types as values by performing partial evaluation on Core.

While the immediate validation through inspection facilitated by the elaboration function allowed to build some initial confidence during the development of the model, more rigorous validation remained necessary. In previous work, the dominant approach has been mechanising proofs of meta-theoretical properties about the C language. We instead opted for a validation by differential testing, both against compilers (i.e. on randomly generated programs, mostly exercising the elaboration of expressions and statements), and on hand crafted testsuites (in particular regarding the memory object model). As we discussed in Chapter 12, we believe this approach is better suited for our model because the ISO standard does not define meta-theoretical results as part of its specification of the C language. On issues relating to the memory object model, for which we had to investigate the de facto semantics arising from programmers and compiler practice, the requirements are better expressed in term of small litmus programs, rather than abstract properties.

This put emphasis on having robust executability, requiring scaling both in terms of the coverage of features of C, and in terms of performance. As a result of this we have opted to develop Cerberus outside of a theorem prover environment, such as Coq, in contrast with most previous work. In the absence of a requirement for meta reasoning, we believe that choosing to develop Cerberus as a more conventional Lem/OCaml project made our goal of building a robust tool better achievable.

PNVI-ae-udi memory object model We have investigated, though surveys, experimental testing, and engagement with WG14 and the GCC and Clang compiler communities, what design requirements for a memory object model that reconciles programmers and compiler practice with the ISO standard. This has resulted in the three PNVI variants we presented in Chapter 9. In collaboration with some members of WG14 a prose version of the PNVI-ae-udi model object model, is at the time of writing in the process of being published as a ISO Technical Specification. This should provide the basis for experimental implementation in existing compilers, which if satisfactory would allow for a possible integration of this memory object model to a future revision of the ISO standard. We have also explored other issues, such as the semantics of uninitialised reads [N2089; N2221; notes98; cmom0006], and how to extend provenance to sub-objects (which then requires clarifying the effective types notion present in the ISO prose) [note30; P1796R0]. However, for these a coherent design that can find support within WG14 remains elusive.

Future work We continue to improve Cerberus, in particular by working toward addressing outstanding issues in our frontend. We also aim to add some of the missing features of C, in particular bitfields and variable length arrays. Finally, we believe that Cerberus in its current state can form the basis for several projects of interest:

- the development of analysis tools for C, such as the ongoing CN project by Pulte et al.;
- the development of tools for testing compilers (in the line of Csmith), using Cerberus as a oracle for differential testing;
- creating a variant of the elaboration function, where integer computation are performed on fixed-width types (instead of the current unbounded type Core is equipped with), which would be better suited for building analysis tools based on SMT;
- in a different direction, developing in a theorem prover a new formalisation of Core's dynamics, to allow for mechanised reasoning of C programs while leveraging the elaboration function.

With this work we showed the feasibility and benefits of using an approach by elaboration for the formalisation of an industrial programming language, and that our detailed design of Core and our elaboration suffices to produce a tractable definition of a large fragment of C.

Appendix A

The memory interface

In this Appendix, we give a complete presentation of memory interface and motivate its design. We directly present the OCaml module signature used in the Cerberus development[1].

A.1 Memory state and monad

The interface places no requirements on the shape of the memory state, which it declares as an opaque type. Memory actions, which we introduce shortly, operate over this state though a monad. In contrast with the state, the choice of the monad is fixed by the interface to support the following features: errors relating to the memory; undefined behaviour; state; and nondeterminism guarded by symbolic constraints. This corresponds to the needs of our most demanding implementation which is symbolic (which we discuss in Section 11.3).

type *mem_state*
val *initial_mem_state* : mem_state
type 'a *memM* =
  ('a
  , string
  , Mem_common.mem_error
  , integer_value Mem_common.mem_constraint, mem_state) Nondeterminism.ndM

The other memory object models do not need to guard their nondeterminism under symbolic constraints. For these, monad that only support errors, undefined behaviour, and state would be sufficient.

A.2 Types of values

As we have seen in Section 3.3, we consolidate the numerous scalar types of C into only three types in Core: `integer`, `floating`, and `pointer`. For example, **`signed int`** and **`unsigned short`** are both modelled with `integer`. These, along with aggregate types built over them (array, struct, and union types), form the *object types* in Core: the only types that can be stored in and read from memory objects. The memory interface declares (but

[1] This can be found at https://github.com/rems-project/cerberus/blob/master/ocaml_frontend/memory_model.ml.

does not define) three corresponding types which are used by the implementation/formalisation of Core's dynamics.

Pointer values In Chapter 2, we showed that the nature of pointer values is subject to divergent views between the ISO standard and various implementations. A design choice of Cerberus was to ensure that the Core language (and, as a result, the elaboration function) does not know about the concrete definition of these values. This allowed us to experiment with different versions of pointer values, with no need to rework the part of the semantics that does not pertain to the memory. The memory interface therefore declares an opaque type for pointer values, along with a small number of constructors, and a destructor:

type *pointer_value*
val *null_ptrval* : Ctype.ctype → pointer_value
val *fun_ptrval* : Symbol.sym → pointer_value

The two constructors are respectively for building null pointer values of a given referenced type, and a function pointer from the symbolic name of a function. Other pointer values will be constructed using an allocating memory action. The destructor has the following signature:

val *case_ptrval* :
  pointer_value →
  (Ctype.ctype → 'a) →
  (Symbol.sym → 'a) →
  (unit → 'a) → 'a

The second argument deals with null pointers; the third, with function pointers; the fourth, with *specified* pointers to object. Additionally, there is a function for extracting the symbolic name of a function referred to by a pointer:

val *case_funsym_opt* : mem_state → pointer_value → Symbol.sym option

Implementations of the interface will typically hold an association map between concrete representation of function pointers and the high-level description of the functions they refer too. We opted to have the state provided as the first operand, instead of placing the operator inside the monad, because the Core construct whose dynamics make use of the operator does not exhibit any effect.

Integer values The type for integer values is also kept opaque by the interface. There are two motivations for this. Firstly, as we discuss in Chapter 8, our initial design for a de facto memory object model associated a provenance to both pointer and integer values. Secondly, our initial implementation of the memory interface was symbolic: the allocator chooses a symbolic address based on size and alignment constraints, and the program execution keeps on accumulating numeric constraints. Like for pointer values, the interface declares several constructors and a destructor.

type *integer_value*
val *integer_ival* : ℤ → integer_value
val *max_ival* : Ctype.integerType → integer_value
val *min_ival* : Ctype.integerType → integer_value
val *sizeof_ival* : Ctype.ctype → integer_value
val *alignof_ival* : Ctype.ctype → integer_value

It is important to note that the integer values provided by the interface are used for the dynamics of Core, and are therefore specified to behave like unbounded integers, hence the signature of *integer_ival*. The *max_ival* and *min_ival* respectively construct the maximal and minimal integer representable by the given C integer type. The *sizeof_ival* and *alignof_ival* are constructors for, respectively the size in bytes, and the alignment of a given C type, as integer values. These allow for the implementation of their direct counterparts in the syntax of Core.

There is an additional constructor for constructing the integer holding the address offset of a given member of a structure type:

val *offsetof_ival* :
(Symbol.sym, Ctype.tag_definition) Pmap.map →
Symbol.sym →
Symbol.identifier → integer_value

Finally, there is a destructor:

val *case_integer_value* :
integer_value →
($\mathbb{Z} \to$ 'a) →
(unit → 'a) → 'a

Ideally, we should only have one case to deal with, the one dealt by the second operand where a concrete integer is extracted; however, to accommodate the symbolic memory object model, the signature allows for another to indicate a "symbolic" integer value that could not fully be reduced to a constant. For all non-symbolic implementations of the interface, the third operand is never called.

Floating values For consistency we expose the type of floating-point values in a similar fashion to the previous two. Cerberus only has minimal treatment of the semantics of floating-point. In all our memory object models implementing the interface, the underlying type is simply OCaml's `Float`.

type *floating_value*
val *zero_fval* : floating_value
val *one_fval* : floating_value
val *str_fval* : string → floating_value

val *case_fval* :
floating_value →
(float → 'a) → 'a

In this section, we have only seen the constructors and destructors of the three value types. There are of course, in addition to these, the necessary operators such as arithmetic and comparison operators, which we discuss in a subsequent section.

Memory values Values stored in a memory object may be of a *derived type*, such as an array, a structure, or a union type. Furthermore, an object may hold an unspecified value. In the interface, this corresponds to the *mem_value* type over which the memory accesses, that we introduce shortly, operate on. Like the previous ones, this type is

declared opaque by the interface, and the necessary constructors and a destructor are provided in the interface.

```
type mem_value
val unspecified_mval : Ctype.ctype → mem_value
val integer_value_mval : Ctype.integerType → integer_value → mem_value
val floating_value_mval : Ctype.floatingType → floating_value → mem_value
val pointer_value_mval : Ctype.ctype → pointer_value → mem_value
val array_mval : mem_value list → mem_value
val struct_mval :
  Symbol.sym →
  (Symbol.identifier * Ctype.ctype * mem_value) list → mem_value
val union_mval :
  Symbol.sym →
  Symbol.identifier →
  mem_value → mem_value
```

Constructors receive, as part of their operands, the C types of the values being constructed. This is necessary because implementations of the memory interface will typically internally convert between abstract values and their byte representations.

```
val case_mem_value :
  mem_value                                                          →
  (Ctype.ctype → 'a)                                                 →
  (Ctype.integerType → integer_value → 'a)                           →
  (Ctype.floatingType → floating_value → 'a)                         →
  (Ctype.ctype → pointer_value → 'a)                                 →
  (mem_value list → 'a)                                              →
  (Symbol.sym → (Symbol.identifier * Ctype.ctype * mem_value) list → 'a)  →
  (Symbol.sym → Symbol.identifier → mem_value → 'a)                  → 'a
```

A.3 Race detection

To allow the detection of unsequenced races, in addition to their usual result, load and store accesses return an abstract *footprint* for that access. In a concrete memory object model this would be implemented as the range of byte address that were accessed. From two footprints, the Core dynamics can check whether they overlap, and raise an undefined behaviour accordingly.

```
type footprint
val overlapping : footprint → footprint → bool
```

A.4 Memory actions

The memory actions of Core have direct counterparts in the interface which are used to define their dynamics. All of these make use of the memory state, and some of them may result in an undefined behaviour. They therefore all operate inside the memory monad.

Allocations of objects and regions

val *allocate_object* :
Mem_common.thread_id
→ Symbol.prefix
→ integer_value
→ Ctype.ctype
→ mem_value option
→ pointer_value memM

val *allocate_region* :
Mem_common.thread_id
→ Symbol.prefix
→ integer_value
→ integer_value
→ pointer_value memM

The action on the left is used to allocate a new object with a known type; this corresponds to allocations from C identifiers. The first argument is the identifier of the thread performing the allocation. This is used for the modelling of C/C++11 concurrency, as we discuss in Chapter 10. The second argument holds the source information about this allocation (typically the C identifier); this is used to produce useful error messages and has no impact on the semantics of the action. The third and fourth arguments are respectively, the alignment constraint of the allocation, given as an integer value, and the type of allocation. From the latter, the memory object model will be able to derive the size. The final argument is an optional initial value for the new object. If it is present, the allocation is initialised but also set to read-only. This is used in the modelling of `const`-qualified identifiers and string literals. Such a call corresponds to Core's **`allocate_object_readonly`**(). If no value is given, the object starts with an unspecified value, and remains writable. The result of this action is a pointer value referring to the newly allocated object.

For the allocation of dynamic regions, the action on the right is used. Its signature only differs from the previous in the absence of the optional initial value, and the fact that the C type is replaced by a second integer value giving the desired size for the region.

Deallocations The Core action performing deallocation also has a direct counterpart in the interface, with the following signature:

val *kill* : Location_ocaml.t → bool → pointer_value → unit memM

The first operand is the C source location for the action, and has no impact on the dynamics; the second operand indicates whether the action comes from the elaboration of dynamic deallocation, e.g. a call to `free()`; the third operand is the pointer value used for the allocation; and the action yields no value.

Accesses The last two actions are the counterparts of Core's **`load`**() and **`store`**() memory accesses. Their signatures follow those of the Core actions, with the addition of a C source location (which again have no impact on their behaviour) as first operand; and, the fact that their results hold the footprint of the accesses (in addition to the loaded value for **`load`**()).

val *load* : Location_ocaml.t → Ctype.ctype → pointer_value →
(footprint ∗ mem_value) memM
val *store* : Location_ocaml.t → Ctype.ctype → bool → pointer_value → mem_value →
footprint memM

The loaded and stored values have the type mem_value, and may therefore be unspecified. As we have seen in Section 4.6, these are then pattern-matched by Core programs as needed.

A.5 Operations on pointer values

Core is equipped with the necessary operations over pointer values to model C's equality and relational operators, and the subtraction operator between two pointer expressions. These appear in effectful expressions as *memory operations*, and have direct counterparts in the interface:

val *eq_ptrval* : pointer_value → pointer_value → bool memM
val *ne_ptrval* : pointer_value → pointer_value → bool memM
val *lt_ptrval* : pointer_value → pointer_value → bool memM
val *gt_ptrval* : pointer_value → pointer_value → bool memM
val *le_ptrval* : pointer_value → pointer_value → bool memM
val *ge_ptrval* : pointer_value → pointer_value → bool memM
val *diff_ptrval* :
Ctype.ctype → pointer_value → pointer_value → integer_value memM

All of these operate within the memory monad to allow implementations in provenance-based memory object model to access their ghost state. Furthermore, in the case of the relational operations, even a naive implementation will need the monad, as these operations may cause an undefined behaviour. The last operation corresponds to Core's `Ptrdiff`, used in the elaboration of C's pointer subtraction. The first argument holds the referenced type of pointer operands, necessary to calculate the integer offset.

The interface additionally exposes two predicates, allowing the Core dynamics to check whether it is well-defined to dereference a pointer value (used in the elaboration of lvalue conversions), and whether a pointer value satisfies the alignment constraint of a given referenced type (used in the elaboration of casts between between two pointer types).

val *validForDeref_ptrval* : Ctype.ctype → pointer_value → bool memM
val *isWellAligned_ptrval* : Ctype.ctype → pointer_value → bool memM

A.6 Casting operations

Conversions between integer and pointer values in Core are made using two explicit operations, with the following counterparts in the interface:

val *ptrfromint* : Ctype.ctype → Ctype.ctype → integer_value →
pointer_value memM
val *intfromptr* : Ctype.ctype → Ctype.integerType → pointer_value →
integer_value memM

The first operand of *ptrfromint*, which converts an integer value into a pointer, holds the C type of the integer value, and the second operand is the referenced type for the pointer. Conversely, for *intfromptr*, which converts a pointer value into an integer, the first operand is the referenced type of the pointer, and the second is the desired integer type. For the same reasons as for the equality and relational operators, both conversions are using the memory monad.

A.7 Pointer arithmetic operators

There are two operators for adding or subtracting from pointers, corresponding directly to the shift operators in Core:

val *array_shift_ptrval* :
  pointer_value → Ctype.ctype → integer_value → pointer_value
val *member_shift_ptrval* :
  pointer_value → Symbol.sym → Symbol.identifier → pointer_value

The first one is used in the elaboration of C's additive operators, when applied to one pointer operand against an integer operand. The C type holds the referenced type of the pointer expression in C. Implementations are expected to provide an operator that behaves such that the address of *array_shift_ptrval*(p, τ, n) is the address of p added with $n * \texttt{sizeof}(\tau)$. The second operator is used in the elaboration of struct and union member access. The first operand is the pointer to the struct/union object; the second operand is the tag of the struct/union type; and the third operand is the identifier of the member.

Note that neither of these operators are within the memory monad; they therefore always succeed. Having the corresponding Core operator be part of the pure language has the advantage of not making them part of the sequencing calculus. This makes for more readable Core programs; in particular, this allows simplifying partial evaluation to achieve better result. For *member_shift_ptrval*, by the typing of Ail and the construction of the elaboration, we know that operator can never fail. For *array_shift_ptrval*, the default behaviour of Cerberus is to relax the ISO C restriction on C's pointer arithmetic by not making undefined the construction of out-of-bound pointer values. This is motivated by the observation that programmers routinely perform such arithmetic (see Question 9 in Chapter 2). However, Cerberus also allows the user to switch to the strict ISO semantics of pointer arithmetic. For this, the interface declares an effectful version of the operator, which behaves like the pure version except for the out-of-bound cases where it raise an undefined behaviour.

val *eff_array_shift_ptrval* : pointer_value → Ctype.ctype → integer_value →
  pointer_value memM

A.8 Operations on integer and floating values

The interface declares the usual arithmetic operations for integer and floating values: `+`, `-`, `*`, `/`. Because integer values are unbounded, these are always successful, and none of these operations are performed within the monad. For the division operator, when the second operand is zero, the result is zero. The corresponding C undefined behaviour is dealt explicitly by the Core expressions produced by the elaboration function. For floating operations, Cerberus does not faithfully model their semantics (merely using the underlying OCaml implementation), and therefore these do not need access to the state either. There are three more operations only available on integer values: the truncating remainder (used for the elaboration of C's modulo operator); the floored remainder; and

the exponent.

val *op_ival* : Mem_common.integer_operator →
integer_value → integer_value → integer_value
val *op_fval* : Mem_common.floating_operator →
floating_value → floating_value → floating_value

Bitwise operations There are counterparts to C's bitwise operators for integer values. Our symbolic memory needs to know (for performance reasons) the bit width of the operations, which is provided by the first operand, as the integer type of the C expression being elaborated.

val *bitwise_complement_ival* :
Ctype.integerType → integer_value → integer_value
val *bitwise_and_ival* :
Ctype.integerType → integer_value → integer_value → integer_value
val *bitwise_or_ival* :
Ctype.integerType → integer_value → integer_value → integer_value
val *bitwise_xor_ival* :
Ctype.integerType → integer_value → integer_value → integer_value

Predicates on integer and floating values The interface declares the equality and comparison operators over integer values as partial functions returning a boolean. The partiality is to accommodate the symbolic memory object model, where it is sometimes not possible to compute the result without accessing some ghost state. We chose to introduce this partiality, to keep these operators outside of the monad and avoid making the corresponding Core operations effectful. In the symbolic memory object model, when the boolean result cannot be decided, `None` is returned; the driver of the Core dynamics then deals with this case by introducing a guarded branch using the nondeterminism monad. For all our other memory object models, these operations always return `Some` boolean value.

val *eq_ival* : integer_value → integer_value → bool option
val *lt_ival* : integer_value → integer_value → bool option
val *le_ival* : integer_value → integer_value → bool option

For floating values, however, there is no such complication, as the symbolic memory object model does not support symbolic branching on them:

val *eq_fval* : floating_value → floating_value → bool
val *lt_fval* : floating_value → floating_value → bool
val *le_fval* : floating_value → floating_value → bool

Casting between integer and floating values There are two counterparts to Core's explicit conversion operators between integer and floating values.

val *fvfromint* : integer_value → floating_value
val *ivfromfloat* : floating_value → integer_value

A.9 Additional actions to support the C standard library

Finally, the interface declares a few memory actions to accommodate the elaboration of key memory management functions from the C standard library:

val *memcpy* : pointer_value → pointer_value → integer_value → pointer_value memM
val *memcmp* : pointer_value → pointer_value → integer_value → integer_value memM
val *realloc* :
Mem_common.thread_id → integer_value → pointer_value → integer_value → pointer_value memM

and for the elaboration of user-defined variadic functions:

val *va_start* : (Ctype.ctype ∗ pointer_value)*list* → integer_value memM
val *va_copy* : integer_value → integer_value memM
val *va_arg* : integer_value → *Ctype.ctype* → pointer_value memM
val *va_end* : integer_value → unit memM
val *va_list* : $\mathbb{Z}$ → ((Ctype.ctype ∗ pointer_value) list) memM

Appendix B

Source of the elaboration function

In this appendix, we show the Lem source code of the elaboration function from Ail to Core. The code is presented without modification. It is broken down into sections for easy access, and we skip some small portions of the code only having to do with boilerplate. The corresponding Lem file can be found at https://github.com/rems-project/cerberus/blob/master/frontend/model/translation.lem.

The top-level functions performing the elaboration of Ail expressions and Ail statements are shown in Sections B.12 and B.14. They are defined by recursion over the Ail AST. The Lem module defining the AST types for Ail is referred to as `A`. Data constructors for the expression AST nodes have the prefix `A.AilE`, and data constructors for statement AST nodes have the prefix `A.AilS`. The Lem module defining the AST types for Core is referred to as `C`. It is mostly not directly used, instead Core expressions are constructed using auxiliary functions exposed by the module referred to as `Caux`. For example, a pure `if` expression is constructed using the function `Caux.mk_if_pe`, and the `unseq` operator (part of the effectful subset of Core) is constructed using `Caux.mk_unseq_e`. To better differentiate "elaboration time" control-flow (i.e. written in Lem, as opposed to control-flow in the generated Core), we indent the Lem `if then else` and `match` expressions independently from the code constructing the Core expressions.

B.1 Elaboration of "compares equal to 0"

```
59  (* STD §6.5.13#3, sentence 1 *)
60  (* STD §6.5.14#3, sentence 1 *)
61  (* STD §6.5.15#4, sentence 2 *)
62  (* STD §6.7.10#2 *)
63  (* STD §6.8.4.1#2, sentence 1-2 *)
64  (* STD §6.8.5#4, sentence 1 *)
65  (* Some C constructs perform tests on scalar expression, with dynamic semantics
66     varying on whether its value "compares equal to 0". The semantics of the quoted
67     sentence implicitly refers to that of the C binary equality operator.
68     This function turns a [e] scalar expression into a [e == 0], so that elaboration
69     of the equality operator can be reused *)
70
71  type test_operator =
72    | TestEq
73    | TestNe
74
75  val mkTestExpression: forall 'a. test_operator -> A.expression GenTypes.genTypeCategory
    ↪  -> A.expression GenTypes.genTypeCategory
```

```
let mkTestExpression op (A.AnnotatedExpression gty annots _ _ as a_expr) =
  let loc = locOf a_expr in
  let gtc =
    (* STD §6.5.9#3, sentence 1 *)
    GenTypes.GenRValueType GenTypes.signedInt_gty in
  let bop = match op with
    | TestEq -> A.Eq
    | TestNe -> A.Ne
  end in
  let zero_const =
    if AilTypesAux.is_integer (ctype_of a_expr) then
      A.ConstantInteger (A.IConstant 0 A.Octal Nothing)
    else if AilTypesAux.is_floating (ctype_of a_expr) then
      A.ConstantFloating ("0.0", Nothing)
    else if AilTypesAux.is_pointer (ctype_of a_expr) then
      A.ConstantNull
    else
      (*BISECT-IGNORE*) error "[Translation.mkTestExpression] must be called on scalar
      ↪  expression" in
  A.AnnotatedExpression gtc annots loc
    (A.AilEbinary a_expr bop (A.AnnotatedExpression gty annots loc (A.AilEconst
    ↪  zero_const)))
```

B.2 Elaboration of constants

B.2.1 Integer constants in `case` statements

This auxiliary function is used to deal with the integer constants when elaborating `case` statements.

```
val translate_integerConstant: A.integerConstant -> C.pexpr
let translate_integerConstant iCst =
  Caux.mk_value_pe begin
    C.Vobject begin
      C.OVinteger begin
        match iCst with
          | A.IConstant n _ _ ->
              Mem.integer_ival n
          | A.IConstantMax ity ->
              Mem.max_ival ity
          | A.IConstantMin ity ->
              Mem.min_ival ity
        end
      end
    end
  end
```

B.2.2 Integer constants used as C11/Linux memory orders

```
val translate_memory_order: A.expression GenTypes.genTypeCategory -> Cmm.memory_order
let translate_memory_order (A.AnnotatedExpression _ _ _ expr as a_expr) =
  (* NOTE: we only support constant directly matching a memory order  *)
  match expr with
    | A.AilEconst (A.ConstantInteger (A.IConstant n _ _)) ->
        match Builtins.decode_memory_order (natFromInteger n) with
          | Just mo -> mo
```

```
136           | Nothing ->
137               error ("Translation.translate_memory_order: " ^ show n)
138         end
139     | (*BISECT-IGNORE*) _ ->
140         error ("Translation.translate_memory_order: " ^ Pp.stringFromAil_expression
            ↪ a_expr)
141   end
142
143
144 val translate_linux_memory_order: A.expression GenTypes.genTypeCategory ->
    ↪ Linux.linux_memory_order
145 let translate_linux_memory_order (A.AnnotatedExpression _ _ _ expr as a_expr) =
146   match expr with
147     | A.AilEconst (A.ConstantInteger (A.IConstant n _ _)) ->
148         match natFromInteger n with
149           | 0 -> Linux.Once
150           | 1 -> Linux.LAcquire
151           | 2 -> Linux.LRelease
152           | 3 -> Linux.Rmb
153           | 4 -> Linux.Wmb
154           | 5 -> Linux.Mb
155           | 6 -> Linux.RbDep
156           | 7 -> Linux.RcuLock
157           | 8 -> Linux.RcuUnlock
158           | 9 -> Linux.SyncRcu
159           | (*BISECT-IGNORE*) _ ->
160               error ("Translation.translate_linux_memory_order: " ^ show n)
161         end
162     | (*BISECT-IGNORE*) _ ->
163         error ("Translation.translate_linux_memory_order: " ^
            ↪ Pp.stringFromAil_expression a_expr)
164   end
```

B.2.3 All other constants

```
167 val     translate_constant: A.constant -> C.pexpr
168 let rec translate_constant cst =
169   match cst with
170     | A.ConstantIndeterminate ty ->
171         (* NOTE: we assume the lack of trap representation *)
172         Caux.mk_unspecified_pe ty
173     | A.ConstantNull ->
174         Caux.mk_specified_pe (Caux.mk_nullptr_pe Cty.void)
175     | A.ConstantInteger (A.IConstant n _ _) ->
176         Caux.mk_value_pe (C.Vloaded (C.LVspecified (C.OVinteger (Mem.integer_ival n))))
177     | A.ConstantInteger (A.IConstantMax ity) ->
178         Caux.mk_value_pe (C.Vloaded (C.LVspecified (C.OVinteger (Mem.max_ival ity))))
179     | A.ConstantInteger (A.IConstantMin ity) ->
180         Caux.mk_value_pe (C.Vloaded (C.LVspecified (C.OVinteger (Mem.min_ival ity))))
181     | A.ConstantFloating (str, _) ->
182         (* TODO: when we support float vs double, we'll need to inspect the suffix *)
183         Caux.mk_value_pe (C.Vloaded (C.LVspecified (C.OVfloating (Mem.str_fval str))))
184     | A.ConstantCharacter (_, str) ->
185         (* NOTE: making an implementation fix here (ASCII) *)
186         Caux.mk_value_pe (C.Vloaded (C.LVspecified (C.OVinteger (Mem.integer_ival
            ↪ (Decode.decode_character_constant str)))))
187     | A.ConstantArray _ csts ->
188         Caux.mk_specified_pe (Caux.mk_array_pe (List.map translate_constant csts))
```

```
189      | A.ConstantStruct tag_sym xs ->
190          Caux.mk_specified_pe begin
191            Caux.mk_struct_pe tag_sym begin
192              List.map (fun (memb_ident, cst) -> (memb_ident, translate_constant cst)) xs
193            end
194          end
195      | A.ConstantUnion tag_sym memb_ident pe ->
196          Caux.mk_specified_pe (Caux.mk_union_pe tag_sym memb_ident (translate_constant
         ↪  pe))
197    end
```

B.3 Elaboration of function designators

```
200  val translate_function_designator:
201    (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
202    translation_stdlib ->
203    A.expression GenTypes.genTypeCategory ->
204    E.elabM (C.expr unit)
205  let translate_function_designator translate_expr stdlib (A.AnnotatedExpression _ _ _
     ↪  expr as a_expr) =
206    let (Ctype.Ctype _ cty as ty) = ctype_of a_expr in
207    match (cty, expr) with
208      | (Ctype.Function _ params is_variadic, A.AilEunary A.Indirection e) ->
209          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun fun_wrp ->
210          translate_expr e                                    >>= fun core_e  ->
211          E.return (Caux.mk_sseq_e fun_wrp.E.sym_pat core_e (Caux.mk_pure_e
             ↪  fun_wrp.E.sym_pe))
212      | (Ctype.Function _ params is_variadic, A.AilEident fid) ->
213          let fid_pe = match fid with
214            | Symbol.Symbol _ _ (Symbol.SD_Id str) ->
215                match Map.lookup str stdlib.ailnames with
216                  | Just sym ->
217                      Caux.mk_value_pe (Core.Vloaded (C.LVspecified (Core.OVpointer
                         ↪  (Mem.fun_ptrval sym))))
218                  | Nothing ->
219                      Caux.mk_value_pe (Core.Vloaded (C.LVspecified (Core.OVpointer
                         ↪  (Mem.fun_ptrval fid))))
220                end
221            | _ ->
222                Caux.mk_value_pe (Core.Vloaded (C.LVspecified (Core.OVpointer
                   ↪  (Mem.fun_ptrval fid))))
223          end in
224          E.return (Caux.mk_pure_e fid_pe)
225      | (*BISECT-IGNORE*) _ ->
226          error ("[Translation.translate_function_designator] wildcard case ==> " ^
227                 Pp.stringFromAil_expression a_expr ^ " and type= " ^
                   ↪  Pp.stringFromAil_ctype Ctype.no_qualifiers ty)
228    end
```

B.4 Elaboration of multiplicative operators

B.4.1 The multiplication operator

```
231  (* STD §6.5.5 Multiplicative operators *)
232  val translate_mul_operator:
```

```
233   Loc.t ->
234   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
235   (Ctype.ctype -> Ctype.ctype -> C.pexpr -> C.pexpr -> C.pexpr * C.pexpr) ->
236   translation_stdlib ->
237   Ctype.ctype ->
238   A.expression GenTypes.genTypeCategory ->
239   A.expression GenTypes.genTypeCategory ->
240   E.elabM (C.expr unit)
241 let translate_mul_operator loc translate_expr usual_arithmetic_conversion stdlib
    ↪  result_ty e1 e2 =
242   let oTy1 = force_core_object_type_of_ctype (ctype_of e1) in
243   let oTy2 = force_core_object_type_of_ctype (ctype_of e2) in
244   translate_expr e1                          >>= fun core_e1  ->
245   translate_expr e2                          >>= fun core_e2  ->
246   E.wrapped_fresh_symbol (C.BTy_loaded oTy1) >>= fun e1_wrp   ->
247   E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp   ->
248   E.wrapped_fresh_symbol (C.BTy_object oTy1) >>= fun obj1_wrp ->
249   E.wrapped_fresh_symbol (C.BTy_object oTy2) >>= fun obj2_wrp ->
250   let (promoted1_pe, promoted2_pe) =
251     Caux.mk_std_pair_pe "§6.5.5#3"
252       (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2) obj1_wrp.E.sym_pe
        ↪  obj2_wrp.E.sym_pe) in
253   E.return begin
254     Caux.add_std "§6.5.5" (
255       Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
        ↪  (Caux.mk_unseq [core_e1; core_e2]) (
256         Caux.mk_pure_e (
257           Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
258             [ (Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
259                                  ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ],
260                (* Both operand are specified *)
261                let core_mul = Caux.mk_std_pe "§6.5.5#4" (Caux.mk_op_pe C.OpMul
                   ↪  promoted1_pe promoted2_pe) in
262                Caux.mk_specified_pe begin
263 if AilTypesAux.is_signed_integer_type result_ty then
264                  stdlib.mkcall_catch_exceptional_condition result_ty core_mul
265 else if AilTypesAux.is_integer result_ty then
266                  stdlib.mkcall_wrapI result_ty core_mul
267 else
268                  core_mul
269                end )
270             ; (Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1; C.BTy_loaded oTy2]),
271                (* If either operand is unspecified, the result is also unspecified is
                   ↪  the
272                   result type of unsigned. Otherwise it is undef, since the
    ↪  multiplication
273                   may overflow *)
274 if AilTypesAux.is_unsigned_integer_type result_ty then
275                Caux.mk_unspecified_pe result_ty
276 else
277                Caux.mk_undef_exceptional_condition loc) ]
278         )
279       )
280     )
281   end
```

B.4.2 The division and modulo operators

```
284 val translate_div_mod_operator:
285   Loc.t ->
286   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
287   (Ctype.ctype -> Ctype.ctype -> C.pexpr -> C.pexpr -> C.pexpr * C.pexpr) ->
288   translation_stdlib ->
289   Ctype.ctype ->
290   A.arithmeticOperator -> (* MUST BE A.Div or A.Mod *)
291   A.expression GenTypes.genTypeCategory ->
292   A.expression GenTypes.genTypeCategory ->
293   E.elabM (C.expr unit)
294 let translate_div_mod_operator loc translate_expr usual_arithmetic_conversion stdlib
    ↪  result_ty aop e1 e2 =
295   (* STD "§6.5.5" *)
296   let oTy1    = force_core_object_type_of_ctype (ctype_of e1) in
297   let oTy2    = force_core_object_type_of_ctype (ctype_of e2) in
298   let oTy_res = force_core_object_type_of_ctype result_ty     in
299   let zero_pe = match oTy_res with
300     | C.OTy_integer ->
301         Caux.mk_integer_pe 0
302     | C.OTy_floating ->
303         Caux.mk_floating_value_pe Mem.zero_fval
304     | (*BISECT-IGNORE*) _ ->
305       illTypedAil loc "AilEbinary Div, Mod"
306   end in
307   translate_expr e1                                >>= fun core_e1   ->
308   translate_expr e2                                >>= fun core_e2   ->
309   E.wrapped_fresh_symbol (C.BTy_loaded oTy1)    >>= fun e1_wrp    ->
310   E.wrapped_fresh_symbol (C.BTy_loaded oTy2)    >>= fun e2_wrp    ->
311   E.wrapped_fresh_symbol (C.BTy_object oTy1)    >>= fun obj1_wrp  ->
312   E.wrapped_fresh_symbol (C.BTy_object oTy2)    >>= fun obj2_wrp  ->
313   E.wrapped_fresh_symbol (C.BTy_object oTy_res) >>= fun conv1_wrp ->
314   E.wrapped_fresh_symbol (C.BTy_object oTy_res) >>= fun conv2_wrp ->
315   let (promoted1_pe, promoted2_pe) = Caux.mk_std_pair_pe "§6.5.5#3"
316     (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2) obj1_wrp.E.sym_pe
      ↪  obj2_wrp.E.sym_pe) in
317   let (ub, core_pe) = match aop with
318     | A.Div ->
319         ( Undefined.UB045a_division_by_zero
320         , Caux.mk_op_pe C.OpDiv promoted1_pe conv2_wrp.E.sym_pe )
321     | A.Mod ->
322         ( Undefined.UB045b_modulo_by_zero
323         , Caux.mk_op_pe C.OpRem_t conv1_wrp.E.sym_pe conv2_wrp.E.sym_pe )
324     | (*BISECT-IGNORE*) _ ->
325         error "[Translation.translate_div_mod_operator], 'aop' must be multiplicative"
326   end in
327   E.return begin
328     Caux.add_std "§6.5.5" (
329       Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
        ↪  (Caux.mk_unseq [core_e1; core_e2]) (
330         Caux.mk_pure_e (
331           Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
332             [ ( Caux.mk_tuple_pat [ Caux.mk_unspecified_pat (Caux.mk_empty_pat
                ↪  C.BTy_ctype)
333                                   ; Caux.mk_empty_pat (C.BTy_loaded oTy2) ]
334               ,
335 if AilTypesAux.is_signed_integer_type result_ty then
```

```
                Caux.mk_undef_exceptional_condition loc
else
                Caux.mk_unspecified_pe result_ty )

            ; ( Caux.mk_tuple_pat [ Caux.mk_empty_pat (C.BTy_loaded oTy1)
                                  ; Caux.mk_unspecified_pat (Caux.mk_empty_pat
                                  ↪  C.BTy_ctype) ]
              , Caux.mk_std_undef_pe loc "§6.5.5#5, sentence 2" ub )

            ; ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
                                  ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
              , Caux.mk_let_pe conv1_wrp.E.sym_pat promoted1_pe (
                  Caux.mk_let_pe conv2_wrp.E.sym_pat promoted2_pe (
                    Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpEq
                    ↪  conv2_wrp.E.sym_pe zero_pe)
                      (Caux.mk_std_undef_pe loc "§6.5.5#5, sentence 2" ub)
                      (* if a/b is representable *)
                      ( Caux.mk_if_pe_ [Annot.Anot_explode]
                      ↪  (stdlib.mkcall_is_representable (Caux.mk_op_pe C.OpDiv
                      ↪  promoted1_pe conv2_wrp.E.sym_pe) result_ty)
                          begin
                            Caux.mk_specified_pe (Caux.mk_std_pe "§6.5.5#5, sentence 1"
                            ↪  begin
                              if AilTypesAux.is_signed_integer_type result_ty then
                                stdlib.mkcall_catch_exceptional_condition result_ty
                                ↪  core_pe
                              else if AilTypesAux.is_integer result_ty then
                                stdlib.mkcall_wrapI result_ty core_pe
                              else
                                core_pe
                            end)
                          end
                          (Caux.mk_undef_pe loc
                          ↪  Undefined.UB045c_quotient_not_representable) )
                    )
                  ) ) ]
          )
        )
      )
    end
```

B.5 Elaboration of relational operators

```
(* STD §6.5.8 Relational operators *)
val translate_relational_operator:
  (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
  (Ctype.ctype -> Ctype.ctype -> C.pexpr -> C.pexpr -> C.pexpr * C.pexpr) ->
  Ctype.ctype ->
  A.binaryOperator -> (* MUST BE in { A.Lt, A.Gt. A.Le, A.Ge } *)
  A.expression GenTypes.genTypeCategory ->
  A.expression GenTypes.genTypeCategory ->
  E.elabM (C.expr unit)
let translate_relational_operator translate_expr usual_arithmetic_conversion result_ty
↪  bop e1 e2 =
  (* STD "§6.5.8" *)
  let oTy1 = force_core_object_type_of_ctype (ctype_of e1) in
  let oTy2 = force_core_object_type_of_ctype (ctype_of e2) in
```

```
384   translate_expr e1                          >>= fun core_e1   ->
385   translate_expr e2                          >>= fun core_e2   ->
386   E.wrapped_fresh_symbol (C.BTy_loaded oTy1) >>= fun e1_wrp    ->
387   E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp    ->
388   E.wrapped_fresh_symbol (C.BTy_object oTy1) >>= fun obj1_wrp  ->
389   E.wrapped_fresh_symbol (C.BTy_object oTy2) >>= fun obj2_wrp  ->
390   E.wrapped_fresh_symbol C.BTy_boolean       >>= fun memop_wrp ->
391   (* The object type on which the Core operator is going to work on. *)
392   (* From Ail's typing it is enough to look at the type of one of the operand (see STD
      ↪  §6.5.8#2) *)
393   let real_bop = match bop with
394     | A.Lt -> C.OpLt
395     | A.Gt -> C.OpGt
396     | A.Le -> C.OpLe
397     | A.Ge -> C.OpGe
398     | (*BISECT-IGNORE*) _ ->
399         error "[Translation.translate_relational_operator], 'bop' must be relational"
400   end in
401   E.return begin
402     Caux.add_std "§6.5.8" (
403       Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
          ↪  (Caux.mk_unseq [core_e1; core_e2]) (
404         Caux.mk_case_e (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
405             [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
406                                   ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
407               ,
408 begin if AilTypesAux.is_real (ctype_of e1) then
409                 let (promoted1_pe, promoted2_pe) =
410                   Caux.mk_std_pair_pe "§6.5.8#3"
411                     (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2)
                        ↪  obj1_wrp.E.sym_pe obj2_wrp.E.sym_pe) in
412                 Caux.add_std "§6.5.8#6" (
413                   Caux.mk_pure_e (
414                     Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe real_bop
                        ↪  promoted1_pe promoted2_pe)
415                       (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
416                       (Caux.mk_specified_pe (Caux.mk_integer_pe 0))
417                   )
418                 )
419 else
420                 let memop = match bop with
421                   | A.Lt -> Mem_common.PtrLt
422                   | A.Gt -> Mem_common.PtrGt
423                   | A.Le -> Mem_common.PtrLe
424                   | A.Ge -> Mem_common.PtrGe
425                   | (*BISECT-IGNORE*) _ -> error
                      ↪  "[Translation.translate_relational_operator], 'bop' must be
                      ↪  relational"
426                 end in
427                 Caux.mk_wseq_e memop_wrp.E.sym_pat (C.Expr [] (C.Ememop memop
                    ↪  [obj1_wrp.E.sym_pe; obj2_wrp.E.sym_pe])) (
428                   Caux.add_std "§6.5.8#6" (
429                     Caux.mk_pure_e (
430                       Caux.mk_if_pe_ [Annot.Anot_explode] memop_wrp.E.sym_pe
431                         (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
432                         (Caux.mk_specified_pe (Caux.mk_integer_pe 0))
433                     )
434                   )
```

```
435                 )
436 end            )
437             ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1; C.BTy_loaded oTy2])
438               , Caux.mk_pure_e (Caux.mk_unspecified_pe result_ty) ) ]
439       )
440    )
441   end
```

B.6 Elaboration of equality operators

```
444 (* STD §6.5.9 Equality operators *)
445 val translate_equality_operator:
446   Loc.t ->
447   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
448   (Ctype.ctype -> Ctype.ctype -> C.pexpr -> C.pexpr -> C.pexpr * C.pexpr) ->
449   Ctype.ctype ->
450   A.binaryOperator -> (* MUST BE in { A.Eq, A.Ne } *)
451   A.expression GenTypes.genTypeCategory ->
452   A.expression GenTypes.genTypeCategory ->
453   E.elabM (C.expr unit)
454 let translate_equality_operator loc translate_expr usual_arithmetic_conversion result_ty
    ↪  bop e1 e2 =
455       (* STD §6.5.9 *)
456 if    Aaux.is_null_pointer_constant e1 && AilTypesAux.is_pointer (ctype_of e2)
457    || AilTypesAux.is_pointer (ctype_of e1) && Aaux.is_null_pointer_constant e2 then
458   (* equality test between a null pointer constant and pointer *)
459   let e = if Aaux.is_null_pointer_constant e1 then e2 else e1 in
460   let nullptr_pe = Caux.mk_std_pe "§6.5.9#5, sentence 2" (Caux.mk_nullptr_pe (ctype_of
    ↪  e)) in
461   translate_expr e                                    >>= fun core_e    ->
462   E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun e_wrp     ->
463   E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj_wrp   ->
464   E.wrapped_fresh_symbol C.BTy_boolean                >>= fun memop_wrp ->
465   let memop = match bop with
466     | A.Eq -> Mem_common.PtrEq
467     | A.Ne -> Mem_common.PtrNe
468     | (*BISECT-IGNORE*) _ ->
469         error "[Translation.translate_equality_operator], 'bop' must be an equality
          ↪  operator"
470   end in
471   E.return begin
472     Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
473       Caux.mk_case_e e_wrp.E.sym_pe
474         [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
475           , Caux.mk_wseq_e memop_wrp.E.sym_pat (Caux.mk_memop_e memop [obj_wrp.E.sym_pe;
            ↪  nullptr_pe]) begin
476               Caux.add_std "§6.5.9#3" begin
477                 Caux.mk_pure_e begin
478                   Caux.mk_if_pe_ [Annot.Anot_explode] memop_wrp.E.sym_pe
479                     (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
480                     (Caux.mk_specified_pe (Caux.mk_integer_pe 0))
481                 end
482               end
483             end )
484           ; ( Caux.mk_empty_pat (C.BTy_loaded C.OTy_pointer)
485             , Caux.mk_pure_e (Caux.mk_undef_pe loc (Undefined.UB_CERB004_unspecified
              ↪  Undefined.UB_unspec_equality_ptr_vs_NULL)) ) ]
```

```
486       end
487     end
488
489   else (* operands both have arithmetic or pointer types *)
490     (* The object type on which the Core operator is going to work on. *)
491     let oTy1 = force_core_object_type_of_ctype (ctype_of e1) in
492     let oTy2 = force_core_object_type_of_ctype (ctype_of e2) in
493     E.wrapped_fresh_symbol (C.BTy_loaded oTy1) >>= fun e1_wrp    ->
494     E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp    ->
495     E.wrapped_fresh_symbol (C.BTy_object oTy1) >>= fun obj1_wrp  ->
496     E.wrapped_fresh_symbol (C.BTy_object oTy2) >>= fun obj2_wrp  ->
497     E.wrapped_fresh_symbol C.BTy_boolean       >>= fun memop_wrp ->
498     translate_expr e1                          >>= fun core_e1   ->
499     translate_expr e2                          >>= fun core_e2   ->
500     E.return begin
501       Caux.mk_wseq_e (Caux.mk_tuple_pat [e1_wrp.E.sym_pat; e2_wrp.E.sym_pat])
        ↪  (Caux.mk_unseq [core_e1; core_e2]) begin
502   if AilTypesAux.is_arithmetic (ctype_of e1) && AilTypesAux.is_arithmetic (ctype_of e2)
    ↪  then
503     let mk_op_pe = match bop with
504       | A.Eq -> Caux.mk_op_pe C.OpEq
505       | A.Ne -> fun x y -> Caux.mk_not_pe (Caux.mk_op_pe C.OpEq x y)
506       | (*BISECT-IGNORE*) _ ->
507           error "[Translation.translate_equality_operator], 'bop' must be an equality
            ↪  operator"
508     end in
509         Caux.mk_pure_e begin
510           Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
511             [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat;
              ↪  Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
512               , let (promoted1_pe, promoted2_pe) =
513                   Caux.mk_std_pair_pe "§6.5.9#4, sentence 1"
514                     (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2)
                      ↪  obj1_wrp.E.sym_pe obj2_wrp.E.sym_pe) in
515                 Caux.mk_std_pe "§6.5.9#3" begin
516                   Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_std_pe "§6.5.9#4, sentence
                    ↪  3" (mk_op_pe promoted1_pe promoted2_pe))
517                     (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
518                     (Caux.mk_specified_pe (Caux.mk_integer_pe 0))
519                 end )
520             ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1; C.BTy_loaded oTy2])
521               , Caux.mk_unspecified_pe result_ty ) ]
522         end
523
524   else (* both operand have pointer type *)
525     let memop = match bop with
526     | A.Eq -> Mem_common.PtrEq
527     | A.Ne -> Mem_common.PtrNe
528     | (*BISECT-IGNORE*) _ ->
529         error "[Translation.translate_equality_operator], 'bop' must be an equality
          ↪  operator"
530     end in
531        (* NOTE: our modelling of ptr <-> ptr casting is the identity,
532                 so nothing is done here for (§6.5.9#5 sentence 3) *)
533         Caux.mk_case_e (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
534           [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat;
            ↪  Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
```

```
535           , Caux.mk_wseq_e memop_wrp.E.sym_pat (C.Expr [] (C.Ememop memop
              ↪  [obj1_wrp.E.sym_pe; obj2_wrp.E.sym_pe])) begin
536               Caux.mk_pure_e begin
537                 Caux.mk_std_pe "§6.5.9#3" begin
538                   Caux.mk_if_pe_ [Annot.Anot_explode] memop_wrp.E.sym_pe
539                     (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
540                     (Caux.mk_specified_pe (Caux.mk_integer_pe 0))
541                 end
542               end
543             end )
544         ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1; C.BTy_loaded oTy2])
545           , Caux.mk_pure_e (Caux.mk_undef_pe loc (Undefined.UB_CERB004_unspecified
              ↪  Undefined.UB_unspec_equality_both_arith_or_ptr)) ) ]
546     end
547   end
```

B.7 Elaboration of bitwise operators

```
550 (* STD §6.5.10  Bitwise AND operator          *)
551 (* STD §6.5.11  Bitwise exclusive OR operator *)
552 (* STD §6.5.12  Bitwise inclusive OR operator *)
553 val translate_bitwise_operator:
554   Loc.t ->
555   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
556   (Ctype.ctype -> Ctype.ctype -> C.pexpr -> C.pexpr -> C.pexpr * C.pexpr) ->
557   translation_stdlib ->
558   Ctype.ctype ->
559   A.arithmeticOperator -> (* MUST BE in { A.Band, A.Bxor, A.Bor } *)
560   A.expression GenTypes.genTypeCategory ->
561   A.expression GenTypes.genTypeCategory ->
562   E.elabM (C.expr unit)
563 let translate_bitwise_operator loc translate_expr usual_arithmetic_conversion stdlib
    ↪  result_ty aop e1 e2 =
564   let (std_id, stdlib_call) = match aop with
565     | A.Band -> ("§6.5.10", (fun ty pe1 pe2 -> C.Pexpr [] () (C.PEctor C.CivAND
      ↪  [Caux.mk_ail_ctype_pe ty; pe1; pe2])))
566     | A.Bxor -> ("§6.5.11", (fun ty pe1 pe2 -> C.Pexpr [] () (C.PEctor C.CivXOR
      ↪  [Caux.mk_ail_ctype_pe ty; pe1; pe2])))
567     | A.Bor  -> ("§6.5.12", (fun ty pe1 pe2 -> C.Pexpr [] () (C.PEctor C.CivOR
      ↪  [Caux.mk_ail_ctype_pe ty; pe1; pe2])))
568     | (*BISECT-IGNORE*) _ ->
569         error "[Translation.translate_bitwise_operator], 'bop' must be a bitwise
          ↪  operator"
570   end in
571   E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e1_wrp   ->
572   E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e2_wrp   ->
573   E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj1_wrp ->
574   E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj2_wrp ->
575   translate_expr e1                                   >>= fun core_e1  ->
576   translate_expr e2                                   >>= fun core_e2  ->
577   let (promoted1_pe, promoted2_pe) =
578     Caux.mk_std_pair_pe (std_id ^ "#3")
579       (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2) obj1_wrp.E.sym_pe
        ↪  obj2_wrp.E.sym_pe) in
580   E.return begin
581     Caux.add_std std_id begin
```

```
582       Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
          ↪ (Caux.mk_unseq_e [core_e1; core_e2]) begin
583         Caux.mk_pure_e begin
584           Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
585             [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat ;
                ↪ Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
586               , (* Both operand are specified *)
587                 Caux.mk_specified_pe (Caux.mk_std_pe (std_id ^ "#4") (stdlib_call
                  ↪ result_ty promoted1_pe promoted2_pe)) )
588             ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded C.OTy_integer; C.BTy_loaded
                ↪ C.OTy_integer])
589               , Caux.mk_unspecified_pe result_ty ) ]
590         end
591       end
592     end
593   end
```

B.8 Elaboration of postfix operators

```
596 val translate_postfix:
597   Loc.t ->
598   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
599   translation_stdlib ->
600   Ctype.ctype ->
601   A.unaryOperator -> (* MUST BE in { A.PostfixIncr, A.PostfixDecr } *)
602   A.expression GenTypes.genTypeCategory ->
603   E.elabM (C.expr unit)
604 let translate_postfix loc translate_expr stdlib result_ty op e =
605   (* TODO: use atomic RMW if the type is atomic *)
606   (* NOTE: if I read N2329 correctly, in C2X this will not be an RMW for the atomic case
      ↪ (but a do while
607            with compare_exchange_weak with seq_cst, seq_cst) *)
608   let (std_para, core_op, ptr_shift_const) =
609     match op with
610       | A.PostfixIncr ->
611           ("#2", C.OpAdd, 1)
612       | A.PostfixDecr ->
613           ("#3", C.OpSub, 0 - 1)
614       | (*BISECT-IGNORE*) _ ->
615           error "[Translation.translate_postfix], 'op' must be a postfix operator"
616     end in
617   let std_sentence_n (n: nat) = "§6.5.2.4" ^ std_para ^ ", sentence " ^ show n in
618   (* STD §6.5.2.4 *)
619   warn_atomic_elaboration (AilTypesAux.is_atomic (ctype_of e)) >>= fun () ->
620   let ty  = AilTypesAux.rvalue_coercion (snd (from_lvalue_type e)) in
621   let oTy = force_core_object_type_of_ctype ty    in
622   let core_ty_e = Caux.mk_ail_ctype_pe ty in
623   E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun lvalue_wrp ->
624   E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun load_wrp   ->
625   E.wrapped_fresh_symbol (C.BTy_object oTy)           >>= fun obj_wrp    ->
626   translate_expr e                                    >>= fun core_e     ->
627   E.return begin
628     Caux.add_stds ["§6.5.2.4"; std_sentence_n 1; std_sentence_n 3] begin
629       Caux.mk_wseq_e lvalue_wrp.E.sym_pat core_e begin
630         Caux.seq_rmw loc false(* return the value of the load *) core_ty_e oTy
          ↪ lvalue_wrp.E.sym_pe load_wrp.E.sym_sym begin
631           Caux.mk_case_pe load_wrp.E.sym_pe
```

```
632             [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
633               , Caux.mk_specified_pe begin
634                   Caux.mk_std_pe (std_sentence_n 2)
635 match Ctype.unatomic_ ty with
636   | Ctype.Basic (Ctype.Integer _) ->
637                     let core_postfix = Caux.mk_op_pe core_op obj_wrp.E.sym_pe
                        ↪  (Caux.mk_integer_pe 1) in
638                     let promoted_ty =
639                       fromJust "Translation.translate_postfix promotion"
640                         (AilTypesAux.promotion integerImpl (ctype_of e)) in
641                     stdlib.mkcall_conv_int result_ty
642       begin if AilTypesAux.is_signed_integer_type promoted_ty then
643                       stdlib.mkcall_catch_exceptional_condition promoted_ty core_postfix
644       else (* is unsigned *)
645                       stdlib.mkcall_wrapI promoted_ty core_postfix
646       end
647   | Ctype.Basic (Ctype.Floating (Ctype.RealFloating _)) ->
648                     (* NOTE: we are not modelling floating UBs *)
649                     Caux.mk_op_pe core_op obj_wrp.E.sym_pe (Caux.mk_floating_value_pe
                        ↪  Mem.one_fval)
650   | Ctype.Pointer _ ref_ty ->
651                     Caux.mk_array_shift obj_wrp.E.sym_pe ref_ty (Caux.mk_integer_pe
                        ↪  ptr_shift_const)
652   | (*BISECT-IGNORE*) _ ->
653       illTypedAil loc "AilEunary PostfixIncr|PostfixDecr"
654 end
655                 end )
656             ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
657               , Caux.mk_unspecified_pe ty ) ]
658         end
659       end
660     end
661   end
```

B.9 Auxiliary function elaborating assignment-like conversions

```
664 val translate_assignment_conversion:
665   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
666   translation_stdlib ->
667   Ctype.ctype ->
668   A.expression GenTypes.genTypeCategory ->
669   E.elabM (C.core_object_type * C.expr unit * (C.pexpr -> C.pexpr))
670 let translate_assignment_conversion translate_expr stdlib ty1 e2 =
671   let ty2 = ctype_of e2 in
672   begin
673     if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_array ty1 then
674       translate_expr e2 >>= fun core_e2 ->
675       E.return
676         ( force_core_object_type_of_ctype ty2
677         , core_e2
678         , fun z -> z )
679     else if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_arithmetic ty1 &&
        ↪  AilTypesAux.is_arithmetic ty2 then
680       translate_expr e2 >>= fun core_e2 ->
681       E.return
```

```
682         ( force_core_object_type_of_ctype ty2
683         , core_e2
684         , conv_loaded_arith stdlib ty2 (Ctype.unatomic ty1) )
685       else if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_struct_or_union ty1
          ↪  then
686       (* NOTE: the two struct/union types could be from two different translation units,
687                but as far as I can things are sufficiently restricted such that no
      ↪   conversion is needed here *)
688         translate_expr e2 >>= fun core_e2 ->
689         E.return
690           ( force_core_object_type_of_ctype ty2
691           , core_e2
692           , fun z -> z )
693       (* NOTE: we apply unatomic to ty1 because the left operand may be an atomic pointer
          ↪  to ... (STD §6.5.16.1#1, bullet 3) *)
694       else match AilTypesAux.referenced_type (Ctype.unatomic ty1) with
695         | Just ref_ty ->
696           begin if Aaux.is_null_pointer_constant e2 then
697             E.return
698               ( C.OTy_pointer
699               , Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe ref_ty))
700               , fun z -> z )
701           else
702             translate_expr e2 >>= fun core_e2 ->
703             E.return
704               ( (*C.OTy_pointer*) force_core_object_type_of_ctype ty2
705               , core_e2
706               , fun z -> z )
707           end
708         | Nothing ->
709           (* By Ail's typing, e1 must have type _Bool and e2 must be a pointer *)
710           E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun conv_wrp ->
711           translate_expr e2 >>= fun core_e2 ->
712           E.return
713             ( C.OTy_integer
714             , Caux.mk_wseq_e conv_wrp.E.sym_pat core_e2
                ↪  (stdlib.mkproc_loaded_pointer_to_Bool conv_wrp.E.sym_pe)
715             , fun z -> z )
716       end
717     end
```

B.10 Elaboration of function calls

```
720 val translate_function_call:
721   Loc.t ->
722   bool -> (* is_used *)
723   (A.expression GenTypes.genTypeCategory -> E.elabM (C.expr unit)) ->
724   translation_stdlib ->
725   A.expression GenTypes.genTypeCategory ->
726   list (A.expression GenTypes.genTypeCategory) ->
727   E.elabM (C.expr unit)
728 let translate_function_call loc is_used translate_expr stdlib e es =
729   (* let is_used_pe = Caux.mk_boolean_pe is_used in *)
730   let (expect_ret_ty, expect_params, expect_is_variadic) =
731     match ctype_of e with
732       | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Function (_, ret_ty) qs_tys
          ↪  is_variadic))) ->
```

```
733           (ret_ty, qs_tys, is_variadic)
734       | (*BISECT-IGNORE*) _ ->
735           illTypedAil loc "AilEcall"
736     end in
737   let expect_param_is_Bool n =
738     match List.index expect_params n with
739       | Just (_, ty, _) ->
740         AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_Bool ty
741       | Nothing ->
742           false
743     end in
744   (* TODO: This is ignoring has_proto, §6.5.2.2#6 is not being considered! *)
745   (* STD §6.5.2.2 *)
746   E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun fun_wrp         ->
747   E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun call_wrp        ->
748   E.wrapped_fresh_symbol C.BTy_ctype                  >>= fun ret_wrp         ->
749   E.wrapped_fresh_symbol (C.BTy_list C.BTy_ctype)     >>= fun params_wrp      ->
750   E.wrapped_fresh_symbol C.BTy_boolean                >>= fun is_variadic_wrp ->
751   E.wrapped_fresh_symbol C.BTy_boolean                >>= fun has_proto_wrp   ->
752   (* elaborate the expression that denotes the called function *)
753   translate_expr e >>= fun core_e ->
754   (* symbolic names for the arguments temporary objects *)
755   let n_args = List.length es in
756   let arg_ptr_syms = mapi (fun i arg_e -> Symbol.fresh_funarg (locOf arg_e) i) es in
757   let arg_ptr_sym_pats = List.map (fun sym -> Caux.mk_sym_pat sym (C.BTy_object
      ↪  C.OTy_pointer)) arg_ptr_syms in
758   let arg_ptr_sym_pes = List.map Caux.mk_sym_pe arg_ptr_syms in
759   (* elaborate each argument *)
760   E.foldlM (fun (n, arg_sym_pats, core_arg_es, args_info) arg_e ->
761     begin if expect_param_is_Bool n && AilTypesAux.is_pointer (ctype_of arg_e) then
762       E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun conv_wrp ->
763       E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun arg_wrp  ->
764       translate_expr arg_e                                >>= fun core_e   ->
765       E.return
766         ( arg_wrp
767         , Caux.mk_wseq_e conv_wrp.E.sym_pat core_e (stdlib.mkproc_loaded_pointer_to_Bool
            ↪  conv_wrp.E.sym_pe) )
768     else
769       let arg_bTy = C.BTy_loaded (force_core_object_type_of_ctype (ctype_of arg_e)) in
770       E.wrapped_fresh_symbol arg_bTy >>= fun arg_wrp ->
771       translate_expr arg_e           >>= fun core_e  ->
772       E.return (arg_wrp, core_e)
773     end >>= fun (arg_wrp, core_arg_e) ->
774     E.return  ( (n+1)
775               , arg_wrp.E.sym_pat :: arg_sym_pats, core_arg_e :: core_arg_es
776               , (ctype_of arg_e, Aaux.is_null_pointer_constant arg_e, arg_wrp.E.sym_pe)
                  ↪  :: args_info )
777   ) (0, [], [], []) es >>= fun (_, rev_arg_sym_pats, rev_core_arg_es, rev_args_info) ->
778   (* create parameters and convert them *)
779   let (args_info, variadic_args_info) = List.splitAt (List.length expect_params)
      ↪  (List.reverse rev_args_info) in
780   (* standard arguments *)
781   E.foldlM (fun (n, rev_core_creates) ((_, expect_param_ty, _), (arg_ty, arg_is_null,
      ↪  arg_sym_pe)) ->
782     E.wrapped_fresh_symbol C.BTy_ctype                  >>= fun param_ty_wrp ->
783     E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun arg_ptr_wrp  ->
784     E.return
785       ( n+1
```

```
786       , (Caux.mk_let_e param_ty_wrp.E.sym_pat (stdlib.mkcall_params_nth
          ↪  params_wrp.E.sym_pe (Caux.mk_integer_pe n))
787         (Caux.mk_if_e_ [Annot.Anot_explode]
788           (Caux.mk_not_pe (Caux.mk_are_compatible (Caux.mk_ail_ctype_pe expect_param_ty)
              ↪  param_ty_wrp.E.sym_pe))
789           (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#9"
              ↪  Undefined.UB041_function_not_compatible))
790           begin
791             let conv_value =
792               if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_Bool
                  ↪  expect_param_ty && AilTypesAux.is_pointer arg_ty then
793                 arg_sym_pe
794               (* NOTE: since the expected type is compatible with the parameter type, if
                  ↪  one is an integer or
795                 floating, the other one must also be an integer or floating
↪  (respectively) *)
796               else if AilTypesAux.is_integer expect_param_ty then
797                 if AilTypesAux.is_integer arg_ty then
798                   stdlib.mkcall_conv_loaded_int_ param_ty_wrp.E.sym_pe arg_sym_pe
799                 else
800                   stdlib.mkcall_loaded_ivfromfloat_ param_ty_wrp.E.sym_pe arg_sym_pe
801               else if AilTypesAux.is_floating expect_param_ty then
802                 if AilTypesAux.is_integer arg_ty then
803                   stdlib.mkcall_loaded_fvfromint_ param_ty_wrp.E.sym_pe arg_sym_pe
804                 else
805                   arg_sym_pe
806               else if AilTypesAux.is_pointer expect_param_ty && arg_is_null then
807                 Caux.mk_specified_pe (Caux.mk_nullptr_pe expect_param_ty)
808               else
809                 arg_sym_pe in
810             let mo =
811               if AilTypesAux.is_atomic expect_param_ty then
812                 (* STD §6.2.6.1#9 *)
813                 Cmm.Seq_cst
814               else
815                 Cmm.NA in
816             Caux.add_std "§6.5.2.2#7, sentence 1" begin
817               Caux.mk_wseq_e arg_ptr_wrp.E.sym_pat
818                 (Caux.pcreate loc (Caux.mk_alignof_pe param_ty_wrp.E.sym_pe)
                    ↪  param_ty_wrp.E.sym_pe (Symbol.PrefFunArg loc (Symbol.digest ())
                    ↪  (intFromInteger n)))
819                 begin
820                   Caux.mk_wseq_e (Caux.mk_empty_pat C.BTy_unit)
821                     (Caux.pstore loc param_ty_wrp.E.sym_pe arg_ptr_wrp.E.sym_pe
                        ↪  conv_value mo)
822                     (Caux.mk_pure_e arg_ptr_wrp.E.sym_pe)
823                 end
824             end
825           end
826         )) :: rev_core_creates )
827   ) (0,[]) (List.zip expect_params args_info) >>= fun (_, rev_core_creates) ->
828   (* standard arguments (CN elaboration switch) *)
829   begin if Global.has_switch SW_inner_arg_temps then
830     E.foldlM (fun (n, cn_core_args) ((_, expect_param_ty, _), (arg_ty, arg_is_null,
      ↪  arg_sym_pe)) ->
831       E.wrapped_fresh_symbol C.BTy_ctype                  >>= fun param_ty_wrp ->
832       E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun arg_ptr_wrp ->
833       E.return
```

```
834         ( n+1
835         , (Caux.mk_let_pe param_ty_wrp.E.sym_pat (stdlib.mkcall_params_nth
            ↪  params_wrp.E.sym_pe (Caux.mk_integer_pe n))
836           (Caux.mk_if_pe_ [Annot.Anot_explode]
837             (Caux.mk_not_pe (Caux.mk_are_compatible (Caux.mk_ail_ctype_pe
                ↪  expect_param_ty) param_ty_wrp.E.sym_pe))
838             (Caux.mk_std_undef_pe loc "§6.5.2.2#9"
                ↪  Undefined.UB041_function_not_compatible)
839             begin
840               (* NOTE: since the expected type is compatible with the parameter type,
841                 if one is an integer or floating, the other one must also be an integer
    ↪  or floating (respectively) *)
842               if AilTypesAux.is_integer expect_param_ty then
843                 if AilTypesAux.is_integer arg_ty then
844                   stdlib.mkcall_conv_loaded_int_ param_ty_wrp.E.sym_pe arg_sym_pe
845                 else
846                   stdlib.mkcall_loaded_ivfromfloat_ param_ty_wrp.E.sym_pe arg_sym_pe
847               else if AilTypesAux.is_floating expect_param_ty then
848                 if AilTypesAux.is_integer arg_ty then
849                   stdlib.mkcall_loaded_fvfromint_ param_ty_wrp.E.sym_pe arg_sym_pe
850                 else
851                   arg_sym_pe
852               else if AilTypesAux.is_pointer expect_param_ty && arg_is_null then
853                 Caux.mk_specified_pe (Caux.mk_nullptr_pe expect_param_ty)
854               else
855                 arg_sym_pe
856             end
857           )) :: cn_core_args )
858     ) (0,[]) (List.zip expect_params args_info)
859   else
860     (* dummy empty list we are not using *)
861     E.return (0, [])
862   end >>= fun (_, rev_cn_core_args) ->
863   (* variadic arguments *)
864   E.foldlM (fun (rev_arg_tys, rev_arg_ty_pes, rev_variadic_core_creates) (arg_ty,
    ↪  arg_is_null, arg_sym_pe) ->
865     let (conv_ty, conv_value) =
866       if AilTypesAux.is_integer arg_ty then
867         let prom_ty = fromJust "translation: default arguments promotion"
            ↪  (AilTypesAux.promotion integerImpl arg_ty) in
868         (prom_ty, stdlib.mkcall_conv_loaded_int prom_ty arg_sym_pe)
869       else if AilTypesAux.is_floating arg_ty then
870         (Ctype.Ctype [] (Ctype.Basic (Ctype.Floating (Ctype.RealFloating
            ↪  Ctype.Double))), arg_sym_pe)
871       else
872         (arg_ty, arg_sym_pe)
873     in E.return ( conv_ty :: rev_arg_tys
874                 , Caux.mk_ail_ctype_pe conv_ty :: rev_arg_ty_pes
875                 , Caux.add_std "§6.5.2.2#7, sentences 2 and 3" begin
876                     stdlib.mkproc_create_and_store (Caux.mk_ail_ctype_pe conv_ty)
                        ↪  conv_value
877                   end :: rev_variadic_core_creates)
878   ) ([], [],[]) variadic_args_info >>= fun (rev_arg_tys, rev_arg_ty_pes,
    ↪  rev_variadic_core_creates) ->
879   (* function call result *)
880   let call_bTy = maybe C.BTy_unit C.BTy_loaded (Caux.core_object_type_of_ctype
    ↪  expect_ret_ty) in
881   E.wrapped_fresh_symbol call_bTy >>= fun call_ret_wrp ->
```

```
882     (* kill temporary objects *)
883     let killall_pat =
884       if List.length arg_ptr_syms < 2 then
885         Caux.mk_empty_pat C.BTy_unit
886       else
887         Caux.mk_empty_pat (C.BTy_tuple (List.replicate (List.length arg_ptr_syms)
          ↪  C.BTy_unit))
888     in
889     (* STD (§6.5.2.2#10, sentence 1) says there is sequence "point after the
890       evaluations of the function designator and the actual arguments but before
891       the actual call." *)
892     E.return begin
893       Caux.add_std "§6.5.2.2#10, sentence 1"
894       (Caux.mk_sseq_e
895         (Caux.mk_tuple_pat begin
896           (Caux.mk_tuple_pat  [ call_wrp.E.sym_pat
897                               ; Caux.mk_tuple_pat [ret_wrp.E.sym_pat;
                                  ↪  params_wrp.E.sym_pat; is_variadic_wrp.E.sym_pat;
                                  ↪  has_proto_wrp.E.sym_pat]])
898             :: (List.reverse rev_arg_sym_pats)
899         end)
900         begin
901           Caux.add_std "§6.5.2.2#4, sentence 2" begin
902             Caux.mk_unseq_e begin
903               (Caux.mk_sseq_e fun_wrp.E.sym_pat core_e
904                 (Caux.mk_pure_e (Caux.mk_tuple_pe [fun_wrp.E.sym_pe; Caux.mk_cfunction_pe
                    ↪  fun_wrp.E.sym_pe])))
905               :: (List.reverse rev_core_arg_es)
906             end
907           end
908         end
909     begin if expect_is_variadic then
910         (* check number of parameters *)
911         (Caux.mk_if_e_ [Annot.Anot_explode]
912           (Caux.mk_not_pe (Caux.mk_op_pe C.OpLe (stdlib.mkcall_params_length
            ↪  params_wrp.E.sym_pe)
913                                                 (Caux.mk_integer_pe (integerFromNat
                                                    ↪  n_args))))
914           (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#6, sentence 3"
            ↪  Undefined.UB038_number_of_args))
915           (* check if function types are compatible *)
916           (Caux.mk_if_e_ [Annot.Anot_explode]
917             (Caux.mk_op_pe C.OpOr (Caux.mk_not_pe is_variadic_wrp.E.sym_pe)
918                           (Caux.mk_not_pe (Caux.mk_are_compatible (Caux.mk_ail_ctype_pe
                              ↪  expect_ret_ty) ret_wrp.E.sym_pe)))
919             (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#9"
              ↪  Undefined.UB041_function_not_compatible))
920             (Caux.mk_sseqs
921               (* create temporary object *)
922               (List.zip arg_ptr_sym_pats (List.reverse rev_core_creates ++ List.reverse
                ↪  rev_variadic_core_creates))
923               (Caux.mk_sseq_e call_ret_wrp.E.sym_pat
924                 (* do the function call *)
925                 (Caux.mk_ccall_e (Caux.mk_ail_ctype_pe (ctype_of e)) call_wrp.E.sym_pe
926                   (let (arg_pes, vararg_pes) = List.splitAt (List.length expect_params)
                      ↪  arg_ptr_sym_pes in
927                     let varargs_ty_pes =
928                       List.map (fun (ty_pe, pe) -> Caux.mk_tuple_pe [ty_pe; pe])
```

```
929                     (List.zip (List.reverse rev_arg_ty_pes) vararg_pes) in
930                   let varargs_ty_pes_type =
931                     C.BTy_tuple [C.BTy_ctype; (C.BTy_object C.OTy_pointer)] in
932                   (*is_used_pe :: *) arg_pes ++ [Caux.mk_list_pe varargs_ty_pes_type
                      ↪  varargs_ty_pes]
933                 )
934               )
935               (Caux.mk_sseq_e killall_pat
936                 (* kill temporary objects *)
937                 (let arg_ptr_syms_tys =
938                   List.zip arg_ptr_syms
939                     (List.map (fun (_, ty, _) -> ty) expect_params ++ List.reverse
                        ↪  rev_arg_tys) in
940                   Caux.mk_unseq (List.map (fun (sym,ct) -> Caux.pkill loc (C.Static ct)
                      ↪  (Caux.mk_sym_pe sym)) arg_ptr_syms_tys))
941                 (* return function call result *)
942                 (Caux.mk_pure_e call_ret_wrp.E.sym_pe)
943               )
944             )
945           )
946         )
947       )
948 else
949       (* check number of parameters *)
950       (Caux.mk_if_e_ [Annot.Anot_explode]
951         (Caux.mk_not_pe (Caux.mk_op_pe C.OpEq (stdlib.mkcall_params_length
          ↪  params_wrp.E.sym_pe)
952                                               (Caux.mk_integer_pe (integerFromNat
                                                  ↪  n_args))))
953         (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#6, sentence 3"
          ↪  Undefined.UB038_number_of_args))
954         (* check if function types are compatible *)
955         (Caux.mk_if_e_ [Annot.Anot_explode]
956           (Caux.mk_op_pe C.OpOr is_variadic_wrp.E.sym_pe
957                          (Caux.mk_not_pe (Caux.mk_are_compatible (Caux.mk_ail_ctype_pe
                               ↪  expect_ret_ty) ret_wrp.E.sym_pe)))
958           (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#9"
            ↪  Undefined.UB041_function_not_compatible))
959 begin if Global.has_switch SW_inner_arg_temps then
960           (Caux.mk_ccall_e (Caux.mk_ail_ctype_pe (ctype_of e)) call_wrp.E.sym_pe
            ↪  ((*is_used_pe :: *)List.reverse rev_cn_core_args))
961 else
962             (* create temporary object *)
963             (Caux.mk_sseqs (List.zip arg_ptr_sym_pats (List.reverse rev_core_creates))
964             (Caux.mk_sseq_e call_ret_wrp.E.sym_pat
965               (* do the function call *)
966               (Caux.mk_ccall_e (Caux.mk_ail_ctype_pe (ctype_of e)) call_wrp.E.sym_pe
                ↪  ((*is_used_pe :: *)arg_ptr_sym_pes))
967               (Caux.mk_sseq_e killall_pat
968                 (* kill temporary objects *)
969                 (let arg_ptr_syms_tys = List.map (fun (sym, (_, ty, _)) -> (sym, ty))
                  ↪  (List.zip arg_ptr_syms expect_params) in
970                   Caux.mk_unseq (List.map (fun (sym,ct) -> Caux.pkill loc (C.Static ct)
                    ↪  (Caux.mk_sym_pe sym)) arg_ptr_syms_tys))
971                 (* return function call result *)
972                 (Caux.mk_pure_e call_ret_wrp.E.sym_pe)
973               )
974             )
```

```
          )
end
        )
      )
end
    )
  end
```

B.11 Elaboration of C11/Linux explicit atomic operations

```
type atomic_explicit =
  | AtomicStoreExplicit
  | AtomicLoadExplicit
  | AtomicThreadFence
  | AtomicCompareExchangeStrongExplicit
  | AtomicCompareExchangeWeakExplicit
  | LinuxStore
  | LinuxLoad
  | LinuxFence
  | LinuxRMW

let translate_atomic_explicit loc translate_expr atomic_op args =
  match (atomic_op, args) with
    | (AtomicStoreExplicit, [lobject_e; desired_e; order_e]) ->
        let mo = translate_memory_order order_e in
        let ref_ty = match ctype_of lobject_e with
          | Ctype.Ctype _ (Ctype.Pointer _ ref_ty) ->
              ref_ty
          | (*BISECT-IGNORE*) _ ->
              illTypedAil loc "AilEcall atomic_store_explicit"
        end in
        let oTy = force_core_object_type_of_ctype (ctype_of desired_e) in
        E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_wrp     ->
        E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp     ->
        E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun desired_wrp    ->
        translate_expr lobject_e                            >>= fun lobject_core_e ->
        translate_expr desired_e                            >>= fun desired_core_e ->
        E.return begin
          Caux.mk_sseq_e (Caux.mk_tuple_pat [loaded_wrp.E.sym_pat;
          ↪  desired_wrp.E.sym_pat])
            begin
              Caux.mk_unseq_e [lobject_core_e; desired_core_e]
            end
            begin
              Caux.mk_case_e loaded_wrp.E.sym_pe
                [ ( Caux.mk_specified_pat object_wrp.E.sym_pat
                  , Caux.pstore loc (Caux.mk_ail_ctype_pe ref_ty) object_wrp.E.sym_pe
                  ↪  desired_wrp.E.sym_pe mo )
                ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
                  , Caux.mk_pure_e (Caux.mk_undef_pe loc
                  ↪  Undefined.UB_unspecified_lvalue) ) ]
            end
        end
    | (AtomicLoadExplicit, [lobject_e; order_e]) ->
        let mo = translate_memory_order order_e in
```

```
1025          let ref_ty = match ctype_of lobject_e with
1026            | Ctype.Ctype _ (Ctype.Pointer _ ref_ty) ->
1027                ref_ty
1028            | (*BISECT-IGNORE*) _ ->
1029                illTypedAil loc "AilEcall atomic_load_explicit"
1030          end in
1031          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_wrp     ->
1032          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp     ->
1033          translate_expr lobject_e                            >>= fun lobject_core_e ->
1034          E.return begin
1035            Caux.mk_sseq_e loaded_wrp.E.sym_pat lobject_core_e
1036              begin
1037                Caux.mk_case_e loaded_wrp.E.sym_pe
1038                  [ ( Caux.mk_specified_pat object_wrp.E.sym_pat
1039                    , Caux.pload loc (Caux.mk_ail_ctype_pe ref_ty) object_wrp.E.sym_pe mo
                      ↪  )
1040                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1041                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                      ↪  Undefined.UB_unspecified_lvalue) ) ]
1042              end
1043          end
1044      | (AtomicThreadFence, [order_e]) ->
1045          (* TODO: allow non trivial call to atomic_thread_fence() ... *)
1046          let mo = translate_memory_order order_e in
1047          E.return begin
1048            C.Expr [] (C.Eaction (C.Paction C.Pos (C.Action loc () (C.Fence mo))))
1049          end
1050      | (AtomicCompareExchangeStrongExplicit, [object_e; expected_e; desired_e;
      ↪  order_success_e; order_failure_e]) ->
1051          let mo_success = translate_memory_order order_success_e in
1052          let mo_failure = translate_memory_order order_failure_e in
1053          let (ty1,ty2) = match (ctype_of object_e, ctype_of expected_e) with
1054            | (Ctype.Ctype _ (Ctype.Pointer _ ty1), Ctype.Ctype _ (Ctype.Pointer _ ty2))
              ↪  ->
1055                (ty1,ty2)
1056            | (*BISECT-IGNORE*) _ ->
1057                illTypedAil loc "AilEcall atomic_compare_exchange_strong_explicit"
1058          end in
1059          let oTy = force_core_object_type_of_ctype (ctype_of desired_e) in
1060          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_object_wrp
              ↪  ->
1061          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp
              ↪  ->
1062          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_expected_wrp
              ↪  ->
1063          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun expected_wrp
              ↪  ->
1064          E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun desired_wrp
              ↪  ->
1065          translate_expr object_e                             >>= fun core_object_e
              ↪  ->
1066          translate_expr expected_e                           >>= fun core_expected_e
              ↪  ->
1067          translate_expr desired_e                            >>= fun core_desired_e
              ↪  ->
1068          (* NOTE: we don't need to convert the arguments because the Ail typing has added
              ↪  casts *)
1069          E.return begin
```

```
1070          Caux.mk_sseq_e (Caux.mk_tuple_pat [ loaded_object_wrp.E.sym_pat;
              ↪  loaded_expected_wrp.E.sym_pat; desired_wrp.E.sym_pat ])
1071            (Caux.mk_unseq_e [ core_object_e; core_expected_e; core_desired_e ])
1072            begin
1073              Caux.mk_case_e (Caux.mk_tuple_pe [ loaded_object_wrp.E.sym_pe;
                ↪  loaded_expected_wrp.E.sym_pe ])
1074                [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat object_wrp.E.sym_pat
1075                                      ; Caux.mk_specified_pat expected_wrp.E.sym_pat ]
1076                  , Caux.pcompare_exchange_strong
1077                      loc (Caux.mk_ail_ctype_pe (Ctype.Ctype [] (Ctype.unatomic_ ty1)))
1078                      object_wrp.E.sym_pe expected_wrp.E.sym_pe desired_wrp.E.sym_pe
1079                      mo_success mo_failure )
1080                ; ( Caux.mk_empty_pat (C.BTy_tuple [ C.BTy_loaded C.OTy_pointer
1081                                                   ; C.BTy_loaded C.OTy_pointer ])
1082                  , Caux.mk_pure_e
1083                      (Caux.mk_undef_pe loc Undefined.UB_unspecified_lvalue) ) ]
1084            end
1085        end
1086    | (AtomicCompareExchangeWeakExplicit, [object_e; expected_e; desired_e;
      ↪  order_success_e; order_failure_e]) ->
1087      let mo_success = translate_memory_order order_success_e in
1088      let mo_failure = translate_memory_order order_failure_e in
1089      let (ty1,ty2) = match (ctype_of object_e, ctype_of expected_e) with
1090        | (Ctype.Ctype _ (Ctype.Pointer _ ty1), Ctype.Ctype _ (Ctype.Pointer _ ty2)) ->
1091          (ty1,ty2)
1092        | (*BISECT-IGNORE*) _ ->
1093          illTypedAil loc "AilEcall atomic_compare_exchange_weak_explicit"
1094      end in
1095      let oTy = force_core_object_type_of_ctype (ctype_of desired_e) in
1096      E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_object_wrp   ->
1097      E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp          ->
1098      E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_expected_wrp ->
1099      E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun expected_wrp        ->
1100      E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun desired_wrp         ->
1101      translate_expr object_e                             >>= fun core_object_e       ->
1102      translate_expr expected_e                           >>= fun core_expected_e     ->
1103      translate_expr desired_e                            >>= fun core_desired_e      ->
1104      (* NOTE: we don't need to convert the arguments because the Ail typing has added
          ↪  casts *)
1105      E.return begin
1106        Caux.mk_sseq_e (Caux.mk_tuple_pat [ loaded_object_wrp.E.sym_pat;
          ↪  loaded_expected_wrp.E.sym_pat; desired_wrp.E.sym_pat ])
1107          (Caux.mk_unseq_e [ core_object_e; core_expected_e; core_desired_e ])
1108          begin
1109            Caux.mk_case_e (Caux.mk_tuple_pe [ loaded_object_wrp.E.sym_pe;
              ↪  loaded_expected_wrp.E.sym_pe ])
1110              [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat object_wrp.E.sym_pat
1111                                    ; Caux.mk_specified_pat expected_wrp.E.sym_pat ]
1112                , Caux.pcompare_exchange_weak
1113                    loc (Caux.mk_ail_ctype_pe (Ctype.Ctype [] (Ctype.unatomic_ ty1)))
1114                    object_wrp.E.sym_pe expected_wrp.E.sym_pe desired_wrp.E.sym_pe
1115                    mo_success mo_failure )
1116              ; ( Caux.mk_empty_pat (C.BTy_tuple [ C.BTy_loaded C.OTy_pointer
1117                                                 ; C.BTy_loaded C.OTy_pointer ])
1118                , Caux.mk_pure_e (Caux.mk_undef_pe loc Undefined.UB_unspecified_lvalue)
                  ↪  ) ]
1119            end
1120      end
```

```
1121      | (LinuxStore, [lobject_e; desired_e; order_e]) ->
1122          let mo = translate_linux_memory_order order_e in
1123          let ref_ty = match ctype_of lobject_e with
1124            | Ctype.Ctype _ (Ctype.Pointer _ ref_ty) ->
1125                ref_ty
1126            | (*BISECT-IGNORE*) _ ->
1127                illTypedAil loc "AilEcall linux_write"
1128          end in
1129          let oTy = force_core_object_type_of_ctype (ctype_of desired_e) in
1130          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_wrp     ->
1131          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp     ->
1132          E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun desired_wrp    ->
1133          translate_expr lobject_e                            >>= fun lobject_core_e ->
1134          translate_expr desired_e                            >>= fun desired_core_e ->
1135          E.return begin
1136            Caux.mk_sseq_e (Caux.mk_tuple_pat [loaded_wrp.E.sym_pat;
              ↪  desired_wrp.E.sym_pat])
1137              begin
1138                Caux.mk_unseq_e [lobject_core_e; desired_core_e]
1139              end
1140              begin
1141                Caux.mk_case_e loaded_wrp.E.sym_pe
1142                  [ ( Caux.mk_specified_pat object_wrp.E.sym_pat
1143                    , Caux.plinux_store loc (Caux.mk_ail_ctype_pe ref_ty)
                      ↪  object_wrp.E.sym_pe desired_wrp.E.sym_pe mo )
1144                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1145                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                      ↪  Undefined.UB_unspecified_lvalue) ) ]
1146              end
1147          end
1148      | (LinuxLoad, [lobject_e; order_e]) ->
1149          let mo = translate_linux_memory_order order_e in
1150          let ref_ty = match ctype_of lobject_e with
1151            | Ctype.Ctype _ (Ctype.Pointer _ ref_ty) ->
1152                ref_ty
1153            | (*BISECT-IGNORE*) _ ->
1154                illTypedAil loc "AilEcall linux_read"
1155          end in
1156          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_wrp     ->
1157          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp     ->
1158          translate_expr lobject_e                            >>= fun lobject_core_e ->
1159          E.return begin
1160            Caux.mk_sseq_e loaded_wrp.E.sym_pat lobject_core_e
1161              begin
1162                Caux.mk_case_e loaded_wrp.E.sym_pe
1163                  [ ( Caux.mk_specified_pat object_wrp.E.sym_pat
1164                    , Caux.plinux_load loc (Caux.mk_ail_ctype_pe ref_ty)
                      ↪  object_wrp.E.sym_pe mo )
1165                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1166                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                      ↪  Undefined.UB_unspecified_lvalue) ) ]
1167              end
1168          end
1169      | (LinuxFence, [order_e]) ->
1170          let mo = translate_linux_memory_order order_e in
1171          E.return begin
1172            C.Expr [] (C.Eaction (C.Paction C.Pos (C.Action loc () (C.LinuxFence mo))))
1173          end
```

```
1174      | (LinuxRMW, [lobject_e; desired_e; order_e]) ->
1175          let mo = translate_linux_memory_order order_e in
1176          let ref_ty = match ctype_of lobject_e with
1177            | Ctype.Ctype _ (Ctype.Pointer _ ref_ty) ->
1178                ref_ty
1179            | (*BISECT-IGNORE*) _ ->
1180                illTypedAil loc "AilEcall linux_rmw"
1181          end in
1182          let oTy = force_core_object_type_of_ctype (ctype_of desired_e) in
1183          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun loaded_wrp     ->
1184          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun object_wrp     ->
1185          E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun desired_wrp    ->
1186          translate_expr lobject_e                            >>= fun lobject_core_e ->
1187          translate_expr desired_e                            >>= fun desired_core_e ->
1188          E.return begin
1189            Caux.mk_sseq_e (Caux.mk_tuple_pat [loaded_wrp.E.sym_pat;
              ↪  desired_wrp.E.sym_pat])
1190              (Caux.mk_unseq_e [lobject_core_e; desired_core_e])
1191              begin
1192                Caux.mk_case_e loaded_wrp.E.sym_pe
1193                  [ ( Caux.mk_specified_pat object_wrp.E.sym_pat
1194                    , Caux.plinux_rmw loc (Caux.mk_ail_ctype_pe ref_ty)
                      ↪  object_wrp.E.sym_pe desired_wrp.E.sym_pe mo )
1195                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1196                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                      ↪  Undefined.UB_unspecified_lvalue) ) ]
1197              end
1198          end
1199      | (*BISECT-IGNORE*) _ ->
1200          error "Translation.translate_explicit_atomic"
1201    end
```

B.12 Top-level function elaborating expressions

```
1204  type expr_ctx =
1205    | ECTX_glob of Symbol.sym * Symbol.sym
1206    | ECTX_logical_operator (* when elborating the desugaring of && or || *)
1207    | ECTX_other
1208
1209  val translate_expression:
1210    bool -> (* whether the value of the expression is used (i.e. the expression is
      ↪  directly applied to AilSexpr) *)
1211    expr_ctx ->
1212    (maybe Symbol.sym * maybe Symbol.sym) ->
1213    translation_stdlib ->
1214    Core.core_tag_definitions ->
1215    A.expression GenTypes.genTypeCategory ->
1216    E.elabM (C.expr unit)
1217  let rec translate_expression is_used ctx variadic_env stdlib tagDefs a_expr =
1218    let self = translate_expression true ctx variadic_env stdlib tagDefs in
1219    let is_lvalue = match GenTypes.genTypeCategoryOf a_expr with
1220      | GenTypes.GenLValueType _ _ _ ->
1221          true
1222      | GenTypes.GenRValueType _ ->
1223          false
1224    end in
1225    let integer_promotion (ty: Ctype.ctype) (e: C.pexpr) : C.pexpr =
```

```
1226      let promoted_ty = fromJust "Translation_aux.integer_promotion"
          ↪  (AilTypesAux.promotion integerImpl ty) in
1227      stdlib.mkcall_conv_int promoted_ty e in
1228    (* STD §6.3.1.8 *)
1229    let usual_arithmetic_conversion (ty1: Ctype.ctype) (ty2: Ctype.ctype) (e1: C.pexpr)
        ↪  (e2: C.pexpr) : C.pexpr * C.pexpr =
1230      match (AilTypesAux.corresponding_real_type ty1, AilTypesAux.corresponding_real_type
          ↪  ty2) with
1231        (* TODO/NOTE: we convert (long double, double and float) to Ocaml float! This is
           ↪  not the C11 behaviour! *)
1232        | (Just _, Just _) ->
1233            (e1, e2)
1234        | (Just _, _) ->
1235            (e1, C.Pexpr [] () (C.PEctor C.Cfvfromint [e2]))
1236        | (_, Just _) ->
1237            (C.Pexpr [] () (C.PEctor C.Cfvfromint [e1]), e2)
1238        | (Nothing, Nothing) ->
1239            (* STD §6.3.1.8#1, bullet 4 *)
1240            match (AilTypesAux.promotion integerImpl ty1, AilTypesAux.promotion
              ↪  integerImpl ty2) with
1241              | (Just (Ctype.Ctype _ (Ctype.Basic (Ctype.Integer ity1')) as ty1'), Just
                  ↪  (Ctype.Ctype _ (Ctype.Basic (Ctype.Integer ity2')) as ty2')) ->
1242                  (* "If both operants have the same type, then no further conversion is
                     ↪  needed." *)
1243                  if ty1' = ty2' then
1244                    (stdlib.mkcall_conv_int ty1' e1, stdlib.mkcall_conv_int ty2' e2)
1245                  (* "Otherwise, if both operands have signed integer types or both have
                     ↪  unsigned integer types,
1246                      the operand with the type of lesser integer conversion rank is
    ↪  converted to the type
1247                      of the operand with greater rank." *)
1248                  else if   (AilTypesAux.is_signed_integer_type ty1' &&
                     ↪  AilTypesAux.is_signed_integer_type ty2')
1249                         || (AilTypesAux.is_unsigned_integer_type ty1' &&
                             ↪  AilTypesAux.is_unsigned_integer_type ty2') then
1250                    if AilTypesAux.lt_integer_rank ity1' ity2' then
1251                      (stdlib.mkcall_conv_int ty2' e1, stdlib.mkcall_conv_int ty2' e2)
1252                    else
1253                      (stdlib.mkcall_conv_int ty1' e1, stdlib.mkcall_conv_int ty1' e2)
1254                  (* "Otherwise, if the operand that has unsigned type has rank greater or
                     ↪  equal to the rank of the
1255                      rank of the type of the other operand, then the operand with signed
    ↪  integer type is converted
1256                      to the type of the operand with unsigned integer type." *)
1257                  else if AilTypesAux.is_unsigned_integer_type ty1' &&
                     ↪  AilTypesAux.ge_integer_rank ity1' ity2' then
1258                    (stdlib.mkcall_conv_int ty1' e1, stdlib.mkcall_conv_int ty1' e2)
1259                  else if AilTypesAux.is_unsigned_integer_type ty2' &&
                     ↪  AilTypesAux.ge_integer_rank ity2' ity1' then
1260                    (stdlib.mkcall_conv_int ty2' e1, stdlib.mkcall_conv_int ty2' e2)
1261                  (* "Otherwise, if the type of the operand with signed integer type can
                     ↪  represent all of the values
1262                      of the type of the operand with unsigned integer type, then the
    ↪  operand with unsigned integer
1263                      type is converted to the type of the operand with signed integer
    ↪  type." *)
1264                  else if AilTypesAux.is_signed_integer_type ty1' then
```

```
1265                  (Caux.mk_if_pe_ [Annot.Anot_explode]
                      ↪  (stdlib.mkcall_all_values_representable_in ty2' ty1')
1266                     (stdlib.mkcall_conv_int ty1' e1)
1267                     (* "Otherwise, both operands are converted to the unsigned integer
                         ↪  type corresponding to the type
1268                         of the operand with signed integer type". *)
1269                     (stdlib.mkcall_conv_int (Ctype.Ctype [] (Ctype.Basic (Ctype.Integer
                         ↪  (AilTypesAux.make_corresponding_unsigned ity1')))) e1)
1270                  ,
1271                  Caux.mk_if_pe_ [Annot.Anot_explode]
                      ↪  (stdlib.mkcall_all_values_representable_in ty2' ty1')
1272                     (stdlib.mkcall_conv_int ty1' e2)
1273                     (stdlib.mkcall_conv_int (Ctype.Ctype [] (Ctype.Basic (Ctype.Integer
                         ↪  (AilTypesAux.make_corresponding_unsigned ity1')))) e2)
1274                  )
1275                else
1276                  (Caux.mk_if_pe_ [Annot.Anot_explode]
                      ↪  (stdlib.mkcall_all_values_representable_in ty1' ty2')
1277                     (stdlib.mkcall_conv_int ty2' e1)
1278                     (stdlib.mkcall_conv_int (Ctype.Ctype [] (Ctype.Basic (Ctype.Integer
                         ↪  (AilTypesAux.make_corresponding_unsigned ity2')))) e1)
1279                  ,
1280                  Caux.mk_if_pe_ [Annot.Anot_explode]
                      ↪  (stdlib.mkcall_all_values_representable_in ty1' ty2')
1281                     (stdlib.mkcall_conv_int ty2' e2)
1282                     (stdlib.mkcall_conv_int (Ctype.Ctype [] (Ctype.Basic (Ctype.Integer
                         ↪  (AilTypesAux.make_corresponding_unsigned ity2')))) e2)
1283                  )
1284            | _ ->
1285                error "Translation.usual_arithmetic_conversion: not (integer vs
                     ↪  integer)"
1286          end
1287      end in
1288  let result_ty = ctype_of a_expr in
1289  if AilTypesAux.is_pointer result_ty && Aaux.is_null_pointer_constant a_expr then
1290    (* NOTE: this is a bit tasteless as it makes the case AilEconst, ConstantNull
       ↪  unreachable *)
1291    E.return (Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe result_ty)))
1292  else
1293    let A.AnnotatedExpression annot std_annots loc expr = a_expr in
1294    Caux.add_loc loc <$> match expr with
```

B.12.1 Elaboration of unary arithmetic operators

```
1295      | A.AilEunary A.Plus e ->
1296          (* STD §6.5.3.3#2 *)
1297          let (oTy, mk_conversion) =
1298            if AilTypesAux.is_integer result_ty then
1299              (C.OTy_integer, integer_promotion (ctype_of e))
1300            else
1301              (C.OTy_floating, fun z -> z) in
1302          E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp ->
1303          E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp   ->
1304          self e                                    >>= fun core_e  ->
1305          E.return begin
1306            Caux.add_std "§6.5.3.3#2" begin
1307              Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
1308                Caux.mk_pure_e begin
```

```
1309                    Caux.mk_case_pe e_wrp.E.sym_pe
1310                      [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1311                        , Caux.mk_specified_pe (mk_conversion obj_wrp.E.sym_pe) )
1312                      ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1313                        , Caux.mk_unspecified_pe result_ty ) ]
1314                  end
1315                end
1316              end
1317            end
1318
1319        | A.AilEunary A.Minus e ->
1320            (* STD §6.5.3.3#3 *)
1321            let (oTy, zero_pe, mk_conversion) =
1322              if AilTypesAux.is_integer result_ty then
1323                (C.OTy_integer, Caux.mk_integer_pe 0, integer_promotion (ctype_of e))
1324              else
1325                (C.OTy_floating, Caux.mk_floating_value_pe Mem.zero_fval, fun z -> z) in
1326            E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp ->
1327            E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp   ->
1328            self e                                    >>= fun core_e  ->
1329            E.return begin
1330              Caux.add_std "§6.5.3.3#3" begin
1331                Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
1332                  Caux.mk_pure_e begin
1333                    Caux.mk_case_pe e_wrp.E.sym_pe
1334                      [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1335                        , let expr =
1336                            Caux.mk_op_pe C.OpSub zero_pe (mk_conversion obj_wrp.E.sym_pe)
                              ↪  in
1337                          Caux.mk_specified_pe (
1338 if AilTypesAux.is_signed_integer_type result_ty then
1339                            stdlib.mkcall_catch_exceptional_condition result_ty expr
1340 else if AilTypesAux.is_integer result_ty then
1341                             stdlib.mkcall_wrapI result_ty expr
1342 else
1343                             expr) )
1344                      ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1345                        , Caux.mk_unspecified_pe result_ty ) ]
1346                  end
1347                end
1348              end
1349            end
1350
1351        | A.AilEunary A.Bnot e ->
1352            (* STD §6.5.3.3#4 *)
1353            let oTy = force_core_object_type_of_ctype (ctype_of e) in
1354            E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp   ->
1355            E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp ->
1356            self e                                    >>= fun core_e  ->
1357            E.return begin
1358              Caux.add_std "§6.5.3.3#4" begin
1359                Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
1360                  Caux.mk_pure_e begin
1361                    Caux.mk_case_pe e_wrp.E.sym_pe
1362                      [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1363                        , let promoted_e = Caux.mk_std_pe "§6.5.3.3#4, sentence 2"
                              ↪  (integer_promotion (ctype_of e) obj_wrp.E.sym_pe) in
1364                          (* NOTE: result_ty == promoted type of e *)
```

```
1365                        Caux.mk_specified_pe begin if
                            ↪  AilTypesAux.is_unsigned_integer_type result_ty then
1366                          (* STD §6.5.3.3#4, sentence 3 *)
1367                          Caux.mk_std_pe "§6.5.3.3#4, sentence 3" (Caux.mk_op_pe C.OpSub
                              ↪  (Caux.mk_ivmax_pe (Caux.mk_ail_ctype_pe result_ty))
                              ↪  promoted_e)
1368                        else
1369                          Caux.bitwise_complement_pe (Caux.mk_ail_ctype_pe result_ty)
                              ↪  promoted_e
1370                        end )
1371                    ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1372                      , Caux.mk_unspecified_pe result_ty ) ]
1373              end
1374            end
1375          end
1376        end
```

B.12.2 Elaboration of the address operator

```
1378      | A.AilEunary A.Address (A.AnnotatedExpression _ _ _ (A.AilEunary A.Indirection
          ↪  e)) ->
1379          (* STD §6.5.3.2#3, sentence 3 *)
1380          (* NOTE: footnote 102 makes it clear that this is valid even if 'e' evaluates
              ↪  to a null pointer *)
1381          Caux.add_std "§6.5.3.2#3, sentence 3" <$> self e
1382
1383      | A.AilEunary A.Address e ->
1384          (* STD §6.5.3.2#3, sentence 5 *)
1385          if AilTypesAux.is_object (ctype_of e) then
1386            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun ptr_wrp ->
1387            self e                                              >>= fun core_e  ->
1388            E.return begin
1389              Caux.add_std "§6.5.3.2#3, sentence 5" begin
1390                Caux.mk_wseq_e ptr_wrp.E.sym_pat core_e begin
1391                  Caux.mk_pure_e (Caux.mk_specified_pe ptr_wrp.E.sym_pe)
1392                end
1393              end
1394            end
1395          else
1396            translate_function_designator self stdlib e
```

B.12.3 Elaboration of postfix operators

See the auxiliary function in Section B.8.

```
1398      | A.AilEunary A.PostfixIncr e ->
1399          translate_postfix loc self stdlib result_ty A.PostfixIncr e
1400      | A.AilEunary A.PostfixDecr e ->
1401          translate_postfix loc self stdlib result_ty A.PostfixDecr e
```

B.12.4 Elaboration of the indirection operator

```
1403      | A.AilEunary A.Indirection e ->
1404  if AilTypesAux.is_pointer_to_function (ctype_of e) then
1405          (* STD 6.5.3.2#4 *)
1406          translate_function_designator self stdlib e
1407  else match AilTypesAux.referenced_type (ctype_of e) with
```

```
1408    | (*BISECT-IGNORE*) Nothing ->
1409        illTypedAil loc "AilEunary Indirection, not a pointer type"
1410    | Just ref_ty ->
1411          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun e_wrp    ->
1412          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj_wrp  ->
1413          E.wrapped_fresh_symbol C.BTy_boolean                >>= fun test_wrp ->
1414          self e                                              >>= fun core_e   ->
1415          E.return begin
1416            Caux.add_std "§6.5.3.2" begin
1417              Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
1418                Caux.mk_case_e e_wrp.E.sym_pe
1419                  [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1420                    , Caux.mk_wseq_e test_wrp.E.sym_pat
1421                        (Caux.mk_memop_e Mem_common.PtrValidForDeref
                            ↪  [Caux.mk_ail_ctype_pe ref_ty; obj_wrp.E.sym_pe]) begin
1422                        Caux.mk_pure_e begin
1423                          Caux.mk_if_pe_ [Annot.Anot_explode] test_wrp.E.sym_pe
1424                            obj_wrp.E.sym_pe
1425                            (Caux.mk_std_undef_pe loc "§6.5.3.3#4, sentence 4"
                                ↪  Undefined.UB043_indirection_invalid_value)
1426                        end
1427                      end )
1428                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1429                    , Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.3.3#4, sentence 4"
                        ↪  Undefined.UB043_indirection_invalid_value) ) ]
1430              end
1431            end
1432          end
1433  end
```

B.12.5 Elaboration of bitwise shift operators

```
1435      | A.AilEbinary e1 (A.Arithmetic A.Shl) e2 ->
1436          (* STD §6.5.7 *)
1437          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e1_wrp        ->
1438          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e2_wrp        ->
1439          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj1_wrp      ->
1440          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj2_wrp      ->
1441          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun promoted1_wrp ->
1442          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun promoted2_wrp ->
1443          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun res_wrp       ->
1444          self e1                                             >>= fun core_e1       ->
1445          self e2                                             >>= fun core_e2       ->
1446          E.return begin
1447            Caux.add_std "§6.5.7" begin
1448              Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
                  ↪  (Caux.mk_unseq [core_e1; core_e2]) begin
1449                Caux.mk_pure_e begin
1450                  Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
1451                    [ ( Caux.mk_tuple_pat [ Caux.mk_empty_pat (C.BTy_loaded
                        ↪  C.OTy_integer)
1452                                          ; Caux.mk_unspecified_pat (Caux.mk_empty_pat
                                              ↪  C.BTy_ctype) ]
1453                      , Caux.mk_undef_exceptional_condition loc )
1454                    ; ( Caux.mk_tuple_pat [ Caux.mk_unspecified_pat (Caux.mk_empty_pat
                        ↪  C.BTy_ctype)
1455                                          ; Caux.mk_empty_pat (C.BTy_loaded
                                              ↪  C.OTy_integer) ]
```

```
1456                         ,
1457 begin if AilTypesAux.is_unsigned_integer_type (ctype_of e1) then
1458                           Caux.mk_unspecified_pe result_ty
1459 else
1460                           Caux.mk_undef_exceptional_condition loc
1461 end
1462                         )
1463                       ; ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat;
                          ↪  Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
1464                         , Caux.mk_let_pe promoted1_wrp.E.sym_pat
1465                             (Caux.mk_std_pe "§6.5.7#3, sentence 1" (integer_promotion
                                ↪  (ctype_of e1) obj1_wrp.E.sym_pe))
1466                           (Caux.mk_let_pe promoted2_wrp.E.sym_pat
1467                             (Caux.mk_std_pe "§6.5.7#3, sentence 1" (integer_promotion
                                ↪  (ctype_of e2) obj2_wrp.E.sym_pe))
1468                           (* (§6.5.7#2) if promoted2 < 0 then undef *)
1469                           (Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpLt
                              ↪  promoted2_wrp.E.sym_pe (Caux.mk_integer_pe 0))
1470                             (Caux.mk_std_undef_pe loc "§6.5.7#3, sentence 3"
                                ↪  Undefined.UB051a_negative_shift)
1471                           (* ctype_width(result_ty) <= promoted2 *)
1472                           (Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpOr
                              ↪  (Caux.mk_op_pe C.OpLt (stdlib.mkcall_ctype_width result_ty)
                              ↪  promoted2_wrp.E.sym_pe)
1473                                                          (Caux.mk_op_pe C.OpEq
                                                              ↪  (stdlib.mkcall_ctype_width
                                                              ↪  result_ty)
                                                              ↪  promoted2_wrp.E.sym_pe))
1474                             (Caux.mk_std_undef_pe loc "§6.5.7#4, sentence 3"
                                ↪  Undefined.UB51b_shift_too_large)
1475 begin if AilTypesAux.is_unsigned_integer_type (ctype_of e1) then
1476                           (Caux.mk_specified_pe (Caux.mk_std_pe "§6.5.7#4, sentence 2" (
1477                             Caux.mk_op_pe C.OpRem_t (Caux.mk_op_pe C.OpMul
                                ↪  promoted1_wrp.E.sym_pe (Caux.mk_op_pe C.OpExp
                                ↪  (Caux.mk_integer_pe 2) promoted2_wrp.E.sym_pe))
1478                                                    (Caux.mk_op_pe C.OpAdd
                                                        ↪  (Caux.mk_ivmax_pe
                                                        ↪  (Caux.mk_ail_ctype_pe result_ty))
                                                        ↪  (Caux.mk_integer_pe 1))
1479                           )))
1480 else
1481                           Caux.mk_std_pe "§6.5.7#4, sentence 3"
1482                             (Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpLt
                                ↪  promoted1_wrp.E.sym_pe (Caux.mk_integer_pe 0))
1483                               (Caux.mk_std_undef_pe loc "§6.5.7#3, sentence 3"
                                  ↪  Undefined.UB052a_negative_left_shift)
1484                             (Caux.mk_let_pe res_wrp.E.sym_pat
1485                               (Caux.mk_op_pe C.OpMul promoted1_wrp.E.sym_pe (Caux.mk_op_pe
                                  ↪  C.OpExp (Caux.mk_integer_pe 2) promoted2_wrp.E.sym_pe))
1486                             (Caux.mk_if_pe_ [Annot.Anot_explode]
                                ↪  (stdlib.mkcall_is_representable res_wrp.E.sym_pe
                                ↪  result_ty)
1487                               (Caux.mk_specified_pe res_wrp.E.sym_pe)
1488                             (Caux.mk_std_undef_pe loc "§6.5.7#3, sentence 3"
                                ↪  Undefined.UB052b_non_representable_left_shift))))
1489 end
1490                            )))) ]
1491                   end
```

```
1492                end
1493              end
1494            end
1495
1496        | A.AilEbinary e1 (A.Arithmetic A.Shr) e2 ->
1497            (* STD §6.5.7 *)
1498            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e1_wrp        ->
1499            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e2_wrp        ->
1500            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj1_wrp      ->
1501            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj2_wrp      ->
1502            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun promoted1_wrp ->
1503            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun promoted2_wrp ->
1504            self e1                                             >>= fun core_e1       ->
1505            self e2                                             >>= fun core_e2       ->
1506            E.return begin
1507              Caux.add_std "§6.5.7" begin
1508                Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
                  ↪  (Caux.mk_unseq [core_e1; core_e2]) begin
1509                  Caux.mk_pure_e begin
1510                    Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
1511                      [ ( Caux.mk_tuple_pat [ Caux.mk_empty_pat (C.BTy_loaded
                          ↪  C.OTy_integer)
1512                                            ; Caux.mk_unspecified_pat (Caux.mk_empty_pat
                                                ↪  C.BTy_ctype) ]
1513                        , Caux.mk_undef_exceptional_condition loc )
1514                      ; ( Caux.mk_tuple_pat [ Caux.mk_unspecified_pat (Caux.mk_empty_pat
                          ↪  C.BTy_ctype)
1515                                            ; Caux.mk_empty_pat (C.BTy_loaded
                                                ↪  C.OTy_integer) ]
1516                        , Caux.mk_unspecified_pe (result_ty) )
1517                      ; ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
1518                                            ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
1519                        , Caux.mk_let_pe promoted1_wrp.E.sym_pat (integer_promotion
                          ↪  (ctype_of e1) obj1_wrp.E.sym_pe)
1520                          (Caux.mk_let_pe promoted2_wrp.E.sym_pat (integer_promotion
                              ↪  (ctype_of e2) obj2_wrp.E.sym_pe)
1521                              (* (§6.5.7#2) if promoted2 < 0 then undef *)
1522                              (Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpLt
                                  ↪  promoted2_wrp.E.sym_pe (Caux.mk_integer_pe 0))
1523                                (Caux.mk_std_undef_pe loc "§6.5.7#3, sentence 3"
                                  ↪  Undefined.UB051a_negative_shift)
1524                              (* ctype_width(result_ty) <= promoted2 *)
1525                              (Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpOr
                                  ↪  (Caux.mk_op_pe C.OpLt (stdlib.mkcall_ctype_width result_ty)
                                  ↪  promoted2_wrp.E.sym_pe)
1526                                                                  (Caux.mk_op_pe C.OpEq
                                                                      ↪  (stdlib.mkcall_ctype_width
                                                                      ↪  result_ty)
                                                                      ↪  promoted2_wrp.E.sym_pe))
1527                                (Caux.mk_std_undef_pe loc "§6.5.7#3, sentence 3"
                                    ↪  Undefined.UB51b_shift_too_large)
1528                                begin
1529                                  let expr = Caux.mk_op_pe C.OpDiv obj1_wrp.E.sym_pe
                                      ↪  (Caux.mk_op_pe C.OpExp (Caux.mk_integer_pe 2)
                                      ↪  promoted2_wrp.E.sym_pe) in
1530                                  Caux.mk_specified_pe
1531  begin if AilTypesAux.is_unsigned_integer_type (ctype_of e1) then
1532                                    Caux.mk_std_pe "6.5.7#5, sentence 2" expr
```

```
1533 else
1534                                 Caux.mk_std_pe "6.5.7#5, sentence 3" begin
1535                                  Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe
                                      ↪  C.OpGe promoted1_wrp.E.sym_pe (Caux.mk_integer_pe
                                      ↪  0))
1536                                    expr
1537                                    (Caux.mk_call_pe (C.Impl
                                        ↪  Implementation.SHR_signed_negative)
1538                                      [Caux.mk_ail_ctype_pe (ctype_of e1) ;
                                          ↪  promoted1_wrp.E.sym_pe; promoted2_wrp.E.sym_pe])
1539                                 end
1540 end
1541                             end))) ) ]
1542                 end
1543               end
1544             end
1545           end
```

B.12.6 Elaboration of identifiers

Note that in Ail, identifiers are always lvalues because the language makes explicit rvalue coercions and the decay of arrays. See Section B.12.32 for the elaboration of these operations.

```
1547       | A.AilEident id ->
1548           let id_sym_pe =
1549             match ctx with
1550               | ECTX_glob glob_sym sym' ->
1551                   if id = glob_sym then
1552                     Caux.mk_sym_pe sym'
1553                   else
1554                     Caux.mk_sym_pe id
1555               | _ (* ECTX_other *) ->
1556                   Caux.mk_sym_pe id
1557             end in
1558           E.return (Caux.mk_pure_e id_sym_pe)
```

B.12.7 Elaboration of cast operators

```
1560       | A.AilEcast _ cast_ty e ->
1561           let e_ty = ctype_of e in
1562           let oTy = force_core_object_type_of_ctype e_ty in
1563           E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp   ->
1564           E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp ->
1565           (* STD §6.3.2.1#2 "lvalue conversion" *)
1566           self e    >>= fun core_e  ->
1567           let let_sym = Symbol.fresh () in
1568           E.return $
1569             Caux.add_std "§6.5.4" (
1570 if AilTypesAux.is_void cast_ty then
1571               Caux.mk_wseq_e e_wrp.E.sym_pat core_e
1572                 Caux.mk_skip_e
1573
1574 else if AilTypesAux.is_pointer cast_ty && Aaux.is_null_pointer_constant e then
1575   match AilTypesAux.referenced_type cast_ty with
1576     | (*BISECT-IGNORE*) Nothing ->
1577               illTypedAil loc "AilEcast, pointer vs null_pointer_constant"
```

```
1578      | Just ref_ty ->
1579                Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe ref_ty))
1580    end
1581
1582  else if AilTypesAux.is_arithmetic cast_ty && AilTypesAux.is_arithmetic e_ty then
1583    if AilTypesAux.is_integer cast_ty then
1584      if AilTypesAux.is_integer e_ty then
1585                Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
1586                  Caux.mk_pure_e (stdlib.mkcall_conv_loaded_int cast_ty e_wrp.E.sym_pe)
1587                )
1588      else (* cast_ty is floating since it is an arithmetic type *)
1589                Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
1590                  Caux.mk_pure_e (stdlib.mkcall_loaded_ivfromfloat cast_ty e_wrp.E.sym_pe)
1591                )
1592    else
1593      if AilTypesAux.is_integer e_ty then
1594                Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
1595                  Caux.mk_pure_e (stdlib.mkcall_loaded_fvfromint cast_ty e_wrp.E.sym_pe)
1596                )
1597      else (* cast_ty is floating since it is an arithmetic type *)
1598                floating_conversion_TODO cast_ty e_ty core_e
1599
1600  else if AilTypesAux.is_pointer cast_ty && AilTypesAux.is_arithmetic e_ty then
1601              (* making a pointer from an integer *)
1602                let ref_ty = fromJust "Translation.translate_expression, AilEcast 1"
                  ↪  (AilTypesAux.referenced_type cast_ty) in
1603                Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
1604                  Caux.mk_case_e e_wrp.E.sym_pe
1605                    [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1606                      , Caux.mk_wseq_e (Caux.mk_sym_pat let_sym (C.BTy_object
                        ↪  C.OTy_pointer))
1607                          (C.Expr [] (C.Ememop Mem_common.PtrFromInt [Caux.mk_ail_ctype_pe
                          ↪  e_ty; Caux.mk_ail_ctype_pe ref_ty; obj_wrp.E.sym_pe]))
1608                          (Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_sym_pe let_sym)))
                          ↪  )
1609                    ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1610                      , (* Casting an unspecified integer to a pointer type gives an
                        ↪  unspecified pointer *)
1611                        Caux.mk_pure_e (Caux.mk_unspecified_pe cast_ty) ) ]
1612                )
1613
1614  else if AilTypesAux.is_arithmetic cast_ty && AilTypesAux.is_pointer e_ty then
1615                (* making an integer from a pointer *)
1616                let ref_ty = fromJust "Translation.translate_expression, AilEcast 2"
                  ↪  (AilTypesAux.referenced_type e_ty) in
1617                Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
1618                    Caux.mk_case_e e_wrp.E.sym_pe
1619                      [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
1620                        , Caux.mk_wseq_e (Caux.mk_sym_pat let_sym (C.BTy_object
                          ↪  C.OTy_integer))
1621                            (C.Expr [] (C.Ememop Mem_common.IntFromPtr
                            ↪  [Caux.mk_ail_ctype_pe ref_ty; Caux.mk_ail_ctype_pe
                            ↪  cast_ty; obj_wrp.E.sym_pe]))
1622                            (Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_sym_pe
                            ↪  let_sym))) )
1623                      ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
1624                        , (* Casting an unspecified pointer to an integer type gives an
                          ↪  unspecified integer *)
```

```
                        Caux.mk_pure_e (Caux.mk_unspecified_pe cast_ty) ) ]
              )

else (* pointer <-> pointer cast *)
              let () = Debug.warn [Debug.DB_elaboration] (fun () ->
                "the elaboration does the identity for casts between pointer types (this
                ↪ is different from ISO)"
              ) in
              let ub_pe = Caux.mk_undef_pe loc
              ↪ Undefined.UB025_misaligned_pointer_conversion in
              match AilTypesAux.referenced_type cast_ty with
                | Just cast_ref_ty ->
  if AilTypesAux.is_void cast_ref_ty || AilTypesAux.is_function cast_ref_ty then
                    core_e
  else
                    Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
                      Caux.mk_case_e e_wrp.E.sym_pe
                        [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
                          , Caux.mk_wseq_e (Caux.mk_sym_pat let_sym C.BTy_boolean)
                              (C.Expr [] (C.Ememop Mem_common.PtrWellAligned
                              ↪ [Caux.mk_ail_ctype_pe cast_ref_ty; obj_wrp.E.sym_pe]))
                              (Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_if_pe_
                              ↪ [Annot.Anot_explode] (Caux.mk_sym_pe let_sym)
                              ↪ obj_wrp.E.sym_pe ub_pe))) )
                        ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
                          , (* we are being daemonic (case where the resulting pointer
                          ↪ would be misaligned) *)
                            Caux.mk_pure_e ub_pe ) ]
                    end
                | _ ->
                    error "Translation AilEcast, ptr vs ptr: just should be impossible"
              end
            )
```

B.12.8 Elaboration of multiplicative operators

See the auxiliary functions in Section B.4.

```
      | A.AilEbinary e1 (A.Arithmetic A.Mul) e2 ->
          translate_mul_operator loc self usual_arithmetic_conversion stdlib
            result_ty e1 e2
      | A.AilEbinary e1 (A.Arithmetic (A.Div as aop)) e2 ->
          translate_div_mod_operator loc self usual_arithmetic_conversion stdlib
            result_ty aop e1 e2
      | A.AilEbinary e1 (A.Arithmetic (A.Mod as aop)) e2 ->
          translate_div_mod_operator loc self usual_arithmetic_conversion stdlib
            result_ty aop e1 e2
```

B.12.9 Elaboration of the addition operator

```
      | A.AilEbinary e1 (A.Arithmetic A.Add) e2 ->
if AilTypesAux.is_arithmetic (ctype_of e1) && AilTypesAux.is_arithmetic (ctype_of e2)
↪ then
            let oTy1 = force_core_object_type_of_ctype (ctype_of e1) in
            let oTy2 = force_core_object_type_of_ctype (ctype_of e2) in
            E.wrapped_fresh_symbol (C.BTy_loaded oTy1) >>= fun e1_wrp   ->
            E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp   ->
```

```
1669              E.wrapped_fresh_symbol (C.BTy_object oTy1) >>= fun obj1_wrp ->
1670              E.wrapped_fresh_symbol (C.BTy_object oTy2) >>= fun obj2_wrp ->
1671              self e1                                    >>= fun core_e1  ->
1672              self e2                                    >>= fun core_e2  ->
1673              let (promoted1_pe, promoted2_pe) =
1674                Caux.mk_std_pair_pe "§6.5.6#4"
1675                  (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2)
                  ↪  obj1_wrp.E.sym_pe obj2_wrp.E.sym_pe) in
1676              E.return begin
1677                Caux.add_std "§6.5.6" begin
1678                  Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat
                  ↪  ]) (Caux.mk_unseq [core_e1; core_e2]) begin
1679                    Caux.mk_pure_e begin
1680                      Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe;
                        ↪  e2_wrp.E.sym_pe])
1681                        [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
1682                                              ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
1683                          , (* Both operand are specified *)
1684                            let core_add = Caux.mk_std_pe "§6.5.6#5" (Caux.mk_op_pe
                              ↪  C.OpAdd promoted1_pe promoted2_pe) in
1685                            Caux.mk_specified_pe (
1686    if AilTypesAux.is_signed_integer_type result_ty then
1687                              stdlib.mkcall_catch_exceptional_condition result_ty core_add
1688    else if AilTypesAux.is_integer result_ty then
1689                              stdlib.mkcall_wrapI result_ty core_add
1690    else
1691                              core_add) )
1692
1693                        ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1;
                          ↪  C.BTy_loaded oTy2])
1694                          , (* If either operand is unspecified, the result is also
                            ↪  unspecified is the
1695                               result type of unsigned. Otherwise it is undef, since the
    ↪  addition
1696                               may overflow *)
1697    if AilTypesAux.is_unsigned_integer_type result_ty then
1698                            Caux.mk_unspecified_pe (result_ty)
1699    else
1700                            Caux.mk_undef_exceptional_condition loc) ]
1701                    end
1702                  end
1703                end
1704              end
1705  else (* Here one of the operands is a pointer *)
1706              E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun ptr_wrp
                ↪  ->
1707              E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun ptr_obj_wrp
                ↪  ->
1708              E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun integer_wrp
                ↪  ->
1709              E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun integer_obj_wrp
                ↪  ->
1710              E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun res_wrp
                ↪  ->
1711              self e1                                             >>= fun core_e1
                ↪  ->
1712              self e2                                             >>= fun core_e2
                ↪  ->
```

```
1713              let (ptr_ty, ptr_core_e, integer_core_e) =
1714    if AilTypesAux.is_arithmetic (ctype_of e1) then
1715                (ctype_of e2, core_e2, core_e1)
1716    else
1717                (ctype_of e1, core_e1, core_e2) in
1718              let ref_ty =
1719                match AilTypesAux.referenced_type ptr_ty with
1720                  | (*BISECT-IGNORE*) Nothing ->
1721                      illTypedAil loc "A.AilEbinary (A.Arithmetic A.Add), one is pointer"
1722                  | Just ref_ty ->
1723                      ref_ty
1724                end in
1725              E.return begin
1726                Caux.add_std "§6.5.6" begin
1727                  Caux.mk_wseq_e (Caux.mk_tuple_pat [ ptr_wrp.E.sym_pat;
                       ↪  integer_wrp.E.sym_pat ]) (Caux.mk_unseq [ptr_core_e;
                       ↪  integer_core_e]) begin
1728                    Caux.mk_case_e (Caux.mk_tuple_pe [ptr_wrp.E.sym_pe;
                         ↪  integer_wrp.E.sym_pe])
1729                      [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat ptr_obj_wrp.E.sym_pat
1730                                            ; Caux.mk_specified_pat
                                              ↪  integer_obj_wrp.E.sym_pat ]
1731                        , (* Both operand are specified *)
1732    begin if Global.has_strict_pointer_arith () || Global.is_PNVI () then
1733                            Caux.mk_wseq_e res_wrp.E.sym_pat
1734                              (Caux.mk_memop_e Mem_common.PtrArrayShift
                                 ↪  [ptr_obj_wrp.E.sym_pe; Caux.mk_ail_ctype_pe ref_ty;
                                 ↪  integer_obj_wrp.E.sym_pe])
1735                              (Caux.mk_pure_e (Caux.mk_specified_pe res_wrp.E.sym_pe))
1736    else
1737                            Caux.mk_pure_e begin
1738                              Caux.mk_specified_pe begin
1739                                Caux.mk_std_pe "§6.5.6#8, sentences 2-3"
                                   ↪  (Caux.mk_array_shift ptr_obj_wrp.E.sym_pe ref_ty
                                   ↪  integer_obj_wrp.E.sym_pe)
1740                              end
1741                            end
1742    end                   )
1743                        ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded C.OTy_pointer;
                            ↪  C.BTy_loaded C.OTy_integer])
1744                          , Caux.mk_pure_e (Caux.mk_undef_pe loc
                              ↪  (Undefined.UB_CERB004_unspecified
                              ↪  Undefined.UB_unspec_pointer_add)) ) ]
1745                  end
1746                end
1747              end
```

B.12.10 Elaboration of the subtraction operator

```
1749        | A.AilEbinary e1 (A.Arithmetic A.Sub) e2 ->
1750            self e1 >>= fun core_e1 ->
1751            self e2 >>= fun core_e2 ->
1752    if AilTypesAux.is_arithmetic (ctype_of e1) && AilTypesAux.is_arithmetic (ctype_of e2)
        ↪  then
1753              let oTy1 = force_core_object_type_of_ctype (ctype_of e1) in
1754              let oTy2 = force_core_object_type_of_ctype (ctype_of e2) in
1755              E.wrapped_fresh_symbol (C.BTy_loaded oTy1) >>= fun e1_wrp   ->
1756              E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp   ->
```

```
1757            E.wrapped_fresh_symbol (C.BTy_object oTy1) >>= fun obj1_wrp ->
1758            E.wrapped_fresh_symbol (C.BTy_object oTy2) >>= fun obj2_wrp ->
1759            let (promoted1_pe, promoted2_pe) =
1760              Caux.mk_std_pair_pe "§6.5.6#4"
1761                (usual_arithmetic_conversion (ctype_of e1) (ctype_of e2) obj1_wrp.E.sym_pe
                  ↪  obj2_wrp.E.sym_pe) in
1762            E.return begin
1763              C.Expr [Annot.Astd "§6.5.6"] (
1764                C.Ewseq (Caux.mk_tuple_pat [e1_wrp.E.sym_pat; e2_wrp.E.sym_pat])
                  ↪  (Caux.mk_unseq [core_e1; core_e2]) (
1765                  Caux.mk_pure_e (
1766                    Caux.mk_case_pe (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
1767                      [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
1768                                            ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
1769                        , (* Both operand are specified *)
1770                          let core_sub = Caux.mk_std_pe "§6.5.6#6" (Caux.mk_op_pe C.OpSub
                            ↪  promoted1_pe promoted2_pe) in
1771                          Caux.mk_specified_pe $
1772  if AilTypesAux.is_signed_integer_type result_ty then
1773                            stdlib.mkcall_catch_exceptional_condition result_ty core_sub
1774  else if AilTypesAux.is_integer result_ty then
1775                            stdlib.mkcall_wrapI result_ty core_sub
1776  else
1777                            core_sub )
1778                      ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded oTy1; C.BTy_loaded
                        ↪  oTy2])
1779                        , (* If either operand is unspecified, the result is also
                          ↪  unspecified is the
1780                             result type of unsigned. Otherwise it is undef, since the
    ↪  addition
1781                             may overflow *)
1782  if AilTypesAux.is_signed_integer_type result_ty then
1783                            Caux.mk_undef_exceptional_condition loc
1784  else
1785                            Caux.mk_unspecified_pe (result_ty) ) ]
1786                  )
1787                )
1788              )
1789            end
1790 else if AilTypesAux.is_pointer (ctype_of e1) && AilTypesAux.is_pointer (ctype_of e2)
    ↪  then
1791            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun e1_wrp    ->
1792            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun e2_wrp    ->
1793            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj1_wrp  ->
1794            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj2_wrp  ->
1795            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun memop_wrp ->
1796            (* NOTE: by Ail typing we can just use the referenced type of either operand
                ↪  *)
1797            let diff_ty_pe = match (ctype_of e1) with
1798              | Ctype.Ctype _ (Ctype.Pointer _ ref_ty1) ->
1799                  Caux.mk_ail_ctype_pe ref_ty1
1800              | (*BISECT-IGNORE*) _ ->
1801                  illTypedAil loc "ptrdiff"
1802            end in
1803            E.return begin
1804              C.Expr [Annot.Astd "§6.5.6"] (
1805                C.Ewseq (Caux.mk_tuple_pat [e1_wrp.E.sym_pat; e2_wrp.E.sym_pat])
                  ↪  (Caux.mk_unseq [core_e1; core_e2]) (
```

```
1806                Caux.mk_case_e (Caux.mk_tuple_pe [e1_wrp.E.sym_pe; e2_wrp.E.sym_pe])
1807                  [ ( Caux.mk_tuple_pat [ Caux.mk_specified_pat obj1_wrp.E.sym_pat
1808                                        ; Caux.mk_specified_pat obj2_wrp.E.sym_pat ]
1809                    , (* Both operand are specified *)
1810                      Caux.mk_wseq_e memop_wrp.E.sym_pat
1811                        (C.Expr [] (C.Ememop Mem_common.Ptrdiff [diff_ty_pe;
                          ↪  obj1_wrp.E.sym_pe; obj2_wrp.E.sym_pe]))
1812                        begin
1813                          Caux.mk_pure_e begin
1814                            Caux.mk_if_pe_ [Annot.Anot_explode]
                              ↪  (stdlib.mkcall_is_representable memop_wrp.E.sym_pe
                              ↪  Ctype.ptrdiff_t)
1815                              (Caux.mk_specified_pe memop_wrp.E.sym_pe)
1816                              (Caux.mk_undef_pe loc
                                ↪  Undefined.UB050_pointers_subtraction_not_representable)
1817                          end
1818                        end )
1819                  ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded C.OTy_pointer;
                      ↪  C.BTy_loaded C.OTy_pointer])
1820                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                      ↪  Undefined.UB050_pointers_subtraction_not_representable) ) ]
1821              )
1822            )
1823          end
1824  else
1825            (* Here one of the operand is pointer *)
1826            let (ptr_ty, ptr_core_e, integer_core_e) =
1827              if AilTypesAux.is_arithmetic (ctype_of e1) then
1828                (ctype_of e2, core_e2, core_e1)
1829              else
1830                (ctype_of e1, core_e1, core_e2) in
1831            let ref_ty = match AilTypesAux.referenced_type ptr_ty with
1832              | (*BISECT-IGNORE*) Nothing ->
1833                  illTypedAil loc "A.AilEbinary (A.Arithmetic A.Sub), one is pointer"
1834              | Just ref_ty ->
1835                  ref_ty
1836            end in
1837            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun ptr_wrp
              ↪  ->
1838            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun integer_wrp
              ↪  ->
1839            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun ptr_obj_wrp
              ↪  ->
1840            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun integer_obj_wrp
              ↪  ->
1841            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun res_wrp
              ↪  ->
1842            E.return begin
1843              C.Expr [Annot.Astd "§6.5.6"] (
1844                C.Ewseq (Caux.mk_tuple_pat [ ptr_wrp.E.sym_pat; integer_wrp.E.sym_pat ])
1845                  (Caux.mk_unseq [ptr_core_e; integer_core_e]) (
1846                    Caux.mk_case_e (Caux.mk_tuple_pe [ptr_wrp.E.sym_pe;
                      ↪  integer_wrp.E.sym_pe])
1847                      [ (Caux.mk_tuple_pat [ Caux.mk_specified_pat
                        ↪  (ptr_obj_wrp.E.sym_pat)
1848                                           ; Caux.mk_specified_pat
                                               ↪  (integer_obj_wrp.E.sym_pat) ],
1849                       (* Both operand are specified *)
```

```
1850 begin if Global.has_strict_pointer_arith () || Global.is_PNVI () then
1851                         Caux.mk_wseq_e res_wrp.E.sym_pat
1852                           (C.Expr [] (C.Ememop Mem_common.PtrArrayShift
                               ↪  [ptr_obj_wrp.E.sym_pe; Caux.mk_ail_ctype_pe ref_ty;
                               ↪  (Caux.mk_neg_pe integer_obj_wrp.E.sym_pe)]))
1853                           (Caux.mk_pure_e (Caux.mk_specified_pe res_wrp.E.sym_pe))
1854 else
1855                         Caux.mk_pure_e begin
1856                           Caux.mk_specified_pe begin
1857                             Caux.mk_std_pe "§6.5.6#8, sentences 2-3"
                                 ↪  (Caux.mk_array_shift ptr_obj_wrp.E.sym_pe ref_ty
                                 ↪  (Caux.mk_neg_pe integer_obj_wrp.E.sym_pe))
1858                           end
1859                         end
1860 end                       )
1861                         ; ( Caux.mk_empty_pat (C.BTy_tuple [C.BTy_loaded C.OTy_pointer;
                             ↪  C.BTy_loaded C.OTy_integer])
1862                           , Caux.mk_pure_e (Caux.mk_undef_pe loc
                               ↪  (Undefined.UB_CERB004_unspecified
                               ↪  Undefined.UB_unspec_pointer_sub)) ) ]
1863                   )
1864                 )
1865               end
```

B.12.11 Elaboration of relational operators

See the auxiliary functions in Section B.5.

```
1867       | A.AilEbinary e1 (A.Lt as bop) e2 ->
1868           translate_relational_operator
1869             self usual_arithmetic_conversion
1870             result_ty bop e1 e2
1871       | A.AilEbinary e1 (A.Gt as bop) e2 ->
1872           translate_relational_operator
1873             self usual_arithmetic_conversion
1874             result_ty bop e1 e2
1875       | A.AilEbinary e1 (A.Le as bop) e2 ->
1876           translate_relational_operator
1877             self usual_arithmetic_conversion
1878             result_ty bop e1 e2
1879       | A.AilEbinary e1 (A.Ge as bop) e2 ->
1880           translate_relational_operator
1881             self usual_arithmetic_conversion
1882             result_ty bop e1 e2
```

B.12.12 Elaboration of equality operators

See the auxiliary functions in Section B.6.

```
1884       | A.AilEbinary e1 (A.Eq as bop) e2 ->
1885           translate_equality_operator loc
1886             self usual_arithmetic_conversion
1887             result_ty bop e1 e2
1888       | A.AilEbinary e1 (A.Ne as bop) e2 ->
1889           translate_equality_operator loc
1890             self usual_arithmetic_conversion
1891             result_ty bop e1 e2
```

B.12.13 Elaboration of bitwise operators

See the auxiliary functions in Section B.7.

```
1893        | A.AilEbinary e1 (A.Arithmetic (A.Band as aop)) e2 ->
1894            translate_bitwise_operator
1895              loc self usual_arithmetic_conversion stdlib
1896              result_ty aop e1 e2
1897        | A.AilEbinary e1 (A.Arithmetic (A.Bxor as aop)) e2 ->
1898            translate_bitwise_operator
1899              loc self usual_arithmetic_conversion stdlib
1900              result_ty aop e1 e2
1901        | A.AilEbinary e1 (A.Arithmetic (A.Bor as aop)) e2 ->
1902            translate_bitwise_operator
1903              loc self usual_arithmetic_conversion stdlib
1904              result_ty aop e1 e2
```

B.12.14 Elaboration of logical operators

```
1906        | A.AilEbinary e1 A.And e2 ->
1907            (* Desugaring e1 && e2 ===> (e1 == 0) ? 0 : (e2 != 0) *)
1908            Caux.add_stds ["6.5.13#3"; "6.5.13#4"] <$>
1909              translate_expression true ECTX_logical_operator variadic_env stdlib tagDefs
                  ↪ begin
1910                A.AnnotatedExpression
1911                  (GenTypes.GenRValueType GenTypes.signedInt_gty) [] loc
1912                  (A.AilEcond (mkTestExpression TestEq e1) zeroAil_tau (mkTestExpression
                      ↪ TestNe e2))
1913              end
1914
1915        | A.AilEbinary e1 A.Or e2 ->
1916            (* Desugaring e1 || e2 ===> (e1 == 0) ? (e2 != 0) : 0 *)
1917            Caux.add_stds ["6.5.14#3"; "6.5.14#4"] <$>
1918            translate_expression true ECTX_logical_operator variadic_env stdlib tagDefs
                ↪ begin
1919                A.AnnotatedExpression
1920                  (GenTypes.GenRValueType GenTypes.signedInt_gty) [] loc
1921                  (A.AilEcond (mkTestExpression TestEq e1) (mkTestExpression TestNe e2)
                      ↪ oneAil_tau)
1922              end
```

B.12.15 Elaboration of conditional operators

```
1924        | A.AilEcond e1 e2 e3 ->
1925            let apply_implicit_conversions expr =
1926              let e_ty = ctype_of expr in
1927              self expr >>= fun core_e ->
1928              if AilTypesAux.is_integer result_ty then
1929                let e_oTy = force_core_object_type_of_ctype e_ty in
1930                E.wrapped_fresh_symbol (C.BTy_loaded e_oTy) >>= fun e_wrp ->
1931                (* NOTE: if result_ty is an integer type, then e2 and e2 are both integers
                      ↪ (so e_ty must be too) *)
1932                E.return begin
1933                  Caux.mk_sseq_e e_wrp.E.sym_pat core_e
1934                    (Caux.mk_pure_e (stdlib.mkcall_conv_loaded_int result_ty
                        ↪ e_wrp.E.sym_pe))
1935                end
1936              else if AilTypesAux.is_floating result_ty then
```

```
1937              begin
1938                if AilTypesAux.is_integer e_ty then
1939                  let e_oTy = force_core_object_type_of_ctype e_ty in
1940                  E.wrapped_fresh_symbol (C.BTy_loaded e_oTy) >>= fun e_wrp ->
1941                  E.return begin
1942                    Caux.mk_sseq_e e_wrp.E.sym_pat core_e
1943                      (Caux.mk_pure_e (stdlib.mkcall_loaded_fvfromint result_ty
                          ↪  e_wrp.E.sym_pe))
1944                  end
1945                else
1946                  E.return (floating_conversion_TODO result_ty e_ty core_e)
1947              end
1948            else if AilTypesAux.is_pointer result_ty then
1949              begin
1950                if AilTypesAux.is_integer e_ty && AilSyntaxAux.is_null_pointer_constant
                    ↪  expr then
1951                  match AilTypesAux.referenced_type result_ty with
1952                    | Just ref_ty ->
1953                        E.return begin
1954                          Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe
                              ↪  ref_ty))
1955                        end
1956                    | _ ->
1957                        error "AilEcond: a pointer must have a referenced type"
1958                  end
1959                else if AilTypesAux.is_pointer e_ty then
1960                  E.return core_e
1961                else
1962                  error "AilEcond: invalid implicit conversion to a pointer type"
1963              end
1964            else
1965              (* NOTE: Ail's typing guarantees that e_ty = result_ty in this case *)
1966              E.return core_e
1967          in
1968          (* STD §6.5.15 *)
1969          (* NOTE: Ail's typing guarantees that e1 is scalar *)
1970          self (mkTestExpression TestEq e1) >>= fun core_e1 ->
1971          (* NOTE: the Core expression 'core_e1' has integer type because we elaborated
              ↪  an equality test *)
1972          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e1_wrp   ->
1973          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun obj1_wrp ->
1974          apply_implicit_conversions e2                       >>= fun conv_e2  ->
1975          apply_implicit_conversions e3                       >>= fun conv_e3  ->
1976          let (seq_quote, test_quote) =
1977            match ctx with
1978              | ECTX_logical_operator ->
1979                  (* the AilEcond was produced as a desugaring of &&, so don't put STD
                      ↪  quotes here *)
1980                  ([], [])
1981              | _ ->
1982                  (["§6.5.15#4, sentence 1"], ["§6.5.15#4, sentence 2"])
1983            end in
1984          E.return begin
1985            Caux.add_stds seq_quote begin
1986            (* STD (§6.5.15#4, sentence 1) says there is a sequenced point between the
1987               evaluation of e1 and the e2/e3. Hence the strong sequencing *)
1988              Caux.mk_sseq_e e1_wrp.E.sym_pat core_e1 begin
1989                Caux.mk_case_e e1_wrp.E.sym_pe
```

```
                  [ ( Caux.mk_specified_pat obj1_wrp.E.sym_pat
                    , Caux.add_stds test_quote begin
                        Caux.mk_if_e (Caux.mk_op_pe C.OpEq obj1_wrp.E.sym_pe
                        ↪  (Caux.mk_integer_pe 0))
                          conv_e2
                          conv_e3
                      end )
                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
                    , Caux.mk_pure_e (Caux.mk_undef_pe loc
                    ↪  (Undefined.UB_CERB004_unspecified
                    ↪  Undefined.UB_unspec_conditional)) ) ]
              end
            end
          end
```

B.12.16 Elaboration of assignment operators

```
      | A.AilEassign e1 e2 ->
          (* STD §6.5.16 *)
          (* TODO: model the non-exact overlap UB *)
          let () = Debug.warn [Debug.DB_elaboration] (fun () ->
            "Cerberus does not currently check the undefined behaviour for non-exactly
            ↪  overlapping assignments (see C11 §6.5.16.1#3)"
          ) in
          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun e1_wrp  ->
          self e1                                             >>= fun core_e1 ->
          self e2                                             >>= fun core_e2 ->
          let ty1 = ctype_of e1 in
          let ty2 = ctype_of e2 in
          begin
if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_arithmetic ty1 &&
↪  AilTypesAux.is_arithmetic ty2 then
            E.return
              ( force_core_object_type_of_ctype ty2
              , core_e2
              , conv_loaded_arith stdlib ty2 (Ctype.unatomic ty1) )

else if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_struct_or_union ty1 then
            (* NOTE: the two struct/union types could be from two different translation
            ↪  units,
                     but as far as I can things are sufficiently restricted such that no
                     conversion is needed here *)
            E.return
              ( force_core_object_type_of_ctype ty2
              , core_e2
              , fun z -> z )

(* NOTE: we apply unatomic to ty1 because the left operand may be an atomic pointer to
↪  ... (STD §6.5.16.1#1, bullet 3) *)
else match AilTypesAux.referenced_type (Ctype.unatomic ty1) with
  | Just ref_ty ->
            E.return
              begin if Aaux.is_null_pointer_constant e2 then
                ( C.OTy_pointer
                , Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe ref_ty))
                , fun z -> z )
              else
                ( force_core_object_type_of_ctype ty2
```

```
                , core_e2
                , fun z -> z )
              end
  | Nothing ->
            (* By Ail's typing, e1 must have type _Bool and e2 must be a pointer *)
            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun conv_wrp ->
            E.return
            ( C.OTy_integer
            , Caux.mk_wseq_e conv_wrp.E.sym_pat core_e2
            ↪  (stdlib.mkproc_loaded_pointer_to_Bool conv_wrp.E.sym_pe)
            , fun z -> z )
end
          end >>= fun (oTy2, core_e2, mk_stored_pe) ->
          E.wrapped_fresh_symbol (C.BTy_loaded oTy2) >>= fun e2_wrp ->
          let object_pe = Caux.mk_std_pe "§6.5.16#3, sentence 1" e1_wrp.E.sym_pe in
          let stored_pe = Caux.mk_std_pe "§6.5.16.1#2, conversion" (mk_stored_pe
          ↪  e2_wrp.E.sym_pe) in
          let core_ty_pe1 = Caux.mk_ail_ctype_pe (AilTypesAux.rvalue_coercion (snd
          ↪  (from_lvalue_type e1))) in
          let mo =
            if AilTypesAux.is_atomic ty1 then
              (* STD §6.2.6.1#9 *)
              Cmm.Seq_cst
            else
              Cmm.NA in
          E.return begin
            Caux.add_std "§6.5.16#3, sentence 4" begin
              Caux.mk_wseq_e (Caux.mk_tuple_pat [ e1_wrp.E.sym_pat; e2_wrp.E.sym_pat ])
                (Caux.add_std "§6.5.16#3, sentence 5" (Caux.mk_unseq_e [core_e1;
                ↪  core_e2])) begin
                  Caux.mk_wseq_e (Caux.mk_empty_pat C.BTy_unit)
                    (C.Expr [Annot.Astd "§6.5.16.1#2, store"] (
                      C.Eaction (C.Paction C.Neg (C.Action loc default (C.Store false(*
                      ↪  not locking *) core_ty_pe1 object_pe stored_pe mo)))
                    ))
                    (Caux.mk_pure_e stored_pe)
              end
            end
          end
```

B.12.17 Elaboration of the comma operator

```
      | A.AilEbinary e1 A.Comma e2 ->
          (* STD §6.5.17 *)
          self e1 >>= fun core_e1 ->
          self e2 >>= fun core_e2 ->
          let bTy =
            let ty_e1 = ctype_of e1 in
            if AilTypesAux.is_void ty_e1 then
              C.BTy_unit
            else
              C.BTy_loaded (force_core_object_type_of_ctype ty_e1) in
          (* STD (§6.5.17, sentence 2) says there is a sequence point between the
             evaluation of the two operand. Hence the strong sequencing *)
          E.return begin
            Caux.add_std "§6.5.17#2, sentence 2" begin
              Caux.mk_sseq_e (Caux.mk_empty_pat bTy) core_e1 core_e2
            end
```

```
2090            end
```

B.12.18 Elaboration of calls to atomic generic functions

```
2095        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBatomic
            ↪  A.AilBAstore))) es ->
2096            translate_atomic_explicit loc self AtomicStoreExplicit es
2097        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBatomic
            ↪  A.AilBAload))) es ->
2098            translate_atomic_explicit loc self AtomicLoadExplicit es
2099        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEident (Symbol.Symbol _ _
            ↪  (Symbol.SD_Id "atomic_thread_fence")))) es ->
2100            translate_atomic_explicit loc self AtomicThreadFence es
2101        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBatomic
            ↪  A.AilBAcompare_exchange_strong))) es ->
2102            translate_atomic_explicit loc self AtomicCompareExchangeStrongExplicit es
2103        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBatomic
            ↪  A.AilBAcompare_exchange_weak))) es ->
2104            translate_atomic_explicit loc self AtomicCompareExchangeWeakExplicit es
2105        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBlinux
            ↪  A.AilBLwrite))) es ->
2106            translate_atomic_explicit loc self LinuxStore es
2107        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBlinux
            ↪  A.AilBLread))) es ->
2108            translate_atomic_explicit loc self LinuxLoad es
2109        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBlinux
            ↪  A.AilBLfence))) es ->
2110            translate_atomic_explicit loc self LinuxFence es
2111        | A.AilEcall (A.AnnotatedExpression _ _ _ (A.AilEbuiltin (A.AilBlinux
            ↪  A.AilBLrmw))) es ->
2112            translate_atomic_explicit loc self LinuxRMW es
```

B.12.19 Elaboration of function calls without arguments

```
2135        | A.AilEcall e [] ->
2136            (* NOTE: when there are no arguments, we don't need all the temporary object
                ↪  creation stuff *)
2137            (* STD §6.5.2.2 *)
2138            let ret_ty = match ctype_of e with
2139              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.FunctionNoParams (_,
                  ↪  ret_ty)))) ->
2140                  ret_ty
2141              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Function (_, ret_ty)
                  ↪  params isVariadic))) ->
2142                  if List.length params = 0 && (not isVariadic) then
2143                    ret_ty
2144                  else
2145                    illTypedAil loc "AilEcall"
2146              | (*BISECT-IGNORE*) _ ->
2147                  illTypedAil loc "AilEcall"
2148            end in
2149            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun fun_wrp    ->
2150            E.wrapped_fresh_symbol C.BTy_ctype                  >>= fun ret_wrp    ->
2151            E.wrapped_fresh_symbol (C.BTy_list C.BTy_ctype)     >>= fun params_wrp ->
2152            self e                                              >>= fun core_e     ->
2153            E.return begin
2154              (* STD §6.5.2.2#10 *)
```

```
2155            Caux.mk_sseq_e fun_wrp.E.sym_pat core_e begin
2156            Caux.mk_let_e
2157              (Caux.mk_tuple_pat [ret_wrp.E.sym_pat; params_wrp.E.sym_pat;
                ↪  Caux.mk_empty_pat C.BTy_boolean; Caux.mk_empty_pat C.BTy_boolean])
2158              (Caux.mk_cfunction_pe fun_wrp.E.sym_pe)
2159              begin
2160                Caux.mk_if_e_ [Annot.Anot_explode]
2161                  (Caux.mk_op_pe C.OpEq (stdlib.mkcall_params_length
                    ↪  params_wrp.E.sym_pe) (Caux.mk_integer_pe 0))
2162                    begin
2163                      Caux.mk_if_e_ [Annot.Anot_explode]
2164                        (Caux.mk_are_compatible (Caux.mk_ail_ctype_pe ret_ty)
                          ↪  ret_wrp.E.sym_pe)
2165                          (Caux.mk_ccall_e (Caux.mk_ail_ctype_pe (ctype_of e))
                            ↪  fun_wrp.E.sym_pe [(*Caux.mk_boolean_pe is_used*)])
2166                          (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#9"
                            ↪  Undefined.UB041_function_not_compatible))
2167                    end
2168                    (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.5.2.2#6, sentence 3"
                      ↪  Undefined.UB038_number_of_args))
2169              end
2170            end
2171          end
```

B.12.20 Elaboration of function calls with arguments

See the auxiliary function in Section B.10.

```
2173      | A.AilEcall e es ->
2174          translate_function_call loc is_used self stdlib e es
```

B.12.21 Elaboration of calls to `assert()`

```
2176      | A.AilEassert e ->
2177          let oTy = force_core_object_type_of_ctype (ctype_of e) in
2178          E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp     ->
2179          E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp   ->
2180          E.wrapped_fresh_symbol C.BTy_boolean      >>= fun memop_wrp ->
2181          self e                                    >>= fun core_e    ->
2182          E.return begin
2183            Caux.mk_sseq_e e_wrp.E.sym_pat core_e
2184  begin if AilTypesAux.is_arithmetic (ctype_of e) then
2185              let zero_pe =
2186                if AilTypesAux.is_integer (ctype_of e) then
2187                  Caux.mk_integer_pe 0
2188                else
2189                  Caux.mk_floating_value_pe Mem.zero_fval in
2190              Caux.mk_pure_e begin
2191                Caux.mk_case_pe e_wrp.E.sym_pe
2192                  [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
2193                    , Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_op_pe C.OpEq
                      ↪  obj_wrp.E.sym_pe zero_pe)
2194                        (Caux.mk_error_pe "assert() failure" Caux.mk_unit_pe)
2195                      Caux.mk_unit_pe )
2196                  ; ( Caux.mk_empty_pat (C.BTy_loaded oTy)
2197                    , Caux.mk_error_pe "assert() unspecified" Caux.mk_unit_pe ) ]
2198              end
```

```
2199 else (* is_pointer *)
2200               Caux.mk_case_e e_wrp.E.sym_pe
2201                 [ ( Caux.mk_specified_pat obj_wrp.E.sym_pat
2202                   , Caux.mk_wseq_e memop_wrp.E.sym_pat
2203                      (Caux.mk_memop_e Mem_common.PtrEq [obj_wrp.E.sym_pe;
                          ↪  Caux.mk_nullptr_pe Cty.void])
2204                      begin
2205                        Caux.mk_pure_e begin
2206                          Caux.mk_if_pe_ [Annot.Anot_explode] memop_wrp.E.sym_pe
2207                            (Caux.mk_error_pe "assert() failure" Caux.mk_unit_pe)
2208                            Caux.mk_unit_pe
2209                        end
2210                      end )
2211                 ; ( Caux.mk_empty_pat (C.BTy_loaded oTy)
2212                   , Caux.mk_pure_e (Caux.mk_error_pe "assert() unspecified"
                      ↪  Caux.mk_unit_pe) ) ]
2213 end
2214           end
```

B.12.22 Elaboration of the `offsetof()` operator

```
2215
2216       | A.AilEoffsetof ty membr_ident ->
2217           let tag_sym = match ty with
2218             | Ctype.Ctype _ (Ctype.Struct x) ->
2219                 x
2220             | Ctype.Ctype _ (Ctype.Union x) ->
2221                 x
2222             | (*BISECT-IGNORE*) _ ->
2223                 illTypedAil loc "AilEoffsetof"
2224           end in
2225           E.return begin
2226             Caux.mk_pure_e begin
2227               Caux.mk_value_pe begin
2228                 C.Vloaded (C.LVspecified (C.OVinteger (Mem.offsetof_ival tagDefs tag_sym
                    ↪  membr_ident)))
2229               end
2230             end
```

B.12.23 Elaboration of compound values

The AST of Ail has explicit nodes for array, struct, and unions values that result from initialisers or compound literals. They are introduced during the desugaring from Cabs to Ail, and have no direct counterparts in the AST of C.

B.12.23.1 Elaboration of array values

```
2237       | A.AilEarray _ _ e_opts ->
2238           let elem_ty = match result_ty with
2239             | Ctype.Ctype _ (Ctype.Array ty _) ->
2240                 ty
2241             | (*BISECT-IGNORE*) _ ->
2242                 illTypedAil loc "AilEarray"
2243           end in
2244           E.foldlM (fun (pe_acc, (pat_acc, core_e_acc)) e_opt ->
2245             match e_opt with
```

```
2246              | Just e ->
2247                  translate_assignment_conversion self stdlib elem_ty e >>= fun
                      ↪  (conv_oTy, core_e, mk_conv_pe) ->
2248                  E.wrapped_fresh_symbol (C.BTy_loaded conv_oTy)       >>= fun e_wrp
                      ↪  ->
2249                  E.return
2250                    ( mk_conv_pe e_wrp.E.sym_pe :: pe_acc
2251                    , (e_wrp.E.sym_pat :: pat_acc, core_e :: core_e_acc) )
2252              | Nothing ->
2253                  E.return
2254                   ( Caux.mk_unspecified_pe elem_ty :: pe_acc
2255                   , (pat_acc, core_e_acc) )
2256            end
2257          ) ([], ([], [])) e_opts >>= fun (rev_pes, (pat_acc, core_e_acc)) ->
2258          E.return begin
2259            match (pat_acc, core_e_acc) with
2260              | ([pat], [core_e]) ->
2261                  Caux.mk_wseq_e pat core_e
2262              | (_ :: _, _ :: _) ->
2263                  (* STD (§6.7.9#23) the evaluations are unsequenced *)
2264                  Caux.mk_wseq_e (Caux.mk_tuple_pat pat_acc) (Caux.add_std "§6.7.9#23"
                      ↪  (Caux.mk_unseq_e core_e_acc))
2265              | _ ->
2266                  (* this is not possible (the Ail would be illformed) *)
2267                  error "AilEarray"
2268            end
2269            begin
2270              Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_array_pe (List.reverse
                  ↪  rev_pes)))
2271            end
2272          end
```

B.12.23.2 Elaboration of struct values

```
2274      | A.AilEstruct tag_sym ident_e_opts ->
2275          let ident_tys = match Map.lookup tag_sym tagDefs with
2276            | Just (Cty.StructDef z _) ->
2277                z
2278            | (*BISECT-IGNORE*) _ ->
2279                illTypedAil loc "AilEstruct"
2280          end in
2281          E.foldlM (fun (acc, (pat_acc, core_e_acc)) (ident, e_opt) ->
2282            match e_opt with
2283              | Just e ->
2284                  let (_, _, _, memb_ty) = fromJust "Translation.translate_expression,
                      ↪  AilEstruct 2" (List.lookup ident ident_tys) in
2285                  translate_assignment_conversion self stdlib memb_ty e >>= fun
                      ↪  (conv_oTy, core_e, mk_conv_pe) ->
2286                  E.wrapped_fresh_symbol (C.BTy_loaded conv_oTy)       >>= fun e_wrp
                      ↪  ->
2287                  E.return
2288                    ( (ident, mk_conv_pe e_wrp.E.sym_pe) :: acc
2289                    , (e_wrp.E.sym_pat :: pat_acc, core_e :: core_e_acc) )
2290              | Nothing ->
2291                  let (_, _, _, ty) = fromJust "Translation.translate_expression,
                      ↪  AilEstruct 3" (List.lookup ident ident_tys) in
2292                  E.return
2293                    ( (ident, Caux.mk_unspecified_pe ty) :: acc
```

```
                    , (pat_acc, core_e_acc) )
          end
        ) ([], ([], [])) ident_e_opts >>= fun (core_xs_rev, (pat_acc, core_e_acc)) ->
        E.return begin
          match (pat_acc, core_e_acc) with
            | ([pat], [core_e]) ->
                Caux.mk_wseq_e pat core_e
            | (_ :: _, _ :: _) ->
                (* STD (§6.7.9#23) the evaluations are unsequenced *)
                Caux.mk_wseq_e (Caux.mk_tuple_pat pat_acc) (Caux.add_std "§6.7.9#23"
                ↪  (Caux.mk_unseq_e core_e_acc))
            | _ ->
                (* this is not possible (the Ail would be illformed) *)
                error "AilEstruct"
          end
          begin
            Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_struct_pe tag_sym
            ↪  (List.reverse core_xs_rev)))
          end
        end
```

B.12.23.3 Elaboration of union values

```
    | A.AilEunion tag_sym memb_ident e_opt ->
        match Map.lookup tag_sym tagDefs with
          | (*BISECT-IGNORE*) Nothing ->
              illTypedAil loc "AilEunion: couldn't find the definition"
          | (*BISECT-IGNORE*) Just (Cty.StructDef _ _) ->
              illTypedAil loc "AilEunion: found a struct definition"
          | Just (Cty.UnionDef ident_tys) ->
              match List.lookup memb_ident ident_tys with
                | (*BISECT-IGNORE*) Nothing ->
                    illTypedAil loc "AilEunion: couldn't find a union definition"
                | Just (_, _, _, memb_ty) ->
                    match e_opt with
                      | Just e ->
                          translate_assignment_conversion self stdlib memb_ty e >>=
                          ↪  fun (conv_oTy, core_e, mk_conv_pe) ->
                          E.wrapped_fresh_symbol (C.BTy_loaded conv_oTy)       >>= fun
                          ↪  e_wrp                              ->
                          E.return begin
                            Caux.mk_wseq_e e_wrp.E.sym_pat core_e
                              (Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_union_pe
                              ↪  tag_sym memb_ident (mk_conv_pe e_wrp.E.sym_pe))))
                          end
                      | Nothing ->
                          E.return begin
                            Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_union_pe
                            ↪  tag_sym memb_ident (Caux.mk_unspecified_pe memb_ty)))
                          end
                    end
              end
        end
```

B.12.24 Elaboration of compound literals

```
2340        | A.AilEcompound qs ty e ->
2341            let core_ty = Caux.mk_ail_ctype_pe ty in
2342            let oTy = force_core_object_type_of_ctype (ctype_of e) in
2343            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj_wrp ->
2344            E.wrapped_fresh_symbol (C.BTy_loaded oTy)           >>= fun e_wrp   ->
2345            self e                                              >>= fun core_e  ->
2346            (* STD §6.5.2.5#3 *)
2347            E.register_compound_literal loc (Symbol.PrefCompoundLiteral (locOf e)
                ↪  (Symbol.digest_of_sym e_wrp.E.sym_sym)) obj_wrp.E.sym_sym qs.Ctype.const
                ↪  ty >>= fun () ->
2348            E.return begin
2349              (* NOTE: the elaboration ensures [ty] and the type of [e] are the same *)
2350              Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
2351              Caux.mk_wseq_e (Caux.mk_empty_pat C.BTy_unit) (Caux.pstore loc core_ty
                  ↪  obj_wrp.E.sym_pe e_wrp.E.sym_pe Cmm.NA) begin
2352                (* STD §6.5.2.5#5, sentence 1 *)
2353                Caux.mk_pure_e obj_wrp.E.sym_pe
2354              end end
2355            end
```

B.12.25 Elaboration of the . operator

```
2357        | A.AilEmemberof e ident ->
2358            (* STD §6.5.2.3 *)
2359            let (tag_sym, oTy) = match ctype_of e with
2360              | Ctype.Ctype _ (Ctype.Struct tag_sym) ->
2361                  (tag_sym, C.OTy_struct tag_sym)
2362              | Ctype.Ctype _ (Ctype.Atomic (Ctype.Ctype _ (Ctype.Struct tag_sym))) ->
2363                  (tag_sym, C.OTy_struct tag_sym)
2364              | Ctype.Ctype _ (Ctype.Union tag_sym) ->
2365                  (tag_sym, C.OTy_union tag_sym)
2366              | Ctype.Ctype _ (Ctype.Atomic (Ctype.Ctype _ (Ctype.Union tag_sym))) ->
2367                  (tag_sym, C.OTy_union tag_sym)
2368              | (*BISECT-IGNORE*) _ ->
2369                  illTypedAil loc "AilEmemberof"
2370            end in
2371            let bTy = if is_lvalue then C.BTy_object C.OTy_pointer else C.BTy_loaded oTy
                ↪  in
2372            E.wrapped_fresh_symbol bTy >>= fun e_wrp  ->
2373            self e                     >>= fun core_e ->
2374
2375            if not is_lvalue then
2376               E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun obj_wrp ->
2377               E.return begin
2378                 Caux.mk_sseq_e e_wrp.E.sym_pat core_e begin
2379                   Caux.mk_pure_e begin
2380                     Caux.mk_case_pe e_wrp.E.sym_pe
2381                       [ ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
2382                         , Caux.mk_undef_pe loc (Undefined.UB_CERB004_unspecified
                            ↪  Undefined.UB_unspec_rvalue_memberof) )
2383                       ; ( Caux.mk_specified_pat obj_wrp.E.sym_pat
2384                         , Caux.mk_memberof_pe tag_sym ident obj_wrp.E.sym_pe ) ]
2385                   end
2386                 end
2387               end
2388            else
```

```
2389            E.return begin
2390              C.Expr [Annot.Astd "§6.5.2.3#3, sentence 2"] (
2391                C.Esseq e_wrp.E.sym_pat core_e (
2392                  Caux.mk_pure_e (Caux.mk_member_shift_pe e_wrp.E.sym_pe tag_sym ident)
2393                )
2394              )
2395            end
```

B.12.26 Elaboration of the -> operator

```
2397        | A.AilEmemberofptr e ident ->
2398            (* STD §6.5.2.3 *)
2399            let (ref_ty, tag_sym) = match ctype_of e with
2400              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Struct tag_sym) as
                  ↪ ref_ty)) ->
2401                  (ref_ty, tag_sym)
2402              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Atomic (Ctype.Ctype _
                  ↪ (Ctype.Struct tag_sym))) as ref_ty)) ->
2403                  (ref_ty, tag_sym)
2404              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Union tag_sym) as
                  ↪ ref_ty)) ->
2405                  (ref_ty, tag_sym)
2406              | Ctype.Ctype _ (Ctype.Pointer _ (Ctype.Ctype _ (Ctype.Atomic (Ctype.Ctype _
                  ↪ (Ctype.Union tag_sym))) as ref_ty)) ->
2407                  (ref_ty, tag_sym)
2408              | (*BISECT-IGNORE*) _ -> illTypedAil loc "AilEmemberofptr"
2409            end in
2410            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_pointer) >>= fun e_wrp    ->
2411            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun obj_wrp  ->
2412            E.wrapped_fresh_symbol C.BTy_boolean                >>= fun test_wrp ->
2413            self e                                              >>= fun core_e   ->
2414            E.return begin
2415              Caux.add_std "§6.5.2.3#4, sentence 2" begin
2416                Caux.mk_sseq_e e_wrp.E.sym_pat core_e begin
2417                  Caux.mk_case_e e_wrp.E.sym_pe
2418                    [ ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
2419                      , Caux.mk_pure_e (Caux.mk_undef_pe loc
                          ↪ (Undefined.UB_CERB004_unspecified
                          ↪ Undefined.UB_unspec_memberofptr)) )
2420                    ; ( Caux.mk_specified_pat obj_wrp.E.sym_pat
2421                      , if Global.has_strict_pointer_arith () then
2422                          Caux.mk_wseq_e test_wrp.E.sym_pat
2423                            (Caux.mk_memop_e Mem_common.PtrValidForDeref
                              ↪ [Caux.mk_ail_ctype_pe ref_ty; obj_wrp.E.sym_pe])
2424                            begin
2425                              Caux.mk_pure_e begin
2426                                Caux.mk_if_pe_ [Annot.Anot_explode] test_wrp.E.sym_pe
2427                                  (Caux.mk_member_shift_pe obj_wrp.E.sym_pe tag_sym ident)
2428                                  (Caux.mk_std_undef_pe loc "§6.5.2.3#4, sentence 4"
                                    ↪ Undefined.UB043_indirection_invalid_value)
2429                              end
2430                            end
2431                        else
2432                          (* NON-ISO: allowing member_shift on "invalid pointer values" *)
2433                          Caux.mk_pure_e (Caux.mk_member_shift_pe obj_wrp.E.sym_pe tag_sym
                              ↪ ident) ) ]
2434                end
2435              end
```

```
2436          end
```

B.12.27 Elaboration of constants

See the auxiliary function in Section B.2.3.

```
2438      | A.AilEconst cst ->
2439          E.return (Caux.mk_pure_e (translate_constant cst))
```

B.12.28 Elaboration of string literals

```
2451      | A.AilEstr (pref_opt, strs) ->
2452          let strs = List.concat (List.map snd strs) in
2453          let elem_ty = match pref_opt with
2454            | Nothing ->
2455                (* STD §6.4.5#6, sentence 3 *)
2456                Cty.char
2457            | Just A.Enc_u8 ->
2458                (* STD §6.4.5#6, sentence 4 *)
2459                Cty.char
2460            | Just A.Enc_u ->
2461                (* STD §6.4.5#6, sentence 6 *)
2462                Cty.char16_t
2463            | Just A.Enc_U ->
2464                (* STD §6.4.5#6, sentence 6 *)
2465                Cty.char32_t
2466            | Just A.Enc_L ->
2467                (* STD §6.4.5#6, sentence 5 *)
2468                Cty.wchar_t
2469          end in
2470          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun lit_wrp ->
2471          let elem_pes = List.map (fun c_str ->
2472             Caux.mk_specified_pe (
2473               stdlib.mkcall_conv_int elem_ty
2474                 (Caux.mk_integer_pe (Decode.decode_character_constant c_str))
2475             )
2476          ) strs ++ [Caux.mk_specified_pe (Caux.mk_integer_pe 0)] in
2477          E.register_string_literal loc lit_wrp.E.sym_sym (Cty.Ctype [] (Cty.Array
          ↪  elem_ty (Just (integerFromNat (List.length elem_pes)))))
2478            (Caux.mk_specified_pe (Caux.mk_array_pe elem_pes))  >>
2479          E.return (
2480            Caux.mk_pure_e lit_wrp.E.sym_pe
2481          )
```

B.12.29 Elaboration of the `sizeof` operator

```
2483      | A.AilEsizeof _ ty ->
2484          E.return begin
2485            if AilTypesAux.is_character ty then
2486              Caux.add_std "§6.5.3.4#4, sentence 1" begin
2487                Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_integer_pe 1))
2488              end
2489            else
2490              Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_sizeof_pe
                  ↪  (Caux.mk_ail_ctype_pe ty)))
2491          end
2492
```

```
2493        | A.AilEsizeof_expr e ->
2494            (* NOTE: from (§6.2.5#26, sentence 3) we know that qualifers do no affect the
                ↪  size of a type,
2495                     so we can use 'Ctype.no_qualifiers' *)
2496            self (A.AnnotatedExpression annot std_annots loc (A.AilEsizeof
                ↪  Ctype.no_qualifiers (ctype_of e)))
```

B.12.30 Elaboration of the `_Alignof` operator

```
2498        | A.AilEalignof _ ty ->
2499            E.return begin
2500              Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_alignof_pe
                  ↪  (Caux.mk_ail_ctype_pe ty)))
2501            end
```

B.12.31 Elaboration of calls to `<stdarg.h>` macros and functions

```
2506        | A.AilEva_start _ last_sym ->
2507            let (variadic_sym, last_arg_sym) =
2508              match variadic_env with
2509              | (Just var_sym, Just last_sym) -> (var_sym, last_sym)
2510              | _ -> error ((Loc.stringFromLocation (Loc.locOf a_expr)) ^ ": va_start not
                  ↪  in a variadic function")
2511              end in
2512            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun va_wrp ->
2513            E.return begin
2514              if last_sym = last_arg_sym then
2515                Caux.mk_sseq_e va_wrp.E.sym_pat (C.Expr [] (C.Ememop Mem_common.Va_start
                    ↪  [Caux.mk_sym_pe variadic_sym]))
2516                  (Caux.mk_pure_e (Caux.mk_specified_pe va_wrp.E.sym_pe))
2517              else
2518                error ((Loc.stringFromLocation (Loc.locOf a_expr)) ^ " : " ^ show last_sym
                    ↪  ^ " is not the last argument") (* it should be UB I think *)
2519            end
2520
2521        | A.AilEva_copy _ e ->
2522            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e_wrp    ->
2523            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun case_wrp ->
2524            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun va_wrp   ->
2525            self e                                              >>= fun core_e   ->
2526            E.return begin
2527              Caux.mk_sseq_e e_wrp.E.sym_pat core_e
2528                (Caux.mk_case_e e_wrp.E.sym_pe
2529                  [ ( Caux.mk_specified_pat case_wrp.E.sym_pat
2530                    , Caux.mk_sseq_e va_wrp.E.sym_pat (C.Expr [] (C.Ememop
                          ↪  Mem_common.Va_copy [case_wrp.E.sym_pe]))
2531                        (Caux.mk_pure_e (Caux.mk_specified_pe va_wrp.E.sym_pe)) )
2532                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
2533                    , Caux.mk_pure_e (Caux.mk_undef_exceptional_condition loc) ) ] )
2534            end
2535
2536        | A.AilEva_arg e ty ->
2537            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e_wrp    ->
2538            E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun ptr_wrp  ->
2539            E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun case_wrp ->
2540            self e                                              >>= fun core_e   ->
2541            E.return begin
```

```
2542            Caux.mk_sseq_e e_wrp.E.sym_pat core_e
2543              (Caux.mk_sseq_e ptr_wrp.E.sym_pat
2544                (Caux.mk_case_e e_wrp.E.sym_pe
2545                  [ ( Caux.mk_specified_pat case_wrp.E.sym_pat
2546                    , (C.Expr [] (C.Ememop Mem_common.Va_arg [case_wrp.E.sym_pe;
                          ↪  Caux.mk_ail_ctype_pe ty])) )
2547                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
2548                    , Caux.mk_pure_e (Caux.mk_undef_exceptional_condition loc) ) ] )
2549                (Caux.pload loc (Caux.mk_ail_ctype_pe ty) ptr_wrp.E.sym_pe Cmm.NA))
2550          end
2551
2552      | A.AilEva_end e ->
2553          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun e_wrp    ->
2554          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer) >>= fun case_wrp ->
2555          self e                                              >>= fun core_e   ->
2556          E.return begin
2557            Caux.mk_sseq_e e_wrp.E.sym_pat core_e
2558              (Caux.mk_case_e e_wrp.E.sym_pe
2559                [ ( Caux.mk_specified_pat case_wrp.E.sym_pat
2560                  , (C.Expr [] (C.Ememop Mem_common.Va_end [case_wrp.E.sym_pe])) )
2561                ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
2562                  , Caux.mk_pure_e (Caux.mk_undef_exceptional_condition loc) ) ] )
2563          end
```

B.12.32 Elaboration of lvalue and function pointer coercions

```
2612      | A.AilErvalue e ->
2613          let mo =
2614            if AilTypesAux.is_atomic (ctype_of e) then
2615              (* STD §6.2.6.1#9 *)
2616              Cmm.Seq_cst
2617            else
2618              Cmm.NA in
2619          self e >>= fun core_e ->
2620          E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun e_wrp ->
2621          E.return begin
2622            Caux.mk_wseq_e e_wrp.E.sym_pat core_e begin
2623              Caux.pload loc (Caux.mk_ail_ctype_pe result_ty) e_wrp.E.sym_pe mo
2624            end
2625          end
2626      | A.AilEarray_decay e ->
2627          match GenTypes.genTypeCategoryOf e with
2628            | GenTypes.GenLValueType _ _ _ ->
2629                (* by Ail typing, [e] is a lvalue. Hence the corresponding Core has a
                      ↪  non-loaded type *)
2630                self e >>= fun core_e ->
2631                E.wrapped_fresh_symbol (C.BTy_object C.OTy_pointer) >>= fun e_wrp ->
2632                E.return (
2633                  Caux.mk_wseq_e e_wrp.E.sym_pat core_e (
2634                    match AilTypesAux.referenced_type result_ty with
2635                      | (*BISECT-IGNORE*) Nothing ->
2636                          illTypedAil loc "AilEarray_decay, result_ty not a pointer"
2637                      | Just ref_ty ->
2638                          C.Expr [Annot.Astd "§6.3.2.1#3"] (
2639                            C.Epure (
2640                              Caux.mk_specified_pe (Caux.mk_array_shift e_wrp.E.sym_pe
2641                                ref_ty (Caux.mk_integer_pe 0)
2642                              )
```

```
                              )
                            )
                      end
                    )
                  )
              | GenTypes.GenRValueType _ ->
                  illTypedAil loc "AilEarray_decay, GenRValueType"
            end
        | A.AilEfunction_decay e ->
            translate_function_designator self stdlib e
```

B.13 Auxiliary functions helping the elaboration of statements

```
module St = State
let inline (>>=) = St.bind
let inline (>>) m1 m2 = St.bind m1 (fun _ -> m2)
let inline (<$>) = State.fmap
let inline (<*>) = State.app
```

B.13.1 Collection of the cases of `switch` statement

```
type collect_cases_state = <|
  found_default: bool;
  case_csts: list A.integerConstant;
|>

val     collect_cases_: A.statement GenTypes.genTypeCategory -> St.stateM unit
↪  collect_cases_state
let rec collect_cases_ (A.AnnotatedStatement loc _ stmt) =
  let register_case ic_n =
    St.update (fun s -> <| s with case_csts= ic_n :: s.case_csts |>) in
  let register_default =
    St.update (fun s -> <| s with found_default= true |>) in
  match stmt with
    | A.AilSskip ->
        St.return ()
    | A.AilSexpr _ ->
        St.return ()
    | A.AilSblock binds ss ->
        St.mapM_ collect_cases_ ss
    | A.AilSif _ s1 s2 ->
        collect_cases_ s1 >> collect_cases_ s2
    | A.AilSwhile _ s _ ->
        collect_cases_ s
    | A.AilSdo s _ _ ->
        collect_cases_ s
    | A.AilSbreak ->
        St.return ()
    | A.AilScontinue ->
        St.return ()
    | A.AilSreturnVoid ->
        St.return ()
    | A.AilSreturn _ ->
        St.return ()
```

```
2697      | A.AilSswitch _ _ ->
2698          St.return ()
2699      | A.AilScase ic_n s ->
2700          register_case ic_n >> collect_cases_ s
2701      | A.AilSdefault s ->
2702          register_default >> collect_cases_ s
2703      | A.AilSlabel _ s _ ->
2704          collect_cases_ s
2705      | A.AilSgoto _ ->
2706          St.return ()
2707      | A.AilSdeclaration _ ->
2708          St.return ()
2709      | A.AilSpar ss ->
2710          St.mapM_ collect_cases_ ss
2711      | A.AilSreg_store _ _ ->
2712          St.return ()
2713      | A.AilSpack _ _ ->
2714          St.return ()
2715      | A.AilSunpack _ _ ->
2716          St.return ()
2717      | A.AilShave _ _ ->
2718          St.return ()
2719      | A.AilSshow _ _ ->
2720          St.return ()
2721      | A.AilSinstantiate _ _ ->
2722          St.return ()
2723    end
2724  let collect_cases s =
2725    snd (St.runStateM (collect_cases_ s) <| found_default= false; case_csts= [] |>)
```

B.13.2 Erasure of loop control statements

```
2728  type erase_loop_control_state = <|
2729    elc_continue: maybe Symbol.sym;
2730    elc_break: maybe Symbol.sym;
2731  |>
2732
2733  val     erase_loop_control_aux: A.statement GenTypes.genTypeCategory -> St.stateM
      ↪  (A.statement GenTypes.genTypeCategory) erase_loop_control_state
2734  let rec erase_loop_control_aux (A.AnnotatedStatement loc attrs stmt_) =
2735    let with_fresh_labels mf =
2736      fun st ->
2737        let continue_sym = Symbol.fresh_pretty_with_id (fun x -> "continue_" ^ show x) in
2738        let break_sym    = Symbol.fresh_pretty_with_id (fun x -> "break_" ^ show x) in
2739        let (ret, st')   = mf <| elc_continue= Just continue_sym; elc_break= Just
          ↪  break_sym |> in
2740        ((continue_sym, break_sym, ret), st) in
2741    A.AnnotatedStatement loc attrs <$> match stmt_ with
2742      | A.AilSskip ->
2743          St.return stmt_
2744      | A.AilSexpr _ ->
2745          St.return stmt_
2746      | A.AilSblock binds ss ->
2747          A.AilSblock binds <$> St.mapM erase_loop_control_aux ss
2748      | A.AilSif e s1 s2 ->
2749          A.AilSif e <$> erase_loop_control_aux s1 <*> erase_loop_control_aux s2
2750      | A.AilSwhile e s loop_id ->
2751          (* STD §6.8.6.2#2 and §6.8.6.3#2 *)
```

```
2752          with_fresh_labels (erase_loop_control_aux s) >>= fun (continue_sym, break_sym,
              ↪  s') ->
2753          St.return begin
2754            A.AilSblock []
2755              [ A.AnnotatedStatement loc Annot.no_attributes (A.AilSwhile e
                  ↪  (A.AnnotatedStatement loc Annot.no_attributes (A.AilSblock []
2756                  [s'; A.AnnotatedStatement loc Annot.no_attributes (A.AilSlabel
                      ↪  continue_sym (A.AnnotatedStatement loc Annot.no_attributes
                      ↪  A.AilSskip) (Just (Annot.LAloop_continue loop_id)))])) loop_id)
2757              ; A.AnnotatedStatement loc Annot.no_attributes (A.AilSlabel break_sym
                  ↪  (A.AnnotatedStatement loc Annot.no_attributes A.AilSskip) (Just
                  ↪  (Annot.LAloop_break loop_id))) ]
2758            end
2759
2760        | A.AilSdo s e loop_id ->
2761            (* STD §6.8.6.2#2 and §6.8.6.3#2 *)
2762            with_fresh_labels (erase_loop_control_aux s) >>= fun (continue_sym, break_sym,
                ↪  s') ->
2763            St.return begin
2764              A.AilSblock []
2765                [ A.AnnotatedStatement loc Annot.no_attributes (A.AilSdo
                    ↪  (A.AnnotatedStatement loc Annot.no_attributes (A.AilSblock []
2766                    [s'; A.AnnotatedStatement loc Annot.no_attributes (A.AilSlabel
                        ↪  continue_sym (A.AnnotatedStatement loc Annot.no_attributes
                        ↪  A.AilSskip) (Just (Annot.LAloop_continue loop_id)))])) e loop_id)
2767                ; A.AnnotatedStatement loc Annot.no_attributes (A.AilSlabel break_sym
                    ↪  (A.AnnotatedStatement loc Annot.no_attributes A.AilSskip) (Just
                    ↪  (Annot.LAloop_break loop_id))) ]
2768              end
2769        | A.AilSbreak ->
2770            St.get >>= function
2771              | <| elc_break= Just break_sym |> ->
2772                 St.return (A.AilSgoto break_sym)
2773              | (*BISECT-IGNORE*) _ ->
2774                  illTypedAil loc "AilSbreak"
2775            end
2776        | A.AilScontinue ->
2777            St.get >>= function
2778              | <| elc_continue= Just continue_sym |> ->
2779                 St.return (A.AilSgoto continue_sym)
2780              | (*BISECT-IGNORE*) _ ->
2781                  illTypedAil loc "AilScontinue"
2782            end
2783        | A.AilSreturnVoid ->
2784            St.return stmt_
2785        | A.AilSreturn _ ->
2786            St.return stmt_
2787        | A.AilSswitch e s ->
2788            with_fresh_labels (erase_loop_control_aux s) >>= fun (_, break_sym, s') ->
2789            St.return begin
2790              A.AilSblock []
2791                [ A.AnnotatedStatement loc Annot.no_attributes (A.AilSswitch e
                    ↪  (A.AnnotatedStatement loc Annot.no_attributes (A.AilSblock []
2792                    [ A.AnnotatedStatement loc Annot.no_attributes (A.AilSgoto break_sym);
                        ↪  s' ])))
2793                ; A.AnnotatedStatement loc Annot.no_attributes (A.AilSlabel break_sym
                    ↪  (A.AnnotatedStatement loc Annot.no_attributes A.AilSskip) (Just
                    ↪  Annot.LAswitch)) ]
```

```
2794            end
2795      | A.AilScase iCst s ->
2796          A.AilScase iCst <$> erase_loop_control_aux s
2797      | A.AilSdefault s ->
2798          A.AilSdefault <$> erase_loop_control_aux s
2799      | A.AilSlabel sym s m_loop_annot ->
2800          erase_loop_control_aux s >>= fun s ->
2801          St.return (A.AilSlabel sym s m_loop_annot)
2802      | A.AilSgoto _ ->
2803          St.return stmt_
2804      | A.AilSdeclaration _ ->
2805          St.return stmt_
2806      | A.AilSpar ss ->
2807          A.AilSpar <$> St.mapM erase_loop_control_aux ss
2808      | A.AilSreg_store _ _ ->
2809          St.return stmt_
2810      | A.AilSpack _ _ ->
2811          St.return stmt_
2812      | A.AilSunpack _ _ ->
2813          St.return stmt_
2814      | A.AilShave _ _ ->
2815          St.return stmt_
2816      | A.AilSshow _ _ ->
2817          St.return stmt_
2818      | A.AilSinstantiate _ _ ->
2819          St.return stmt_
2820    end
2821
2822  let erase_loop_control stmt =
2823    let (stmt', _) = State.runStateM (erase_loop_control_aux stmt) <|
2824      elc_continue= Nothing;
2825      elc_break= Nothing;
2826    |> in stmt'
```

B.13.3 Collection of the visible identifiers from label bodies

```
2829  type collect_visibles_state = <|
2830    visible_syms: list (Symbol.sym * Ctype.ctype);
2831    label_visibles_: map Symbol.sym (list (Symbol.sym * Ctype.ctype));
2832  |>
2833
2834  val     collect_visibles_: A.statement GenTypes.genTypeCategory -> St.stateM unit
      ↪  collect_visibles_state
2835  let rec collect_visibles_ (A.AnnotatedStatement loc _ stmt) =
2836    match stmt with
2837      | A.AilSskip ->
2838          St.return ()
2839      | A.AilSexpr _ ->
2840          St.return ()
2841      | A.AilSblock binds ss ->
2842          St.get >>= fun st ->
2843          let saved_syms = st.visible_syms in
2844          St.update (fun st ->
2845            <| st with visible_syms= List.map (fun (sym, (_, _, _, ty)) -> (sym ,ty))
              ↪  binds ++ st.visible_syms |>
2846          ) >>
2847          St.mapM_ collect_visibles_ ss >>
2848          St.update (fun st ->
```

```
2849            <| st with visible_syms= saved_syms |>
2850          )
2851      | A.AilSif _ s1 s2 ->
2852          collect_visibles_ s1 >> collect_visibles_ s2
2853      | A.AilSwhile _ s _ ->
2854          collect_visibles_ s
2855      | A.AilSdo s _ _ ->
2856          collect_visibles_ s
2857      | A.AilSbreak ->
2858          St.return ()
2859      | A.AilScontinue ->
2860          St.return ()
2861      | A.AilSreturnVoid ->
2862          St.return ()
2863      | A.AilSreturn _ ->
2864          St.return ()
2865      | A.AilSswitch _ s ->
2866          collect_visibles_ s
2867      | A.AilScase _ s ->
2868          collect_visibles_ s
2869      | A.AilSdefault s ->
2870          collect_visibles_ s
2871      | A.AilSlabel label s _ ->
2872          St.update (fun st -> <| st with
2873            label_visibles_= Map.insert label st.visible_syms st.label_visibles_
2874          |>) >>
2875          collect_visibles_ s
2876      | A.AilSgoto label ->
2877          St.return ()
2878      | A.AilSdeclaration _ ->
2879          St.return ()
2880      | A.AilSpar ss ->
2881          St.mapM_ collect_visibles_ ss
2882      | A.AilSreg_store _ _ ->
2883          St.return ()
2884      | A.AilSpack _ _ ->
2885          St.return ()
2886      | A.AilSunpack _ _ ->
2887          St.return ()
2888      | A.AilShave _ _ ->
2889          St.return ()
2890      | A.AilSshow _ _ ->
2891          St.return ()
2892      | A.AilSinstantiate _ _ ->
2893          St.return ()
2894    end
2895
2896  val collect_visibles: A.statement GenTypes.genTypeCategory -> collect_visibles_state
2897  let collect_visibles stmt =
2898      snd begin
2899        State.runStateM (collect_visibles_ stmt)
2900          <| visible_syms= []
2901           ; label_visibles_= Map.empty |>
2902      end
```

B.13.4 Elaboration of implicit allocations/deallocations when jumping in or out of a block

```
val mk_run_with_lifetime_e: Loc.t -> Symbol.sym -> list (Symbol.sym * Ctype.ctype) ->
  list (Symbol.sym * Ctype.ctype) -> list C.pexpr -> C.expr unit
let mk_run_with_lifetime_e loc sym visibles_before visibles_after pes =
  let killed_syms_tys =
    List.filter (fun (sym,_ty) ->
      Maybe.isNothing (List.lookup sym visibles_after)
    ) visibles_before in

  let mk_kills_e =
    Caux.mk_unit_sseq (
      List.map (fun (sym,ty) ->
        Caux.pkill loc (C.Static ty) (Caux.mk_sym_pe sym)
      ) killed_syms_tys
    ) in

  let created_sym_tys =
    List.filter (fun (sym, _) ->
      not (List.any (fun (sym',_) -> sym = sym') visibles_before)
    ) visibles_after in

  let mk_creates_e =
    Caux.mk_sseqs begin
      List.map (fun (sym, ty) ->
        (C.Pattern [] (C.CaseBase (Just sym, C.BTy_object C.OTy_pointer)),
         let core_ty = Caux.mk_ail_ctype_pe ty in
         Caux.pcreate loc (Caux.mk_alignof_pe core_ty) core_ty (Symbol.PrefSource loc
          [(*f; *)sym]))
      ) created_sym_tys
    end in
  mk_kills_e (mk_creates_e (Caux.mk_run_e sym ((List.map (fun (sym, _) -> Caux.mk_sym_pe
  sym) visibles_after) ++ pes)))
```

B.14 Top-level function elaborating statements

```
open Operators

type translate_stmt_env = <|
  return_ty: Ctype.ctype;
  is_Noreturn: bool;

  variadic_sym_opt: maybe Symbol.sym;
  last_arg_sym_opt: maybe Symbol.sym;

  return_lab  : Symbol.sym;
  default_lab : maybe Symbol.sym;
  case_labs   : list (A.integerConstant * Symbol.sym);

  (* The lists are the objects visible from the loop and break labels *)
  loop:  maybe (
              Symbol.sym (* loop continuation symbol *)
            * Symbol.sym (* continue continuation symbol *)
            * Symbol.sym (* break continuation symbol *)
            * list (Symbol.sym * Ctype.ctype)
  );
```

```
2955    break: maybe (Symbol.sym * list (Symbol.sym * Ctype.ctype));
2956
2957    label_visibles: map Symbol.sym (list (Symbol.sym * Ctype.ctype));
2958  |>
2959
2960
2961  let wrapped_translate_expression is_used ctx variadic_env stdlib tagDefs e =
2962    translate_expression is_used ctx variadic_env stdlib tagDefs e >>= fun core_e ->
2963    E.return begin
2964      C.Expr [Annot.Astd "§6.5#2"] (C.Ebound core_e)
2965    end
2966
2967
2968  val translate_stmt:
2969    translation_stdlib ->
2970    C.core_tag_definitions ->
2971    A.ail_identifier ->
2972    translate_stmt_env ->
2973    A.statement GenTypes.genTypeCategory ->
2974    E.elabM (C.expr unit)
2975
2976  let rec translate_stmt stdlib tagDefs f env (A.AnnotatedStatement loc stmt_attrs stmt) :
      ↪  E.elabM (C.expr unit) =
2977    let translate_expression is_used = wrapped_translate_expression is_used ECTX_other
        ↪  (env.variadic_sym_opt, env.last_arg_sym_opt) stdlib tagDefs in
2978    let translate_cases_block case_labs default_lab =
2979      translate_stmt stdlib tagDefs f <| env with case_labs=   case_labs;
2980                                                  default_lab= default_lab |>
2981    in
2982    let self = translate_stmt stdlib tagDefs f env in
2983
2984    (Caux.add_loc loc -| Caux.add_attrs stmt_attrs) <$>
2985    match stmt with
```

B.14.1 Elaboration of empty and expression statements

```
2986      | A.AilSskip ->
2987          E.return Caux.mk_skip_e
2988
2989      | A.AilSexpr e ->
2990          translate_expression false e >>= fun core_e ->
2991          E.return (
2992            Caux.mk_sseq_e (Caux.mk_empty_pat (maybe C.BTy_unit C.BTy_loaded
              ↪  (Caux.core_object_type_of_ctype (ctype_of e))))
2993              core_e
2994              (Caux.mk_pure_e Caux.mk_unit_pe)
2995          )
```

B.14.2 Elaboration of block statements

```
3025      | A.AilSblock binds ss ->
3026          let decls_with_loc = [ (sym, ident_loc, (align_opt, qs, ty)) | forall ((sym,
              ↪  ((ident_loc, _, _), align_opt, qs, ty)) MEM binds ) | true ] in
3027          let decls = List.map (fun (sym, _, (_, qs, ty)) -> (sym, (qs, ty)))
              ↪  decls_with_loc in
3028          E.with_block_objects decls begin
3029            E.mapM self ss
```

```
3030          end >>= fun (compound_lits, core_ss) ->
3031          let lit_pats_core_creates : list (C.pattern * C.expr unit) =
3032            List.map (fun (loc, prefix, sym, is_const, ty) ->
3033              let core_ty = Caux.mk_ail_ctype_pe ty in
3034              ( Caux.mk_sym_pat sym (C.BTy_object C.OTy_pointer)
3035              , Caux.pcreate loc (Caux.mk_alignof_pe core_ty) core_ty prefix )
3036            ) compound_lits
3037          in
3038          (* the symbolic names and create actions for the local variables *)
3039          let pats_core_creates : list (C.pattern * C.expr unit) =
3040            List.map (fun (sym, ident_loc, (align_opt, qs, ty)) ->
3041              let c_ty = Caux.mk_ail_ctype_pe ty in
3042              let align_ival  =
3043              match align_opt with
3044                | Just (Ctype.AlignInteger n) ->
3045                    Caux.mk_integer_pe n
3046                | Just (Ctype.AlignType al_ty) ->
3047                    Caux.mk_alignof_pe (Caux.mk_ail_ctype_pe al_ty)
3048                | Nothing ->
3049                    Caux.mk_alignof_pe c_ty
3050              end in
3051              ( Caux.mk_sym_pat sym (C.BTy_object C.OTy_pointer)
3052              , Caux.pcreate (Loc.with_cursor_from loc ident_loc) align_ival c_ty
                  ↪  (Symbol.PrefSource ident_loc [f; sym]) )
3053            ) decls_with_loc in
3054          let pat_core_kills : list (C.expr unit) =
3055            List.map (fun (loc, _, sym, _, ty) ->
3056              Caux.pkill (Loc.with_cursor_from loc loc) (C.Static ty) (Caux.mk_sym_pe sym)
3057            ) compound_lits in
3058          (* NOTE: doing the kills here is now redundant if there is are returns before
              ↪  all exit point.
3059             but it may be nasty to do the check.
3060             For non-void function however we know (?) that their must be these returns
    ↪  ?? so we could drop the kills here *)
3061          (* the kill actions for the local variables *)
3062          let core_kills : list (C.expr unit) =
3063            List.map (fun (sym, ident_loc, (_, _, ty)) ->
3064              Caux.pkill (Loc.with_cursor_from loc ident_loc) (C.Static ty)
                  ↪  (Caux.mk_sym_pe sym)
3065            ) decls_with_loc in
3066          E.return (
3067            (* NOTE: we sequence (left-to-right) the creates and kills of the block-scoped
                ↪  objects *)
3068            Caux.mk_sseqs (lit_pats_core_creates ++ pats_core_creates) begin
3069              Caux.mk_unit_sseq (core_ss ++ core_kills ++ pat_core_kills)
3070                Caux.mk_skip_e
3071            end
3072          )
```

B.14.3 Elaboration of `if` statements

```
3074      | A.AilSif e s1 s2 ->
3075          E.wrapped_fresh_symbol C.BTy_boolean                >>= fun do_then_wrp ->
3076          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer)  >>= fun test_wrp    ->
3077          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer)  >>= fun case_wrp    ->
3078          translate_expression true (mkTestExpression TestEq e) >>= fun core_test   ->
3079          self s1                                             >>= fun core_s1     ->
3080          self s2                                             >>= fun core_s2     ->
```

```
3081          (* NOTE: here we fix the strictness of unspecified values *)
3082          E.return begin
3083            (* NOTE: the case-of producing a boolean (instead of directly doing the
                  control
3084               with the case) is to prevent possible combinatorial explosions of the
    generated Core code *)
3085            Caux.mk_sseq_e test_wrp.E.sym_pat core_test begin
3086              Caux.mk_sseq_e do_then_wrp.E.sym_pat begin
3087                Caux.mk_case_e test_wrp.E.sym_pe
3088                  [ ( Caux.mk_specified_pat case_wrp.E.sym_pat
3089                    , Caux.mk_pure_e begin
3090                        Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_not_pe (Caux.mk_op_pe
                              C.OpEq case_wrp.E.sym_pe (Caux.mk_integer_pe 1)))
3091                          (Caux.mk_boolean_pe true) (Caux.mk_boolean_pe false)
3092                      end )
3093                    (* non-deterministic branching if the test expression had unspecified
                          value *)
3094                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
3095                    , Caux.mk_nd_e [Caux.mk_pure_e (Caux.mk_boolean_pe true);
                          Caux.mk_pure_e (Caux.mk_boolean_pe false)] ) ]
3096              end
3097              (Caux.mk_if_e do_then_wrp.E.sym_pe core_s1 core_s2)
3098            end
3099          end
```

B.14.4 Elaboration of `while` statements

```
3101      | A.AilSwhile e s loop_id ->
3102          (* NOTE: the object type is OTy_integer since we are using mkTestExpression
                which turns [e] into [e == 0] *)
3103          let sym_loop = Symbol.fresh_pretty_with_id (fun x ->  "while_" ^ show x) in
3104          let sym_loop_body = Symbol.fresh_pretty_with_id (fun x ->  "while_body_" ^ show
                x) in
3105          E.wrapped_fresh_symbol C.BTy_boolean                  >>= fun do_loop_wrp  ->
3106          E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer)   >>= fun test_wrp     ->
3107          E.wrapped_fresh_symbol (C.BTy_object C.OTy_integer)   >>= fun case_wrp     ->
3108          translate_expression true (mkTestExpression TestEq e) >>= fun core_test    ->
3109          self s                                                >>= fun core_s       ->
3110          E.get_visible_objects                                 >>= fun visible_syms ->
3111          (* TODO: the types of the annotations in Esave are dummy, but this is not
                observable for now *)
3112          let core_s_loop =
3113            Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit) core_s
3114              (Caux.mk_run_e sym_loop [ Caux.mk_sym_pe sym | forall (sym MEM visible_syms)
                    | true ]) in
3115          E.mapM (fun sym ->
3116            E.resolve_object_type sym >>= fun (_, ty) ->
3117            E.return (sym, ((C.BTy_object C.OTy_pointer, Just (ty, true)), Caux.mk_sym_pe
                  sym))
3118          ) visible_syms >>= fun args ->
3119          E.return begin
3120            Caux.mk_save_e_ [Annot.Alabel (Annot.LAloop_prebody loop_id)] (sym_loop,
                  C.BTy_unit) args begin
3121              Caux.mk_sseq_e test_wrp.E.sym_pat core_test begin
3122                Caux.mk_sseq_e do_loop_wrp.E.sym_pat begin
3123                  Caux.mk_case_e test_wrp.E.sym_pe
3124                    [ ( Caux.mk_specified_pat case_wrp.E.sym_pat
3125                      , Caux.mk_pure_e begin
```

```
3126                        Caux.mk_if_pe_ [Annot.Anot_explode] (Caux.mk_not_pe
                            ↪  (Caux.mk_op_pe C.OpEq case_wrp.E.sym_pe (Caux.mk_integer_pe
                            ↪  1)))
3127                          (Caux.mk_boolean_pe true) (Caux.mk_boolean_pe false)
3128                      end )
3129                    (* non-deterministic branching if the test expression had
                        ↪  unspecified value *)
3130                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
3131                    , Caux.mk_nd_e [Caux.mk_pure_e (Caux.mk_boolean_pe true);
                        ↪  Caux.mk_pure_e (Caux.mk_boolean_pe false)] ) ]
3132              end
3133                (Caux.mk_if_e do_loop_wrp.E.sym_pe
3134                   begin
3135                     Caux.mk_save_e_ [Annot.Alabel (Annot.LAloop_body loop_id)]
                        ↪  (sym_loop_body, C.BTy_unit) args
3136                       core_s_loop
3137                   end
3138                   Caux.mk_skip_e)
3139            end
3140          end
3141        end
```

B.14.5 Elaboration of do statements

```
3143      | A.AilSdo s e loop_id ->
3144          (* TODO: make the elab of AilSdo use mkTestExpression *)
3145          let sym_loop = Symbol.fresh_pretty_with_id (fun x ->  "do_" ^ show x) in
3146          let sym_case = Symbol.fresh () in
3147          let sym_e    = Symbol.fresh () in
3148          translate_expression true e >>= fun core_e       ->
3149          self s                      >>= fun core_s       ->
3150          E.get_visible_objects       >>= fun visible_syms ->
3151
3152          let core_loop =
3153            Caux.mk_run_e sym_loop [ Caux.mk_sym_pe sym | forall (sym MEM visible_syms) |
                ↪  true ] in
3154
3155          E.mapM (fun sym ->
3156              E.resolve_object_type sym >>= fun (_, ty) ->
3157              E.return (sym, ((C.BTy_object C.OTy_pointer, Just (ty, true)),
                    ↪  Caux.mk_sym_pe sym))
3158            ) visible_syms >>= fun args ->
3159          E.return begin
3160            Caux.mk_save_e_ [Annot.Alabel (Annot.LAloop_body loop_id)] (sym_loop,
                ↪  C.BTy_unit) args (
3161              (* loop body *)
3162              Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit) core_s
3163                (* controlling expression *)
3164                begin
3165                  Caux.mk_sseq_e (Caux.mk_sym_pat sym_e (C.BTy_loaded C.OTy_integer))
                      ↪  core_e (
3166                    Caux.mk_case_e (Caux.mk_sym_pe sym_e)
3167                      [ ( Caux.mk_specified_pat (Caux.mk_sym_pat sym_case (C.BTy_object
                          ↪  C.OTy_integer))
3168                        , Caux.mk_if_e (Caux.mk_not_pe (Caux.mk_op_pe C.OpEq
                          ↪  (Caux.mk_sym_pe sym_case) (Caux.mk_integer_pe 0)))
3169                            core_loop Caux.mk_skip_e )
```

```
                    (* non-deterministic branching if the test expression had
                    ↪  unspecified value *)
                  ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
                    , (* Caux.mk_nd_e [core_loop; Caux.mk_skip_e] *)
                      Caux.mk_pure_e (Caux.mk_undef_pe loc (Undefined.DUMMY
                      ↪  "unspecified AilSdo")) ) ]
                )
            end
        )
    end
```

B.14.6 Elaboration of `return` statements

```
  | A.AilSreturnVoid ->
      E.get_visible_objects >>= fun visible_syms ->
      E.mapM (fun sym ->
          E.resolve_object_type sym >>= fun (_, ty) ->
          E.return (sym, ty)
        ) visible_syms >>= fun visible_syms_tys ->
      E.return begin
        if env.is_Noreturn then
          Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.7.4#8"
          ↪  Undefined.UB071_noreturn)
        else
          let bTy =
            if List.length visible_syms < 2 then
              C.BTy_unit
            else
              C.BTy_tuple (List.replicate (List.length visible_syms) C.BTy_unit) in
          Caux.mk_sseq_e (Caux.mk_empty_pat bTy)
            (Caux.mk_unseq (List.map (fun (sym,ty) -> Caux.pkill loc (C.Static ty)
            ↪  (Caux.mk_sym_pe sym)) visible_syms_tys))
            (Caux.mk_run_e env.return_lab [Caux.mk_unit_pe])
      end

  | A.AilSreturn e ->
      E.get_visible_objects >>= fun visible_syms ->
      E.mapM (fun sym ->
          E.resolve_object_type sym >>= fun (_, ty) ->
          E.return (sym, ty)
      ) visible_syms >>= fun visible_syms_tys ->
      if env.is_Noreturn then
        E.return (Caux.mk_pure_e (Caux.mk_std_undef_pe loc "§6.7.4#8"
        ↪  Undefined.UB071_noreturn))
      else if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_pointer
      ↪  env.return_ty && Aaux.is_null_pointer_constant e then
        E.return begin
          mk_run_with_lifetime_e loc env.return_lab visible_syms_tys []
          ↪  [Caux.mk_specified_pe (Caux.mk_nullptr_pe env.return_ty)]
        end
      else
        let oTy = force_core_object_type_of_ctype (ctype_of e) in
        translate_expression true e                >>= fun core_e ->
        E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp  ->
      (* All the visible objects from the current function need to be killed. *)
      if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_Bool env.return_ty &&
      ↪  AilTypesAux.is_pointer (ctype_of e) then
          (* By Ail's typing we know that [e] is a pointer *)
```

```
3226            E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun conv_wrp ->
3227            E.return begin
3228              Caux.mk_sseq_e e_wrp.E.sym_pat core_e begin
3229                Caux.mk_sseq_e conv_wrp.E.sym_pat (stdlib.mkproc_loaded_pointer_to_Bool
                    ↪  e_wrp.E.sym_pe)
3230                (mk_run_with_lifetime_e loc env.return_lab visible_syms_tys []
                    ↪  [conv_wrp.E.sym_pe])
3231              end
3232            end
3233          else
3234            let ret_pe =
3235              if AilTypesAux.atomic_qualified_unqualified AilTypesAux.is_arithmetic
                  ↪  env.return_ty && AilTypesAux.is_arithmetic (ctype_of e) then
3236                conv_loaded_arith stdlib (ctype_of e) env.return_ty e_wrp.E.sym_pe
3237              else
3238                e_wrp.E.sym_pe in
3239            E.return begin
3240              Caux.mk_sseq_e e_wrp.E.sym_pat core_e begin
3241                mk_run_with_lifetime_e loc env.return_lab visible_syms_tys [] [ret_pe]
3242              end
3243            end
```

B.14.7 Elaboration of `switch` statements

```
3245      | A.AilSswitch e s ->
3246          (* Translate the controlling expression *)
3247          translate_expression true e >>= fun core_e ->
3248          let oTy = force_core_object_type_of_ctype (ctype_of e) in
3249          E.wrapped_fresh_symbol (C.BTy_loaded oTy) >>= fun e_wrp ->
3250          (* Case in specified values *)
3251          E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun control_wrp ->
3252          (* Integer promotion *)
3253          (* STD §6.8.4.2#5, sentence 1 *)
3254          E.wrapped_fresh_symbol (C.BTy_object oTy) >>= fun promoted_wrp ->
3255          let promoted_pe =
3256            Caux.mk_std_pe "§6.8.4.2#5, sentence 1" (
3257              stdlib.mkcall_conv_int (fromJust "Translation.translate_stmt: switch expr
                    ↪  promotion"
3258                (AilTypesAux.promotion integerImpl (ctype_of e))) control_wrp.E.sym_pe
3259            ) in
3260          (* Create case symbols and translate cases block *)
3261          let cases    = collect_cases s in
3262          let nb_cases = List.length cases.case_csts in
3263          let case_syms = list_init nb_cases (fun _ -> Symbol.fresh_pretty_with_id (fun x
              ↪  -> "case_" ^ show x)) in
3264          let default_sym = Symbol.fresh_pretty_with_id (fun x -> "default_" ^ show x) in
3265          let case_labs = List.zip cases.case_csts case_syms in
3266          translate_cases_block case_labs (Just default_sym) s >>= fun core_s ->
3267          E.get_visible_objects >>= fun visible_syms ->
3268          let visible_pes = List.map (fun sym -> Caux.mk_sym_pe sym) visible_syms in
3269          (* Build translated switch *)
3270          E.return
3271            (* Get control expression *)
3272            (Caux.mk_sseq_e e_wrp.E.sym_pat core_e
3273              (* Check if unspecified *)
3274              (Caux.mk_case_e e_wrp.E.sym_pe
3275                [ ( Caux.mk_specified_pat control_wrp.E.sym_pat
3276                  , (* Do integer promotion *)
```

```
3277                  Caux.mk_sseq_e promoted_wrp.E.sym_pat
3278                    (Caux.mk_pure_e promoted_pe)
3279                    (* For every case.... *)
3280                    (List.foldl (fun acc (iCst, case_lab) ->
3281                      Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit)
3282                        (* Check if equal to the constant expression *)
3283                        (Caux.mk_if_e (Caux.mk_op_pe C.OpEq promoted_wrp.E.sym_pe
                            ↪  (translate_integerConstant iCst))
3284                          (* TODO: not sure if the visible afters are the same *)
3285                          (Caux.mk_run_e case_lab visible_pes)
3286                          (Caux.mk_skip_e))
3287                        acc)
3288                    (* default branch *)
3289                    (Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit)
3290                      (if cases.found_default then
3291                        Caux.mk_run_e default_sym visible_pes
3292                        else
3293                        Caux.mk_skip_e)
3294                      core_s)
3295                    case_labs) )
3296                (* UB if unspecified *)
3297              ; ( Caux.mk_unspecified_pat (Caux.mk_empty_pat C.BTy_ctype)
3298                , Caux.mk_pure_e (Caux.mk_undef_exceptional_condition loc) ) ]))
3299
3300    | A.AilScase iCst s ->
3301        self s                >>= fun core_s       ->
3302        E.get_visible_objects >>= fun visible_syms ->
3303        E.mapM (fun sym ->
3304            E.resolve_object_type sym >>= fun (_, ty) ->
3305            E.return (sym, ((C.BTy_object C.OTy_pointer, Just (ty, true)),
            ↪  Caux.mk_sym_pe sym))
3306          ) visible_syms >>= fun visible_pes ->
3307        match List.lookup iCst env.case_labs with
3308          | Just lab ->
3309              E.return (Caux.mk_save_e (lab, C.BTy_unit) visible_pes core_s)
3310          | (*BISECT-IGNORE*) Nothing ->
3311              error "Translation.translate_stmt: case label not found."
3312        end
3313
3314    | A.AilSdefault s ->
3315        self s >>= fun core_s ->
3316        match env.default_lab with
3317          | Just lab ->
3318              E.return (Caux.mk_save_e (lab, C.BTy_unit) [] core_s)
3319          | (*BISECT-IGNORE*) Nothing ->
3320              error "Translation.translate_stmt: default label not found."
3321        end
```

B.14.8 Elaboration of label and `goto` statements

```
3323    | A.AilSlabel sym s m_label_annot ->
3324        self s                >>= fun core_s       ->
3325        E.get_visible_objects >>= fun visible_syms ->
3326        E.mapM (fun sym ->
3327            E.resolve_object_type sym >>= fun (_, ty) ->
3328            E.return (sym, ((C.BTy_object C.OTy_pointer, Just (ty, true)),
            ↪  Caux.mk_sym_pe sym))
3329          ) visible_syms >>= fun args ->
```

```
3330          let annots = match m_label_annot with
3331            | Just loop_annot -> [Annot.Alabel loop_annot]
3332            | Nothing -> []
3333          end in
3334          E.return (Caux.mk_save_e_ annots (sym, C.BTy_unit) args core_s)
3335
3336      | A.AilSgoto sym ->
3337          E.get_visible_objects >>= fun visible_syms ->
3338          E.mapM (fun sym ->
3339              E.resolve_object_type sym >>= fun (_, ty) ->
3340              E.return (sym, ty)
3341          ) visible_syms >>= fun visible_syms_tys ->
3342          let visibles_after = fromJust "Translation.translation_statement, AilSgoto"
             ↪  (Map.lookup sym env.label_visibles) in
3343          E.return (mk_run_with_lifetime_e loc sym visible_syms_tys visibles_after [])
```

B.14.9 Elaboration of declaration statements

```
3345      | A.AilSdeclaration sym_es ->
3346          (* This pass translates the declarations *)
3347          E.mapM (fun (ptr_sym, e) ->
3348            E.resolve_object_type ptr_sym >>= fun (qs, ty) ->
3349            let e_bTy = C.BTy_loaded (force_core_object_type_of_ctype (ctype_of e)) in
3350            E.wrapped_fresh_symbol e_bTy >>= fun e_wrp ->
3351            match Ctype.unatomic_ ty with
3352              | Ctype.Pointer _ ref_ty ->
3353                  let mk_store =
3354                    if qs.Ctype.const then Caux.pstore_lock else Caux.pstore in
3355                  if Aaux.is_null_pointer_constant e then
3356                    E.return (
3357                      mk_store loc (Caux.mk_ail_ctype_pe ty) (Caux.mk_sym_pe ptr_sym)
3358                        (Caux.mk_specified_pe (Caux.mk_nullptr_pe ref_ty)) Cmm.NA
3359                    )
3360                  else
3361                    translate_expression true e >>= fun core_e ->
3362                    E.return (
3363                      Caux.mk_sseq_e e_wrp.E.sym_pat core_e
3364                        (mk_store loc (Caux.mk_ail_ctype_pe ty) (Caux.mk_sym_pe ptr_sym)
                           ↪  e_wrp.E.sym_pe Cmm.NA)
3365                    )
3366              | ty ->
3367                  let mk_store =
3368                    if qs.Ctype.const then Caux.pstore_lock else Caux.pstore in
3369                  let cty = Ctype.Ctype [] ty in
3370                  translate_expression true e >>= fun core_e ->
3371                  if AilTypesAux.is_pointer (ctype_of e) then
3372                    (* we are dealing with the case {_Bool} = {pointer} *)
3373                    E.wrapped_fresh_symbol (C.BTy_loaded C.OTy_integer) >>= fun conv_wrp
                     ↪  ->
3374                    E.return begin
3375                      Caux.mk_sseq_e e_wrp.E.sym_pat core_e begin
3376                        Caux.mk_sseq_e conv_wrp.E.sym_pat
                           ↪  (stdlib.mkproc_loaded_pointer_to_Bool e_wrp.E.sym_pe)
3377                          (mk_store loc (Caux.mk_ail_ctype_pe cty) (Caux.mk_sym_pe
                             ↪  ptr_sym) conv_wrp.E.sym_pe Cmm.NA)
3378                      end
3379                    end
3380                  else
```

```
                  (* we are not dealing with a pointer on either sides *)
                  E.return (
                    Caux.mk_sseq_e e_wrp.E.sym_pat core_e
                      (mk_store loc (Caux.mk_ail_ctype_pe cty) (Caux.mk_sym_pe ptr_sym)
                      (if AilTypesAux.is_arithmetic cty then conv_loaded_arith stdlib
                      ↪  (ctype_of e) cty e_wrp.E.sym_pe else e_wrp.E.sym_pe) Cmm.NA)
                  )
            end
          ) sym_es >>= fun z ->

          (* This pass combines the translated declarations *)
          match z with
            | [] ->
                E.return Caux.mk_skip_e
            | z::zs' ->
                E.foldlM (fun x y -> E.return (Caux.concat_sseq x y)) z zs'
          end
```

B.15 Top-level function elaborating Ail programs

```
val translate_tag_definitions:
  list (A.ail_identifier * (Annot.attributes * Ctype.tag_definition)) ->
  C.core_tag_definitions
let translate_tag_definitions ctx =
  Map.fromList (List.map (fun (x, (_, y)) -> (x, y)) ctx)

import Cerb_attributes

val translate_program:
  translation_stdlib ->
  maybe Symbol.sym * A.sigma GenTypes.genTypeCategory ->
  E.elabM (  C.core_tag_definitions
           * list (Symbol.sym * (C.generic_globs unit unit))
           * C.generic_fun_map unit unit
           * map Symbol.sym (Loc.t * Annot.attributes * Ctype.ctype * list (maybe
           ↪  Symbol.sym * Ctype.ctype) * bool * bool) )

let translate_program stdlib (startup_sym_opt, sigm) =
  let core_tagDefs = translate_tag_definitions sigm.A.tag_definitions in

  E.foldlM (fun (gacc, facc, finfoacc) (sym, (loc, decl_attrs, decl)) ->
    (* for each Ail declaration *)
    match decl with
```

B.15.1 Elaboration of global objects

```
      | A.Decl_object _ align qs ty ->
          (* elaboration of a global variables *)
          let core_ty     = Caux.mk_ail_ctype_pe ty in
          let align_ival  =
            match align with
              | Just (Ctype.AlignInteger n) ->
                  Caux.mk_integer_pe n
              | Just (Ctype.AlignType al_ty) ->
                  Caux.mk_alignof_pe (Caux.mk_ail_ctype_pe al_ty)
              | Nothing ->
```

```
3527                Caux.mk_alignof_pe (Caux.mk_ail_ctype_pe ty)
3528            end in
3529          let core_create =
3530            match Cerb_attributes.decode_with_address decl_attrs with
3531              | Right (Just addr) ->
3532                  Caux.add_annot (Annot.Acerb (Annot.ACerb_with_address addr))
3533              | _ ->
3534                  (fun z -> z)
3535            end (Caux.pcreate loc align_ival core_ty (Symbol.PrefSource loc [sym])) in
3536          let sym_global  = Symbol.fresh () in
3537
3538          match List.lookup sym sigm.A.object_definitions with
3539            | Nothing ->
3540                (* we are dealing with an external object *)
3541                E.return ( (sym, C.GlobalDecl (C.BTy_object C.OTy_pointer, ty)) :: gacc,
                    ↪  facc, finfoacc )
3542
3543            | Just expr ->
3544                begin
3545                  if AilTypesAux.is_pointer ty && Aaux.is_null_pointer_constant expr
                      ↪  then
3546                    E.return (
3547                      C.BTy_loaded C.OTy_pointer,
3548                      Caux.mk_pure_e (Caux.mk_specified_pe (Caux.mk_nullptr_pe ty))
3549                    )
3550                  else
3551                    (* NOTE: we use `with_block_objects`, for the compound_literal
                        ↪  tracking *)
3552                    E.with_block_objects [] begin
3553                      wrapped_translate_expression true (ECTX_glob sym sym_global)
                          ↪  ((Nothing: maybe Symbol.sym), (Nothing: maybe Symbol.sym))
                          ↪  stdlib core_tagDefs expr
3554                    end >>= fun (compound_lits, core) ->
3555                    let lit_pats_core_creates : list (C.pattern * C.expr unit) =
3556                      List.map (fun (loc, prefix, sym, is_const, ty) ->
3557                        let core_ty = Caux.mk_ail_ctype_pe ty in
3558                        ( Caux.mk_sym_pat sym (C.BTy_object C.OTy_pointer)
3559                        , Caux.pcreate loc (Caux.mk_alignof_pe core_ty) core_ty prefix )
3560                      ) compound_lits in
3561                    E.return (
3562                      C.BTy_loaded (force_core_object_type_of_ctype (ctype_of expr)),
3563                      Caux.mk_sseqs lit_pats_core_creates core
3564                    )
3565                end >>= fun (e_bTy, core_e) ->
3566                let e_sym = Symbol.fresh () in
3567                let core_init_e =
3568                  if AilTypesAux.is_integer ty || AilTypesAux.is_floating ty then
3569                    conv_loaded_arith stdlib (ctype_of expr) ty (Caux.mk_sym_pe e_sym)
3570                  else
3571                    Caux.mk_sym_pe e_sym in
3572                let mk_store =
3573                  (* const-qualified globals are made read-only *)
3574                  if qs.Ctype.const then Caux.pstore_lock else Caux.pstore in
3575                let core_e =
3576                  Caux.add_loc (locOf core_e) begin
3577                    Caux.mk_sseq_e (Caux.mk_sym_pat sym_global (C.BTy_object
                        ↪  C.OTy_pointer)) core_create (
3578                      Caux.mk_sseq_e (Caux.mk_sym_pat e_sym e_bTy) core_e (
```

```
3579                      Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit)
3580                        (* TODO: proper memory order *)
3581                        (mk_store loc core_ty (Caux.mk_sym_pe sym_global) core_init_e
                            ↪  Cmm.NA)
3582                        (Caux.mk_pure_e (Caux.mk_sym_pe sym_global))
3583                    ))
3584                end in
3585              E.return
3586                ( (sym, C.GlobalDef (C.BTy_object C.OTy_pointer, ty) core_e) :: gacc
3587                , facc, finfoacc )
3588        end
```

B.15.2 Elaboration of function definitions

```
3590      | A.Decl_function has_proto (_, return_ty) params is_variadic is_inline
          ↪  is_Noreturn ->
3591          let is_using_inner_arg_temps =
3592            (* NOTE: we exclude main because the driver allocates the objects for argc
                ↪  and argvs *)
3593            (* with this switch the argument temporary objects are allocated in the
                ↪  function *)
3594            Global.has_switch SW_inner_arg_temps && startup_sym_opt <> Just sym in
3595          (* elaboration of a function *)
3596          let ret_bTy =
3597            if AilTypesAux.is_void return_ty then
3598              C.BTy_unit
3599            else
3600              C.BTy_loaded (force_core_object_type_of_ctype return_ty) in
3601          let param_bTys =
3602            if is_using_inner_arg_temps then
3603              List.map (fun (_, ty, _) -> C.BTy_loaded (force_core_object_type_of_ctype
                  ↪  ty)) params
3604            else
3605              List.replicate (List.length params) (C.BTy_object C.OTy_pointer) in
3606          match List.lookup sym sigm.A.function_definitions with
3607            | Nothing ->
3608                (* if the function has no definition, we create a Core procedure
                    ↪  declaration *)
3609                let finfo =     (* TODO: plug in non-empty attributes *)
3610                  ( loc, decl_attrs, return_ty
3611                    (* TODO: check if we need qualifiers too *)
3612                  , List.map (fun (_, ty, _) -> (Nothing, ty)) params
3613                  , is_variadic, has_proto ) in
3614                E.return
3615                  ( gacc
3616                  , Map.insert sym (C.ProcDecl loc ret_bTy param_bTys) facc
3617                    (* get the correct symbol if a proxy exists *)
3618                  , match sym with
3619                      | Symbol.Symbol _ _ (Symbol.SD_Id str) ->
3620                          match Map.lookup str stdlib.ailnames with
3621                            | Just sym_proxy -> Map.insert sym_proxy finfo finfoacc
3622                            | Nothing -> Map.insert sym finfo finfoacc
3623                          end
3624                      | _ -> Map.insert sym finfo finfoacc
3625                    end )
3626
3627            | Just (loc, _, param_syms, stmt) ->
3628                E.mapM (fun (s, bTy) ->
```

```
3629                let descr =
3630                  if not (Global.has_switch SW_inner_arg_temps) then Symbol.SD_None
                      ↪  else
3631                    match Symbol.symbol_description s with
3632                    | Symbol.SD_ObjectAddress v -> Symbol.SD_FunArgValue v
3633                    | _ -> Assert_extra.failwith "function argument does not have
                        ↪  SD_ObjectAddress description"
3634                    end
3635                in
3636                E.wrapped_fresh_symbol_ descr bTy >>= fun (sym, _, sym_pe) ->
3637                E.return (sym, sym_pe)
3638              ) (List.zip param_syms param_bTys) >>= fun arg_value_decls ->
3639              let finfo =
3640                (* NOTE: the attributes in the Ail function declarations includes the
                    ↪  ones only present on both
3641                   the Cabs function declaration and definition *)
3642                ( loc, decl_attrs, return_ty
3643                (* TODO: check if we need qualifiers too *)
3644                , List.map (fun (sym, (_, ty, _)) ->
3645                      if Global.has_switch SW_inner_arg_temps then
3646                        let descr = match Symbol.symbol_description sym with
3647                          | Symbol.SD_ObjectAddress v -> Symbol.SD_FunArgValue v
3648                          | _ -> Assert_extra.failwith "function argument does not
                              ↪  have SD_ObjectAddress description"
3649                        end in
3650                        (Just (Symbol.set_symbol_description sym descr), ty)
3651                      else
3652                        (Just sym, ty)
3653                    ) (List.zip param_syms params)
3654                , is_variadic, has_proto ) in
3655              let ret_label = Symbol.fresh_pretty_with_id (fun x -> "ret_" ^ show x)
                  ↪  in
3656              let stmt = erase_loop_control stmt in
3657              let visibles = collect_visibles stmt in
3658              let (variadic_sym_opt, last_arg_sym_opt) =
3659                if is_variadic then
3660                  match List.reverse param_syms with
3661                    | [] -> error "variadic functions need to have at least one
                        ↪  parameter"
3662                    | sym::_ -> (Just (Symbol.fresh ()), Just sym)
3663                  end
3664                else
3665                  (Nothing, Nothing)
3666              in
3667              let (mk_body_wrapper, label_visibles) =
3668                if is_using_inner_arg_temps then
3669                  let xs = List.map (fun (sym, (_, ty, _)) -> (sym, ty)) (List.zip
                      ↪  param_syms params) in
3670                  ( (fun z -> E.with_block_objects (List.map (fun (sym, ty) -> (sym,
                      ↪  (Ctype.no_qualifiers, ty))) xs) z >>= fun (_, ret) -> E.return
                      ↪  ret)
3671                  , Map.map (fun z -> z ++ xs) visibles.label_visibles_ )
3672                else
3673                  ((fun z -> z), visibles.label_visibles_) in
3674              mk_body_wrapper
3675                (translate_stmt stdlib core_tagDefs sym <|
3676                  return_ty= return_ty;
3677                  return_lab= ret_label;
```

```
3678                    variadic_sym_opt= variadic_sym_opt;
3679                    last_arg_sym_opt= last_arg_sym_opt;
3680                    is_Noreturn= is_Noreturn;
3681                    default_lab= Nothing;
3682                    case_labs= [];
3683                    loop= Nothing;
3684                    break= Nothing;
3685                    label_visibles= label_visibles;
3686                  |> stmt) >>= fun core_body ->
3687                let add_prelude_and_epilogue z =
3688                  if is_using_inner_arg_temps then
3689                    Caux.mk_sseqs
3690                      begin
3691                        List.concat begin
3692                          List.mapi (fun i ((_, ty, _), (ptr_sym, (_, value_sym_pe))) ->
3693                            let ty_pe = Caux.mk_ctype_pe ty in
3694                            [ ( Caux.mk_sym_pat ptr_sym (C.BTy_object C.OTy_pointer)
3695                              , Caux.pcreate loc (Caux.mk_alignof_pe ty_pe) ty_pe
                                ↪  (Symbol.PrefFunArg loc (Symbol.digest ()) (intFromNat
                                ↪  i)) )
3696                            ; ( Caux.mk_empty_pat C.BTy_unit
3697                              , Caux.pstore loc ty_pe (Caux.mk_sym_pe ptr_sym)
                                ↪  value_sym_pe Cmm.NA ) ]
3698                          ) (List.zip params (List.zip param_syms arg_value_decls))
3699                        end
3700                      end
3701                      begin
3702                        Caux.mk_unit_sseq
3703                          begin
3704                            z ::
3705                            List.map (fun (ptr_sym, (_, ty, _)) ->
3706                              Caux.pkill loc (C.Static ty) (Caux.mk_sym_pe ptr_sym)
3707                            ) (List.zip param_syms params)
3708                          end
3709                          (Caux.mk_pure_e Caux.mk_unit_pe)
3710                      end
3711                  else
3712                    z in
3713                (* let is_used_sym = Symbol.fresh () in
3714                let is_used_arg_type = (is_used_sym, C.BTy_boolean) in *)
3715                let ret_sym = Symbol.fresh () in
3716                let core_return =
3717                  let ret_pe =
3718                    if startup_sym_opt = Just sym then
3719                      (* STD §5.1.2.2.3#1 sentence 1*)
3720                      Caux.mk_specified_pe (Caux.mk_integer_pe 0)
3721                    else if AilTypesAux.is_void return_ty && is_Noreturn then
3722                      Caux.mk_std_undef_pe loc "§6.7.4#8" Undefined.UB071_noreturn
3723                    else if AilTypesAux.is_void return_ty then
3724                      Caux.mk_unit_pe
3725                    else
3726                      (* Caux.mk_if_pe (Caux.mk_sym_pe is_used_sym)
3727                        (Caux.mk_std_undef_pe loc "§6.9.1#12"
    ↪  Undefined.UB088_reached_end_of_function)
3728                        (Caux.mk_unspecified_pe return_ty) in *)
3729                      Caux.mk_std_undef_pe loc "§6.9.1#12"
                        ↪  Undefined.UB088_reached_end_of_function in
3730                  Caux.mk_save_e_ [Annot.Alabel Annot.LAreturn]
```

```
                                    (ret_label, ret_bTy)
                                    [(ret_sym, ((ret_bTy, Just (return_ty, false)),
                                    ↪  ret_pe))]
                    (Caux.mk_pure_e (Caux.mk_sym_pe ret_sym))
                in
                let variadic_arg_type =
                  match variadic_sym_opt with
                    | Just sym -> [(sym, C.BTy_list (C.BTy_tuple [C.BTy_ctype;
                    ↪  C.BTy_object C.OTy_pointer]))]
                    | Nothing -> []
                  end
                in
                let param_syms =
                  if is_using_inner_arg_temps then
                    List.map fst arg_value_decls
                  else
                    param_syms in
                E.return
                  ( gacc
                  , Map.insert sym
                     (C.Proc loc ret_bTy
                             ((*is_used_arg_type :: *)List.zip param_syms param_bTys ++
                             ↪  variadic_arg_type)
                             (Caux.mk_sseq_e (Caux.mk_empty_pat C.BTy_unit)
                             ↪  (add_prelude_and_epilogue core_body) core_return)
                     ) facc
                  , Map.insert sym finfo finfoacc )
            end
```

B.15.3 Final construction of the Core program

```
    end
  ) ([], Map.empty, Map.empty) (List.reverse sigm.A.declarations) >>= fun (globs, cfuns,
  ↪  funinfo) ->

  (* adding string literals as Core globals *)
  E.get_string_literals >>= fun xs ->
  E.foldlM (fun acc (loc, sym, ty, e_init) ->
    let expr =
      Caux.pcreate_readonly loc
        (Caux.mk_alignof_pe (Caux.mk_ctype_pe ty))
        (Caux.mk_ctype_pe ty)
        e_init
        (Symbol.PrefStringLiteral loc (Symbol.digest_of_sym sym)) in
    E.return ((sym, C.GlobalDef (C.BTy_object C.OTy_pointer, ty) expr) :: acc)
  ) globs (List.reverse xs) >>= fun globs' ->
  E.return (core_tagDefs, globs', cfuns, funinfo)


let translate_extern_map (_, sigm) =
  Map.map (fun (sym, kind) ->
    match kind with
      | A.IK_declaration -> ([sym], C.LK_none)
      | A.IK_tentative   -> ([sym], C.LK_tentative sym)
      | A.IK_definition  -> ([sym], C.LK_normal sym)
    end) sigm.A.extern_idmap
```

```
3781 (* This is the entry function (called from main.ml) *)
3782 val translate:
3783     (map string Symbol.sym) * C.fun_map unit ->
3784     C.impl ->
3785     A.ail_program GenTypes.genTypeCategory ->
3786     C.file unit
3787 let translate (ailnames, stdlib_fun_map) impl prog =
3788   let translation_stdlib = mk_translation_stdlib (ailnames, stdlib_fun_map) in
3789   let ((core_tagDefs, cglobs, (*cdecls, *) cfuns, funinfo), st) =
3790     E.runStateM (translate_program translation_stdlib prog) (E.elab_init ())
3791   in
3792   <| C.main= fst prog;
3793      C.tagDefs= core_tagDefs;
3794      C.stdlib= stdlib_fun_map;
3795      C.impl= impl;
3796      C.globs= Core_linking.merge_globs cglobs [] []; (* topological sort *)
3797      C.funs= cfuns;
3798      C.extern = translate_extern_map prog;
3799      C.funinfo= funinfo;
3800      C.loop_attributes= (snd prog).A.loop_attributes; |>
```

Bibliography

[And94] Lars Ole Anderson. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, 1994.

[ANSICrationale] *Rationale for American National Standard for Information Systems – Programming Language – C*. Tech. rep.

[BA08] Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ Concurrency Memory Model". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 68–78. ISBN: 9781595938602. DOI: 10.1145/1375581.1375591.

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ Concurrency". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 55–66. ISBN: 9781450304900. DOI: 10.1145/1926385.1926394.

[Bat+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. "Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 509–520. ISBN: 9781450310833. DOI: 10.1145/2103656.2103717.

[BBW14] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values". In: *Programming Languages and Systems*. Ed. by Jacques Garrigue. Springer International Publishing, 2014, pp. 449–468. ISBN: 978-3-319-12736-1. DOI: 10.1007/978-3-319-12736-1_24.

[BBW15] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Concrete Memory Model for CompCert". In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Springer International Publishing, 2015, pp. 67–83. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_5.

[BDW16a] Mark Batty, Alastair F. Donaldson, and John Wickerson. "Overhauling SC Atomics in C11 and OpenCL". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, 634=648. ISBN: 9781450335492. DOI: 10.1145/2837614.2837637.

[BDW16b] Mark Batty, Alastair F. Donaldson, and John Wickerson. "Overhauling SC Atomics in C11 and OpenCL". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 634–648. ISBN: 9781450335492. DOI: 10.1145/2837614.2837637.

[BL09] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288. DOI: 10.1007/s10817-009-9148-3.

[Bof98] Mark Bofinger. "Reasoning about C programs". PhD thesis. University of Queensland, 1998.

[BW96] Paul E. Black and Phillip J. Windley. "Inference Rules for Programming Languages with Side Effects in Expressions". In: *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 51–60. ISBN: 3540615873. URL: https://hissa.nist.gov/~black/Papers/hol96.pdf.

[BW98] Paul E. Black and Phillip J. Windley. "Formal Verification of Secure Programs in the Presence of Side Effects". In: *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*. Vol. 3. HICSS '98. IEEE Computer Society, 1998, pp. 327–. ISBN: 0-8186-8239-6. DOI: 10.1109/HICSS.1998.656295. URL: https://hissa.nist.gov/~black/Papers/hicss31.pdf.

[Cam12] Brian Campbell. "An Executable Semantics for CompCert C". In: *Certified Programs and Proofs*. Ed. by Chris Hawblitzel and Dale Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 60–75. ISBN: 978-3-642-35308-6.

[Chi+15] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 117–130. ISBN: 9781450328357. DOI: 10.1145/2694344.2694367.

[cmom0006] Kayvan Memarian and Peter Sewell. *Clarifying uninitialised reads v5 - working draft*. Mar. 2021. URL: https://github.com/C-memory-object-model-study-group/c-mom-sg/blob/master/notes/cmom-0006-2021-03-08-clarifying-uninitialised-reads-v5.md.

[CS94] Jeffrey V. Cook and Sakthi Subramanian. *A Formal Semantics for C in Nqthm*. Tech. rep. 517D. http://web.archive.org/web/19971120123425/http://www.tis.com/docs/research/assurance/ps/nqsem.ps. 10: Trusted Information Systems, 1994.

[Dav+19] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450362405. DOI: 10.1145/3297858.3304042.

[DK21] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Comput. Surv.* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952.

[DR260] WG14. *Defect Report 260*. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm. Sept. 2004.

[DR451] WG14. *Defect Report 451*. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_451.htm. Aug. 2013.

[Ell12] Chucky Ellison. "A Formal Semantics of C with Applications". PhD thesis. University of Illinois, 2012. DOI: http://hdl.handle.net/2142/34297.

[ER12] Chucky Ellison and Grigore Rosu. "An Executable Formal Semantics of C with Applications". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 533–544. ISBN: 9781450310833. DOI: 10.1145/2103656.2103719.

[GCC-arrays] FSF. *Using the GNU Compiler Collection (GCC) / 4.7 Arrays and pointers*. https://gcc.gnu.org/onlinedocs/gcc/Arrays-and-pointers-implementation.html.

[GCC-ints] FSF. *Using the GNU Compiler Collection (GCC) / 4.5 Integers*. https://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html.

[GCC-tests] FSF. *GNU Compiler Collection, C Torture Test Suite*. https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite/gcc.c-torture/execute.

[GH92] Yuri Gurevich and James K. Huggins. "The Semantics of the C Programming Language". In: *Selected Papers from the Workshop on Computer Science Logic*. CSL '92. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 274–308. ISBN: 3540569928.

[HACL*] *HACL*, a formally verified cryptographic library written in F**. https://github.com/hacl-star/hacl-star.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 336–345. ISBN: 9781450334686. DOI: 10.1145/2737924.2737979.

[HM98] Graham Hutton and Erik Meijer. "Monadic parsing in Haskell". In: *Journal of Functional Programming* 8.4 (1998), pp. 437–444. DOI: 10.1017/S0956796898003050.

[INT02-C] *SEI CERT C Coding Standard / INT02-C. Understand integer conversion rules*. https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules.

[ISO-C11] *Programming Languages — C*. ISO/IEC 9899:2011. The last public draft is available at https://open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf. 2011.

[ISO94] ISO. *ISO/IEC 10967-1 (1994): Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*. Dec. 1994.

[JM09] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Springer Berlin Heidelberg, 2009, pp. 661–667. ISBN: 978-3-642-02658-4.

[JP17] Jacques-Henri Jourdan and François Pottier. "A Simple, Possibly Correct LR Parser for C11". In: *ACM Trans. Program. Lang. Syst.* 39.4 (Sept. 2017). ISSN: 0164-0925. DOI: 10.1145/3064848. URL: https://doi.org/10.1145/3064848.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. "Validating LR(1) Parsers". In: *European Symposium on Programming (ESOP)*. Springer, 2012, pp. 397–416. DOI: 10.1007/978-3-642-28869-2_20.

[Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. "A Formal C Memory Model Supporting Integer-Pointer Casts". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 326–335. ISBN: 9781450334686. DOI: 10.1145/2737924.2738005.

[KCC18] KCC. *Example Test Suite*. https://github.com/kframework/c-semantics/tree/master/examples/c. 2018.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C Semantics: CompCert and the C Standard". In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Springer International Publishing, 2014, pp. 543–548. ISBN: 978-3-319-08970-6. DOI: 10.1007/978-3-319-08970-6_36.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (K&R)*. Prentice Hall, 1978. ISBN: 978-0-131-10163-0.

[Kre13] Robbert Krebbers. "Aliasing Restrictions of C11 Formalized in Coq". In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Springer International Publishing, 2013, pp. 50–65. ISBN: 978-3-319-03545-1. DOI: 10.1007/978-3-319-03545-1_4.

[Kre14a] Robbert Krebbers. "An Operational and Axiomatic Semantics for Non-Determinism and Sequence Points in C". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, 101=112. ISBN: 9781450325448. DOI: 10.1145/2535838.2535878.

[Kre14b] Robbert Krebbers. "Separation Algebras for C Verification in Coq". In: *Verified Software: Theories, Tools and Experiments*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Springer International Publishing, 2014, pp. 150–166. ISBN: 978-3-319-12154-3.

[Kre15] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud University Nijmegen, Dec. 2015.

[Kre16] Robbert Krebbers. "A Formal C Memory Model for Separation Logic". In: *Journal of Automated Reasoning* 57.4 (Dec. 2016), pp. 319–387. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9369-1.

[KW13] Robbert Krebbers and Freek Wiedijk. "Separation Logic for Non-Local Control Flow and Block Scope Variables". In: *Proceedings of the 16th International Conference on Foundations of Software Science and Computation Structures*. FOSSACS'13. Rome, Italy: Springer-Verlag, 2013, pp. 257–272. ISBN: 9783642370748. DOI: 10.1007/978-3-642-37075-5_17.

[KW15] Robbert Krebbers and Freek Wiedijk. "A Typed C11 Semantics for Interactive Theorem Proving". In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. CPP '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 15–27. ISBN: 9781450332965. DOI: 10.1145/2676724.2693571.

[Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing Sequential Consistency in C/C++11". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 618–632. ISBN: 9781450349888. DOI: 10.1145/3062341.3062352.

[Lau+19] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. "Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C". In: *Proc. 31st International Conference on Computer-Aided Verification*. July 2019. DOI: 10.1007/978-3-030-25540-4_22.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations". In: *J. Autom. Reason.* 41.1 (July 2008), pp. 1–31. ISSN: 0168-7433. DOI: 10.1007/s10817-008-9099-0.

[Lee+18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. "Reconciling High-level Optimizations and Low-level Code with Twin Memory Allocation". In: *Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '2018. Boston, MA, USA, Nov. 2018.

[Lep+22] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498681.

[Ler+12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26. URL: https://hal.inria.fr/hal-00703441.

[Ler09] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[Mat11] J. Matthiesen. "Mathematizing the C programming language". Part II dissertation. May 2011.

[Mat12] J. Matthiesen. "Elaborating C". MPhil dissertation. June 2012.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the de Facto Standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 1–15. ISBN: 9781450342612. DOI: 10.1145/2908080.2908081.

[Mem+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. "Exploring C Semantics and Pointer Provenance". In: vol. 3. POPL. New York, NY, USA: Association for Computing Machinery, Jan. 2019. DOI: 10.1145/3290380.

[Mul+14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. "Lem: Reusable Engineering of Real-world Semantics". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, Sept. 2014, pp. 175–188. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628143.

[musl-libc] *musl libc*. https://musl.libc.org/.

[N1637] Robbert Krebbers and Freek Wiedijk. *N1637: Subtleties of the ANSI/ISO C standard*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf. Sept. 2012.

[N1747] Freek Wiedijk and Robbert Krebbers. *N1747*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1747.htm. Aug. 2013.

[N2012] Kayvan Memarian and Peter Sewell. *N2012: Clarifying the C memory object model*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2012.htm. Mar. 2016.

[N2013] David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. *C memory object and value semantics: the space of de facto and ISO standards*. https://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf (a revision of ISO SC22 WG14 N2013). Mar. 2016.

[N2014] Kayvan Memarian and Peter Sewell. *What is C in practice? (Cerberus survey v2): Analysis of Responses*. https://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html (a revision of ISO SC22 WG14 N2014). Mar. 2016.

[N2015] Kayvan Memarian and Peter Sewell. *What is C in practice? (Cerberus survey v2): Analysis of Responses - with Comments*. https://www.cl.cam.ac.uk/~pes20/cerberus/analysis-2016-02-05-anon.txt (a revision of ISO SC22 WG14 N2015). Mar. 2016.

[N2089] Kayvan Memarian and Peter Sewell. *N2089: Clarifying Unspecified Values (Draft Defect Report or Proposal for C2x)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2089.htm. Sept. 2016.

[N2090] Kayvan Memarian and Peter Sewell. *N2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2090.htm. Sept. 2016.

[N2091] Kayvan Memarian and Peter Sewell. *N2091: Clarifying Trap Representations (Draft Defect Report or Proposal for C2x)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2091.htm. Sept. 2016.

[N2219] Victor Gomes Kayvan Memarian and Peter Sewell. *N2219: Clarifying Pointer Provenance (Q1-Q20) v3*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2219.htm. Mar. 2018.

[N2220] Victor Gomes Kayvan Memarian and Peter Sewell. *N2220: Clarifying Trap Representations (Q47) v3*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2220.htm. Mar. 2018.

[N2221] Victor Gomes Kayvan Memarian and Peter Sewell. *N2221: Clarifying Unspecified Values (Q47-Q59) v3*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2221.htm. Mar. 2018.

[N2222] Victor Gomes Kayvan Memarian and Peter Sewell. *N2222: Further Pointer Issues (Q21-Q46)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2222.htm. Mar. 2018.

[N2223] Victor Gomes Kayvan Memarian and Peter Sewell. *N2223: Clarifying the C Memory Object Model: Introduction to N2219 - N2222*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2223.htm. Mar. 2018.

[N2263] Victor Gomes Kayvan Memarian and Peter Sewell. *N2263: Clarifying Pointer Provenance v4*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2263.htm. May 2018.

[N2335] Aaron Ballman. *N2335: Attributes in C*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2335.pdf. Mar. 2019.

[N2362] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. *N2362: Moving to a provenance-aware memory object model for C*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2362.pdf. Mar. 2019.

[N2364] Peter Sewell, Kayvan Memarian, and Victor B. F. Gomes. *N2364: C provenance semantics: detailed semantics (for PNVI-plain, PNVI address-exposed, PNVI address-exposed user-disambiguation, and PVI models)*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2364.pdf. Apr. 2019.

[N2369] Paul E. McKenney, Maged Michael, and Peter Sewell. *N2369: Pointer lifetime-end zap*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf. Apr. 2019.

[N3005] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. *N3005: A Provenance-aware Memory Object Model for C - Draft Technical Specification*. https://open-std.org/JTC1/SC22/WG14/www/docs/n3005.pdf. June 2022.

[Nec+02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Compiler Construction*. Ed. by R. Nigel Horspool. Springer Berlin Heidelberg, 2002, pp. 213–228. ISBN: 978-3-540-45937-8.

[NMS16] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. "An operational semantics for C/C++11 concurrency". In: *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Amsterdam, The Netherlands: ACM, Nov. 2016. DOI: 10.1145/2983990.2983997.

[Nor98] Michael Norrish. *C formalised in HOL*. Tech. rep. UCAM-CL-TR-453. University of Cambridge, Computer Laboratory, Dec. 1998. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf.

[Nor99] Michael Norrish. "Deterministic Expressions in C". In: *Proceedings of the 8th European Symposium on Programming Languages and Systems*. ESOP '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 147–161. ISBN: 3540656995.

[note30] David Chisnall, Justus Matthiesen, Kayvan Memarian, Peter Sewell, and Robert N. M. Watson. *C memory object and value semantics: the space of de facto and ISO standards*. https://www.cl.cam.ac.uk/~pes20/cerberus/notes30-full.pdf (a revision and extension of ISO SC22 WG14 N2013). Mar. 2016.

[notes98] Kayvan Memarian, Victor Gomes, and Peter Sewell. *Clarifying Uninitialised Values (Q47-Q59) v4 - working draft*. https://www.cl.cam.ac.uk/~pes20/cerberus/notes98-2018-04-21-uninit-v4.html. Apr. 2018.

[P1726R4] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams. *P1726R4: Pointer lifetime-end zap*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1726r4.pdf. Aug. 2020.

[P1796R0] Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Jens Gustedt, and Hubert Tong. *P1796R0: Effective types: examples*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1796r0.pdf. Apr. 2019.

[Pap98] Nikolaos S Papaspyrou. "A formal semantics for the C programming language". PhD thesis. National Technical University of Athens, Department of Electrical and Computer Engineering, Division of Computer Science, Feb. 1998.

[PR05] François Pottier and Yann Régis-Gianas. *Menhir LR(1) parser generator for OCaml*. https://cambium.inria.fr/~fpottier/menhir/. 2005.

[Pul+23] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. "CN: Verifying systems C code with separation-logic refinement types". In: *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '2023. Conditionally Accepted. 2023.

[Reg+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-Case Reduction for C Compiler Bugs". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 335–346. ISSN: 0362-1340. DOI: 10.1145/2345156.2254104.

[Rid+15] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. "SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 38–53. ISBN: 9781450338349. DOI: 10.1145/2815400.2815411.

[Rit93] Dennis M. Ritchie. "The Development of the C Language". In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 201–208. ISSN: 0362-1340. DOI: 10.1145/155360.155580. URL: https://doi.org/10.1145/155360.155580.

[Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 158–174. ISBN: 9781450383912. DOI: 10.1145/3453483.3454036.

[Šev+11] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "Relaxed-Memory Concurrency and Verified Compilation". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 43–54. ISBN: 9781450304900. DOI: 10.1145/1926385.1926393.

[Šev+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency". In: *J. ACM* 60.3 (June 2013). ISSN: 0004-5411. DOI: 10.1145/2487241.2487248.

[SMM15] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. "Test suites for benchmarks of static analysis tools". In: *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 12–15. DOI: 10.1109/ISSREW.2015.7392027.

[TinyCC] *Tiny C Compiler*. https://repo.or.cz/w/tinycc.git.

[TK05] Harvey Tuch and Gerwin Klein. "A Unified Memory Model for Pointers". In: *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'05. Montego Bay, Jamaica: Springer-Verlag, 2005, pp. 474–488. ISBN: 354030553X. DOI: 10.1007/11591191_33.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. "Types, Bytes, and Separation Logic". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: Association for Computing Machinery, 2007, pp. 97–108. ISBN: 1595935754. DOI: 10.1145/1190216.1190234.

[Tuc08] Harvey Tuch. "Formal Memory Models for Verifying C Systems Code". PhD thesis. Sydney, Australia: UNSW, Aug. 2008.

[Vaf+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. "Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do about It". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 209–220. ISBN: 9781450333009. DOI: 10.1145/2676726.2676995.

[Wan+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Undefined Behavior: What Happened to My Code?" In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: Association for Computing Machinery, 2012. ISBN: 9781450316699. DOI: 10.1145/2349896.2349905.

[Wan+13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 260–275. ISBN: 9781450323888. DOI: 10.1145/2517349.2522728.

[WG14-DR] WG14. *Defect Report Summary for ISO/IEC 9899:1999*. https://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm.

[Win+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. "Mind the Gap". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Springer Berlin Heidelberg, 2009, pp. 500–515. ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_34.

[Woo+14] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468. ISBN: 9781479943944.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. ISBN: 9781450306638. DOI: 10.1145/1993498.1993532.