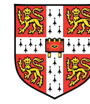
Technical Report

UCAM-CL-TR-799
ISSN 1476-2986

Number 799



UNIVERSITY OF CAMBRIDGE

Computer Laboratory

A separation logic framework for HOL

Thomas Tuerk

June 2011

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

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

© 2011 Thomas Tuerk

This technical report is based on a dissertation submitted December 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Downing College.

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

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

ISSN 1476-2986

A Separation Logic Framework for HOL

Thomas Tuerk

Summary

Separation logic is an extension of Hoare logic due to O'Hearn and Reynolds. It was designed for reasoning about mutable data structures. Because separation logic supports local reasoning, it scales better than classical Hoare logic and can easily be used to reason about concurrency. There are automated separation logic tools as well as several formalisations in interactive theorem provers. Typically, the automated separation logic tools are able to reason about shallow properties of large programs. They usually consider just the shape of datastructures, not their data-content. The formalisations inside theorem provers can be used to prove interesting, deep properties. However, they typically lack automation. Another shortcomming is that there are a lot of slightly different separation logics. For each programming language and each interesting property a new kind of separation logic seems to be invented.

In this thesis, a general framework for separation logic is developed inside the HOL4 theorem prover. This framework is based on *Abstract Separation Logic*, an abstract, high level variant of separation logic. Abstract Separation Logic is a general separation logic such that many other separation logics can be based on it. This framework is instantiatiated in a first step to support a stack with read and write permissions following ideas of Parkinson, Bornat and Calcagno. Finally, the framework is further instantiated to build a separation logic tool called Holfoot. It is similar to the tool Smallfoot, but extends it from reasoning about shape properties to fully functional specifications.

To my knowledge this work presents the first formalisation of Abstract Separation Logic inside a theorem prover. By building Holfoot on top of this formalisation, I could demonstrate that Abstract Separation Logic can be used as a basis for realistic separation logic tools. Moreover, this work demonstrates that it is feasable to implement such separation logic tools inside a theorem prover. Holfoot is highly automated. It can verify Smallfoot examples automatically inside HOL4. Moreover, Holfoot can use the full power of HOL4. This allows Holfoot to verify fully functional specifications. Simple fully functional specifications can be handled automatically using HOL4's tools and libraries or external SMT solvers. More complicated ones can be handled using interactive proofs inside HOL4. In contrast, most other separation logic tools can reason just about the shape of data structures. Others reason only about data properties that can be solved using SMT solvers.

Contents

1 Introduction **11**

1.1 Introduction of Separation Logic 11

1.2 Smallfoot 12

1.3 Short overview of Separation Logic Tools 13

1.4 Contributions of this work 14

1.4.1 General overview 14

1.4.2 Capabilities of Holfoot 14

1.4.3 Contributions in Detail 15

1.4.4 Contributions to HOL4 17

1.5 Structure of the thesis 17

2 Holfoot **19**

2.1 Input Language 19

2.1.1 States 20

2.1.2 Pure Expressions 21

2.1.3 Predicates 22

2.1.4 Statements 26

2.1.5 Conditions 27

2.1.6 HOL4 Syntax 27

2.1.7 Programs 28

2.1.8 Specifications 28

2.2 Introductory Examples 29

2.2.1 Recursive Implementation of List-Length 29

2.2.1.1 Local reasoning 30

2.2.1.2 Read/Write Permissions 30

2.2.1.3 Internal Representation 31

2.2.1.4 Fully-Functional Specifications 32

2.2.2 Pointer Transferring Buffer Example . . . 32
2.3 Annotating While-Loops . . . 33
2.3.1 Loop Invariants . . . 33
2.3.2 Loop Specifications . . . 34
2.3.3 Examples . . . 35
2.3.3.1 Array Increment Example . . . 36
2.3.3.2 List Filtering Example . . . 37
2.3.3.3 List Copy Example . . . 38
2.3.3.4 Partial Datastructures . . . 39
2.3.4 Unrolling Loops . . . 41
2.4 Additional Constructs . . . 42
2.4.1 assume / assert . . . 43
2.4.2 diverge, fail . . . 43
2.4.3 Block Specifications . . . 44
2.4.4 Annotating Memory Allocation . . . 45
2.4.5 Assuming Procedures . . . 45
2.4.6 Global Specification Variables . . . 46
2.5 Interactive Proofs . . . 47
2.5.1 General Overview . . . 47
2.5.2 Sum and Maximal Element of an Array Example . . . 48
2.5.3 List Remove Example . . . 49
2.5.4 Mergesort Example . . . 50
2.5.5 Circular List Example . . . 52
2.5.6 Binary Search Tree Example . . . 53
2.5.7 Insertion into Red-Black Tree Example . . . 55
2.6 Extending Holfoot . . . 56
2.6.1 Amortised Queue Example . . . 57
2.7 Conclusion . . . 58

3 Theoretical Foundation and Implementation 59
3.1 Notations . . . 60
3.1.1 Sets . . . 60
3.1.2 Finite Maps . . . 60
3.1.3 Multisets . . . 61
3.1.4 Lists . . . 61

3.2 Abstract Separation Logic . . . 61
3.2.1 States and Predicates on States . . . 61
3.2.1.1 Separation Combinators . . . 62
3.2.1.2 Predicates . . . 62
3.2.1.3 Separation Algebras . . . 63
3.2.1.4 Product Separation Combinators . . . 64
3.2.2 Actions . . . 65
3.2.2.1 Semantic Hoare triples . . . 65
3.2.2.2 Common Actions . . . 66
3.2.2.3 Local Actions . . . 67
3.2.2.4 Total Lattice of Local Actions . . . 68
3.2.2.5 Best Local Action . . . 69
3.2.2.6 Semaphore operations / Precise Predicates . . . 70
3.2.2.7 Quantified Best Local Action . . . 71
3.2.2.8 *assume* . . . 71
3.2.3 Programs . . . 72
3.2.3.1 Programs, Proto Traces, Traces 73
3.2.3.2 Semantics of Programs, Proto Traces, Traces 75
3.2.3.3 Comments on Semantics . . . 76
3.2.4 Common Programming Constructs . . . 77
3.2.4.1 Sequential Composition . . . 77
3.2.4.2 Nondeterministic Choice . . . 78
3.2.4.3 Conditional Execution / While Loops . . . 78
3.2.4.4 Conditional Critical Regions . . . 78
3.2.4.5 Infinite Nondeterministic Choice . . . 79
3.2.5 Inference Rules . . . 79
3.2.5.1 Frame Rule . . . 80
3.2.5.2 Structural Rules . . . 80
3.2.5.3 Basic commands . . . 81
3.2.5.4 Basic Program Compositions . . . 82
3.2.5.5 Control Structures . . . 82
3.2.5.6 Symbolic Execution . . . 83
3.2.5.7 *assume* . . . 84
3.2.6 Program Abstraction . . . 85
3.2.7 Recursive Procedures . . . 88

3.2.8 Summary . . . 89
3.3 Variables as Resource . . . 89
3.3.1 Stacks with Read / Write Permissions . . . 90
3.3.2 Expressions . . . 92
3.3.3 Predicates . . . 93
3.3.3.1 Stack-Imprecise Predicates . . . 93
3.3.3.2 Pure Predicates . . . 95
3.3.3.3 Separating Conjunction on Lists . . . 96
3.3.4 Normal Forms . . . 97
3.3.5 Inference Rules . . . 101
3.3.6 Program Constructs . . . 102
3.3.6.1 Assume . . . 103
3.3.6.2 Control Structures . . . 104
3.3.6.3 Semaphore Operations . . . 105
3.3.6.4 Procedure Calls . . . 105
3.3.6.5 Assignments . . . 107
3.3.6.6 Local Variables . . . 109
3.3.6.7 Quantified Best Local Actions . . . 110
3.3.7 Frame Inference . . . 110
3.3.7.1 Informal Discussion . . . 110
3.3.7.2 Basic Definitions . . . 111
3.3.7.3 Inference Rules . . . 113
3.3.7.4 Solving Frame Inference Predicates . . . 114
3.3.7.5 Frame Inference Algorithm . . . 114
3.3.8 Implicit Information . . . 115
3.4 Holfoot . . . 116
3.4.1 States . . . 116
3.4.2 Predicates . . . 117
3.4.2.1 Points-To . . . 118
3.4.2.2 Singly-Linked Lists . . . 118
3.4.2.3 Trees . . . 119
3.4.2.4 Arrays . . . 120
3.4.3 Program Constructs . . . 121
3.4.3.1 Memory Allocation . . . 122
3.4.3.2 Memory Deallocation . . . 122

3.4.3.3 Heap Lookup . . . 123
3.4.3.4 Heap Assignment . . . 124
3.4.4 Implicit Information . . . 124
3.4.5 Frame Inference . . . 126
3.5 Holfoot Implementation . . . 129
3.5.1 Overview . . . 129
3.5.2 Consequence Conversions . . . 130
3.5.3 Quantifier Heuristics . . . 131

4 Conclusion **133**
4.1 Summary . . . 133
4.2 Conclusion . . . 134
4.3 Future Work . . . 135

Bibliography **137**

A Holfoot Installation **141**
A.1 Installation of HOL4 . . . 141
A.2 Installation of Holfoot . . . 142
A.3 Testing Holfoot . . . 143

B Example Specifications **145**
B.1 Automatic Examples . . . 145
B.1.1 General List Example . . . 145
B.1.2 List Length . . . 146
B.1.3 List Reverse . . . 147
B.1.4 List Copy . . . 147
B.1.5 List Append . . . 148
B.1.6 List Allocation and Deallocation by Length . . . 148
B.1.7 List Filter . . . 149
B.1.8 Queue . . . 150
B.1.9 Binary Tree Copy / Deallocate . . . 151
B.1.10 Races . . . 151
B.1.11 Buffers . . . 153
B.1.12 Memory Manager . . . 153
B.1.13 Shape Property Versions of Interactive Examples . . . 155

B.2 Interactive Examples 157
B.2.1 Tree Map 157
B.2.2 Tree Depth 157
B.2.3 List Remove 159
B.2.4 Circular List 161
B.2.5 List Filter 162
B.2.6 List Rotating 165
B.2.7 Factorial 166
B.2.8 Tree Sum 167
B.2.9 Array Increment 168
B.2.10 Array Copy 170
B.2.11 Array Reverse 172
B.2.12 Binary Search 173
B.2.13 Mergesort 175
B.2.14 Insertion Sort 176
B.2.15 Quicksort 178
B.2.16 Binary Search Tree 181
B.2.17 Red-Black Tree 184
B.3 VSTTE'10 Competition 190
B.3.1 Problem 1 190
B.3.2 Problem 2 192
B.3.3 Problem 3 193
B.3.4 Problem 4 194
B.3.5 Problem 5 198

C HOL4-Theorem Index **203**
C.1 holfootTheory 203
C.2 separationLogicTheory 221
C.3 vars_as_resourceTheory 244

Chapter 1

Introduction

Separation logic [33] has become popular in recent years. In this work, a framework for separation logic inside the HOL4 [13, 34] theorem prover is presented. The main focus is generality. The framework is intended to be easily instantatiable for different programming languages and different separation logics. As a case study a tool called *Holfoot* is implemented in this framework. Holfoot is able to reason about the partial correctness of programs written in a simple, low-level imperative language. It provides a high level of automation. The main focus, however, is combining the automation of separation logic with the power of interactive theorem provers. Holfoot can reason about fully functional specifications using all the libraries and infrastructure provided by HOL4. Before going into any details, let's have a look at separation logic in general:

1.1 Introduction of Separation Logic

Separation logic is an extension of Hoare logic. It was introduced by Reynolds [33] based on previous work by Burstall [6] and O'Hearn et al. [30]. It aims at reasoning about mutable data structures in combination with low level imperative programming languages that use pointers and explicit memory management. Usually statements of a programming language operate on a well defined part of the current state. Everything outside this local state does not affect the execution of the statement and is itself not affected by the statement. The main idea of separation logic is to exploit this locality.

Consider for example the statement `x := [y] + 1`, which looks up the value stored in memory at the current value of variable `y`, increases it by one and stores the result in variable `x`. This statement just needs to access the variables `x` and `y` as well as the memory location `[y]`. All other variables as well as all other memory locations do not influence the execution of the statement and remain unmodified. If the statement can not access one of the resources `x`, `y` and `[y]`, it fails. Therefore, whenever a Hoare triple $\{P\}$ `x := [y] + 1` $\{Q\}$ holds, one knows that P somehow mentions `x`, `y` and `[y]` and that the specification can be safely extended by a *frame* that is *separate* to P, i. e. describes a separate part of the state.

In order to exploit such local reasoning, separation logic introduces a *separating conjunction operator* $*$ and the *frame* inference rule:

$$\begin{array}{ll}\forall \mathsf{prog}, P, Q. & \{P\}\ \mathsf{prog}\ \{Q\} \quad\quad \Longleftrightarrow \\ & \forall R.\ \{P * R\}\ \mathsf{prog}\ \{Q * R\}\end{array}$$

The definition of the $*$-operator fixes the notion of separation. For different programming languages, different representations of states, different verification goals, the concrete definition of $*$ changes considerably. The frame rule uses this notion of separation to extend a specification with an unrelated part of the state. It is sound iff all statements of the programming language are local with respect to the notion of separation used.

The idea of local reasoning extends to concurrency as well. For the simple case of non interfering parallel composition the following rule can be used:

$$\begin{array}{l}\forall \mathsf{prog}_1,\, P_1,\, Q_1,\, \mathsf{prog}_2,\, P_2,\, Q_2. \\ \quad (\{P_1\}\ \mathsf{prog}_1\ \{Q_1\}\ \wedge\ \{P_2\}\ \mathsf{prog}_2\ \{Q_2\}) \implies \\ \quad \{P_1 * P_2\}\ \mathsf{prog}_1 \mid\mid \mathsf{prog}_2\ \{Q_1 * Q_2\}\end{array}$$

1.2 Smallfoot

The first separation logic tool was *Smallfoot* [2, 3]. Smallfoot uses a simple, imperative, low level programming language. This language operates on a stack that maps variables to values and a heap that maps locations to records of values. The entries in the record are indexed by tags represented by strings, values are integers and locations are positive integers. There are statements for assigning a value to a variable, lookup of a heap entry, updating a heap entry and explicit allocation and deallocation of heap cells. As control structures there are conditional execution and while-loops. Smallfoot supports procedures that can use local variables, call-by-reference and call-by value parameters and recursion. Concurrency is supported by parallel procedure calls as well as conditional critical regions, which protect access to certain resources.

Smallfoot can automatically reason about shallow properties of programs written in this simple language. It is mainly interested in the shape of datastructures in memory. Smallfoot does not prove termination, i. e. only partial correctness is considered. The specification is given in terms of pre- and postconditions for procedures. In order to prove such specifications, Smallfoot requires programs to be annotated with loop invariants and invariants for the resources used by conditional critical regions. Given such annotations, Smallfoot works automatically.

Instead of using backwards analysis and the generation of verification conditions, Smallfoot uses symbolic execution. Moreover, local updates as well as an automatic method to calculate frames were introduced by Smallfoot. A typical Smallfoot-example is the above mentioned reversal of a singly-linked list:

```
list_reverse(i;) [list(i)] {
  local p, x;
  p = NULL;
  while (i != NULL) [list(i) * list (p)] {
    x = i->tl;
    i->tl = p;
    p = i;
    i = x;
  }
  i = p;
} [ list (i)]
```

This pseudocode algorithm in the syntax of Smallfoot takes an argument `i` that points to the beginning of a singly-linked list and reverses this list in-place. In order to do that, an auxiliary list is constructed that is pointed to by the local variable `p`, which is initialised by the null-pointer. The loop then chops off the first element of the list starting at `i` and adds it as the head of the list `p`. When the loop terminates the list pointed to by `i` is empty and `p` contains the reversed list.

Notice, that the specification of this algorithm talks just about the shape of data structures. At the beginning there is a singly-linked list starting at `i` and at the end there is still a list starting at `i`. Nothing is said about the data content or that the list gets reversed by this algorithm. Similarly, the loop invariant just states that there are two lists in separate parts of the memory.

1.3 Short overview of Separation Logic Tools

Since Smallfoot several other separation logic tools have been developed for various purposes. One extension was to extend automation by infering loop invariants [11, 22]. Moreover, the simple pseudo-code language of Smallfoot was replaced by real-world programming languages. Results are tools like *Space Invader* or *SLAyer*, which can automatically reason about shallow properties of large C programs. Using a new technique called *abduction* [8] the tool *Space Invader Abductor* is even able to discover procedure specifications. Other tools focus on object orientation. Tools like *JStar* [10] can reason about shallow properties of Java programs.

Smallfoot supports *concurrent separation logic* [5], which provides a parallel composition operator and conditional critical regions. Locks for these critical regions have to be predefined and annotated with a *lock-invariant* that describes the part of the state protected by the lock. Interest in fine grained concurrency has led to the introduction of permissions by Bornat, Calcagno, O'Hearn and Parkinson [4, 31]. Furthermore, Vafeiadis introduced *a marriage of rely/guarantee and separation logic* [37] and implemented his ideas in the tool *SmallfootRG*. This line of work has recently resulted in *Deny-Guarantee* reasoning [12]. Another approach by Gotsman et al. is to allow storeable locks, i. e. locks that can be dynamically created at runtime, and fork-join primitives for parallel programs [14, 15].

Besides improving reasoning about concurrency, automatically infering specifications and moving to real world programming languages, another focus of tool development are deeper properties. Examples are the tools *VeriFast* [18, 19] and *HIP* [26]. Even using powerful automated tools like external SMT solvers, it is very hard to prove deep properties automatically. Therefore, these tools are to varying degrees interactive. HIP allows users to verify remaining proof obligations interactively with external provers. VeriFast supports a very rich annotation language. Users can even add lemma-functions to the program which act like interactive proof scripts. So, the distinction between automatic and interactive tools becomes fuzzy as soon as deep properties are considered.

There are formalisations of separation logic inside theorem provers that go all the way and are clearly interactive. There are several formalisations of separation logic in Coq [24]. Appel and Blazy [1], Tuch, Klein and Norrish [35] as well as McCreight [25] use formalisations of separation logic in Coq to reason about Cminor [21]; Marti, Affeldt and Yonezawa [23] reason about programs in their own, low-level imperative language. A

formalisation of separation logic in Isabelle/HOL [27] by Kolanski and Klein [20] is able to reason about a large subset of C. Another formalisation in Isabelle/HOL by Weber [38] uses a simple while language. All these formalisations allow interactive reasoning about deep properties. However, there is little automation. Another formalisation of separation logic is *YNot* [9]. *YNot* extends Coq's functional programming language with imperative constructs and allows reasoning about these imperative constructs. Compared with the other formalisations, there is a high level of automation.

1.4 Contributions of this work

1.4.1 General overview

I developed a separation logic framework inside the HOL4 [13, 34] theorem prover. In contrast to other formalisations in theorem provers, this framework is intended to be usable for different programming languages and different flavours of separation logic. The framework is based on *Abstract Separation Logic* [7], an abstract, high-level variant of separation logic by Calcagno, O'Hearn and Yang. Reasoning about partial correctness and *concurrent separation logic* [5] are built deep into Abstract Separation Logic. Termination proofs and different concepts of concurrency like fork-join constructs are not supported.

I instantiated this framework to build *Holfoot*, a tool similar to Smallfoot [3]. This instantiation consists of two steps: first a stack with with read / write permissions is added following ideas of Parkinson, Bornat and Calcagno [31]. This still rather abstract layer allows reasoning about most of the concepts needed by Holfoot. For example, local variables and procedure calls with call-by-reference and call-by-value parameters can be handled at this layer. Holfoot supports only pure expressions and conditions of control structures, i. e. expressions and conditions that involve just the stack. Therefore, all the reasoning about variable assignments and control structures like while-loops or conditional execution can be handled at this layer. Moreover, most of the frame computation rules can be formalised here.

In the second step, a heap is added to the model. This allows defining commands like heap-lookups and heap-assignments as well as commands for the allocation and deallocation of heap-cells. Additionally, all the predicates involving the heap like predicates describing a singly-linked list or a binary tree in the heap are defined at this layer. However, there is comparably little effort needed at this layer. Most of the work is done at the upper layers. Since this is the view a user of Holfoot gets of the system, there is a parser and a pretty printer for this layer.

1.4.2 Capabilities of Holfoot

Holfoot is highly automated. It can verify most Smallfoot examples automatically inside HOL4. However, it uses the infrastructure provided by HOL4 to go beyond the features of Smallfoot. Holfoot allows reasoning about the content of data-structures instead of just their shape. Thus, one can use Holfoot to verify fully functional specifications. Many simple fully functional specifications can be handled automatically. More complicated specifications, like a fully functional specification of quicksort or insertion into a red-black

tree, can be tackled interactively. The quicksort example also demonstrates that Holfoot can — in contrast to Smallfoot — handle arrays and pointer arithmetic.

If Holfoot is used interactively, all the libraries and tools of HOL4 can be utilised. This includes infrastructure for reasoning about datatypes like lists or sets, some support for arithmetic reasoning or even support for calling external SMT solvers. Often using HOL4's infrastructure for introducing new definitions and the organisation of lemmata is useful as well.

1.4.3 Contributions in Detail

To my knowledge this work presents the first formalisation of Abstract Separation Logic inside a theorem prover. A formalisation inside a theorem prover increases the trust in Abstract Separation Logic, i. e. it increases the trust that all the detailed definitions and constructions of Abstract Separation Logic really fit together without problems. More importantly though, by building Holfoot on top of this formalisation, I could demonstrate the power and flexibility of Abstract Separation Logic. Smallfoot is a relatively simple separation logic tool and therefore a good choice for a formalisation. Its programming language has few instructions and there is a simple model of the state. Despite that, the techniques used by Smallfoot are still at the core of more recent separation logic tools like JStar or Space Invader. Building a formalisation of Smallfoot on top of Abstract Separation Logic therefore demonstrates that the other tools can be based on it, too.

In order to build Holfoot, a lot of minor technical and theoretical problems had to be solved. These problems and their solutions will be explained later in detail, here I will just point out a few highlights. I extend Abstract Separation Logic slightly. The most noticeable change is adding procedures. Ignoring some minor technical details, the formalisation of Abstract Separation Logic is otherwise close to its original presentation [7].

The second layer of abstraction, i. e. the introduction of a stack, follows ideas of Parkinson, Bornat and Calcagno presented in *Variables as Resource in Hoare Logics* [31]. In contrast to the Abstract Separation Logic layer this layer is only loosely related to the original paper [31], because I adapted these ideas to Abstract Separation Logic. For example, I had to express local variable declarations and call-by-value arguments of procedure calls in the Abstract Separation Logic setting. I also extended the notion of stack-imprecise predicates and formalised a notion of semantic substitutions. There are a lot of minor problems that needed addressing. The most important one is how I address the frame problem. By adding a context I eliminated the need to be extremely careful about the order of steps taken. This technique has meanwhile been adapted by other tools like JStar. Moreover, I use a continuation style that allows unifying the frame and entailment problems.

The Holfoot layer is a formalisation of Smallfoot. I could use the syntax of the programming language including parts of the grammar for the parser generator, some rough ideas about the semantics and a general idea of high-level inference rules needed. A lot had to be modified considerably, though. Other parts of Holfoot are hardly related to Smallfoot at all. There are good reasons for these differences. One reason is that Smallfoot [2, 3] defines the semantics of the programming language in terms of high-level inference rules for symbolic execution. In contrast, Holfoot is using Abstract Separation Logic, which means

that an operational semantics is given for instructions like a heap-lookup. The high-level inference rules are derived from these definitions and the semantics of Abstract Separation Logic. In general, Smallfoot is mainly interested in reasoning about so called *verification conditions*. These verification conditions consist of Hoare triples for straight-line code, i. e. code without loops, procedures and local variable declarations and constructs for concurrency like parallel compositions or conditional critical regions. It is not well documented how these verification conditions are generated, i. e. how loops, procedure calls and synchronisation primitives are removed from the input. Holfoot replaces all procedure calls during preprocessing with their specifications. Similarly, conditional critical regions are abstracted. The result is similar to Smallfoot's verification conditions. However, all these steps are done by proof and the stack is an integral part of these transformations.

These differences in how the semantics are defined result in additional effort on the side of Holfoot to come up with high-level inference rules. A much more important reason why little of Smallfoot can be used for Holfoot is that Holfoot is interested in fully-functional specifications and interactive proofs. In order to allow fully-functional specifications, data-content was added to the predicates used by Smallfoot. Moreover, arbitrary HOL4 terms are allowed for describing this data-content and sideconditions. This data-content, handling stack-variables explicitly and using a context for frame calculations require significant modifications and extensions to Smallfoot's inference rules. Moreover, Holfoot supports additional constructs. There are arrays and pointer arithmetic. These require generalisations of the heap allocation and deallocation statements as well as new predicates and inference rules.

Interactivity is another important reason why Holfoot differs considerably from Smallfoot. I assume an experienced, intelligent user. The philosophy of Holfoot is to give this user all the power and as much information as possible, while of course trying to maintain an as high level of automation as possible. In contrast, Smallfoot puts the emphasis on automation. Holfoot's philosophy results in simple things like preserving as much of the structure of the input during proofs in order to allow the user to use its intuition about the program. With the same goal comments are introduced that help understanding the origin of proof obligations. More importantly though, Holfoot's automation should never do any guessing that might lead the verification process down a wrong track. The inferences used are usually proper implications. When applying them, some information is lost. Only inferences rules that are unlikely to loose important information should be applied automatically. Effort has been spent to develop such inference rules. Good examples are Smallfoot's inferences for the frame and entailment problems. To avoid problems, Smallfoot is applying two groups of inference rules in the right order. I was able to avoid most of these problems by adding a context to the frame / entailment calculations. Another example for Holfoot's philosophy are case-splits. Sometimes a case-split is necessary and there are a few heuristics about likely candidates for a case split. For Smallfoot and Holfoot running automatically these case-splits are very useful. Interactively, it is often much better to ask the user, who hopefully has some insight in how the program is supposed to work, for help instead of trying possibly the wrong case-split.

With the same motivation of empowering the user, I implemented additional annotations in Holfoot. I added assume and assert statements to Holfoot. Assume statements are used to model conditional execution and while-loops. They can be used on their own as well, though. Assume statements are limited to assuming pure predicates, i. e. they can't talk

about the heap. Asserts, on the other hand, can use arbitrary predicates. They trigger a frame calculation and can easily be used to fold or unfold predicates manually or derive facts that are only implicitly present. More interesting and useful are block-specifications, though. They allow annotating a block of code with a pre- and a post-condition. This code is then abstracted with this specification. A special case are block-specifications, whose first statement is a while-loop. These *loop-specifications* are an alternative to loop-invariants. They allow the user to exploit local reasoning for while-loops and lead often to simpler, more natural specifications. I presented this idea at the VSTTE'10 Theory workshop [36].

1.4.4 Contributions to HOL4

Besides this work on separation logic constructs, I worked on the general theorem proving infrastructure of HOL4 as well. There are a lot of minor technical additions that proved useful for Holfoot. For example, I extended HOL4's pretty-printer in order to implement syntax highlighting for Holfoot. Other noticeable additions include work on HOL4's list and pair libraries. Most prominent and by far the largest parts of my work on general theorem proving infrastructure are however the *consequence conversion* and *quantifier heuristic* libraries.

As mentioned above, most of the inferences used by Holfoot are implications. The automation mainly consists of applying these inference until either the problem is solved or no further inference can be applied. This represents a kind of rewriting using implications instead of equalities. There was no infrastructure in HOL4 for this. My library for *consequence conversions* provides the necessary infrastructure. It is at the very heart of Holfoot's automation as it is used to apply Holfoot's inference rules.

The *quantifier heuristics* library is used to guess quantifier instantiations. Especially due to considering data-content, many existential quantifiers occur during verification. HOL4 provides infrastructure for instantiating them in simple cases. Unfortunately, these simple methods were not sufficient to achieve a high level of automation for Holfoot. Therefore, I implemented a new library for quantifier instantiations. This library uses user-definable heuristics to guess instantiations. It is much more powerful than previous tools available in HOL4. In particular it is able to use consequence conversions in order to implement guesses that it can not prove to be justified. Moreover, it is able to instantiate quantifiers only partly. If, for example, it is searching for a list and it can be determined that the list is not empty, then it can instantiate that list with a non empty list whose head and tail are still quantified. The quantifier heuristics library is an essential part of Holfoot. Especially partial instantiations are crucial for Holfoot's automation.

1.5 Structure of the thesis

The rest of this thesis is separated into two main parts: Chapter 2 and Chapter 3. The first part, i. e. Chapter 2, consists of a high-level presentation of Holfoot. No implementation details and no theoretical background are discussed here. The reader does not need to know HOL4 or separation logic.

I present the features of Holfoot using concrete example specifications. Many of these specifications can be verified completely automatically. Thanks to the availability of

a precompiled command-line version of Holfoot, the user does not even need to know that Holfoot is implemented using HOL4 for these automated examples. For interactive examples, some exposure to HOL4 cannot be avoided. I try to explain the basic proof ideas while avoiding HOL4 details as much as possible.

Chapter 3 contains the theoretical foundations of the framework and of Holfoot. These foundations are presented in a mathematical style. Readers not familiar with HOL4 should be able to read and understand it. Important definitions and lemmata are, however, cross-referenced with their HOL4 counterparts. This should enable the interested reader to check that the mathematical concepts are faithfully formalised in HOL4. Moreover, this cross-referencing enables HOL4 developers to use the mathematical descriptions as a documentation of the HOL4 theories. Besides the theoretical foundations, Chapter 3 also briefly mentions technical background, especially the consequence conversion and quantifier heuristic libraries.

Chapter 4 contains conclusions and future work. The index of HOL4 theorems can be found in the appendix. The appendix contains instructions on installing Holfoot and additional Holfoot example specifications as well.

Chapter 2

Holfoot

Holfoot is a formalisation of Smallfoot [2, 3] inside the HOL4 [13, 34] theorem prover. Smallfoot is an automated separation logic tool. It is able to reason about the partial correctness of programs written in a simple, low-level imperative language, which is designed to resemble C. This language contains pointers, local and global variables, dynamic memory allocation/deallocation, conditional execution, while-loops and recursive procedures with call-by-value and call-by-reference arguments. Moreover, concurrency is supported by conditional critical regions and a parallel composition operator. Smallfoot-specifications are concerned with the shape of datastructures in memory. Their content is not considered.

Smallfoot comes with a selection of example specifications. There are common algorithms about singly-linked lists like copying, reversing or deallocating them. Another set of examples contains similar algorithms for trees. There is an implementation of mergesort, some code about queues, circular-lists, buffers and similar examples.

Holfoot is a formalisation of Smallfoot. Thus, it can verify most Smallfoot examples completely automatically (see Appx. B.1). However, it extends Smallfoot to reason about the content of datastructures. Moreover, there is support for arrays and pointer arithmetic. Being aware of the data-content allows Holfoot to reason about fully-functional specifications. Simple fully-functional specifications like reversal of a singly-linked list can be verified automatically. For more complicated examples like a fully-functional specification of mergesort or insertion into a red-black-tree, Holfoot supports user-interaction (see Appx. B.2). In interactive mode, all the libraries and tools of HOL4 can be used.

This chapter contains a high level presentation of Holfoot. Many examples are used to illustrate Holfoot's features. This chapter does not explain Holfoot's semantic foundation or its implementation. Instead, the theoretical foundations and a few glimpses at its implementation are presented in Chapter 3.

2.1 Input Language

Holfoot can reason about the partial correctness of programs written in a simple, low-level imperative programming language. Its input consists of a list of specified procedures

written in this language. A simple example, which can be verified automatically by Holfoot, is a recursive implementation of determining the length of a singly-linked list (see Appx. B.1.2).

```
list_length(r;c) [data_list(c, cdata)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [ data_list (c, cdata) * (r == ``LENGTH cdata``)]
```

Smallfoot uses a very similar input language. The languages differ mainly in the type of specifications that are allowed. Holfoot allows talking about the data-content and supports arrays. Moreover, Holfoot provides a richer annotation language. In contrast, Smallfoot supports only predicates describing the shape of data-structures. However, there are predicates for double- and XOR-linked lists that are not supported by Holfoot.

2.1.1 States

Holfoot's programs operate on states that consist of a stack and a heap. Stacks are finite maps from variables to values and permissions. Permissions are read- or write-permissions. Values are natural numbers, i. e. non-negative integers including 0. Variables are just identifiers, i. e. they are represented as strings.

$$\begin{aligned} \mathit{Values} &\stackrel{\text{def}}{=} \mathbb{N}_0 \\ \mathit{Perms} &\stackrel{\text{def}}{=} \{\text{read}, \text{write}\} \\ \mathit{Vars} &\stackrel{\text{def}}{=} \mathit{Strings} \\ \mathit{Stacks} &\stackrel{\text{def}}{=} \mathit{Vars} \xrightarrow{\text{fin}} (\mathit{Values} \times \mathit{Perms}) \end{aligned}$$

A heap is a finite map from locations to named records of values. Locations are natural numbers excluding 0. The named record is represented as a map from tags to values. These tags are identifiers used to index the entry in the record. They are represented as strings.

$$\begin{aligned} \mathit{Locations} &\stackrel{\text{def}}{=} \mathit{Values} \setminus \{0\} = \mathbb{N} \\ \mathit{Tags} &\stackrel{\text{def}}{=} \mathit{Strings} \\ \mathit{Heaps} &\stackrel{\text{def}}{=} \mathit{Locations} \xrightarrow{\text{fin}} (\mathit{Tags} \to \mathit{Values}) \end{aligned}$$

Example 2.1.1. Figure 2.1 illustrates an example Holfoot state. There are stack variables `x`,`y` and `z` with some given values. The stack contains read-permission for the variables `x` and `y` and write-permission for `z`. The heap is allocated at four locations (22,65,34,12). For each of these locations the stored record consists of entries for the tags `l`, `r` and `dta`.

On the semantic level, the state contains a binary tree with root `x`. At each allocated location of the heap one node of the tree is stored. These nodes contain a pointer to their

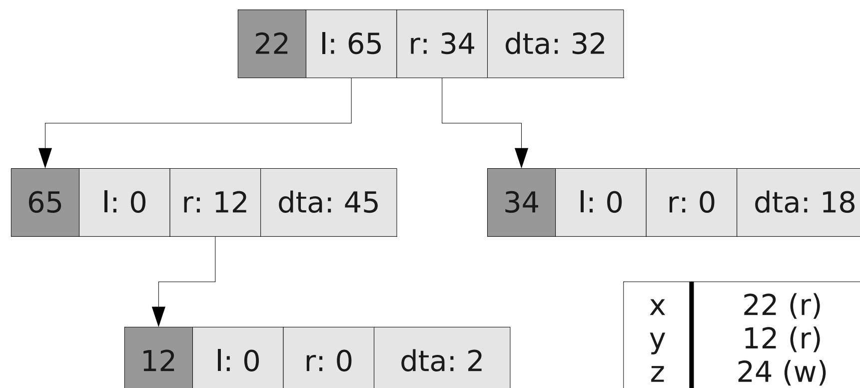


Figure 2.1: Holfoot example state 1

left and right children and a data element. These parts of the nodes are labeled with the tags `l`, `r` and `dta`, respectively. Notice, that the arrows in the diagram are just for illustration purposes.

2.1.2 Pure Expressions

All expressions supported by Holfoot are pure, i.e. they do not depend on the heap. Therefore, *pure expressions* in Holfoot are partial functions from stacks to values. They are partial, because the evaluation of a pure expression fails, if it can't access all the stack-variables it needs. Holfoot supports the following pure expressions:

`x` (variables)
: A variable-expression `x` looks up the value of `x` in the stack. If the variable is present, its value is returned. Otherwise the expression fails.

`NULL`, 0, 1, . . . , `#c` (constants)
: A constant always returns its value. `NULL` is a synonym for `0`. The expression `#c` allows to use a specification variable `c` as a constant.

`_c` (existentially quantified constant)
: `_c` represents an existentially quantified constant `c`.

e_1 `op` e_2 (binary operations)
: Binary operations like addition (`+`), monus (`-`), integer devision (`/`) and modulo (`%`) are supported between expressions. Notice, that expressions evaluate to natural numbers. Therefore monus and integer division are used.

`‘‘hol‘‘` (HOL4 expression)
: HOL4 quotations can be used to introduce additional expressions and especially additional operations between expressions. `hol` has to be a HOL4 term that evaluates to a natural number. Program variables that occur in `hol` are replaced with their value. To express for example the maximum of two program variables `x` and `y`, the expression `‘‘MAX x y‘‘` can be used.

`old(x)` (old value of a variable)
: `old` can under certain circumstances be used to get the value of a variable before some operation. It will be discussed later.

Example 2.1.2. Let $e \hat{=} v$ denote that the expression e is evaluated to the value v in the stack illustrated in Figure 2.1. Then the following pure expressions are evaluated as follows:

`x`	$\hat{=}$ 22	`12`	$\hat{=}$ 12	`x + w`	$\hat{=}$ fail
`y`	$\hat{=}$ 12	`#c`	$\hat{=}$ c	`x - 50`	$\hat{=}$ 0
`w`	$\hat{=}$ fail	`x + 1`	$\hat{=}$ 23	`2 * (x + 1)`	$\hat{=}$ 46
`NULL`	$\hat{=}$ 0	`x + y`	$\hat{=}$ 34	``` ``MAX x y`` ```	$\hat{=}$ 22

2.1.3 Predicates

Predicates on states are sets of states. They can also be seen as functions that given a stack and a heap return whether this state is accepted by the predicate. Some of these predicates describe datastructures in the heap. For the data content of these datastructures, *data expressions* are used. A data expression is either a constant `data`, an existentially quantified constant `_data` or a HOL4-quotation ``` ``hol`` ```.

`emp` **(empty heap)**

`emp` demands that the heap is empty, i. e. that no memory cell is allocated. The stack does not need to be empty! Arbitrary stacks are accepted.

`e`$_1$ `op e`$_2$ **(pure comparison)**

These predicates compare the values of two expressions. All stacks that satisfy this comparison are accepted; the heap has to be empty. Valid comparison operators are `==` (equal), `!=` (not equal), `<` (less), `<=` (less or equal), `>` (greater), `>=` (greater or equal). For parsing purposes, `e`$_1$ has to be a simple expression, i. e. it has to be either a variable or a constant.

``` ``hol`` ``` **(pure HOL4 predicate)**

This predicate is similar to a pure comparison. It allows, however, using HOL4 predicates. The heap has to be empty again and the stack has to satisfy the predicate described by `hol`. Notice, that `hol` may contain stack-variables. This predicate is very useful for expressing Boolean sideconditions.

`e |-> [t`$_1$`:e`$_1$`,t`$_2$`:e`$_2$`,...]` **(points-to)**

This predicate describes a single heap-cell. Any stack that is able to evaluate all the expressions is accepted. The heap needs to consist of a single cell at location `e`. The record stored at this location contains for tag `t`$_1$ the value of `e`$_1$, for `t`$_2$ the value of `e`$_2$, etc. Notice, that the record contains additional tags as well, since by definition it contains an entry for every tag.

For parsing purposes, `e` has to be a simple expression, i. e. it has to be either a variable or a constant. Since this is sometimes inconvenient, there is the alternative notation `pointsTo(e, [t`$_1$`:e`$_1$`,t`$_2$`:e`$_2$`,...])` that supports all expressions.

Example 2.1.3. Consider the state shown in Figure 2.2. This state satisfies the predicate `x |-> [r:y,l:65]`. The extended state shown in Figure 2.3 does not satisfy this predicate, however.
```

Figure 2.2: Holfoot example state 2

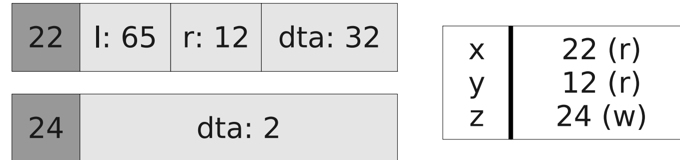


Figure 2.3: Holfoot example state 3

$\mathtt{p}_1$ `*` $\mathtt{p}_2$ **(separating conjunction)** The separating conjunction operator `*` requires that the state can be split into two separate parts such that one part satisfies the predicate $\mathtt{p}_1$ while the other satisfies $\mathtt{p}_2$. Heaps and stacks are split independently.

Splitting a heap h is easy. A subset of the locations of h form a new heap h_1, the remaining locations form the second heap h_2.

Splitting stacks is more complicated, since permissions have to be taken into consideration. Luckily, most of the time it is not necessary to split stacks. All predicates used by Holfoot only need read permissions on stack variables. Since a read-permission can be split into arbitrarily many read-permissions, the stack can be split into arbitrarily many parts that are equivalent to the original stack with respect to predicates. Thus, there is no need to split the stack. The original stack can be used for $\mathtt{p}_1$ and $\mathtt{p}_2$.

Example 2.1.4. Let's consider the state shown in Figure 2.3. This state does not satisfy the predicate `x |-> [r:y,l:65]`, because the location `z` is not described by this predicate. The state satisfies `x |-> [r:y,l:65] * z |-> [dta:2]`. Pure predicates can be added as well, since pure predicates take an empty part of the heap. So, `x |-> [r:y,l:65] * z |-> [dta:2] * x != y` is satisfied as well.

`data_lseg(tag;`$\mathtt{e}_1$`,dtag:data,`$\mathtt{e}_2$`)` **(singly-linked list segment)**
The predicate `data_lseg(tag;`$\mathtt{e}_1$`,dtag:data,`$\mathtt{e}_2$`)` describes a segment of a non-cyclic singly-linked list. The first node of the list-segment is stored in the heap at the location described by $\mathtt{e}_1$. Each node points to the next node via the tag `tag`. The last node points to the value of $\mathtt{e}_2$. If $\mathtt{e}_1$ and $\mathtt{e}_2$ evaluate to the same value, then the list-segment is empty. The list-segment contains the data `data` indexed by tag `dtag`.

Singly-linked lists are very common in Holfoot. Therefore, there are many abbreviations and variants:

- `tag` can be omitted. It then defaults to `tl`.
- `dtag` can be omitted. It then defaults to `dta`.
- `data_list(e,data)` is an abbreviation of `data_lseg(e,data,NULL)`. This represents a null terminated singly-linked list.

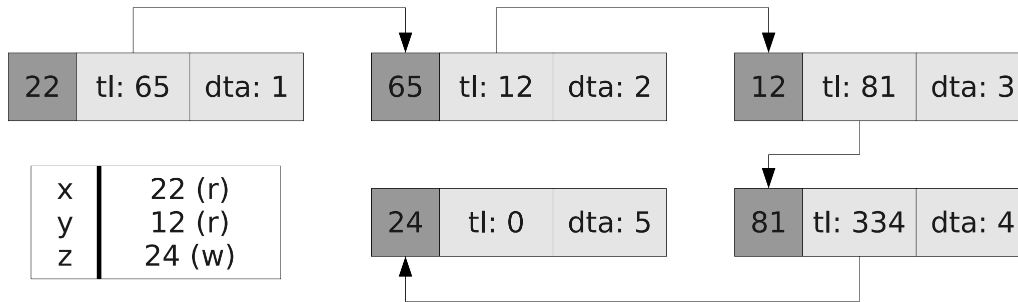


Figure 2.4: Holfoot example state 4

- `lseg(`e_1`,`e_2`)` describes a singly-linked list segment without specifying the data-content.
- `list(e)` is an abbreviation of `lseg(e,NULL)`. So, it represents a null terminated singly-linked list without data.

Example 2.1.5. Consider the state shown in Figure 2.4. This state satisfies the predicate `data_list(x,‘‘[1;2;3;4;5]‘‘)`. The predicate `list(x)` that does not specify the data-content of the list holds on this state as well. Moreover, the predicate `data_lseg(x,‘‘[1;2]‘‘,y) * data_list(y,‘‘[3;4;5]‘‘)` is satisfied.

`data_tree(tagL;e,dtagL:data)` **(n-array trees)**

The predicate `data_tree(tagL;e,dtagL:data)` describes a tree. The root of the tree is stored in the heap at the location described by `e`. Each node points to its children via the tags in `tagL`. `NULL` is used to denote leaves. Each node contains the data indexed by the tags in `dtagL`.

The argument `data` is a functional representation of the tree. It is either a leaf `leaf` or a node `node valueL treeL`. The list `valueL` contains the values indexed by `dtagL`. `treeL` is a list of functional representations of the trees, whose roots are indexed by `tagL`.

Trees are common in Holfoot, especially binary ones. Therefore, there are many abbreviations and variants:

- `tagL` can be omitted. It then defaults to `[l,r]`.
- `dtagL` can be omitted. It then defaults to `[dta]`.
- `tree(`t_1`,`t_2`,e)` is a binary tree without data that uses that tags t_1 and t_2 to point to child nodes.
- `tree(e)` is an abbreviation of `tree(l,r,e)`.

Example 2.1.6. Consider the state shown in Figure 2.1. This state satisfies the predicate `tree(x)`. Considering the data-content, the state is as well described by

```
data_tree(x, ‘‘node [32] [node [45] [leaf; node [2] [leaf; leaf]];
                         node [18] [leaf; leaf]]‘‘)
```

22	dta: 1
23	dta: 2
24	dta: 3

x	22 (r)
y	12 (r)
z	24 (w)

Figure 2.5: Holfoot example state 5

data_array(e_1,e_2,dtag:data) **(array)**

The predicate data_array(e_1,e_2,dtag:data) describes an array. The expression e_1 evaluates to the starting location l, the expression e_2 to the length n. The array is stored in the heap at locations l, $l+1$, $\ldots l+(n-1)$. The array contains the data data in these heap-locations indexed by the tag dtag.

There are many abbreviations and variants of arrays:

- dtag can be omitted. It then defaults to dta.
- array(e_1,e_2) describes an array without specifying its data content.
- data_interval(e_1,e_2,dtag:data) uses the last location of the array instead of its length. It is defined as data_array(e_1,(e_2+1)-e_1,dtag:data).
- interval(e_1,e_2) describes an interval without specifying its data content.

Example 2.1.7. Consider the state shown in Figure 2.5. This state satisfies the predicates data_array(x,3,‘‘[1,2,3]‘‘) and array(x,3). The state is described by data_interval(x,z,‘‘[1,2,3]‘‘) as well.

if (e_1 == e_2) then $pred_1$ else $pred_2$ **(conditional predicate)**

This predicate checks whether the expressions e_1 and e_2 evaluate to the same value. If this is the case the predicate $pred_1$ has to hold, otherwise $pred_2$ is required to hold. If the else part is omitted, $pred_2$ defaults to emp.

if (e_1 != e_2) then $pred_1$ else $pred_2$ **(conditional predicate)**

This predicate is equivalent to if (e_1 == e_2) then $pred_2$ else $pred_1$.

map (\v_1 ... v_n. pred) data **(separating map)**

This predicate describes mapping the predicate pred over the list data and combining all resulting predicates using the separating conjunction operator *.

Example 2.1.8. data_array(x,3,‘‘[1,2,3]‘‘) describes an array (see Fig. 2.5). This array can also be described by pointsTo(x + 0, 1) * pointsTo(x + 1, 2) * pointsTo(x + 2, 3) and therefore by

```
map (\n v.  pointsTo(x + #n, #v)) ‘‘[(0,1);(1,2);(2,3)]‘‘
```

.

2.1.4 Statements

Holfoot supports the following statements:

`x = e` **(Assignment)**
: evaluates the expression `e` and stores the result in stack-variable `x`. The assignment fails, if `e` cannot be evaluated or there is no write-permission for `x`.

`x = e->t` **(Heap Look-Up)**
: `e` is evaluated and interpreted as a heap-location l. The value indexed by tag `t` at this location l is looked up and stored in variable `x`. The execution of this statement fails, if l is not allocated in the heap, the stack does not contain write permission for `x` or the expression `e` cannot be evaluated.

$\mathtt{e}_1$`->t =` $\mathtt{e}_2$ **(Heap Assignment)**
: $\mathtt{e}_1$ is evaluated and interpreted as a heap-location l. The value indexed by tag `t` at this location l is updated by the value of expression $\mathtt{e}_2$. This statement fails, if $\mathtt{e}_1$ or $\mathtt{e}_2$ cannot be evaluated or if l is not allocated in the heap.

`x = new(e)` **(Memory Allocation)**
: The expression `e` is evaluated and interpreted as a size s. Then s consecutive heap cells are allocated and the location of the first cell is stored in variable `x`. This means that after the successful execution of this statement, the locations `x`, $\mathtt{x} + 1$, ... $\mathtt{x} + (s - 1)$ have been added to the heap. This command fails, if `e` cannot be evaluated or there is no write-permission for `x`.

`x = new()` **(Memory Allocation of Single Cell)**
: shorthand for `x = new(1)`

`dispose(`$\mathtt{e}_1, \mathtt{e}_2$`)` **(Memory Deallocation)** $\mathtt{e}_1$ and $\mathtt{e}_2$ are evaluated. $\mathtt{e}_1$ is interpreted as a heap location l and $\mathtt{e}_2$ as a size s. Then s consecutive heap cells starting at location l are deallocated, i.e. the locations $l, l + 1, \ldots l + (s - 1)$ are removed from the heap. This command fails, if $\mathtt{e}_1$ or $\mathtt{e}_2$ cannot be evaluated or if one of the locations $l, l + 1, \ldots l + (s - 1)$ is not allocated in the heap.

`dispose(e)` **(Memory Deallocation of Single Cell)**
: shorthand for `dispose(e,1)`

Remark 2.1.9. Notice that a statement of the form $\mathtt{e}_1$`->`$\mathtt{t}_1$ `=` $\mathtt{e}_2$`->`$\mathtt{t}_2$ does not exist and that for example $\mathtt{e}_1$`->`$\mathtt{t}_1 + 1$ is not a valid expression.

There are also constructs which are technically statements, but which are used for specification purposes. These constructs include:

`assume(cond)` **(Assume)**
: `assume(cond)` assumes that the condition `cond` holds. If this condition holds in the current state, `assume(cond)` skips. Otherwise it diverges.

`assert(pred)` **(Assert)**
: `assert(pred)` asserts that a substate of the current state satisfies the predicate `pred`. If this is the case, it skips. Otherwise it fails.

`diverge`, `fail`
: `diverge` always diverges. `fail` fails.

2.1.5 Conditions

These statements are extended to programs by adding sequential composition, control structures, procedure calls and conditional critical regions. Before these can be presented, conditions for the control structures have to be introduced first.

Holfoot supports the following conditions:

`true`, `false`
: the constant true and false conditions

e_1 `op` e_2 **(Comparison)**
: This condition compares the values of two expressions. Valid comparison operators are `==` (equal), `!=` (not equal), `<` (less), `<=` (less or equal), `>` (greater), `>=` (greater or equal). For parsing purposes, e_1 has to be a simple expression, i. e. it has to be either a variable or a constant. If either e_1 or e_2 cannot be evaluated, the evaluation of the condition fails.

`not(cond)`, $cond_1$ `and` $cond_2$, $cond_1$ `or` $cond_2$ **(Boolean Operators)**
: The Boolean operators `not`, `and` and `or` can be used to combine conditions.

`‘‘hol‘‘` **(HOL4 Condition)**
: Quoted HOL4 terms that may contain program variables can be used as conditions.

2.1.6 HOL4 Syntax

Expressions, conditions and data expressions can contain quoted HOL4 terms. These terms can use arbitrary HOL4 constructs. They may even contain user-defined functions. An introduction to HOL4 and its libraries is outside the scope of this presentation. Here just a few of the most common constructs are listed:

- `/\`, `\/`, `~` and `==>` denote the Boolean conjunction, disjunction, negation and implication operators.
- Universal quantification is denoted by '`!`', existential quantification by '`?`'. The term `!x y. P x y` for example means that `P` holds for all arguments `x` and `y`.
- `[]` denotes the empty list, `[e1;e2;e3]` the list with 3 element `e1`, `e2` and `e3`.
- `e::l` denotes the list consisting of an element `e` followed by a list `l`.
- `EL n l` denotes the `n`-th element of the list `l`. As counting starts with 0 the term `EL 1 [0;1;2]` evaluates to `1`.
- `MEM e l` denotes that `e` is an element of the list `l`.
- `l1 ++ l2` denotes appending the lists `l1` and `l2`.
- `MAP f l` denotes mapping the function `f` over the list `l`.
- `EVERY P l` denotes that the predicate `P` holds for all elements of `l`.
- `PERM l1 l2` denotes that `l1` and `l2` are permutations of each other.
- `SORTED $<= l` denotes that `l` is sorted according to the less or equal relation.

2.1.7 Programs

Holfoot supports the following programs:

`s` **(single statements)**
A statement is a program.

`p`$_1$`;` `p`$_2$`;` `...` `p`$_n$`;` **(sequential composition)**
The sequential composition of programs is a program.

`if cond then prog`$_1$ `else prog`$_2$ **(conditional execution)**
There is conditional execution. The else-clause is optional.

`if (*) then prog`$_1$ `else prog`$_2$ **(non-deterministic choice)**
A non-deterministic choice construct is available.

`while cond prog` **(while loop)**
There are while loops.

`with r when cond prog` **(conditional critical region)**
A conditional critical region waits until it can acquire the lock that protects resource `r` and until the condition `cond` holds. Then program `prog` is executed and the lock released.

`procedureName(refArgs;valArgs)` **(procedure call)**
Holfoot supports procedure calls. Procedures posses call-by-reference and call-by-value arguments. Call-by-reference arguments have to be variables, call-by-value arguments expressions. These expressions are evaluated and the procedure called with the resulting values.

`procName`$_1$`(rargs`$_1$`;vargs`$_1$`)` || `procName`$_2$`(rargs`$_2$`;vargs`$_2$`)` **(parallel procedure call)**
Procedures can be called in parallel. The call-by-value arguments of both procedure calls are evaluated before the concurrent execution starts.

2.1.8 Specifications

Programs provide control structures, procedure calls and operations for concurrency. However, it remains to be seen, how to declare procedures and locks. Moreover, these procedures need to be specified. Holfoot's input are *specifications*. A specification consists of a list of resource and procedure declarations. Procedure declarations are of the form:

```
assume procedureName(refArgs;valArgs) [w/r: rwvars] [precondition] {
  local vars;
  prog;
} [postcondition]
```

This declares a new procedure with the given name, arguments and local variables. The local variable declaration, the explicit declaration of variable permissions and the qualifier `assume` are optional. The procedure is specified by the pair of predicates `precondition`, `postcondition`. Additionally, there is the implicit precondition that all call-by-reference

arguments are distinct from each other. If the procedure is executed in a state satisfying its precondition, this specification requires that it does not fail. The execution of the procedure may diverge, but if it terminates, the resulting state has to satisfy the post-condition. The procedure-name `init` is reserved. It is only used to initialise resource invariants.

Resource-declarations are expressed by:

```
resource resName (varList) [invariant]
```

This declares a new resource/lock with the given name. The resource protects a state that satisfies the given invariant and has exclusive access (write access) to the variables in `varList`.

2.2 Introductory Examples

So far, the syntax and some rough ideas about the meaning of the input language have been presented. However, the meaning of the top-level input, i. e. of specifications has not been discussed in detail yet. In this section, examples are used to illustrate this meaning.

2.2.1 Recursive Implementation of List-Length

First, let's consider a slightly simpler specification of the list-length algorithm used as an introductory example (see Sec. 2.1 and Appx. B.1.2):

```
list_length(r;c) [list(c)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [ list (c)]
```

This specification can be handled by Smallfoot. It declares a single procedure. This procedure `list_length` gets two arguments. `r` is a call-by-reference argument used as the return value. `c` is a call-by-value argument. Moreover, the procedure declares a local variable `t`.

The precondition states that `c` contains the start location of a null terminated singly-linked list. This list is empty if and only if `c` equals `NULL`. If `c` equals `NULL`, `0` is assigned to `r`. Otherwise, the location of the next node is stored in variable `t` via the heap-lookup `t = c->tl`. Then, the procedure is called recursively to determine the length of the list starting at `t`. Finally, the result of this recursive call is incremented using the assignment `r = r + 1`. The postcondition guarantees that at the end there is still a list starting at `c`. Implicitly it is also guaranteed that no faults like accessing an unallocated heap-location occur. However, termination is not proved.

2.2.1.1 Local reasoning

Looking closer, the specification of the list-length algorithm states even more. `list(c)` describes a state that contains only a list starting at `c`. Except for this list, no other locations are allocated in the heap. Therefore, the specification guarantees that no junk is left by the list-length procedure on the heap.

However, this tight specification might be surprising with respect to the recursive procedure call. For verification purposes, a procedure-call is abstracted by the procedure specification. Before calling `list_length(r;t)` in the running example, it was determined that the list starting at `c` is not empty. Moreover, the location of the next node of the list was assigned to the local variable `t`. Thus, the current state before the recursive procedure call `list_length(r;t)` can be described by `c |-> t * list(t)`. According to the original specification, this call satisfies the Hoare triple {`list(t)`} `list_length(r;t)` {`list(t)`}. However, `c |-> t * list(t)` does not imply the precondition `list(t)` of this specification. The heap is too large!

As a separation logic tool, Holfoot supports local reasoning. Any triple {`P`} *prog* {`Q`} can be extended by a context `R` to {`P * R`} *prog* {`Q * R`}. Expressed differently, the programming language of Holfoot is designed in such a way, that statements fail if they cannot access all the resources, i. e. all the stack-variables and heap-locations, that might influence their behaviour or might be influenced by them. This means that if {`P`} *prog* {`Q`} holds, all the relevant resources are described by `P`. Any state separate to the one described by `P` can safely be added as a context.

In the running example {`list(t)`} `list_length(r;t)` {`list(t)`} can be extended to {`list(t) * c |-> t`} `list_length(r;t)` {`list(t) * c |-> t`}. This allows the procedure to be called recursively. Finding a frame R such that $P * R$ implies Q for given predicates P and Q is an essential operation in Holfoot. Besides other uses, it is used to reason about procedure calls and conditional critical region.

2.2.1.2 Read/Write Permissions

I claimed before that most of the time one does not need to consider the exact semantics of the spatial conjunction operator $*$ on stacks (see introduction of $*$ in Sec. 2.1.3). In particular, I claimed that read and write-permissions can often be ignored. Here, they are important. The frame `R` has to be separate from the original precondition. If this precondition requires exclusive access to a stack-variable, i. e. a write permission, then the frame is not allowed to mention this variable.

The condition that the frame is not allowed to mention variables with write-permissions will be formally justified later. Here, lets just try to understand the intuition behind this condition. If a procedure needs write-permission to a call-by-reference argument, it might update this argument. Therefore, this variable has in general a different value before and after the execution of the procedure. A frame describes an unmodified part of the state. Because the variable is changing, it must not be used by the frame.

The necessary permissions on stack variables are usually not specified explicitly. Holfoot is normally able to determine the necessary permissions automatically during parsing by examining the body of the procedure. Permissions can, however, be explicitly specified using a declaration of the form `[w/r: write-var list; read-var list]` before the precondition.

2.2.1.3 Internal Representation

Holfoot uses HOL terms as its internal representation. This internal representation contains the necessary permissions on stack variables explicitly. Moreover, it contains other information gathered during parsing. Most prominently, while call-by-value arguments are handled like stack-variables in the input language, the internal representation regards them as constants. In order to use them like variables in the procedure body, new local variables are introduced and appropriately initialised. The internal representation also removes the `old` construct by introducing new specification variables. Furthermore, implicit arguments like tags are made explicit.

Otherwise, the internal representation looks similar to the input language. There are some minor differences, though. The equality check `c == NULL` is for example in the internal representation written as `c = 0`. These differences exist due to friction between Smallfoot's and HOL's syntax. The input language is designed to be compatible with Smallfoot. Therefore, it has to use `==`. Since Holfoot is implemented inside HOL, the internal representation is a HOL term. This term is pretty-printed to resemble the input language. Using HOL's infrastructure, it is much easier to pretty-print the equality check as `c = 0`.

The internal representation of our running example is:

```
list_length(r; c_const) [w/r: r; |  list ( tl ; #c_const)] {
   local (c = c_const), t;
   if (c = 0) {
      r = 0
   } else {
      t =  ctl ; list_length(r; t); r = (r + 1);
   }
} [w/r: r;   |  list ( tl ; #c_const)]
```

The first step in verifying this specification consists of eliminating recursive procedure calls by replacing them with their specification. Similarly, the lock invariants are incorporated into the program. This results in a conjunction of Hoare triples. Notice, that comments get automatically introduced and maintained that show the origin of the Hoare triples.

```
[[w/r: r; |  list ( tl ; #c_const)]]
   /* list_length */
   local (c = c_const), t;
   if (c = 0) {
      r = 0
   } else {
      t = c->tl; abstracted list_length(r; t) ; r = (r + 1);
   }
[[w/r: r; |  list ( tl ; #c_const)]]
```

During the process of verifying these Hoare triples, frame inference predicates are likely to occur. The recursive procure call in the running example triggers for example the following frame inference calculation:

```
/* list_length − case (not (c = 0)) 3 − abstracted list_length (r; t) − final */
[[w/r: t, c, !r;  |
   (r = #r_const) * (c = #c_const) * (t = #c_const_tl) |
```

```
  #c_const |-> tl: #c_const_tl * list (tl; #c_const_tl) -->
   list (tl; #c_const_tl) | ... ]]
------------------------------------
  c_const <> 0
```

Let *context* be `(r = #r_const) * (c = #c_const) * (t = #c_const_tl)`. Further, let P denote `#c_const |-> tl: #c_const_tl * list(tl; #c_const_tl)` and Q be `list(tl; #c_const_tl)`. Then this statement searches for a frame R such that *context* $*$ P implies *context* $*$ Q $*$ R. Furthermore, the state has write-permission to the variables `t`, `c` and `r`. The exclamation mark before `r` denotes that the frame is not allowed to refer to `r`.

2.2.1.4 Fully-Functional Specifications

The running example of recursively determining the length of a singly-linked list can automatically be verified using Smallfoot or Holfoot. Its specification is quite weak though. It just guarantees that if there is a list at the beginning, there is a list at the end and no errors occur during execution. It is not specified that the length of the list is calculated. It is not even stated that the original list is preserved.

In order to write a fully-functional specification of the list-length procedure, one needs to talk about the data-content of the list. This is not possible using Smallfoot. Holfoot however, can automatically verify a fully-functional specification:

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [ data_list (c,cdata) * (r == ``LENGTH cdata``)]
```

Compared with the previous example, just the pre- and postcondition changed. The precondition now states that the list starting at `c` contains some data `cdata`. Furthermore, the postcondition states that the list contains still the same data and `r` now contains the length of `cdata`. Thus, this is a fully-functional specification. Notice, how the value of `r` is expressed. ``r == ``LENGTH cdata`` `` requires an empty heap. Therefore, it is combined with the rest of the postcondition using spatial conjunction instead of normal conjunction as one might expect. The HOL-quotation ``` ``LENGTH cdata`` ``` is used to calculate the length of the list using the HOL function `LENGTH`. `cdata` is free specification variable. Therefore, it is implicitly universally quantified. The specification holds for all possible values of `cdata`.

2.2.2 Pointer Transferring Buffer Example

A good example to illustrate the usage of resources, conditional critical regions and parallel procedure calls is a simple implementation of a pointer transferring buffer (see Appx. B.1.11):

```
resource buf (c) [if c==NULL then emp else c|->]

init() { c = NULL; }
put(x) [x|->] { with buf when (c==NULL) { c = x; } } [emp]
get(y;) [emp] { with buf when (c!=NULL) { y = c; c = NULL; } } [y|->]
putter() [emp] { local x; x = new(); put(x); putter(); } [emp]
getter() [emp] { local y; get(y;); dispose(y); getter(); } [emp]
main() [emp] { putter() || getter(); } [emp]
```

The example declares a resource `buf`. This resource has write-access to a variable `c` and protects a part of the state that satisfies `if c==NULL then emp else c |->`. This means that the heap is empty, if `c` is 0. Otherwise the heap contains a single location `c`.

The special procedure `init` is used to initialise all lock-invariants. `init` has no arguments and no explicit specification. Implicitly, its precondition states that it has write-access to all variables used by locks and that the heap is empty. The postcondition is the combination of all lock-invariants. There has to be a `init` procedure present if resources are used.

Besides the `init` procedure there are the procedures `put` and `get`. `put` puts a heap-cell into the buffer, `get` gets a cell out of the buffer. Towards this end, conditional critical regions are used. `get` tries to acquire the resource `buf` when the buffer is not empty (condition `c != NULL`). When the critical region is entered, access is granted to the protected state, i. e. the state described by `c |->` . The value of variable `c` is copied to `y` and `c` is set to `NULL`. When leaving the conditional critical region, the resource `buf` therefore protects the empty heap. Access to the location that was stored in the buffer remains with `get`. `put` works similarly. It acquires the resource `buf` when the buffer is empty and puts a location into the buffer.

The procedures `put` and `get` are extended to procedures `putter` and `getter` which constantly allocate new heap locations and put them into the buffer or get them out of the buffer and deallocate them. Finally, the procedure `main` calls these procedures in parallel. All three procedures `putter`, `getter` and `main` have very simple specifications. They need to access only the empty heap and ensure that the heap stays empty.

Notice, that in this example the memory allocation and deallocation statements `new` and `dispose` are used for the first time. Moreover, notice, that `getter`, `putter` and `main` do not terminate. Smallfoot and Holfoot are just interested in partial correctness.

2.3 Annotating While-Loops

Until now, the examples did not contain loops. Holfoot requires while-loops to be annotated in order to reason about them.

2.3.1 Loop Invariants

The most common annotation of loops is an invariant. Loop-invariants in Holfoot have their common meaning. A loop-invariant has to hold before the loop and after each loop iteration. When the loop is exited, one therefore knows that the invariant still holds and

that the condition of the loop does not hold. Let's consider an iterative implementation of the list-length example (see Appx. B.1.2).

```
list_length(r;c) [list(c)] {
  local t;
  r = 0; t = c;
  while (t != NULL) [lseg(c, t) * list (t)] {
    t = t->tl; r = r + 1;
  }
} [ list (c)]
```

In the concrete example, the loop is used to move through the list starting at `t`. The variable `t` is initialised with `c`. It is then updated to point to the tail of the list, until the list is empty. Therefore, the while-loop of this example can be annotated with the invariant `lseg(c,t) * list(t)`. There is always a list-segment from `c` to `t` and a list starting at `t`. At the beginning, `t` equals `c`. Thus the list-segment is empty and the list describes the whole state. In each loop-iteration `t` is updated to point to the next node of the list. Thus, one node is removed from the list and added to the list-segment. The loop terminates, if `t` equals `NULL`, i. e. if the list starting at `t` is empty. Then the list is empty and the whole state is described by the list-segment.

This loop-invariant is already complicated enough. It becomes even worse, if a fully-functional specification is used:

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  r = 0; t = c;
  while (t != NULL)
      [ data_lseg (c, _cdata1, t) * data_list (t, _cdata2) *
       (r == ``LENGTH _cdata1``) * ``cdata = _cdata1 ++ _cdata2``] {
    t = t->tl; r = r + 1;
  }
} [ data_list (c,cdata) * (r == ``LENGTH cdata``)]
```

Now the invariant states that there exists some data `cdata1` and `cdata2` (existential quantification is denoted by the underscore) such that the list-segment contains `cdata1` and the list `cdata2`. The variable `r` contains the length of `cdata1`. Moreover, appending `cdata1` and `cdata2` results in some list `cdata`. The existential quantification of `cdata1` and `cdata2` means that different values can be chosen for each iteration of the loop. In contrast, `cdata` is a free specification variable and therefore implicitly universally quantified. It can once be chosen, but then stays the same for all loop-iterations.

2.3.2 Loop Specifications

Remember the specification of a recursive implementation of the same algorithm (see Sec. 2.2.1.4 or Appx. B.1.2). The recursive and the interactive implementation have the same interface with exactly the same procedure specifications. However, while this specification is sufficient for the recursive implementation, the iterative one needs to be annotated with a complicated loop invariant. This invariant is not just lengthy and complicated, it even needs additional concepts. Only the invariant needs to talk about list-segments, a partial datastructure. The reason why the loop-invariant is so complicated is that loop invariants in contrast to recursive procedure calls do not exploit local reasoning.

So, if procedure calls can exploit local reasoning, lets translate the loop into a recursive function. A program `while cond prog1; prog2` can easily be translated in a new recursive function `whilefun` with body `if cond { prog1; whilefun } else prog2`. In the running example, we can translate the loop into an recursive function with a much simpler specification:

```
whilefun(t,r;) [data_list(t, data)] {
  if (t != NULL) {
    t = t->tl; r = r + 1;
    whilefun(t,r;);
  }
} [ data_list (old(t), data) * (r == "LENGTH data + old(r)")]

list_length(r;c) [data_list(c,cdata)] {
  local t;
  r = 0; t = c;
  whilefun (t,r;);
} [ data_list (c,cdata) * (r == "LENGTH cdata")]
```

The recursive function has a similar specification as the main one. It just considers the list starting at `t` and adds the length of this list to the value of `r`. The list-segment between `c` and `t` disappears. It is implicitly handled by local reasoning.

Of course, one does not want to perform such transformations explicitly. Instead, the ideas of this transformation are used to introduce a new annotation for while-loops. These annotations are called loop-specifications. I presented this idea at the VSTTE'10 theory workshop [36].

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  r = 0; t = c;
  loop_spec [ data_list (t, data)] {
    while (t != NULL) {
      t = t->tl; r = r + 1;
    }
  } [ data_list (old(t), data) * (r == "LENGTH data + old(r)")]
} [ data_list (c,cdata) * (r == "LENGTH cdata")]
```

2.3.3 Examples

For the example of calculating the length of a singly-linked list loop-specifications are advantageous. There are similar results for reversing a singly-linked list (see Appx. B.1.3), copying a singly-linked list (see Appx. B.1.4), appending two singly-linked lists (see Appx. B.1.5), removing an element from a singly-linked list (see Appx. B.2.3), etc. These examples are all very similar to the list-length example. Therefore, they are just listed in the appendix, but not discussed here in detail.

Instead of considering such examples that are all very similar, let's discuss the general properties of loop-specifications. Loop-specifications were introduced in order to exploit local reasoning. However, even without local reasoning they are still useful. In contrast to invariants, the pre- and post-condition specify the behaviour of the block containing

the while loop. Therefore, loop-specifications are closely related to Eric Hehner's *specified blocks* [16]. Hehner uses single Boolean expressions instead of a pre- and postcondition. Moreover, his work is much more general. However, he is not using local reasoning. Allowing for these differences, his method of reasoning about loops is very similar to the one proposed here.

2.3.3.1 Array Increment Example

Similar to Hehner's specified blocks, loop specifications slightly change how to think about loops. As a rule of thumb, loop invariants express what the loop has already done, whereas loop specifications express what it will still do. Talking about what still needs doing instead of what has already been done, often leads to more natural specifications. Even without local reasoning, Hehner prefers loops specified as blocks to invariants. He claims *that it is simpler and more direct to say what's left to be done, rather than to formulate an invariant* [16]. This difference between loop invariants and loop specifications is demonstrated by one of Hehner's examples (see Appx. B.2.9):

```
inc(;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   while (i < n) {
      tmp = (a + i) -> dta; (a + i) -> dta = tmp + 1;
      i = i + 1;
   }
} [data_array (a,n,‘‘MAP SUC data‘‘)]
```

This procedure increments every element of an array. The loop can be specified with the following invariant:

```
data_array(a, n, _data2) *
‘‘(!id. id < i            ==> (EL id data2 = SUC (EL id data))) /\
  (!id. i <= id /\ id < n ==> (EL id data2 =      EL id data ))‘‘
```

This invariant states that there is an array of length `n` starting at `a` and containing some existentially quantified data `data2`. For all indices up to `i` the array contains the incremented value, for all other indices it still contains the original one. If a loop specification is used, it is the other way round:

pre: `data_array(a,n,data)`
post:

```
data_array(a, n, _data2) *
‘‘(!id. id < old(i)            ==> (EL id data2 =      EL id data )) /\
  (!id. old(i) <= id /\ id < n ==> (EL id data2 = SUC (EL id data)))‘‘
```

This specification states that all the indices starting at the value of `i` will be updated, while all smaller than `i` are not touched. Notice, that no local reasoning is involved here, yet. Using local reasoning, the loop specification can however be simplified by implicitly handling the part of the array that is not touched.

pre: `data_array(a+i,n-i,data)`
post: `data_array(a+old(i), n-old(i), ‘‘MAP SUC data‘‘)`

This specification now states that given an array starting from a + i of length n − i − i. e. just the part of the original array starting at index i – all elements of this array are incremented. There is no need any more for some complicated expressions about indices.

2.3.3.2 List Filtering Example

The last example demonstrates that loop invariants usually specify what has already been done, whereas loop specifications specify what will be done. However, both views were easy to express. The following example of filtering a list (see Appx. B.1.7 and Appx. B.2.5) demonstrates that it might be much simpler to express what the loop will still do. Notice that this example is not exploiting local reasoning.

```
list_filter(l;x) [data_list(l,data)] {
  local y, z, e;
  y = l; z = NULL;
  while(y != NULL) {
    e = y->dta;
    if (e == x) {
      if (y == l) {
        l = y->tl; dispose y; y = l;
      } else {
        e = y->tl; z->tl = e; dispose y; y = z->tl;
      }
    } else {
      z = y; y = y->tl;
    }
  }
} [ data_list (l, ``FILTER (\n:num. ~(n = x)) data``) ]
```

The loop invariant describes that parts of the list got already filtered. This partial filtering is complicated to express:

```
if (y == l) then
   data_list(l,_data1) *
   ``?data_fc. (EVERY (\n. n = x) data_fc) /\
               (data = data_fc ++ _data1)``
else
   data_lseg(l,``FILTER (\n:num. ~(n = x)) _data1``,z) *
   z |-> [tl:y, dta:_date] * (_date != x) * data_list(y,_data2) *
   ``?data_fc. (EVERY (\n. n = x) data_fc) /\
               (data = _data1 ++ _date::(data_fc++_data2))``
```

If the filtering still happens at the beginning of the list (y equals l), then there is a list containing some data1. The original data data can be expressed as an arbitrary number of x's followed by data1. If the filtering currently happens inside the list, then there is a list-segment from l to z that is already completely filtered. z is itself filtered, i. e. at this location a different value than x is stored. Furthermore, z points to the still unfiltered part of the list starting at y. The original data can be expressed as a part that when filtered results in the data present in the list-segment from l to z, followed by the data stored at location z, an arbitrary number of x's and finally the data of the list starting

at y. Not using Holfoot's notation, the invariant can be described by:

$$\begin{array}{l}\textbf{if } (\mathtt{y} = \mathtt{l}) \textbf{ then} \\ \quad \exists data_1.\ (data = (\text{some } \mathtt{xs}) + data_1) * \text{list}(\mathtt{l}, data_1) \\ \textbf{else} \\ \quad \exists data_1, date, data_2.\ \ (data = data_1 + date + (\text{some } \mathtt{xs}) + data_2) * \\ \qquad \text{lseg}(\mathtt{l}, \text{filtered } data_1, \mathtt{z}) * (z \mapsto [\mathtt{tl} : \mathtt{y},\ \mathtt{dta} : date]) * \\ \qquad date \neq \mathtt{x} * \text{list}(\mathtt{y}, data_2)\end{array}$$

In contrast to this complicated loop invariant, the loop specification is straightforward, because it describes that the whole list starting at y will be filtered.

```
pre:   data_list (y, data2) *
       (if (y != l) then data_lseg (l, data, z) * (z |-> tl:y,dta:#zdata))
post:  if (old(y) == old(l)) then
          data_list(l, ``FILTER (\n. ~(n = x)) data2``)
       else
          (data_list(l, ``data ++ [zdata] ++
                              (FILTER (\n. ~(n = x)) data2)``))
```

2.3.3.3 List Copy Example

After considering examples for which loop specifications proved beneficial even without local reasoning, let's have a look at another example with local reasoning (see Appx. B.1.4):

```
list_copy(z;c) [data_list(c,data)] {
  local x,y,w,d;
  if (c == NULL) {
    z=NULL;
  } else {
    z=new(); z->tl=NULL; x = c->dta; z->dta = x;
    w=z; y=c->tl;
    while (y!=NULL) {
      d=new(); d->tl=NULL; x=y->dta; d->dta=x;
      w->tl=d; w=d; y=y->tl;
    }
  }
} [ data_list (c,data) *  data_list (z,data)]
```

This procedure copies a singly-linked list that starts at c and updates the call-by-reference argument z such that z points to the copy after execution. The procedure first checks, whether the list is empty. In this case, nothing needs to be copied. Otherwise, the first element is copied and auxiliary variables w and y initialised. After this initialisation, z points to the beginning of the copy, w points to its last element and y points to the part of the original list that still needs to be copied. Then a while loop is used to copy the remainder of the list by copying the element pointed to by y and then advancing y and w.

The while-loop can be specified with the following invariant:

```
data_lseg(c,``_data1++[_cdate]``,y) * data_list(y,``_data2``) *
data_lseg(z,_data1,w) * w |-> tl:0,dta:_cdate *
``data:num list = _data1 ++ _cdate::_data2``
```

This invariant states that the original data can be split into three parts: two lists `data1`, `data2` and a single element `cdate`. There is a list-segment from `c` to `y` containing `data1` followed by `cdate`. This part of the original list has already been copied. The data `data1` has been copied to a list-segment from `z` to `w`. The last entry `cdate` is stored at location `w`. Finally, `data2` still needs to be copied. It is stored in a list starting at `y`. Using a loop specification simplifies reasoning about the loop significantly:

pre: `w |-> [tl:0,dta:#date] * data_list (y, data2)`

post: `data_list(old(w), ‘‘date::data2‘‘) *  data_list(old(y), data2)`

This specification states that if before the loop is executed `w` points to some data `date` and there is a list starting at `y` containing `data2`, then the list starting at `y` is copied such that the old value of `w` points to a list containing `date` followed by `data2` after the execution of the loop. The part of the list that has already been copied, i. e. the list-segment from `c` to `y` does not need to be mentioned explicitly. It is handled implicitly using local reasoning.

2.3.3.4 Partial Datastructures

Loop specifications can utilise local reasoning in order to implicitly handle some parts of the state that loop invariants mention explicitly. These implicitly handled parts of the state usually consist of partial datastructures. For the examples so far, these partial datastructures are easy to express. For lists, the corresponding partial datastructure is a list-segment and for arrays it is an array. Let’s now consider a slightly more complicated datastructure: trees. For trees, the corresponding partial datastructure is a tree with a hole for some other tree. This is difficult to express. Separation logic’s magic-wand operator can be used, but reasoning about this additional operator is not straightforward and Holfoot is not able to do it. Therefore, Holfoot usually can’t handle the invariants of loops that operate on trees. However, loop specifications can be used to avoid the partial datastructure. This allows Holfoot to reason about additional examples.

Deleting the Minimal Node of a Binary Search Tree Example Later, once interactive Holfoot proofs are discussed, the example of a binary-search tree will be discussed in detail (see Sec. 2.5.6 and Appx. B.2.16). Let’s consider deleting the minimal element of a binary search tree from the point of loop-specifications for now:

```
search_tree_delete_min (t,m;) [data_tree(t,data) *
      ‘‘BIN_SEARCH_TREE_SET data keys /\ ~(keys = EMPTY)‘‘] {
   local tt, pp, p;
   p = t->l;
   if (p == 0) {
      m = t->dta; tt = t->r;
      dispose (t); t = tt;
   } else {
      pp = t; tt = p->l;
      loop_spec  [(pp |-> [l:p,  r:#rc2,dta:#dc2]) *
                 (p  |-> [l:tt,r:#rc,  dta:#dc]) * (pp == #ppc) *
                 data_tree(tt ,data_l) * data_tree(#rc,data_r) *
                 ‘‘BIN_SEARCH_TREE_SET (node [dc] [data_l;data_r]) keys‘‘] {
         while (tt != NULL) {
```

```
          pp = p; p = tt; tt = p->l;
        }
        m = p->dta; tt = p->r;
        dispose (p); pp->l = tt;
      } [(m == _mk) * (#ppc |-> [l:_new_p,r:#rc2,dta:#dc2]) *
        data_tree(_new_p,_data) *
        ``BIN_SEARCH_TREE_SET _data (keys DELETE _mk) /\
         (_mk IN keys) /\ (!k. k IN keys ==> _mk <= k)``]
  }
} [data_tree (t, _data) * (m == _mk) *
    ``BIN_SEARCH_TREE_SET data (keys DELETE mk) /\
      (mk IN keys) /\ (!k. k IN keys ==> mk <= k)``]
```

This procedure deletes the minimal key from a non-empty binary search tree. The while-loop is used to search for the node storing the minimal key. After the loop has been executed, the original binary-search tree is unmodified and the variable p points to the node holding the minimal key and pp to its parent node. However, expressing these properties of p and pp is complicated and would require some kind of partial tree datastructure. Therefore, the code that deletes the minimal element is included in the loop specification. Thus, the post-condition of the loop specification can state, that the minimal key of the original tree has been deleted. In contrast to the corresponding loop invariant, the loop specification does not need partial tree datastructures.

Besides demonstrating that loop specifications can be used to eliminate the need for partial datastructures, the last example also demonstrates why it is useful that loop specifications allow code after the while-loop. This code after the loop is the else part in the corresponding recursive procedure. Code after the loop is not used by most of the examples. However, as this example illustrates, it sometimes results in much simpler post-conditions and enables Holfoot to handle additional problems.

Summing all Nodes of a Binary Tree Example Another example that demonstrates the power of loop specifications is summing all the nodes of a binary tree (see Appx. B.2.8). A recursive implementation is straightforward. The recursion provides an implicit stack for traversing the tree:

```
tree_sum(r;t) [data_tree(t,data)] {
  local i;
  if (t == NULL) { r = 0; } else {
    r = t->dta;
    i = t->l; tree_sum(i;i); r = r + i;
    i = t->r; tree_sum(i;i); r = r + i;
  }
} [data_tree (t,data) * (r == ``TREE_SUM data``)]
```

An iterative implementation on the other hand, is quite complicated. Now, the user has to keep track of the parts of the tree that still need processing. Essentially, one needs to explicitly maintain a stack. Reasoning about this stack is tricky. There is the invariant that all the trees on the stack combined with the partial trees that have already been processed form the original tree. Most tools use complicated constructs like the magic-wand operator. Holfoot can avoid this by using a loop specification and local reasoning:

```
assume pop(sp,r;) [w/r: sp,r;]
   [ data_list  (sp ,`` v :: vs ``) ] [ data_list  (sp,  vs) * (r == #v)]
assume push(sp;v) [w/r: sp;]
   [ data_list  (sp,data)] [ data_list  (sp,  `` v :: data ``) ]

tree_sum_depth (r;t) [data_tree(t, data)] {
  local sp, c, i;
  r = 0;
  if (t != 0) {
    sp = 0; push(sp;t);
    loop_spec [ data_list (sp, trees ) *  ``~( MEM 0 trees)`` *
                ``LENGTH trees_data = LENGTH trees`` *
               map (\t d.  data_tree (t,d))  ``ZIP ( trees , trees_data )`` ] {
      while (sp != 0) {
        pop(sp,c;);
        i = c->l; if (i != 0) push(sp;i);
        i = c->r; if (i != 0) push(sp;i);
        i = c->dta; r = r + i;
      }
    } [map (\t d.  data_tree (t,d))  ``ZIP ( trees , trees_data )``  *
       (r == ``old(r) + SUM (MAP TREE_SUM trees_data)``)]
  }
} [data_tree (t,data) * (r == ``TREE_SUM data``)]
```

The procedures push and pop are standard and can easily be implemented and verified. Therefore, they are omitted here. Instead the keyword assume is used to define these push and pop operations on stacks. Notice, that the necessary variable permissions are stated explicitly, because Holfoot can't analyse the body of these procedures to figure out the correct permissions.

The interesting part is the while-loop in procedure tree_sum_depth. The precondition states that the stack contains a list trees. This list is a list of root nodes of binary trees containing the data stored in the list trees_data. The separating map operator map is used to establish this connection. None of these trees is empty, i. e. no root pointer is NULL. As a technical side-condition the list trees_data has to have the same length as trees. Given this precondition, the loop guarantees that the trees remain in the heap. However, the stack is not mentioned in the postcondition, i. e. the stack is now empty. Moreover, the sum of all nodes of all the trees in the stack has been added to r.

2.3.4 Unrolling Loops

So far two possibilities for annotating loops have been presented: loop invariants and loop specifications. Sometimes, these annotations are complicated, because the first few iterations of the loop have to be handled specially. A simple example is an implementation of calculating the factorial (see Appx. B.2.7):

```
fact(r;n) {
   local i;
   r = 1; i = 1;
   while (i < n) [(r == ``FACT i``) *  ``( i <= n) \/ (i = 1)``] {
      i = i + 1; r = r * i;
```

```
   }
} [r == “FACT n“]
```

This procedure calculates the factorial of `n` and stores it in `r`. This is done by initialising `r` with 1 and then using a loop to multiply it with $2, 3, \ldots, n$. Let's consider the invariant of this loop. `r` holds the factorial of the counter `i` and `i` is either 1 or less or equal than `n`. The special case `i` equals 1 is needed, because in the first iteration `n` might be 0, while `i` equals 1. This is fine, since $0! = 1! = 1$ holds. One can eliminate the need to consider this special case by unrolling the loop once:

```
fact(r;n) {
   local i;
   r = 1; i = 1;
   if (i < n) {
      i = i + 1; r = r * i;
      while (i < n) [(r == “FACT i“) * (i <= n)] {
         i = i + 1; r = r * i;
      }
   }
} [r == “FACT n“]
```

This idea is used by the `[unroll x]` modifier for loop-invariants. It tells Holfoot to unroll the loop `x` times and then use the given loop-invariant. Using this modifier, the specification of factorial becomes:

```
fact(r;n) {
   local i;
   r = 1; i = 1;
   while (i < n) [unroll 1] [r == “FACT i“ * (i <= n)] {
      i = i + 1; r = r * i;
   }
} [r == “FACT n“]
```

Unrolling can also be used with loop specifications:

```
fact(r;n) {
   local i;
   r = 1; i = 1;
   loop_spec [ unroll 1] [(r == “FACT i“) * (i <= n)] {
     while (i < n) {
        i = i + 1; r = r * i;
     }
   } [r == “FACT n“]
} [r == “FACT n“]
```

Another example for unrolling loops is appending two singly-linked lists (see Appx. B.1.5).

2.4 Additional Constructs

By now all the important and frequently used constructs of Holfoot's input language have been introduced. There are, however, a few additional constructs that are useful in certain situations. These are presented briefly in this section.

2.4.1 assume / assert

There are `assume` and `assert` statements available. `assume(cond)` skips, if the condition `cond` holds. Otherwise, it diverges. Since Holfoot is reasoning about partial correctness, this amounts to just considering states that satisfy `cond`.

`assert(pred)` is more complicated. It skips, if the predicate `pred` is satisfied by a substate of the current state. Otherwise, it fails. This behaviour results in the need to show that there is a substate satisfying `pred`. `assert` needs to consider substates, since it uses predicates whereas `assume` uses conditions. Conditions are pure, they just talk about the stack. In contrast, predicates can describe the heap as well.

Similar to a procedure call, the check, whether a substate satisfies the predicate `pred`, requires searching a frame `R` such that the current state satisfies `pred * R`. If this search succeeds, the automation uses `pred * R` to describe the state after the execution of `assert`. This behaviour of Holfoot's automation allows `assert` to be for example used to roll and unroll recursively defined datastructures like lists, trees or arrays.

An example is a fully functional specification of quicksort (see Appx. B.2.15). The while loop operates on the interval except the first element. After the loop, `assert` is used to reintroduce the first element into the interval representation.

```
quicksort(;b,e) [data_interval(b, e, data)] {
  local piv, l, r;
  if (e > b) {
    piv = b->dta; l = b + 1; r = e;
     loop_spec [ data_interval (l,r,data) * (l <= r + 1)] {
       ...
    } [ data_interval (old(l), old(r), _data2) * ... ]
    assert [ data_interval (b, e, data3)];
    ...
  }
} [ data_interval (b, e, _rdata ) * ``( SORTED $<= _rdata) /\ (PERM data _rdata)``]
```

Often it is sufficient to use `assert` to derive additional information about the stack. An example is the following program that allocates a new memory location and directly deallocates it again. This deallocation looses the information that it was once allocated and therefore is not equal to `NULL`. `assert` is used to preserve this information.

```
dummy(x;) [] {
  x = new();
  assert [x != NULL];
  dispose x;
} [x != NULL]
```

2.4.2 diverge, fail

There are statements that always diverge and fail. These can be used for annotating programs. Since Holfoot's specifications state that the program will not fail, the statement `fail` marks unreachable code. `diverge` marks code that should be ignored, i. e. once this code is reached, the verification stops successfully. Thus, `if (not(cond)) { fail }` is similar to an assertion, while `if (not(cond)) { diverge }` is similar to an assume statement.

2.4.3 Block Specifications

`assume` and `assert` operate on a single state. They either assume or assert that this single state satisfies some property. In this sense they are similar to loop-invariants. As seen with loop-invariants this concept of describing a single state does not mix well with separation logic's local reasoning. Instead it proved beneficial to use a pre- and a postcondition, i. e. a pair of states.

Similarly, it is often beneficial to annotate a block of code with a pre- and a postcondition. An example is getting the minimum and maximum depth of a tree (see Appx. B.2.2). This example needs to calculate the maximum of two values. Since Holfoot does not support this operation directly, conditional execution is used:

```
if (di1 < dj1) {
  r1 = dj1 + 1;
} else {
  r1 = di1 + 1;
}
```

This looks complicated. It can be annotated with a block specification to state that it is calculating the incremented maximum:

```
block_spec [emp] {
  if (di1 < dj1) {
    r1 = dj1 + 1;
  } else {
    r1 = di1 + 1;
  }
} [r1 == ``(MAX di1 dj1) + 1``]
```

This annotation states that this block of code does not access any locations on the heap and after execution the value of `r1` has been updated to contain the incremented maximum. Introducing the annotation helps structuring the code and understanding it. Moreover, it speeds up the verification process considerably, because the case split is now contained inside the block.

Block specifications can also be used to forget unimportant information that would just clutter the verification process. An example is the implementation of binary search (see Appx. B.2.2).

```
binsearch(f;a,n,e) [array(a,n)] {
   local l, r, m, tmp;
   l = 0; r = n; f = 0;
   while ((f == 0) and (l < r)) [array(a,n) * (r <= n)] {
      block_spec [l < r] {
         m = l + ((r - l) / 2);
      } [l <= m * m < r]
      tmp = (a+m)->dta;
      if (tmp < e) { l = m+1; } else
         if (e < tmp) { r = m; } else { f = 1; }
   }
} [array(a,n)]
```

HOL4's automation is not good at reasoning about integer devision. In fact, the external SMT-solver Yices is used to verify this specification. The block specification allows hiding the exact definition of `m` and just expose the fact that `m` lies between `l` and `r`.

Block specifications are closely related to loop specifications. A loop specification can be seen as a block specification that starts with a while-loop.

2.4.4 Annotating Memory Allocation

According to the semantics of Holfoot's programming language, at each location in the heap there are values for all possible tags stored. Sometimes, it is convenient to make this knowledge explicit for some tags when allocating new heap cells. This is especially the case for allocating arrays.

Consider for example copying an array (see Appx. B.2.10).

```
copy(r;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   r = new(n) [dta];
   while (i < n) [data_array(a,n,data) * data_array (r,n,_data_new) * (i <= n) *
                 ``! x. x < i ==> (EL x data = EL x _data_new)``] {
      tmp = (a + i) -> dta; (r + i) -> dta = tmp;
      i = i + 1;
   }
} [ data_array (a,n,data) * data_array (r, n, data)]
```

The memory allocation `r = new(n)` is annotated with the list of tags `[dta]`. Without annotation, the statement would allocate heap cells that satisfy the predicate `array(r,n)`. With the additional annotation, the data content of the array is explicitly represented. A new specification variable `tdata` is introduced, such that `data_array(r,n,dta:tdata)` holds.

The loop-invariant requires that there is some existentially quantified data `data_new` in the array starting at `r`. If the specification variable `tdata` has been introduced, one has to show that for all `tdata` a corresponding `data_new` exists. This is trivial, just set `data_new` to `tdata`. If, however, this specification variable is not introduced, the automation ends up in a bad state. Then one has to show that there exists a `data_new` that is valid for all `tdata`. The quantifiers have swapped order and no such `data_new` can be found. This is discussed in more detail in Sec. 3.3.7.5.

2.4.5 Assuming Procedures

Procedures can be annotated with the keyword `assume`. In this case, the specification of the procedure is just assumed instead of proved. This mechanism can be used to define new operations. An example can be found in the iterative implementation of summing all the nodes of a tree (see Sec. 2.3.3.4 and Appx. B.2.8). There push and pop operations are defined using `assume`. Notice, that the necessary write-permissions on the call-by-reference arguments are stated explicitly.

```
assume pop(sp,r;) [w/r: sp,r;]
   [ data_list  (sp ,'' v :: vs '') ] [ data_list  (sp, vs) * (r == #v)]

assume push(sp;v) [w/r: sp;]
   [ data_list  (sp,data)] [ data_list  (sp,  '' v :: data '') ]
```

2.4.6 Global Specification Variables

Normally, the scope of specification variables is limited to a pair of pre- and postconditions or even a single predicate. They get explicitly universally quantified inside this scope. Most of the time, this behaviour is sensible. Sometimes, however, it is useful to use global specification variables, i. e. specification variables whose scope is the entire specification, spanning multiple procedure definitions and even the body of procedures. An example is filtering a list (see Appx. B.1.7). The original problem removes all occurrences of some value `x`. However, the same algorithm can be used in order to filter with respect to an arbitrary predicate. This can be expressed in Holfoot using a higher order global specification variable P. Holfoot is even able to verify a recursive implementation automatically.

```
global P;

list_filter(l;) [data_list(l,data)] {
  local e, m;
  if (l == NULL) {
  } else {
    e = l->dta; m = l->tl;
    list_filter(m;);
    if (``~(P e)``) {
      dispose l; l = m;
    } else {
      l->tl = m;
    }
  }
} [ data_list (l,  '' FILTER P data '') ]
```

Using global specification variables might be useful for determining the behaviour of a procedure in the first place. Consider, for example, an implementation of reversing a singly-linked list (see Appx. B.1.3). Assume, one could figure out the shape of the needed datastructures, but not their content. In this case, one can introduce two global specification variables `f1` and `f2` and run Holfoot on the following input:

```
global f1, f2;

list_reverse(i;) [data_list(i,data)] {
  local p, x;
  p = NULL;
  loop_spec [ data_list (i,data) *  data_list (p, data2)] {
    while (i != NULL) {
      x = i->tl; i->tl = p; p = i; i = x;
    }
  } [ data_list (p,  '' f1 data data2 '') ]
```

```
  i = p;
} [ data_list (i ,'' f2 data '') ]
```

As this specification does not hold for arbitrary functions `f1` and `f2`, Holfoot will fail. However, the remaining proof obligations are interesting, because they describe a tail-recursive implementation of reversing a list.

- $\forall l.\ \texttt{f1}\ []\ l\ =\ l$
- $\forall e, l_1, l_2.\ \texttt{f1}\ (e :: l_1)\ l_2\ =\ \texttt{f1}\ l_1\ (e :: l_2)$
- $\forall l.\ \texttt{f2}\ l\ =\ \texttt{f1}\ l\ []$

Sometimes this trick of using global specification variables to figure out the real specification is useful. However, at the current stage it is really just a trick that might or might not give decent results. In combination with techniques for guessing the shape of datastructures, it might be interesting to experiment with this technique. It could result in semi-automatically translating low-level imperative programs into functional ones.

2.5 Interactive Proofs

Most Holfoot examples presented so far can be handled automatically. A collection of such examples can be found in Appendix B.1. However, the full power of Holfoot is only available when using it interactively inside HOL4 [13, 34].

To use Holfoot interactively, one needs to be familiar with HOL4, its user-interface and libraries. Since an introduction to HOL4 would be lengthy and in any case outside the scope of this thesis, I will try to explain some key ideas and observations on interactive proofs without referring to too many HOL4 details. HOL4 proof scripts containing all the details can be found in Appendix B.2 and B.3.

2.5.1 General Overview

HOL4 is implemented in ML [32]. The user interacts with HOL4 through an interactive ML session. Holfoot provides commands to parse specification files, which are written in the syntax described above. The result of this parsing is a HOL4 term that states the validity of the specification. Holfoot provides a pretty-printer that prints this term in a form similar to the input language. Details can be found in Section 2.2.1.3.

In order to verify the parsed specification, the term is send as a new goal to HOL4's *goalstack*. This goalstack is a HOL4 mechanism for backward proofs. A *goal* is proved by repeatedly reducing it to a several subgoals using *tactics* until these subgoals become simple enough to be proved directly. Holfoot provides tactics for proving specifications. The most important tactic performs forward analysis on Hoare triples and evaluates frame calculations. There are several versions of this tactic. The most common one tries to do as much work as possible. It can solve many interesting examples automatically. In fact, this tactic is used by the command-line and web-interface[1] of Holfoot and provides

[1] `http://holfoot.heap-of-problems.org`

Holfoot's automatic verification facilities. If this tactic can't make any more progress it stops and allows the user to call other tactics. Using such tactics, the user can for example reason about pure side-conditions, perform case-splits or provide witnesses for existential quantifiers.

Additionally, there are several versions of this tactic that just perform a certain number of steps. The user can for example instruct Holfoot to just symbolically evaluate the next statement or evaluate everything up to the next loop. Other customisations include providing the automation with problem-specific rewrite rules or turning features like case-splitting or arithmetic simplifications off.

Instead of discussing tactics and details of the proof-scripts, I will try to provide a high level view of some interesting, interactive proofs.

2.5.2 Sum and Maximal Element of an Array Example

During the *Verified Software: Theories, Tools and Experiments* conference in August 2010 in Edinburgh there was an informal verification competition[2] organised by Natarajan Shankar and Peter Mueller. Problem 1 from this competition (see Appx. B.3.1) is a good first example to show the benefits of interactive proofs.

Given a simple program that calculates the sum *sum* and the maximal element *max* of an array of size n, the competition challenges participants to prove $sum \leqslant n * max$. In Holfoot this specification can be written as:

```
vscomp1(sum,max;a,n) [data_array(a,n,data)] {
   local i, tmp;
   sum = 0; max = 0; i = 0;
   while (i < n) [data_array(a,n,data) * i <= n * (sum <= (i * max))] {
      tmp = (a + i) -> dta;
      if (max < tmp) { max = tmp; }
      sum = sum + tmp;
      i = i + 1;
   }
} [ data_array (a,n,data) * (sum <= (n * max))]
```

After calling Holfoot's automation, there are two remaining proof obligations:

- The first proof obligation is created by trying to show that the loop invariant still holds after executing the body of the loop, if `max` was updated.

$$\forall i, n, max, sum, a. \quad (i < n) \;\wedge\; (sum \leqslant i * max) \;\wedge\; (max < a[i]) \implies (sum + a[i] \leqslant (i+1) * a[i])$$

- The second proof obligation arises from proving that the loop invariant implies the postcondition of the procedure, once the loop is exited.

$$\forall i, n, max, sum. \quad (n \leqslant i) \;\wedge\; (i \leqslant n) \;\wedge\; (sum \leqslant i * max) \implies (sum \leqslant max * n)$$

[2] http://www.macs.hw.ac.uk/vstte10/Competition.html

The second proof obligation is trivial. In fact, it's rather disappointing that it is not solved automatically. In principle, HOL4's automation is able to solve problems like this automatically. However, for performance reasons Holfoot's automation is only using a carefully selected subset of HOL4's automation.

However, the first proof obligation is more interesting. It is a rather simple arithmetic property. However, it is not a linear problem. Therefore, SMT-solvers like Yices cannot solve this problem automatically. However, it can be solved with a short HOL4 proof-script (see Appx. B.3.1).

The specification shown answers the original challenge. However, it is rather weak. It does not state that `max` contains the maximal element and that `sum` contains the sum of all elements. Here, Holfoot can benefit for HOL4's infrastructure. One can easily define functions that compute the sum of all elements and the maximal element.

```
val LIST_SUM_def = Define `
   (LIST_SUM [] = 0) /\
   (LIST_SUM (n::ns) = n + LIST_SUM ns)`;

val LIST_MAX_def = Define `
   (LIST_MAX [] = 0) /\
   (LIST_MAX (n::ns) = MAX n (LIST_MAX ns))`;
```

Then the main statement can be proved independently from the implementation as a lemma on the semantics of `LIST_MAX` and `LIST_SUM`.

$$\forall l.\ \texttt{LIST_SUM}\ l\ \leqslant\ \texttt{LENGTH}\ l\ *\ \texttt{LIST_MAX}\ l$$

The new functions `LIST_SUM` and `LIST_MAX` can be used in an improved specification.

```
vscomp1(sum,max;a,n) [data_array(a,n,data)] {
   local i, tmp;
   sum = 0; max = 0; i = 0;
   while (i < n) [data_array(a,n,data) * i <= n *
                  (max == ``LIST_MAX (FIRSTN i data)``) *
                  (sum == ``LIST_SUM (FIRSTN i data)``)] {
      tmp = (a + i) -> dta;
      if (max < tmp) { max = tmp; }
      sum = sum + tmp;
      i = i + 1;
   }
} [data_array (a,n,data) * (sum <= (n * max)) *
   (max == ``LIST_MAX data``) * (sum == ``LIST_SUM data``)]
```

The proof-script for this specification (see Appx. B.3.1) consists of calling Holfoot's automation followed by some rewrites using the definitions of the new functions as well as the proved lemma.

2.5.3 List Remove Example

Another simple example that shows the benefits of user defined functions is removing the first occurrence of an element from a singly-linked list (see Appx. B.2.3). Unluckily, HOL4's list library does not contain a `REMOVE` function. However, it is easily defined:

```
val REMOVE_def = Define ‘
   (REMOVE x [] = []) /\
   (REMOVE x (y::ys) = if (x = y) then ys else (y::REMOVE x ys))‘;
```

This new definition is used by the following specification. If Holfoot's automation is provided with the definition of REMOVE, it is able to prove this specification automatically.

```
list_remove(l;x) [data_list(l,data)] {
  local v,t;
  if (l != NULL) {
    v = l->dta;
    if (v == x) {
      t = l; l = l->tl; dispose(t);
    } else {
      t = l->tl; list_remove(t;x); l->tl = t;
    }
  }
} [ data_list (l, ``REMOVE x data``)]
```

2.5.4 Mergesort Example

A specification of mergesort that specifies only the shape of datastructures is one of Smallfoot's examples (see Appx. B.1.13). Smallfoot and Holfoot can verify such a specification automatically. Let's now consider a fully functional specification of mergesort (see Appx. B.2.13).

```
merge(r;p,q) [data_list(p,pdata) * data_list (q,qdata) *
             ``SORTED $<= pdata /\ SORTED $<= qdata``] {
  local t, q_date, p_date;
  if (q == NULL) { r = p; } else
  if (p == NULL) { r = q; } else {
    q_date = q->dta; p_date = p->dta;
    if (q_date < p_date) { t = q; q = q->tl; } else
                         { t = p; p = p->tl; }
    merge(r;p,q);
    t->tl = r; r = t;
  }
} [ data_list (r, _rdata) * ``(SORTED $<= _rdata) /\ (PERM (pdata ++ qdata) _rdata)``]

split(r;p) [data_list(p,data)] {
  local t1,t2;
  if (p == NULL) { r = NULL; } else {
    t1 = p->tl;
    if (t1 == NULL) { r = NULL; } else {
      t2 = t1->tl; split(r;t2);
      p->tl = t2; t1->tl = r; r = t1;
    }
  }
} [ data_list (p,_pdata) * data_list (r,_rdata) * ``PERM (_pdata ++ _rdata) data``]

mergesort(r;p) [data_list(p,data)] {
```

```
  local q,q1,p1;
  if (p == NULL) { r = p; } else {
    split(q;p);
    mergesort(q1;q); mergesort(p1;p);
    merge(r;p1,q1);
  }
} [ data_list (r, _rdata ) * ‘‘( SORTED $<= _rdata) /\ (PERM data _rdata)‘‘]
```

After calling Holfoot's automation, one ends up with the following verification conditions:

- the procedure `merge` requires
  - $\forall l.\ \texttt{PERM}\ l\ l$,
    i. e. permutations are reflexive
  - $\forall e, l.\ \texttt{SORTED}\ (e :: l) \Longrightarrow \texttt{SORTED}\ l$,
    i. e. if a non-empty list $e :: l$ is sorted than its tail l is sorted as well.
  - $\forall e_1, e_2, l_1, l_2, l_3.\ \ \texttt{SORTED}\ (e_1 :: l_1)\ \wedge\ \texttt{SORTED}\ (e_2 :: l_2)\ \wedge\ (e_2 < e_1)\ \wedge$
    $\texttt{SORTED}\ l_3\ \wedge\ \texttt{PERM}\ (e_1 :: (l_1{+}{+}l_2))\ l_3\ \Longrightarrow$
    $\texttt{SORTED}\ (e_2 :: l_3)\ \wedge\ \texttt{PERM}\ (e_1 :: (l_1{+}{+}e_2 :: l_2))\ (e_2 :: l_3)$
  - $\forall e_1, e_2, l_1, l_2, l_3.\ \ \texttt{SORTED}\ (e_1 :: l_1)\ \wedge\ \texttt{SORTED}\ (e_2 :: l_2)\ \wedge\ (e_1 \leqslant e_2)\ \wedge$
    $\texttt{SORTED}\ l_3\ \wedge\ \texttt{PERM}\ (l_1{+}{+}e_2 :: l_2)\ l_3\ \Longrightarrow$
    $\texttt{SORTED}\ (e_1 :: l_3)\ \wedge\ \texttt{PERM}\ (e_1 :: (l_1{+}{+}e_2 :: l_2))\ (e_1 :: l_3)$
- the procedure `split` requires
  - $\forall e.\ \texttt{PERM}\ [e]\ [e]$,
    i. e. permutations are reflexive for lists of length one
  - $\forall e_1, e_2, l_1, l_2, l_3.\ \ \texttt{PERM}\ (l_1{+}{+}l_2)\ l_3\ \Longrightarrow$
    $\texttt{PERM}\ (e_1 :: (l_1{+}{+}e_2 :: l_2))\ (e_1 :: e_2 :: l_3)$
- the procedure `split` requires
  - $\forall l_1, l_2, l_3, l'_1, l'_2, l'_3.\ \ \texttt{PERM}\ l_1\ l'_1\ \ \wedge\ \texttt{PERM}\ l_2\ l'_2\ \ \wedge$
    $\texttt{PERM}\ (l_1{+}{+}l_2)\ l_3\ \ \wedge\ \texttt{PERM}\ (l'_1{+}{+}l'_2)\ l'_3 \Longrightarrow$
    $\texttt{PERM}\ l_3\ l'_3$

Most of these proof obligations are straightforward. Providing the automation with some knowledge about sorted lists and permutations solves most of these obligations. Just simplified versions of the last two `merge` proof-obligations remain.

- $\forall e_1, e_2, l_1, l_2, l_3.\ \ \texttt{SORTED}\ (e_1 :: l_1)\ \wedge\ \texttt{SORTED}\ (e_2 :: l_2)\ \wedge\ (e_2 < e_1)\ \wedge$
  $\texttt{SORTED}\ l_3\ \wedge\ \texttt{PERM}\ (e_1 :: (l_1{+}{+}l_2))\ l_3\ \Longrightarrow$
  $\texttt{SORTED}\ (e_2 :: l_3)$
- $\forall e_1, e_2, l_1, l_2, l_3.\ \ \texttt{SORTED}\ (e_1 :: l_1)\ \wedge\ \texttt{SORTED}\ (e_2 :: l_2)\ \wedge\ (e_1 \leqslant e_2)\ \wedge$
  $\texttt{SORTED}\ l_3\ \wedge\ \texttt{PERM}\ (l_1{+}{+}e_2 :: l_2)\ l_3\ \Longrightarrow$
  $\texttt{SORTED}\ (e_1 :: l_3)$

These two proof obligations capture the essence of the algorithmic idea of `merge`. They can be verified using just a few lines of proof-script. However, this verification requires combining the concepts of sorted lists and permutations. That's why they cannot be easily discharged automatically. Informally, the first first one can be justified as follows.

If we want to show that $e_2 :: l_3$ is sorted and we know that l_3 is sorted, it remains to show $e_2 \leqslant e$ forall elements e of l_3. We know that l_3 is a permutation of $(e_1 :: (l_1{+}{+}l_2))$. Therefore, e is either e_1 or an element of l_1 or l_2. $e_2 < e_1$ is stated explicitly in the precondition. Moreover, e_2 is not greater than any element of l_2, because $e_2 :: l_2$ is sorted. Finally, we know $\forall e \in l_1.\ e_1 \leqslant e$, because $e_1 :: l_1$ is sorted. Combined with $e_2 < e_1$ this results in $e_2 < e$.

The proof of the second proof obligation is very similar. I hope this simple proof on high-level concepts of the algorithm convinces you that the proof obligations really talk about the essence of the `merge` algorithm.

2.5.5 Circular List Example

The last few examples demonstrated the ideal case of using Holfoot interactively. The automation takes care of the program structure and details of the memory layout. This leaves the user to reason about a functional representation of the algorithm. There are many examples that can be handled like that.

Other examples, however, require the user to provide manual case-splits or provide witnesses to existential quantifiers in order to reason about the program structure. A simple example is an implementation of circular lists (see Appx. B.2.4).

```
push(r) [r|->_tf * lseg(_tf,r)] {
  local t, u;
  t = new();
  u = r->tl;
  t->tl = u;
  r->tl = t;
} [r|->_b * _b|->_tf * lseg(_tf,r)]
```

```
pop_dequeue(r)
  [r!=_tf * r|->_tf * lseg(_tf,r)] {
  local t, u;
  t = r->tl;
  u = t->tl;
  r->tl = u;
  dispose t;
} [r|->_b * lseg(_b,r)]
```

```
enqueue(r;) [r|->_tf * lseg(_tf,r)] {
  push(r);
  r = r->tl;
} [r|->_tf * lseg(_tf,_b) * _b|->r]
```

```
test(r;) [r|->_tf * lseg(_tf,r)] {
  push(r);
  pop_dequeue(r);
  enqueue(r;);
  pop_dequeue(r);
} [r|->_a * lseg(_a,r)]
```

The procedures `push` and `pop_dequeue` are straightforward and can be verified automatically. `enqueue` looks trivial. However, it requires the user to provide an existential witness. After the symbolic execution, the following Hoare triple remains to be proved.

```
[[w/r: r; | (r |-> tl: #tf) * lseg(tl; #tf, #r_const) * (#r_const |-> tl: r)]]
    /* enqueue 3 */
[[w/r: r; | (r |-> tl: #_tf) * lseg(tl; #_tf, #_b) * (#_b |-> tl:r)]]
```

Since the body of the Hoare triple is empty, this means that it has to be shown that the precondition of this Hoare triple implies its postcondition. The pre- and postcondition look very similar. For a human, it is trivial to instantiate the existentially quantified variables `tf` and `b` in the postcondition such that the pre- and postcondition become identical. Holfoot is unluckily just able to figure out the instantiation for `tf`. The instantiation of `b` with `r_const` needs to be provided by the user.

Similarly, the proof of the procedure `test` needs user guidance. After evaluating the first three procedure calls, the following Hoare triple remains:

```
[[w/r: r; | lseg(tl; #tf, #b) * (#b |-> tl: r) * (r |-> tl: #tf) * (r != #b)]]
   /* test 4 */
   abstracted pop_dequeue(; r)
[[w/r: r; | (r |-> tl: #_a) * lseg(tl; #_a, r)]]
```

In order to call `pop_dequeue` the property `r != #tf` needs to be shown. Holfoot is not able to do this automatically. The user needs to instruct Holfoot to perform a case-split on whether `tf` and `b` are equal. If they are equal, then `r != #tf` holds trivially. Otherwise, the list segment described by `lseg(#tf,#b)` is not empty and therefore contains the heap-location `tf`. Since the location `r` is contained in the separate heap described by `r |-> #tf`, it can be concluded that `r` is not equal to `tf`.

2.5.6 Binary Search Tree Example

An algorithmically more challenging problem is implementing binary search trees (see Appx. B.2.16). Before specifying algorithms operating on binary search trees, a representation of binary search trees needs to be introduced. Holfoot uses a two-layered representation. First, Holfoot's standard tree-predicate relates the concrete representation of a tree as a dynamic datastructure in memory to a functional representation of the tree. Then, a user-defined predicate states that this functional tree represents a binary search tree containing some set of keys.

For Holfoot, a functional tree is either a leaf `leaf` or a node `node valueL treeL` containing a list of values and pointing to a list of subtrees. Let `data` be some functional representation of a tree that satisfies for some state the predicate `data_tree(t,data)` (or more verbosely the predicate `data_tree([l,r];t,[dta]:data)`). Then, all nodes of `data` are of the form `node [v] [lt; rt]` where `v` is the value stored in the node (tag `dta` in the heap), `lt` is the left subtree (tag `l`) and `rt` is the right subtree (tag `r`). In order to reason about binary search trees, this functional representation of trees has to be related to an abstract view of binary search trees containing a set of keys. This is done by introducing a new predicate `BIN_SEARCH_TREE_SET` that satisfies the following equations:

$$\begin{array}{ll}
\texttt{BIN_SEARCH_TREE_SET leaf } \mathit{keys} & = (\mathit{keys} = \emptyset) \\
\texttt{BIN_SEARCH_TREE_SET (node } [k]\ [t_1; t_2]\texttt{)}\ \mathit{keys} & = \\
\quad \exists k_1 k_2.\ \ (\mathit{keys} = \{k\} \cup k_1 \cup k_2) \wedge & \\
\quad\quad (\forall k' \in k_1.\ k' < k) \ \wedge\ (\forall k' \in k_2.\ k' > k) \wedge & \\
\quad\quad \texttt{BIN_SEARCH_TREE_SET}\ t_1\ k_1 \wedge & \\
\quad\quad \texttt{BIN_SEARCH_TREE_SET}\ t_2\ k_2 &
\end{array}$$

Using this new predicate, it is easy to specify for example inserting a key into a binary search tree (see Appx. B.2.16).

```
search_tree_insert(t;k) [data_tree(t,data) * "BIN_SEARCH_TREE_SET data keys"] {
  local k0, tt;
  if (t == NULL) { t = new(); t->l = 0; t->r = 0; t->dta = k; } else {
    k0 = t->dta;
    if (k0 == k) { } else {
      if (k < k0) {
        tt = t->l; search_tree_insert(tt;k); t->l = tt;
      } else {
        tt = t->r; search_tree_insert(tt;k); t->r = tt;
      }
    }
  }
} [data_tree(t,_data) * "BIN_SEARCH_TREE_SET data (k INSERT keys)"]
```

The precondition states that `t` is pointing to the root of a binary tree, whose functional representation is `data`. It is further stated that this tree is a binary search tree containing a set of keys `keys`. The postcondition demands that `t` points to the root of some modified binary tree. The exact structure of this tree is not specified. However, it is stated that this tree is a binary search tree containing the set of keys `k INSERT keys`. This means that the key `k` has been added to the binary search tree.

Separating the layout of the datastructure in memory from its abstract interpretation is important for Holfoot's automation. Knowing about the concrete representation of the tree in memory is sufficient to symbolically execute the body of `search_tree_insert`. That the binary tree in question is a binary search tree is a pure side-condition that can easily by passed around. The user can use this side-condition later to establish that the modification of the concrete datastructure relates to changes in the abstract view.

In the running example of inserting a new key into a binary search tree, there are four proof-obligations after running the automation. These correspond to the cases of inserting a new node, doing nothing, because the key is already present, inserting the key into the left subtree and inserting it into the right subtree.

- $\forall k,\ keys.$
  $\texttt{BIN_SEARCH_TREE_SET leaf}\ keys \Longrightarrow$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [leaf;leaf])}\ (\{k\} \cup keys)$

- $\forall l,\ r,\ k,\ keys.$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l\texttt{;}r\texttt{])}\ keys \Longrightarrow$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l\texttt{;}r\texttt{])}\ (\{k\} \cup keys)$

- $\forall l,\ r,\ k,\ k',\ keys.$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l\texttt{;}r\texttt{])}\ keys\ \wedge (k' < k) \Longrightarrow$
  $\exists keys'.\ \forall \texttt{l'}.\ \texttt{BIN_SEARCH_TREE_SET}\ l\ keys' \wedge$
  $\texttt{BIN_SEARCH_TREE_SET}\ l'\ (\{k'\} \cup keys') \Longrightarrow$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l'\texttt{;}r\texttt{])}\ (\{k'\} \cup keys)$

- $\forall l,\ r,\ k,\ k',\ keys.$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l\texttt{;}r\texttt{])}\ keys\ \wedge (k' > k) \Longrightarrow$
  $\exists keys'.\ \forall \texttt{r'}.\ \texttt{BIN_SEARCH_TREE_SET}\ r\ keys' \wedge$
  $\texttt{BIN_SEARCH_TREE_SET}\ r'\ (\{k'\} \cup keys') \Longrightarrow$
  $\texttt{BIN_SEARCH_TREE_SET (node [}k\texttt{] [}l\texttt{;}r'\texttt{])}\ (\{k'\} \cup keys)$

Using the definition of BINARY_SEARCH_TREE_SET all these proof obligations can easily be discarded. I hope this example convinces you that, thanks to the separation of concrete representation and abstract view, Holfoot is able to reason about even complicated datastructures like binary search trees with a high level of automation. The user usually just has to reason about the essence of the algorithm manually.

2.5.7 Insertion into Red-Black Tree Example

Another example that demonstrates the benefits of a two layered representation of high-level datastructures is red-black trees (see Appx. B.2.17). Again, the standard tree-predicate is used to connect the concrete representation in memory with a functional representation of the tree. This time, a node of the tree stores three values: a key k, a value v and a colour c. Using the functional tree representation a predicate RED_BLACK_-TREE is defined such that RED_BLACK_TREE t f holds for a tree t and a finite map f, iff

- t is a binary search tree representing the finite map f from keys to values.
- All nodes of the tree are well-formed, i. e. they are of the form node [k;v;c] [l;r]. As c represents the colour of the node, there are only two choices. 0 denotes black and 1 red.
- The root of t is a leaf or a black node.
- No red node has a red child.
- All paths through the tree contain the same number of black nodes.

Using this new predicate for red-black trees, inserting a new key/value pair into a red-black tree can easily be specified.

```
rb_tree_insert (r; k, v) [data_tree(r,[k,v,c]:data) * ``RED_BLACK_TREE data f``] {
   rb_tree_insert_r (r; k, v);
   r->c = 0;
} [data_tree (r,[k,v,c]:_data) * ``RED_BLACK_TREE _data (f |+ (k,v))``]
```

Inserting a new key/value pair into a red-black tree is a complicated operation. Compared to inserting it into a binary search tree, the tree needs to be rebalanced. The implementation uses several auxiliary procedures for operations like balancing, rotating or determining the colour of a tree (see Appx. B.2.17). In order to reason about red-black trees new functions are defined in HOL4 that capture the behaviour of these auxiliary procedures.

A simple example is the procedure rb_tree_is_red, which determines, whether a node is red. A new HOL4 function RED_BLACK_TREE___IS_RED is defined to capture the behaviour of this procedure.

```
val RED_BLACK_TREE___IS_RED_def = Define `
   (RED_BLACK_TREE___IS_RED leaf = F) /\
   (RED_BLACK_TREE___IS_RED (node [k;v;c] [t1;t2]) = (c = 1)) /\
   (RED_BLACK_TREE___IS_RED _ = F)`;
```

```
rb_tree_is_red (r;t) [data_tree(t,[k,v,c]:data)] {
   local x;
   if (t == 0) { r = 0; } else {
     x = t->c;
     if (x == 1) { r = 1; } else { r = 0; }
   }
} [data_tree (t,[k,v,c]:data) * (r == ``BOOL_TO_NUM (RED_BLACK_TREE___IS_RED data)``)]
```

Other procedures require an additional definition to capture the necessary precondition as well. The procedure `rb_tree_left_rotate` requires, for example, that it operates on a non-empty tree that has a non-empty right subtree.

```
val PROGRAM_PRED___can_left_rotate_def = Define `
   PROGRAM_PRED___can_left_rotate t =
     IS_RED_BLACK_TREE_NODE t /\
     IS_RED_BLACK_TREE_NODE (RED_BLACK_TREE___RIGHT_SUBTREE t)`

val PROGRAM_FUN___left_rotate_def = Define `
   PROGRAM_FUN___left_rotate
      (node [k1;v1;c1] [a; node [k2;v2;c2] [b;c]]) =
      (node [k2;v2;0] [node [k1;v1;1] [a;b]; c])`

rb_tree_left_rotate (r;)
   [data_tree (r,[k,v,c]:data) * ``PROGRAM_PRED___can_left_rotate data``] {
   local s, x;
   s = r->r; x = s->l; r->r = x; s->l = r;
   r->c = 1; s->c = 0; r = s;
} [data_tree (r,[k,v,c]:``PROGRAM_FUN___left_rotate data``)]
```

It is straightforward to define similar functions for all auxiliary procedures. Proving that the procedures implement their functional representation is simple as well. It mainly consists of calling Holfoot's automation and rewriting with the definitions of the new functions. This treatment of the auxiliary procedures essentially translates the program into a functional representation inside HOL4.

It remains to be shown that these operations on trees really implement inserting a key into a red-black tree. This proof is completely independent of the concrete, low-level implementation. Instead, it uses the functional representation of the auxiliary functions. Therefore, it can concentrate on the essence of the algorithm, i. e. on the core of the verification problem.

It might be interesting that introducing new definitions and showing their correspondence with the auxiliary procedures was straightforward and took about one hour. In contrast, the algorithm underlying inserting a key/value pair into a red-black tree is complicated. Its verification took me about one week.

2.6 Extending Holfoot

Defining new HOL4 functions that are used in pure side-conditions is extensively used in interactive proofs (see Sec. 2.5.6). Besides being used in pure side-conditions, i. e. in predicates, such functions can also be used in conditions of control structures or in pure

expressions. Moreover, procedure declarations annotated with the keyword `assume` can be used to simulate the effect of introducing new statements. These simple possibilities to extend Holfoot's input language are frequently used. Moreover, they are supported by Holfoot's automation.

It is, however, also possible to extend Holfoot at a deeper level. For example, one can add real new statements instead of simulating them with procedures. New control structures or predicates that describe datastructures in the heap can be added as well. It is even possible to add new annotations that guide the verification process. Adding a new construct requires defining it in HOL4 and adapting Holfoot's parser such that the new construct can easily be used. Except in the parser, there is no fixed set of constructs defined anywhere in Holfoot. All constructs are shallowly embedded in HOL4. In the presence of a new construct Holfoot's automation will operate as usual as long as it does not need any information about this new construct. If information about the new construct is required, the automation stops. This allows the user to reason about it manually.

It is possible to extend Holfoot's automation. There is a set of inference rules written in ML. Simplified, the automation can be seen as a loop that applies the inferences rules in this set. Holfoot's automation can be extended by implementing new inference rules in ML and adding them to the set. Some of the existing inference rules also use certain parameters that can be modified. It is for example possible to provide Holfoot with additional rewrite-rules for user-defined predicates.

While Holfoot is in principle extensible in these ways, considerable knowledge of HOL4, its libraries, ML, Holfoot and the separation logic framework is necessary to do it in practise. It is, however, comparably easy to define new non-pure predicates in terms of existing predicates. An example is problem 5 of the VSTTE'10 competition (see Appx. B.3.5).

2.6.1 Amortised Queue Example

Problem 5 of the VSTTE'10 competition (see Appx. B.3.5) is concerned with amortised queues. In the implementation, an amortised queue consists of two singly-linked lists called *front* and *rear*. The abstract data present in an amortised queue consists of *front* `++ REVERSE` *rear*. If the front is not empty, this allows efficient access to the first element of the queue. Similarly, an element can easily be added at the end of the queue by adding it in front of *rear*. One has to be careful though, that the front does not become empty. Therefore, the competition problem enforces the invariant that *rear* is never longer than *front*.

In Holfoot, an amortised queue at location `q` containing a list `data` can be described by the following predicate:

```
q |-> [front:_f, rear:_r, front_length:_fl, rear_length:_rl] *
data_list(_f, _f_data) * data_list(_r, _r_data) * (_rl <= _fl) *
(_fl == ``LENGTH _f_data``) * (_rl == ``LENGTH _r_data``) *
``data = _f_data ++ REVERSE (_r_data)``
```

This predicate describes that at location `q` in the heap there are four values stored: pointers to the front and rear lists indexed by the tags `front` and `rear` and the explicitly stored lengths of these lists indexed by `front_length` and `rear_length`. It further states

that the rear list is not longer than the front one and that the data content of both lists can be combined in the described way to form `data`.

This predicate can be used to define amortised lists in Holfoot. However, using this lengthy form in procedure specifications is hard to read, error-prone and tiresome. Instead, the proof script (see Appx. B.3.5) introduces new predicates for amortised queues. A strong amortised queue is defined as above, a weak one lacks the condition that the front is not shorter than the rear. Holfoot's parser allows adding these new predicates. Similarly, HOL4's pretty printer can easily be made aware of them. In order to get Holfoot's automation to handle the new predicates decently, it also needs to be shown that they are well-behaved with respect to expressions (a technical condition that is explained in Sec. 3.3).

After introducing the new predicates, specifications like the following can be verified.

```
queue_length(re;q) [amortized_queue(q, data)] {
   local rl,fl;
   rl = q->rear_length; fl = q->front_length;
   re = rl + fl;
} [amortized_queue(q, data) * (re == ``LENGTH data``)]
```

Since the new predicate is defined as an abbreviation for predicates known to the automation, the proof script mainly consists of expanding the definition of the new predicates and calling Holfoot's automation.

2.7 Conclusion

In this chapter Holfoot is presented from a user's perspective. Its input language and features are introduced using many example specifications. More examples can be found in Appendix B or at Holfoot's webpage[3]. In case you want to try out Holfoot, Appendix A explains how to obtain and install it.

[3] http://holfoot.heap-of-problems.org

Chapter 3

Theoretical Foundation and Implementation

In Chapter 2 a high level view of Holfoot has been presented. This chapter explains the theoretical foundation as well as technical details of its implementation.

Holfoot is built as an instantiation of a general separation logic framework based on *Abstract Separation Logic* [7]. This framework is first instantiated to support a stack with read / write permissions following ideas of Parkinson, Bornat and Calcagno [31]. Then, a heap is added in a second instantiation step. The presentation of the theoretical foundations follows this structure.

Section 3.2 presents Abstract Separation Logic. The version of Abstract Separation Logic used in this work is very close to its original presentation [7]. It has however been slightly extended. Most noticeable and most important is the addition of procedures. This section introduces many fundamental concepts. Abstract Separation Logic is important for the understanding of the other parts of the theoretical foundations. Readers familiar with Abstract Separation Logic might want to skip this section though, as it contains only minor additions to the original work [7].

Section 3.3 presents a first instantiation of the Abstract Separation Logic framework. A stack with read / write permissions is added. This allows reasoning about concepts like pure expressions, assignments, local variables, procedures with call by reference and call by value parameters. Moreover, the general infrastructure for Holfoot's frame calculations is provided at this level. This layers follows ideas of Parkinson, Bornat and Calcagno [31]. Readers familiar with this work will discover many connections, but should still read this section, because the main ideas are adapted to an Abstract Separation Logic setting. Moreover, they are extended and additional concepts are added.

Finally, Section 3.4 describes the final instantiation of the framework in order to build Holfoot. This layer adds a heap to the model of states. This allows reasoning about explicit memory allocation and deallocation, heap lookups and heap-assignments. Moreover, predicates for datastructures like singly-linked lists, trees and arrays are defined at this layer.

These three sections (Sec. 3.2, 3.3 and 3.4) present Holfoot's theoretical foundations. They have been formalised using the HOL4 [13, 34] theorem prover. However, these sections do not require the reader to be familiar with HOL4. All concepts are presented using general mathematical notations. There are, however, often references to the corresponding HOL4

theorems. These references might be interesting for readers familiar with HOL4 that want to understand the formalisation in detail. For the sake of such readers, there are also some *HOL4 remarks* that describe implementation details in HOL4. Readers not familiar with HOL4 are encouraged to skip these remarks. They assume knowledge about HOL4 and are not important for the understanding of the theoretical foundations.

After the presentation of the theoretical foundations, some technical details are discussed in Section 3.5. As this section does not assume any knowledge about HOL4, the details are discussed at an abstract level. Therefore, this section mainly contains a discussion about how Holfoot's automation applies its inference rules and how Holfoot's quantifier instantiation heuristics work.

3.1 Notations

Before, the theoretical background is presented, some notations have to be introduced briefly. Function application is written as either $f(x)$ or $f\ x$. Often curried functions are used. These are written as $f(x)(y)$ or $f\ x\ y$, respectively. Sometimes, square brackets are used for the first argument: $f[x](y)$, $f[x]\ y$. Abstractions are written as $\lambda x.\ f(x)$ or with multiple arguments as $\lambda x, y.\ f(x)(y)$. Several datatypes are used in the following. Some of these need a short introduction.

3.1.1 Sets

Standard set notations are used in the following. $\emptyset$ denotes the empty set; $\{e_1, e_2, \ldots e_n\}$ the set that contains the elements $e_1, \ldots e_n$. Further, let $\{x \mid P(x)\}$ denote the set containing all x for which $P(x)$ holds. $e \in \mathcal{S}$ denotes that e is an element of the set $\mathcal{S}$. Additional notations include union of sets $\mathcal{S}_1 \cup \mathcal{S}_2$, intersection of sets $\mathcal{S}_1 \cap \mathcal{S}_2$ and the subset relation $\mathcal{S}_1 \subseteq \mathcal{S}_2$. Two sets are disjoint, iff they don't share any elements, i. e. iff $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$. The image of a set $\mathcal{S}$ under a function f is denoted by $\mathit{image}\ f\ \mathcal{S}$. The difference of two sets is denoted by $\mathcal{S}_1 \backslash \mathcal{S}_2$.

3.1.2 Finite Maps

Finite maps are functions with a finite domain. A finite map $m : \alpha \xrightarrow{\text{fin}} \beta$ is therefore a function, that is only defined for a finite subset $\mathit{dom}(m)$ of α. $\emptyset$ is used to denote empty finite maps, i. e. finite maps m with $\mathit{dom}(m) = \emptyset$. The disjoint union of two finite maps m_1 and m_2 is given by

$$(m_1 \uplus m_2)(x) = \begin{cases} m_1(x) & \text{if } x \in \mathit{dom}(m_1) \ \wedge\ x \notin \mathit{dom}(m_2) \\ m_2(x) & \text{if } x \notin \mathit{dom}(m_1) \ \wedge\ x \in \mathit{dom}(m_2) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The restriction of a finite map m by a set $\mathcal{S}$ is given by

$$m \backslash \mathcal{S} = \begin{cases} m(x) & \text{if } x \in \mathit{dom}(m) \ \wedge\ x \notin \mathcal{S} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

That a finite map m is updated at entry x by an value v is denoted by

$$update[x, v](m)(x_2) = \begin{cases} v & \text{if } x_2 = x \\ m(x) & \text{otherwise} \end{cases}$$

3.1.3 Multisets

Multisets (also called bags) are sets that may contain multiple instances of an element. Multisets can be seen as functions from the element type to natural numbers. $\varnothing$ is used to denote the empty multiset, i. e. $\varnothing = \lambda x.\ 0$. Let $\{e_1, \ldots, e_n\}$ denote the multiset that contains the elements $e_1, \ldots, e_n$. Notice, that if some elements are equal, they are multiple times in the multiset. A set S can be interpreted as a multiset $\lambda x.\ \mathit{if}\ x \in S\ \mathit{then}\ 1\ \mathit{else}\ 0$, i. e. a multiset that contains each element of S once. A value x is an element of a multiset $\mathcal{S}$ (denoted by $x \in \mathcal{S}$) iff $\mathcal{S}$ contains x at least once, i. e. iff $\mathcal{S}(x) > 0$. Multiset union is defined by $\big(\mathcal{S}_1 \cup \mathcal{S}_2\big)(x) := \mathcal{S}_1(x) + \mathcal{S}_2(x)$. In contrast merging of two multisets takes the maximum number of entries: $\big(\mathcal{S}_1 \sqcup \mathcal{S}_2\big)(x) := \mathit{max}(\mathcal{S}_1(x), \mathcal{S}_2(x))$. Using the minimum leads to intersection of multisets: $\big(\mathcal{S}_1 \cap \mathcal{S}_2\big)(x) := \mathit{min}(\mathcal{S}_1(x), \mathcal{S}_2(x))$. Two multisets $\mathcal{S}_1$ and $\mathcal{S}_2$ are disjoint, iff $\forall x.\ x \notin \mathcal{S}_1 \vee x \notin \mathcal{S}_2$ holds, i. e. iff $\mathcal{S}_1 \cap \mathcal{S}_2 = \varnothing$ holds. $\mathcal{S}_1$ is a subset of $\mathcal{S}_2$ (denoted by $\mathcal{S}_1 \subseteq \mathcal{S}_2$), iff $\forall x.\ \mathcal{S}_1(x) \leqslant \mathcal{S}_2(x)$ holds. Multiset difference is defined by $\big(\mathcal{S}_1 \backslash \mathcal{S}_2\big)(x) := \mathit{max}(0, \mathcal{S}_1(x) - \mathcal{S}_2(x))$. Finally, all elements of a multiset $\mathcal{S}$ are said to be distinct, iff $\forall x.\ \mathcal{S}(x) \leqslant 1$ holds. A multiset $\mathcal{S}$ is finite iff the set $\{x \mid \mathcal{S}(x) > 0\}$ is finite.

3.1.4 Lists

The empty list is denoted by $[]$. $x :: \mathit{xs}$ denotes a list consisting of an element x followed by a list xs. $[x_0, \ldots, x_n]$ denotes the list consisting of the elements $x_n, \ldots, x_n$. The n-th element of such a list l is denoted by $\mathit{el}(n, l)$. Counting starts at 0, i. e. $\mathit{el}(i, [x_0, \ldots x_n]) = x_i$. The function *length* returns the length of a list. $\mathit{hd}(l)$ denotes the head and $\mathit{tl}(l)$ the tail of a list, i. e. $\mathit{hd}(x :: \mathit{xs}) = x$ and $\mathit{tl}(x :: \mathit{xs}) = \mathit{xs}$. Two lists are appended using the function *append*. Sometimes, $\mathit{append}(l_1, l_2)$ is also written as $l_1 +\!\!+\ l_2$. Mapping a function f over a list l is denoted by $\mathit{map}(f, l)$, i. e. $\mathit{map}(f, [x_0, \ldots, x_n]) = [f(x_0), \ldots, f(x_n)]$. Finally, $\mathit{take}(n, l)$ denotes the list consisting of the first n elements of the list l and $\mathit{drop}(n, l)$ the list consisting of the remaining elements, i. e. $\mathit{take}(i, [x_0, \ldots, x_n]) = [x_0, \ldots x_{i-1}]$ and $\mathit{drop}(i, [x_0, \ldots, x_n]) = [x_i, \ldots x_n]$.

3.2 Abstract Separation Logic

Abstract Separation Logic as introduced by Calcagno, O'Hearn, and Yang [7] is the foundation of the separation logic framework in HOL4. In the following, this foundation will be described.

3.2.1 States and Predicates on States

As the name suggests, Abstract Separation Logic is an abstract version of separation logic. It abstracts from both the concrete specification and the concrete programming

language. The programming language of Abstract Separation Logic manipulates some abstract states, the specification language is based on predicates on these states.

3.2.1.1 Separation Combinators

Since nothing is known about these states, a partial function $\circ$, called the *separation combinator*, is used to combine states and define whether two states are separate.

Definition 3.2.1 (Separation Combinator (HOL4-Thm 217))**.** A *separation combinator* on a set of states Σ is a partially defined function $\circ : \Sigma \times \Sigma \rightharpoonup \Sigma$ that satisfies the following properties:

- $\circ$ is partially associative, i. e.
  $\forall s_1, s_2, s_3.\ \text{Defined}(s_1 \circ (s_2 \circ s_3)) \Leftrightarrow \text{Defined}((s_1 \circ s_2) \circ s_3) \quad \wedge$
  $\forall s_1, s_2, s_3.\ \text{Defined}(s_1 \circ (s_2 \circ s_3)) \Longrightarrow (s_1 \circ (s_2 \circ s_3) = (s_1 \circ s_2) \circ s_3)$
- $\circ$ is partially commutative, i. e.
  $\forall s_1, s_2.\ \text{Defined}(s_1 \circ s_2) \Leftrightarrow \text{Defined}(s_2 \circ s_1) \quad \wedge$
  $\forall s_1, s_2.\ \text{Defined}(s_1 \circ s_2) \Longrightarrow (s_1 \circ s_2 = s_2 \circ s_1)$
- $\circ$ is partially cancellative, i. e.
  $\forall s_1, s_2, s_3.\ \text{Defined}(s_1 \circ s_2) \wedge \text{Defined}(s_1 \circ s_3) \wedge$
  $(s_1 \circ s_2 = s_1 \circ s_3) \Longrightarrow (s_2 = s_3)$
- for all states $s \in \Sigma$ there exists a neutral element $u_s \in \Sigma$ with $u_s \circ s = s$

HOL4 remark 3.2.2. HOL4 supports only total functions. In order to formalise separation combinators, which are only partially defined, option-types are used. The value `NONE` is used to model undefined, whereas `SOME(x)` represents the defined value `x`.

Definition 3.2.3 (Separateness, Substates, Superstates (HOL4-Thms 132, 133))**.** The definition of separation combinators induces notions of *separateness* (#), *substates* ($\preceq$) and *superstates* ($\succeq$).

$$\begin{array}{lll} s_1 \#_\circ s_2 & \text{iff} & s_1 \circ s_2 \text{ is defined} \\ s_1 \preceq_\circ s_3 & \text{iff} & \exists s_2.\ s_3 = s_1 \circ s_2 \\ s_3 \succeq_\circ s_1 & \text{iff} & s_1 \preceq_\circ s_3 \end{array}$$

3.2.1.2 Predicates

Predicates over the set of states Σ are as usual elements of the powerset $\mathcal{P}(\Sigma)$. The separating conjunction operator $*_\circ$ on such predicates and its neutral element $emp_\circ$ are defined as follows:

Definition 3.2.4 ($*$, *emp* (HOL4-Thms 183, 81))**.**

$$\begin{array}{rcl} P *_\circ Q & := & \{s \mid \exists p, q.\ (p \circ q = s) \ \wedge\ p \in P \ \wedge\ q \in Q\} \\ emp_\circ & := & \{u \mid \exists s.\ u \circ s = s\} \end{array}$$

Most of the time it is clear from the context or does not matter which separation combinator $\circ$ is used. The additional argument in omitted for the sake of brevity in these cases. An example are the following important properties of $*$ and *emp*:

Lemma 3.2.5 ((HOL4-Thm 213))**.** For all separation combinators the separating conjunction operator $*$ forms together with *emp* a commutative monoid, i. e. the following properties hold:

$$\begin{array}{rcl} P * emp & = & P \\ P * Q & = & Q * P \\ (P * Q) * R & = & P * (Q * R) \end{array}$$

Other standard separation logic constructs can be defined in a natural way as well. *magic wand* $-\!\!*$ and *septraction* $-\!\!\circ$ can for example be defined as follows:

Definition 3.2.6 (Magic Wand / Septraction)**.**

$$\begin{array}{rcll} P -\!\!*_{\circ} Q & := & \{s \mid \forall p, q.\ (p \circ s = q)\ \wedge\ p \in P\ \Longrightarrow q \in Q\} & \text{(HOL4-Thm 134)} \\ P -\!\!\circ_{\circ} Q & := & \{s \mid \exists p, q.\ (p \circ s = q)\ \wedge\ p \in P\ \wedge\ q \in Q\} & \text{(HOL4-Thm 177)} \end{array}$$

As usual, common Boolean operators are lifted to predicates:

Definition 3.2.7 (Lifted predicates)**.**

$$\begin{array}{rcll} \mathit{true} & := & \{s \mid \mathit{true}\} & \text{(HOL4-Thm 189)} \\ \mathit{false} & := & \{s \mid \mathit{false}\} = \varnothing & \text{(HOL4-Thm 84)} \\ \neg P & := & \{s \mid s \notin P\} & \text{(HOL4-Thm 135)} \\ P \wedge Q & := & \{s \mid s \in P \wedge s \in Q\} & \text{(HOL4-Thm 77)} \\ P \vee Q & := & \{s \mid s \in P \vee s \in Q\} & \text{(HOL4-Thm 136)} \\ c \,\&\, P & := & \{s \mid c \wedge s \in P\} & \text{(HOL4-Thm 188)} \\ \exists x.\ P(x) & := & \{s \mid \exists x.\ s \in P(x)\} & \text{(HOL4-Thm 83)} \\ \forall x.\ P(x) & := & \{s \mid \forall x.\ s \in P(x)\} & \text{(HOL4-Thm 85)} \\ & \vdots & & \end{array}$$

Example 3.2.8. Heaps, modelled as finite partial functions, can be expressed in Abstract Separation Logic. In this model Σ is the set of all heaps. Two heaps are separate, if their domains are disjoint. The combination of two separate heaps is their disjoint union. Thus, the separation combinator $\circ$ for heaps (HOL4-Thms 220, 196) is given by

$$h_1 \circ h_2 \quad = \quad \begin{cases} h_1 \uplus h_2 & \text{iff } dom(h_1) \cap dom(h_2) = \varnothing \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The empty heap $\varnothing$ is the neutral element for all states, i. e. $emp_{\circ} = \{\varnothing\}$ (HOL4-Thm 82).

3.2.1.3 Separation Algebras

Abstract Separation Logic [7] originally used *separation algebras* instead of the separation combinators presented here. Separation combinators are a minor generalisation of separation algebras. While most instantiations like Holfoot use only separation algebras, using the slightly weaker concept of separation combinators is sufficient. An example illustrating the usefulness of using the weaker concept of separation combinators is given below by the *identity separation combinator*.

Definition 3.2.9 (Separation Algebra (HOL4-Thm 214))**.** A *separation algebra* is a cancellative, partial commutative monoid $(\Sigma,\ \bullet,\ u)$. This means that $(\Sigma,\ \bullet,\ u)$ is a separation algebra iff $\bullet$ is a separation combinator and u is the neutral element with respect to $\bullet$ for all states in Σ (HOL4-Thm 215).

The difference between separation algebras and combinators is smaller than one might expect, because even the neutral elements of separation combinators satisfy properties similar to the uniqueness demanded for algebras:

- $\forall s, s_1, s_2.\ (s \circ s_1 = s)\ \wedge\ (s \circ s_2 = s) \implies (s_1 = s_2)$, i. e. for each state $s \in \Sigma$ there is exactly one neural element (HOL4-Thm 218). This element will be denoted by u_x in the following.
- $\forall s_1, s_2.\ s_1\ \#\ s_2 \implies \left(u_{s_1} = u_{s_2} = u_{s_1 \circ s_2}\right)$ (HOL4-Thm 219)
- $\forall s_1, s_2.\ u_{s_1}\ \#\ s_2 \implies \left(u_{s_2} = u_{s_1}\right)$ (HOL4-Thm 222)
- $\forall x.\ u_x \circ u_x = u_x$ (HOL4-Thm 221)

This implies that for a separation algebra $(\Sigma,\ \bullet,\ u)$ the neutral element *emp* with respect to $\circ$ evaluates to $\{u\}$ (HOL4-Thm 80), which is the original definition of *emp* [7].

So, there is only a small difference between separation algebras and separation combinators. However, it is sometimes useful to allow combinators as the following discussion about products of separation combinators will demonstrate.

3.2.1.4 Product Separation Combinators

In this work Abstract Separation Logic is used as a basis for a general framework. For such a framework it is useful to be able to construct separation combinators and states from simpler components and reason about these components separately. For example, a separation combinator for heaps was defined previously (Example 3.2.8). It is useful to be able to use this definition and extend it to a larger state that contains a heap and for example a stack. The following definition of *Product Separation Combinators* allows such extensions:

Definition 3.2.10 (Product Separation Combinator (HOL4-Thm 224))**.** Let $\circ_1$ and $\circ_2$ be two separation combinators on Σ_1 and Σ_2 respectively. Then their product $\circ_1 \times \circ_2$ is defined as

$$(s_1, s_2)\ (\circ_1 \times \circ_2)\ (t_1, t_2) \ := \ \begin{cases} (s_1 \circ_1 t_1, s_2 \circ t_2) & \text{iff } s_1 \ \#_{\circ_1}\ t_1 \ \wedge\ s_2 \ \#_{\circ_2}\ t_2 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$\circ_1 \times \circ_2$ is a separation combinator on $\Sigma_1 \times \Sigma_2$ (HOL4-Thm 225). If $(\Sigma_1, \bullet_1, u_1)$ and $(\Sigma_2, \bullet_2, u_2)$ are separation algebras, then $(\Sigma_1 \times \Sigma_2, \bullet_1 \times \bullet_2, (u_1, u_2))$ is a separation algebra as well (HOL4-Thm 226).

As motivated before, the product of two separation combinators is used to build combinators on complicated states component-wise. A simple, but still useful example is augmenting a state with a static component. This component could for example contain some environment information like global constants or more interestingly definitions of procedures. The following *identity separation combinator* $\circleddash$ can be used in that way to add arbitrary static data:

Definition 3.2.11 (Identity Separation Combinator (HOL4-Thm 207))**.** The *identity separation combinator* $\circleddash$ is defined as:

$$s_1 \circleddash s_2 \quad := \quad \begin{cases} s_1 & \text{iff } s_1 = s_2 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$\circleddash$ is a separation combinator (HOL4-Thm 208) but in general not a separation algebra.

3.2.2 Actions

The elementary constructs of Abstract Separation Logic's programming language are *actions*. They are defined as follows:

Definition 3.2.12 (Action)**.** An *action* $\mathit{act} : \Sigma \to \mathcal{P}(\Sigma)^\top$ is a function from a state to a set of states or a special failure state $\top$.

If an action *act* may fail when executed in a state s, the result of $\mathit{act}(s)$ is $\top$. Otherwise, $\mathit{act}(s)$ results in the set of all possible states after executing the action. The empty set indicates that the action diverges.

HOL4 remark 3.2.13. In the HOL4 implementation, actions are shallowly embedded. $\mathcal{P}(\Sigma)^\top$ is represented using option-types. `NONE` is used to model $\top$, whereas `SOME(P)` represents the state set `P`.

3.2.2.1 Semantic Hoare triples

Given this notion of actions and the previous definitions of predicates, Hoare triples for actions are defined as follows:

Definition 3.2.14 (Semantic Hoare Triple (HOL4-Thm 206))**.** For predicates P, Q and an action *act*, a *Semantic Hoare triple* $\langle\!\langle P \rangle\!\rangle\ \mathit{act}\ \langle\!\langle Q \rangle\!\rangle$ holds, iff for all states p that satisfy the *precondition* P the action does not fail, i. e. $\forall p \in P.\ \mathit{act}(p) \neq \top$, and leads to a state that satisfies the *postcondition* Q, i. e. $\forall p \in P.\ \mathit{act}(p) \subseteq Q$. Notice, that this describes partial correctness, since a semantic Hoare triple is trivially satisfied, if *act* does not terminate, i. e. if $\mathit{act}(s) = \emptyset$ holds.

These definitions of actions and semantic Hoare triples illustrate that Abstract Separation Logic is used to verify the partial correctness of nondeterministic, imperative programs. Its programming language is an abstraction of a concrete programming language with this verification goal in mind. For example, it is not possible to express that an action nondeterministically fails or succeeds. For verification purposes it is sufficient that it

might fail and therefore these cases are combined in a single failure state $\top$. Divergence is handled similarly. Since Abstract Separation Logic is concerned with *partial* correctness, i. e. with statements about what happens if a program terminates, diverging means that the verification effort succeeds. Therefore, diverging computations are not added to the set of resulting states.

3.2.2.2 Common Actions

Even in this abstract setting without concrete states and a concrete separation combinator, basic actions can be defined. The most basic ones are probably *skip*, *diverge* and *fail*:

Definition 3.2.15 (*skip*, *diverge*, *fail*)**.**

$$\begin{array}{rcll} \text{skip}(s) & := & \{s\} & \text{(HOL4-Thm 76)} \\ \text{diverge}(s) & := & \varnothing & \text{(HOL4-Thm 72)} \\ \text{fail}(s) & := & \top & \text{(HOL4-Thm 73)} \end{array}$$

Actions can be combined to form new actions. The most common combinations are sequential composition and nondeterministic choice. In order to define these, it is handy to extend the union and intersection of sets to operate on $\mathcal{P}(\Sigma)^\top$:

Definition 3.2.16. For a set $\mathcal{S} \subseteq \mathcal{P}(\Sigma)^\top$ union and intersection with respect to $\top$ are defined as:

$$\begin{array}{rcll} \bigcup_\top \mathcal{S} & := & \begin{cases} \top & \text{if } \top \in \mathcal{S} \\ \bigcup \mathcal{S} & \text{otherwise} \end{cases} & \text{(HOL4-Thm 235)} \\ \bigcap_\top \mathcal{S} & := & \begin{cases} \top & \text{if } \forall P \in \mathcal{S}.\ P = \top \\ \bigcap\ (\mathcal{S} \backslash \{\top\}) & \text{otherwise} \end{cases} & \text{(HOL4-Thm 212)} \end{array}$$

Definition 3.2.17 (Sequential composition (HOL4-Thm 75))**.**

$$(act_1; act_2)(s) := \begin{cases} \top & \text{if } act_1(s) = \top \\ \bigcup_\top \{act_2(s') \mid s' \in act_1(s)\} & \text{otherwise} \end{cases}$$

So, the sequential composition $act_1; act_2$ fails in a state s if the first action act_1 fails in s. Otherwise, act_2 is executed on all the nondeterministic results. If act_2 fails on any of those, the whole sequential composition fails. Otherwise, it returns nondeterministically one of the results.

Definition 3.2.18 (Nondeterministic Choice (HOL4-Thm 71))**.**

$$\left(\bigsqcup act\text{-}set\right)(s) \quad := \quad \bigcup_\top \ \{act(s) \mid act \in act\text{-}set\}$$

Since nondeterministic choice between two actions is a very common special case, special syntax is introduced for it:

$$act_1 + act_2 \quad := \quad \bigsqcup \{act_1,\ act_2\}$$

The nondeterministic choice between some actions fails, if any of these actions fails. Otherwise, it nondeterministically returns one of the nondeterministic outcomes of one of the actions.

3.2.2.3 Local Actions

Local reasoning is an essential concept of separation logic. It is closely connected to separation logic's *frame rule*. This inference rule allows a semantic Hoare triple to be extended with an arbitrary context:

Semantic Frame Rule

$$\frac{\langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P * R \rangle\!\rangle\ act\ \langle\!\langle Q * R \rangle\!\rangle}$$

The idea is that the precondition P describes all the resources (like for example memory, stack variables, locks) needed to execute *act*. If therefore, there are separate resources R available as well, they don't affect the execution of the action. Moreover, these additional resources are not affected by the execution. The action operates locally on the state described by P.

Most actions used by common programming languages satisfy this frame rule. In order to provide local reasoning, Abstract Separation Logic allows only those actions. They are called *local actions*.

Definition 3.2.19 (Local Actions (HOL4-Thm 128))**.** An action *act* is called *local*, iff it satisfies the frame rule, i. e. iff

$$\forall P, Q, R.\ \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle \Longrightarrow \langle\!\langle P * R \rangle\!\rangle\ act\ \langle\!\langle Q * R \rangle\!\rangle$$

This definition of local actions represents the intention of local actions. However, it is difficult to use this definition directly. Another way of defining local actions is via safety monotonicity and the frame property:

Lemma 3.2.20. (HOL4-Thm 223) An action *act* is *local*, iff it satisfies

safety monotonicity (HOL4-Thm 237):
$\forall s_1, s_2.\ s_1 \preceq s_2 \ \wedge\ act(s_1) \neq \top \Longrightarrow act(s_2) \neq \top$

frame property (HOL4-Thm 236):
$\forall s_1, s_2, s_3, t, t'.\ \ s_1 \circ s_2 = s_3 \ \wedge\ act(s_1) \neq \top \ \wedge\ act(s_3) \neq \top \ \wedge\ t \in act(s_3) \Longrightarrow$
$\exists t'.\ t' \in act(s_1) \ \wedge\ t' \circ s_2 = t$

Safety monotonicity states that if an action has enough resources to succeed in a state s, then it will also succeed in any superstate of s, i. e. in any state that provides more resources. The frame property states that the execution on the superstate keeps these additional resources untouched.

There is an even more concise characterisation of locality:

Lemma 3.2.21. (HOL4-Thm 117) An action *act* is *local*, iff

$$\forall s_1, s_2.\ s_1 \ \#\ s_2 \ \Longrightarrow\ act(s_1 \circ s_2) \subseteq_\top \Big(act(s_1) *_\top s_2\Big)$$

holds, where $\subseteq_\top$ and $*_\top$ are extensions of $\subseteq$ and $*$ that respect the failure state set $\top$:

$$P \subseteq_\top Q \ := \ (Q = \top) \vee (P \neq \top \ \wedge P \subseteq Q) \qquad \text{(HOL4-Thm 203)}$$

$$P *_\top Q \ := \ \begin{cases} \top & \text{if } P = \top \text{ or } Q = \top \\ P * Q & \text{otherwise} \end{cases} \qquad \text{(HOL4-Thm 205)}$$

Notice, that this definition of local actions – like Abstract Separation Logic in general – is with respect to partial correctness. A local action may diverge on the larger state while it terminates on the smaller one. Similarly, operating on a larger state may reduce the number of possible resulting states.

Lemma 3.2.22 (Basic local actions)**.** The actions *skip*, *diverge* and *fail* are local actions (HOL4-Thms 123, 120, 121) Moreover, the sequential combination $a_1; a_2$ of two local actions a_1 and a_2 as well as the nondeterministic choice $\bigsqcup_{act \in local\text{-}act\text{-}set}\, act$ between a set of local actions *local-act-set* are local actions themselves (HOL4-Thms 122, 119).

Example 3.2.23 (Example actions on heaps (HOL4-Thm 130))**.** Consider the model of heaps from Example 3.2.8. In this model the action $act_1(h) :=$ if $h = \varnothing$ then $\{h\}$ else $\top$ is not a local action, because it violates safety monotonicity. It succeeds for the empty heap but fails for all other heaps. The similar action $act_2(h) :=$ if $h = \varnothing$ then $\{h\}$ else $\{\}$ however is a local action. Instead of failing act_2 diverges, which is fine as Abstract Separation Logic is just concerned with partial correctness.

3.2.2.4 Total Lattice of Local Actions

$\subseteq_\top$ is a partial order of $\mathcal{P}(\Sigma)^\top$ (HOL4-Thm 204), i. e. it is reflexive, antisymmetric and transitive. It can be easily used to define an order of actions:

Definition 3.2.24 (Order of Actions (HOL4-Thm 201))**.** For two actions act_1 and act_2 let $act_1 \sqsubseteq act_2$ be defined by

$$act_1 \sqsubseteq act_2 := \forall s.\ act_1(s) \subseteq_\top act_2(2)$$

$\sqsubseteq$ is a partial order of actions (HOL4-Thm 200). If $act_1 \sqsubseteq act_2$ holds, act_2 allows more behaviour than act_1, i. e. act_2 is an abstraction of act_1. This is expressed formally by the following lemma:

Lemma 3.2.25 ((HOL4-Thm 199))**.**

$$act_1 \sqsubseteq act_2 \iff \forall P, Q.\ \langle\!\langle P \rangle\!\rangle\ act_2\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ act_1\ \langle\!\langle Q \rangle\!\rangle$$

Lemma 3.2.26 (Lattice of local actions (HOL4-Thm 202))**.** The set of local actions *LocAct* forms together with $\sqsubseteq$ a complete lattice. This means that $\sqsubseteq$ is a partial order on *LocAct* and for each non empty subset $\mathcal{L}$ of *LocAct* there exists an infimum and a supremum.

$$\begin{array}{lllll} sup(\mathcal{L}) & = & \bigsqcup \mathcal{L} & = & \lambda s.\ \bigcup_\top \{act(s) \mid act \in \mathcal{L}\} \quad \text{(HOL4-Thms 234, 233, 232)} \\ inf(\mathcal{L}) & = & \bigsqcap \mathcal{L} & := & \lambda s.\ \bigcap_\top \{act(s) \mid act \in \mathcal{L}\} \quad \text{(HOL4-Thms 211, 209, 210)} \end{array}$$

Let's try to clarify the statement of this lemma. The supremum $sup(\mathcal{L})$ is defined by

$$\forall act \in \mathcal{L}.\ act \sqsubseteq sup(\mathcal{L})$$

$$\forall act'.\ (\forall act \in \mathcal{L}.\ act \sqsubseteq act') \Longrightarrow sup(\mathcal{L}) \sqsubseteq act'$$

According to Lemma 3.2.25 $\sqsubseteq$ is closely related to semantic Hoare triples. Therefore, the supremum $sup(\mathcal{L})$ can also be characterised by

$$\forall act \in \mathcal{L}, P, Q.\ \langle\!\langle P \rangle\!\rangle\ sup(\mathcal{L})\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle$$

$$\begin{array}{ll} \forall act'. & (\forall act \in \mathcal{L}, P, Q.\ \langle\!\langle P \rangle\!\rangle\ act'\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle) \Longrightarrow \\ & \forall P, Q.\ \langle\!\langle P \rangle\!\rangle\ act'\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ sup(\mathcal{L})\ \langle\!\langle Q \rangle\!\rangle \end{array}$$

So the supremum of $\mathcal{L}$ is the most specific action that is more general than all action in $\mathcal{L}$. This explains, why the supremum of $\mathcal{L}$ is the nondeterministic choice between the actions in $\mathcal{L}$.

Similarly, the infimum is the most general action that is more specific than any $act \in \mathcal{L}$:

$$\forall act \in \mathcal{L}, P, Q.\ \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle \implies \langle\!\langle P \rangle\!\rangle\ inf(\mathcal{L})\ \langle\!\langle Q \rangle\!\rangle$$

$$\begin{array}{ll} \forall act'. & (\forall act \in \mathcal{L}, P, Q.\ \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ act'\ \langle\!\langle Q \rangle\!\rangle) \Longrightarrow \\ & \forall P, Q.\ \langle\!\langle P \rangle\!\rangle\ inf(\mathcal{L})\ \langle\!\langle Q \rangle\!\rangle \Rightarrow \langle\!\langle P \rangle\!\rangle\ act'\ \langle\!\langle Q \rangle\!\rangle \end{array}$$

3.2.2.5 Best Local Action

This lattice of local actions is used to define a *best local action*:

Definition 3.2.27 (Best Local Action (HOL4-Thm 195))**.** Given a precondition P and a postcondition Q the *best local action* $bla[P, Q]$ is the supremum of the set of all local actions act that satisfy the $\langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle$:

$$bla[P, Q] := \bigsqcup \{act \mid act \text{ is local } \wedge\ \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle\}$$

As the supremum of a set of local actions $\mathcal{L}$, the best local action is itself local and more general than any action in $\mathcal{L}$. Moreover, $bla[P, Q]$ satisfies the semantic Hoare triple $\langle\!\langle P \rangle\!\rangle\ bla[P, Q]\ \langle\!\langle Q \rangle\!\rangle$:

Lemma 3.2.28 (Best Local Action Properties (HOL4-Thm 194))**.** For two predicates P and Q, the best local action $bla[P, Q]$ satisfies the following properties:

- $bla[P, Q]$ is a local action
- $\langle\!\langle P \rangle\!\rangle\ bla[P, Q]\ \langle\!\langle Q \rangle\!\rangle$
- $\forall act, P, Q. (act \text{ is local} \wedge \langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle) \Longrightarrow act \sqsubseteq bla[P, Q]$

Since $bla[P, Q]$ is a local action, it can be safely used without violating the frame rule. The triple $\langle\!\langle P \rangle\!\rangle\ bla[P, Q]\ \langle\!\langle Q \rangle\!\rangle$ provides a handle for reasoning about it. Finally, the last property allows actions act that satisfy $\langle\!\langle P \rangle\!\rangle\ act\ \langle\!\langle Q \rangle\!\rangle$ to be abstracted with $bla[P, Q]$.

Similar to the definition of local actions, there is also a more concise characterisation of best local actions:

Lemma 3.2.29. (HOL4-Thm 193) The best local action bla can also be defined by

$$bla[P, Q](s) = \bigcap_{\top} \{Q *_{\top} \{s_0\} \mid s_0 \circ s_1 = s\ \wedge\ s_1 \in P\}$$

3.2.2.6 Semaphore operations / Precise Predicates

The best local action is frequently used to define new actions. Examples are the definition of *materialisation* and *annihilation* which are used to handle semaphore operations.

Abstract Separation Logic supports simple semaphores. There are predefined locks with the usual operations P and V for allocating and releasing a lock. However, since Abstract Separation Logic uses an abstraction of a real programming language, their semantics is unusual. Instead of updating and checking some lock and perhaps blocking the current thread, they grant access to some part of the state protected by the lock. To this end, each lock is annotated with a predicate called *lock-invariant*. Acquiring the lock grants access to a part of the state that is described by this lock-invariant, releasing the lock removes this access. To this end, the local actions *materialisation* and *annihilation* are used:

Definition 3.2.30 (*materialisation*, *annihilation* (HOL4-Thms 74, 67, 129))**.** For a lock-invariant $I \in \mathcal{P}(\Sigma)$, the local actions *materialisation* and *annihilation* are defined by

$$\begin{array}{lcl} \mathit{materialisation}[I] & := & \mathit{bla}[\mathit{emp}, I] \\ \mathit{annihilation}[I] & := & \mathit{bla}[I, \mathit{emp}] \end{array}$$

This demonstrates nicely, that Abstract Separation Logic uses an abstraction of a programming language and that it is sometimes hard to see that this abstraction is sound. Even without considering the soundness of the abstraction, it is tricky to see that the new actions have the intended semantics.

Consider for example the semantics of *annihilation* with an invariant I on a state s. Let $\mathcal{S}_s$ be the set of states that results from removing I from s, i. e.:

$$\mathcal{S}_s = \{s_0 \mid s_1 \in I \ \wedge s = s_0 \circ s_1\}$$

It's easy to describe the result of $\mathit{annihilation}[I](s)$ using $\mathcal{S}_s$ (HOL4-Thm 68):

$$\mathit{annihilation}[I](s) = \begin{cases} \top & \text{if } |\mathcal{S}_s| = 0 \\ \mathcal{S}_s & \text{if } |\mathcal{S}_s| = 1 \\ \varnothing & \text{otherwise} \end{cases}$$

The first two cases are as one would expect. If $\mathcal{S}_s$ is empty, i. e. if no substate of s satisfies I, the *annihilation* action fails. Otherwise one might expect $\mathcal{S}_s$ as the result. This is however only true, if $\mathcal{S}_s$ has exactly one element. Otherwise the action diverges. This perhaps surprising behaviour is implied by the locality of *annihilation*.

In order to avoid such unintuitive behaviour, usually just *precise* predicates are used with annihilation:

Definition 3.2.31 (Precise Predicates (HOL4-Thm 131))**.** A predicate P is called *precise* iff for every state there is at most one substate that satisfies P, i. e. iff

$$\forall s, s_1, s_2.\ (s_1 \in P \wedge s_1 \preceq s) \ \wedge \ (s_2 \in P \wedge s_2 \preceq s) \implies s_1 = s_2$$

Using a precise invariant guarantees that the *annihilation* action does not diverge. However, the main reason for using precise invariants is, that the abstraction of semaphore operations is unsound for arbitrary invariants. Instead, Brookes uses only precise predicates as invariants to define the semantics of concurrent separation logic [5]. Showing that the programming language used by Abstract Separation Logic is a sound abstraction of a real programming language is outside the scope of this work, though. Therefore, the notion of precise predicates occurs only infrequently in this work.

3.2.2.7 Quantified Best Local Action

As shown with *materialisation* and *annihilation*, best local actions are used to define new actions. Another usage is to abstract blocks of code. Especially for the later purpose it is often useful to consider not just a single Hoare triple $\langle\!\langle P \rangle\!\rangle \,.\, \langle\!\langle Q \rangle\!\rangle$, but whole families of triples $\langle\!\langle P_1 \rangle\!\rangle \,.\, \langle\!\langle Q_1 \rangle\!\rangle$, $\langle\!\langle P_2 \rangle\!\rangle \,.\, \langle\!\langle Q_2 \rangle\!\rangle$, ... Such families are represented by $\forall i.\ \langle\!\langle P(i) \rangle\!\rangle \,.\, \langle\!\langle Q(i) \rangle\!\rangle$ using higher order quantification and *specification variables*. The lattice of actions as described by Lemma 3.2.26 provides concepts to extend best local actions to families:

Definition 3.2.32 (Quantified Best Local Action (HOL4-Thm 228))**.** Given two functions P_f and Q_f from an arbitrary argument type to predicates, i. e. given a family of pre- and post-conditions, the *quantified best local action* $qbla[P_f, Q_f]$ is the infimum of the set of best local actions with pre- and postconditions from this family:

$$qbla[P_f, Q_f] := \bigsqcap \{bla[P_f(x), Q_f(x)] \mid x \text{ arbitrary}\}$$

qbla is an extended version of *bla*. It has very similar properties. While the definition using the infimum is intuitive, definitions that are very similar to the definition of *bla* can be used as well:

Lemma 3.2.33 ((HOL4-Thms 231, 229))**.**

$$\begin{array}{rcl} qbla[P_f, Q_f] & = & \bigsqcup \{act \mid act \text{ is local } \wedge\ \forall x.\ \langle\!\langle P_f(x) \rangle\!\rangle\ act\ \langle\!\langle Q_f(x) \rangle\!\rangle\} \\ & = & \lambda s.\ \bigcap_{\top} \{Q_f(x) *_{\top} \{s_0\} \mid s_0 \circ s_1 = s\ \wedge\ s_1 \in P_f(x)\} \end{array}$$

Lemma 3.2.34 (Quantified Best Local Action Properties (HOL4-Thm 230))**.** For families of pre- and post-conditions P_f and Q_f the best local action $qbla[P_f, Q_f]$ satisfies the following properties:

- $qbla[P_f, Q_f]$ is a local action
- $\forall x.\ \langle\!\langle P_f(x) \rangle\!\rangle\ qbla[P_f, Q_f]\ \langle\!\langle Q_f(x) \rangle\!\rangle$
- $\forall act, P, Q.\ (act \text{ is local} \wedge \forall x.\ \langle\!\langle P_f(x) \rangle\!\rangle\ act\ \langle\!\langle Q_f(x) \rangle\!\rangle) \Longrightarrow act \sqsubseteq qbla[P_f, Q_f]$

3.2.2.8 *assume*

One important action of Abstract Separation Logic still needs to be introduced: *assume*. It is used to define programming constructs that need conditions. Examples are conditional execution or while loops. Given a predicate B, $assume[B](s)$ should skip, if $s \in B$ holds and diverge otherwise. However, *assume* should also be a local action. This means that if $assume[B]$ diverges for a state s_1, it has to diverge for all superstates s_2. In order to guarantee this, only special predicates are used with *assume*:

Definition 3.2.35 (Intuitionistic Predicate (HOL4-Thms 114, 116))**.** A predicate P is called *intuitionistic*, iff $P * true = P$ holds, i. e. iff

$$\forall s_1, s_2.\ s_1 \preceq s_2 \wedge s_1 \in P \Longrightarrow s_2 \in P$$

Intuitionistic predicates guarantee that once a predicate holds, it holds for all superstates. To guarantee that if it does not hold, it holds for no superstate as well, *intuitionistic negation* is used.

Definition 3.2.36 (Intuitionistic Negation (HOL4-Thm 113)). The *intuitionistic negation* $\neg_i P$ of a predicate P holds in a state s_1, if P does not hold for any superstate s_2 of s_1:

$$s_1 \in \neg_i P \;:=\; \forall s_2.\ s_1 \preceq s_2 \Longrightarrow (s_2 \notin P)$$

This concept of intuitionistic predicates and negation allows the definition of a local *assume* action with the intended semantics:

Definition 3.2.37 (*assume* (HOL4-Thms 69, 118)). For a predicate $B \in \mathcal{P}(\Sigma)$ the action *assume* is defined as follows:

$$\mathit{assume}[B](s) = \begin{cases} \{s\} & \text{if } s \in B \\ \varnothing & \text{if } s \in \neg_i B \\ \top & \text{otherwise} \end{cases}$$

$\mathit{assume}[B]$ is a local action for intuitionistic B.

There are three cases now. If B holds in all superstates $\mathit{assume}[B]$ skips. If it does not hold in any superstate, $\mathit{assume}[B]$ diverges. If – however – there are some superstates for which P holds and some for which it does not hold, i. e. if there are not sufficient resources to decide the intuitionistic predicate B, $\mathit{assume}[B]$ fails.

Figuring out, whether $\mathit{assume}[B]$ fails in a state or not is important. This motivates the following definition:

Definition 3.2.38 (Decided Predicate (HOL4-Thm 137)). A predicate P is called *decided* in a set of states $\mathcal{S}$, iff $\forall s \in \mathcal{S}.\ s \in P \ \vee s \in \neg_i P$ holds.

3.2.3 Programs

Now all the necessary actions have been presented to define the programming language of Abstract Separation Logic. The basic constructs of this language are local actions. Besides local actions, the language contains the usual control structures like conditional execution and while-loops. Additionally, nondeterminism, concurrency and simple semaphore operations are supported.

The language of the original work [7] is extended with procedures here. While the semantics still follow ideas from Brookes [5] about Concurrent Separation Logic, their presentation differs from the original one. This is partly due to adding procedures and partly due to the formalisation in HOL4.

It is not obvious, whether to use shallow or deep embeddings for the HOL4 formalisation. Actions are a good example: on the one hand, one would like to allow every HOL4 function of the right type to be used as a primitive construct of the programming language. That would provide a lot of flexibility and the possibility to extend the language very easily. On the other hand, each primitive construct should be a local action. Given a suitable definition of programs, the semantics of any program will then be a local action as well.

So, it appears easiest to use a deep embedding and a fixed set of local actions. Similar friction occurs on a higher level as well. In order to handle locks and procedure calls, it is useful to have a deep embedding and a set of dedicated operations which are the only ones that use locks and procedures. On the other hand, a shallow embedding with its flexibility is useful.

In this work, programs are formalised in a mixture of deep and shallow embeddings in order to combine the benefits of both. On the lowest level of the HOL4 formalisation, there are *primitive commands* – a wrapper around a shallow embedding of local actions. On the next layer there is a deep embedding of *traces*, which are sequences of primitive commands as well as special actions to take care of interleaving and lock operations. One layer up are *proto traces* which correspond to programs in the original work [7]. Proto traces are translated to a set of traces. This translation eliminates procedure calls and parallel composition. Finally, *programs* are shallowly embedded as sets of proto traces.

3.2.3.1 Programs, Proto Traces, Traces ...

Definition 3.2.39 (Proto-Trace)**.** The set of *proto-traces* PTr is inductively defined to be the smallest set with

- $act \in PTr$ for all local actions act
- $pt_1\ ;\ pt_2 \in PTr$ (sequential composition) for $pt_1, pt_2 \in PTr$
- $pt_1 \mid\mid pt_2 \in PTr$ (parallel composition) for $pt_1, pt_2 \in PTr$
- $\text{proccall}(name, arg) \in PTr$ (procedure call) for all procedure-names $name$ and all arguments arg
- $l.pt \in PTr$ (lock declaration) for a lock l and $pt \in PTr$
- $with\ l\ do\ pt \in PTr$ (critical region) for a lock l and $pt \in PTr$

Definition 3.2.40 (Program)**.** A program is a set of proto-traces. The set of all programs is denoted by *Prog*.

HOL4 remark 3.2.41. The definition of proto traces uses local actions. As motivated, these local actions are represented by a wrapper in HOL4 (HOL4-Thm 198). Given an action *act* the wrapper returns the action itself if it is local and *fail* otherwise. As *fail* is a local action (HOL4-Thm 121), this guarantees that the wrapper always returns a local action (HOL4-Thm 127). Similarly, there is a wrapper for intuitionistic predicates (HOL4-Thm 197). For a intuitionistic predicate B the wrapper returns B itself otherwise it returns *false*. Moreover the wrapper allows intuitionistic negation, conjunction and disjunction of predicates. All predicates returned by the wrapper are intuitionistic (HOL4-Thm 115).

The semantics of programs and proto traces is given by translating them to traces, i. e. to sequences of atomic actions.

Definition 3.2.42 (Atomic Action)**.** An *atomic action* is either a local action, a check $check(act_1, act_2)$ for local actions act_1, act_2 or a lock operation $P(l)$ or $V(l)$ for a lock l.

Definition 3.2.43 (Trace)**.** A trace is a list of atomic actions. Let ϵ denote the empty trace. The concatenation of two traces t_1, t_2 is denoted as $t_1 \cdot t_2$.

To define the traces of a program, an environment is needed that fixes the semantics of procedure calls. The idea is, that the procedure environment provides a body for each procedure call, i. e. for each procedure and all arguments the environment returns a program. The procedure call is replaced by this program.

Definition 3.2.44 (Procedure Environment)**.** A *procedure environment* is a finite map $penv : \textit{procedure-names} \xrightarrow{\text{fin}} (\textit{arguments} \rightarrow \textit{Prog})$ from procedure-names to a function from procedure arguments to programs.

There is support for mutual recursive procedures. However, traces are defined as lists and are therefore finite. Thus, one has to be careful to avoid unfolding procedures infinitely often which would result in infinite traces. This problem is solved by considering all traces that need at most depth n of nested procedure calls and then uniting these sets of traces. This results in an infinite set of finite traces. Traces that would need an infinite number of procedure unfoldings can be ignored, since Abstract Separation Logic considers partial correctness and these traces would not terminate.

Besides procedure calls, proto traces also take care of parallel composition. The parallel composition of traces is the set of all traces resulting from interleavings of the original traces. Besides just interleaving, *check* actions are inserted to enforce race-freedom. For all local actions that might be executed in parallel, a check is inserted to guarantee that these actions don't interfere with each other.

Definition 3.2.45 (Interleaving Traces (HOL4-Thm 187))**.**

$$
\begin{array}{rcl}
\textit{add-check}(a_1, a_2, t) & = & \begin{cases} \textit{check}(a_1, a_2) \cdot t & \text{if } a_1 \text{ and } a_2 \text{ are local actions} \\ t & \text{otherwise} \end{cases} \\
\epsilon \; \textit{zip} \; t & = & t \; \textit{zip} \; \epsilon = \{t\} \\
(a_1; t_1) \; \textit{zip} \; (a_2; t_2) & = & \Big\{ \textit{add-check}(a_1, a_2, t) \mid t \in \begin{array}{l} \{a_1; u \mid u \in t_1 \; \textit{zip} \; (a_2; t_2)\} \cup \\ \{a_2; u \mid u \in (a_1; t_1) \; \textit{zip} \; t_2\} \end{array} \Big\}
\end{array}
$$

Critical regions remain to be handled. Let *remove-locks(l,t)* (HOL4-Thm 185) remove all atomic actions concerned with the lock l, i. e. $P(l)$ and $V(l)$, from the trace t. Finally, a trace is l-*synchronised* (HOL4-Thm 184), iff the lock-actions $P(l)$ and $V(l)$ are properly matched. This allows the set of traces of a proto trace and a program to be defined as:

Definition 3.2.46 (Traces of Proto-traces / Programs (HOL4-Thms 176, 175, 161))**.** Given a procedure environment *penv*, the traces of a proto-trace that need at most nesting-

depth n for procedure calls (denoted as $T^n_{penv}(t)$) are given by:

$$
\begin{array}{rcl}
T^n_{penv}(act) & = & \{act\} \\
T^n_{penv}(pt_1 \ ; \ pt_2) & = & \{t_1 \cdot t_2 \mid t_1 \in T^n_{penv}(pt_1) \ \wedge \ t_2 \in T^n_{penv}(pt_2)\} \\
T^n_{penv}(pt_1 \ || \ pt_2) & = & \bigcup_{t_1 \in T^n_{penv}(pt_1), t_2 \in T^n_{penv}(pt_2)} t_1 \ zip \ t_2 \\
T^n_{penv}(\text{proccall}(name, arg)) & = & \begin{cases} \{fail\} & \text{if } name \notin dom(penv) \\ \emptyset & \text{if } name \in dom(penv) \wedge n = 0 \\ \bigcup_{pt \in penv(name, arg)} T^{n-1}_{penv}(pt) & \text{otherwise} \end{cases} \\
T^n_{penv}(l.pt) & = & \{remove\text{-}locks(l,t) \mid t \in T^n_{penv}(pt) \ \wedge \ t \text{ is } l\text{-synchronised}\} \\
T^n_{penv}(with \ l \ do \ pt) & = & \{P(l) \cdot t \cdot V(l) \mid t \in T^n_{penv}(pt)\}
\end{array}
$$

The traces of a proto-trace pt and a program p with respect to $penv$ are defined as

$$
\begin{array}{rcl}
T_{penv}(pt) & = & \bigcup_{n \in \mathbb{N}_0} T^n_{penv}(pt) \\
T_{penv}(p) & = & \bigcup_{pt \in p} T_{penv}(pt)
\end{array}
$$

3.2.3.2 Semantics of Programs, Proto Traces, Traces ...

After defining how to translate programs into a set of traces, it remains to define the semantics of traces in order to get a semantics for programs. Traces are sequences of local actions, checks and lock operations. A sequential composition operator for local actions as well as an informal semantics for checks is presented above. Moreover, it is discussed in Section 3.2.2.6 that Abstract Separation Logic uses precise lock-invariants and that the local actions *materialisation* and *annihilation* are used to model the semantics of semaphore operations.

Let $lenv : locks \rightarrow \mathcal{P}(\Sigma)$ be a *lock-environment*, i. e. a function that assigns a lock invariant to each lock. Then the semantics of an atomic action with respect to $lenv$ can be defined by:

Definition 3.2.47 (Semantics of Atomic Actions (HOL4-Thms 78, 70))**.** The semantics of an atomic action with respect to a *lock-environment* $lenv : locks \rightarrow \mathcal{P}(\Sigma)$ is given by

$$
\begin{array}{rcl}
[\![act]\!]_{lenv} & = & act \\
[\![check(act_1, act_2)]\!]_{lenv}(s) & = & \begin{cases} \{s\} & \text{if } \exists s_1, s_2. \ \ s = s_1 \circ s_2 \ \wedge \\ & \phantom{\text{if } \exists s_1, s_2. \ \ } act_1(s_1) \neq \top \ \wedge \ act_2(s_2) \neq \top \\ \top & \text{otherwise} \end{cases} \\
[\![P(l)]\!]_{lenv} & = & materialise(lenv(l)) \\
[\![V(l)]\!]_{lenv} & = & annihilate(lenv(l))
\end{array}
$$

This semantics is extended to the semantics of traces by using sequential composition and to the semantics programs using nondeterministic choice.

Definition 3.2.48 (Semantics of Programs, Proto-Traces, Traces (HOL4-Thms 159, 186))**.** The semantics of a trace with respect to a lock-environment is the sequential

combination of the semantics of its atomic actions. The semantics of a proto-traces and program is given by the nondeterministic choice between the semantics of its traces.

$$\begin{array}{rcl} \llbracket \epsilon \rrbracket_{lenv} & = & skip \\ \llbracket a \cdot t \rrbracket_{lenv} & = & \llbracket a \rrbracket_{lenv} \ ; \ \llbracket t \rrbracket_{lenv} \\ \llbracket pt \rrbracket_{(penv,lenv)} & = & \bigsqcup \ \{\llbracket t \rrbracket_{lenv} \mid t \in T_{penv}(pt)\} \\ \llbracket prog \rrbracket_{(penv,lenv)} & = & \bigsqcup \ \{\llbracket t \rrbracket_{lenv} \mid t \in T_{penv}(prog)\} \end{array}$$

Notice that the semantics of a program is a always a local action. This allows concepts for actions to be easily lifted to programs:

Lemma 3.2.49 ((HOL4-Thm 125))**.** For all procedure- and lock-environments *penv*, *lenv* and all programs *prog*, the semantics of the program $\llbracket prog \rrbracket_{(penv,lenv)}$ is a local action.

This is due to the construction. The semantics of an atomic action is a local action (HOL4-Thm 124). Because sequential composition and nondeterministic choice of local actions result in local actions (HOL4-Thms 122, 119) the semantics of traces (HOL4-Thm 126) and finally the semantics of programs (HOL4-Thm 125) are local actions.

Definition 3.2.50 (Hoare triple (HOL4-Thm 138))**.** A *Hoare triple* $\rhd_{env}\ \{P\}\ prog\ \{Q\}$ holds, iff $\langle\!\langle P \rangle\!\rangle\ \llbracket prog \rrbracket_{env}\ \langle\!\langle Q \rangle\!\rangle$ holds. If a Hoare triple holds for all environments, it is written as $\{P\}\ prog\ \{Q\}$.

Definition 3.2.51 (Program Abstractions (HOL4-Thms 139, 140))**.** A program p_2 is an abstraction of a program p_1 with respect to some environment *env* (denoted as $p_1 \sqsubseteq_{env} p_2$), iff $\llbracket p_1 \rrbracket_{env} \sqsubseteq \llbracket p_2 \rrbracket_{env}$ holds. Similar to Lemma 3.2.25, this can also be expressed as

$$\begin{array}{ll} p_1 \sqsubseteq_{env} p_2 & \Longleftrightarrow \\ \forall P, Q.\ \ \rhd_{env}\ \{P\}\ p_2\ \{Q\}\ \Rightarrow \rhd_{env}\ \{P\}\ p_1\ \{Q\} & \end{array}$$

$p_1 \sqsubseteq p_2$ is used to denote that p_2 is an abstraction of p_1 for all environments.

3.2.3.3 Comments on Semantics

The definition of the semantics of programs shows clearly that Abstract Separation Logic uses an abstraction of a programming language instead of an abstract programming language. The most obvious example is semaphore operations. One would expect that a simple real programming language acquires a lock before entering a critical section and returns it at the end. Acquiring the lock may involve waiting. An abstract language might model this in some way or, for example, just consider synchronised traces, i. e. traces that acquire and release locks in the right order. One would not expect the behaviour of Abstract Separation Logic, i. e. one would not expect some lock invariant *magically* appearing and disappearing. This is a high level abstraction of the behaviour of a real programming language. As discussed in section 3.2.2.6 it is not even obvious that this abstraction is sound. Brookes [5] uses precise predicates in order to guarantee it.

The restriction to synchronised traces in the definition of the lock declaration of proto-traces, which looks very sensible at first glance, might cause trouble as well. Imagine two proto-traces pt_{fail} and pt_l such that pt_l is not synchronised for the lock l and $\llbracket pt_{\text{fail}} \rrbracket_{env}$ fails when executed in a state s. Consider further the proto-trace $pt := (pt_{\text{fail}}\ ;\ l.pt_l)$. One would expect $\llbracket pt \rrbracket_{env}$ to fail in s as well, because first the failing proto-trace is executed.

However, this is not the case: $l.pt_l$ has no traces, i. e. $T_{penv}(l.pt_l) = \emptyset$, and therefore $T_{penv}(pt) = \emptyset$. This implies $[\![pt]\!]_{env} = diverge$. The problem is empty sets of traces. This may be caused by procedure calls and using the empty set as a program as well. The problem is circumvented by a suitable definition of sequential composition of programs.

Abstract Separation Logic is general, flexible and powerful. It is a good basis for a separation logic framework. However, the semantics of its programming language is far from intuitive. Brookes [5] proves that this programming language is, with certain side-conditions, a sound abstraction of a real programming language. In order to increase the trust in the separation logic framework developed here, it might be worthwhile for future work to formalise a programming language with an intuitive semantics inside HOL4 and prove that the programming language of Abstract Separation Logic is a sound abstraction of this language.

3.2.4 Common Programming Constructs

The programs introduced so far do not resemble the programs of standard imperative languages. Common constructs like loops or conditional execution are missing. However, these can be easily defined.

Every proto-trace pt can be regarded as the program $\{pt\}$. This immediately enriches the programming language with procedure calls and local actions. In particular, one can use *skip*, *fail*, *assume*, *diverge*, *bla* and *qbla* as programs. Since a shallow embedding of local actions is used in the HOL4 formalisation, it is very easy to define additional actions as well. Other constructs for proto-traces can be lifted to programs in the natural way:

Definition 3.2.52 (Parallel composition, Lock Declaration, Critical Region)**.**

$$\begin{array}{rcll} p_1 \,||\, p_2 & := & \{pt_1 \,||\, pt_2 \mid pt_1 \in p_1 \;\wedge\; pt_2 \in p_2\} & \text{(HOL4-Thm 171)} \\ l.p & := & \{l.pt \mid pt \in p\} & \text{(HOL4-Thm 169)} \\ \mathit{with}\ l\ \mathit{do}\ p & := & \{\mathit{with}\ l\ \mathit{do}\ pt \mid pt \in p\} & \text{(HOL4-Thm 166)} \end{array}$$

3.2.4.1 Sequential Composition

However, lifting sequential composition needs careful consideration. As discussed in Section 3.2.3.3 one has to take care to avoid programs with an empty trace-set. This is achieved by implicitly inserting *diverge* into the set of proto-traces:

Definition 3.2.53 (Sequential Composition of Programs (HOL4-Thm 173))**.**

$$p_1 \;;\; p_2 \;\;:=\;\; \{pt_1 \;;\; pt_2 \mid pt_1 \in p_1 \;\wedge\; pt_2 \in p_2 \cup \{diverge\}\}$$

Consider two programs p_1 and p_2 and an environment env such that $T_{env}(p_2) = \emptyset$. Without inserting *diverge* into the set of proto-traces of p_2, the set of traces $T_{env}(p_1 \;;\; p_2)$ would be empty as well, regardless of p_1. This would prevent errors in p_1 showing. Adding *diverge* is safe, because Abstract Separation Logic is for partial correctness. Moreover, it solves the problem and leads to the desired semantics as the following lemma demonstrates:

Lemma 3.2.54 ((HOL4-Thm 160))**.** The semantics of the sequential composition of two programs p_1 and p_2 in some environment env is the sequential composition of their semantics:

$$[\![p_1 \;;\; p_2]\!]_{env} = [\![p_1]\!]_{env} \;;\; [\![p_2]\!]_{env}$$

As usual sequential composition is extended to repetition and Kleene star:

Definition 3.2.55 (Repetition, Kleene Star)**.**

$$\begin{array}{rcll} p^0 & := & \mathit{skip} & \text{(HOL4-Thm 172)} \\ p^{n+1} & := & p \; ; \; p^n & \text{(HOL4-Thm 172)} \\ p^* & := & \bigcup_{n \in \mathbb{N}_0} p^n & \text{(HOL4-Thm 168)} \end{array}$$

3.2.4.2 Nondeterministic Choice

The original work on Abstract Separation Logic [7] explicitly defines nondeterministic choice as a command of the programming language. Here, nondeterministic choice is handled implicitly. Programs are sets of proto-traces, which correspond to the programs of the original work. The semantics of a program is the nondeterministic choice between all it's proto-traces. This allows the nondeterministic choice between two programs to be defined as the union operation on sets:

Definition 3.2.56 (Binary Nondeterministic Choice (HOL4-Thm 162))**.**

$$p_1 + p_2 = p_1 \cup p_2$$

However, while the original work is limited to a finite number of nondeterministic choices, the concept of programs as sets allows an infinite number. This is, for example, used to define the Kleene star operation above. While the original work introduces a special construct and a special semantics for Kleene star, this work can define it as nondeterministically choosing a number of repetitions.

3.2.4.3 Conditional Execution / While Loops

The combination of nondeterministic choice with *assume* allows the standard conditional execution and while-loops to be defined:

Definition 3.2.57 (Conditional Execution, While Loop (HOL4-Thms 165, 174))**.**

$$\begin{array}{rcl} \text{if } B \text{ then } p_1 \text{ else } p_2 & := & (\mathit{assume}(B); p_1) \; + \; (\mathit{assume}(\neg_i B); p_2) \\ \text{while } B \text{ do } p & := & (\mathit{assume}(B); p)^* \; ; \; \mathit{assume}(\neg_i B) \end{array}$$

These definitions of conditional execution and loops might be surprising. Remember however, that Abstract Separation Logic is reasoning about partial correctness. If the wrong branch of the conditional execution or the wrong number of iterations is chosen, a guarding *assume* statement causes the execution to diverge. Because only partial correctness is considered, diverging executions are ignored.

3.2.4.4 Conditional Critical Regions

Another control structure that can easily be defined is conditional critical regions. There is built-in support for critical regions. These can easily be equipped with conditions using *assume*:

Definition 3.2.58 (Conditional Critical Region (HOL4-Thm 164))**.**

$$\mathit{with}\ l\ \mathit{when}\ B\ \mathit{do}\ p := \mathit{with}\ l\ \mathit{do}\ (\mathit{assume}[B] \; ; \; p)$$

3.2.4.5 Infinite Nondeterministic Choice

When Kleene star is defined above, it is argued that being able to nondeterministically choose between an infinite number of choices is useful. Another example for this usefulness are procedure calls with call-by-value parameters.

The semantics of a procedure call proccall(*name*, *arg*) in an environment *penv* is defined by looking up the definition of the procedure in *penv*. The result of this lookup, i. e. the body of the procedure is instantiated with the argument *arg* and the semantics of the procedure call is defined by the semantics of the resulting program *penv*(*name*)(*arg*).

The argument *arg* can be considered as a call-by-reference argument. The procedure gets the argument and can do with it whatever it likes. A call-by-value argument would be evaluated before being passing to the procedure. This can be achieved by nondeterministically choosing a value and assuming that the argument evaluates to this value:

Definition 3.2.59 (Choose Constants (HOL4-Thm 163))**.** Let $e_1, \ldots, e_n$ be a list of functions that given a state either fail or returns some value. Then *choose-constants* for a program *prog* that depends on a list of values $c_1, \ldots, c_n$ is defined as:

$$\begin{array}{l} \mathit{choose\text{-}constants}([e_1, \ldots, e_n])(\lambda[c_1, \ldots, c_n].\mathit{prog}([c_1, \ldots, c_n])) \quad := \\ \bigcup\limits_{v_1, \ldots, v_n} \left(\mathit{assume}[\lambda s. \bigwedge\limits_{i=1,\ldots,n} v_i = e_i(s)] \ ; \ \mathit{prog}([v_1, \ldots, v_n]) \right) \end{array}$$

Wrapped around procedure calls this *choose-constants* construct is used to model call-by-value parameters.

3.2.5 Inference Rules

Using the semantics of Abstract Separation Logic as presented above, high level inference rules are proved. Instead of using the low-level semantics, these inference rules are used to reason about larger programs.

An inference rule represents an implication. The *program abstraction rule* states, for example, that if prog_2 is an abstraction of a program prog_1 and a Hoare triple $\rhd_{\mathit{env}}$ $\{P\}\ \mathit{prog}_2\ \{Q\}$ holds for this abstraction prog_2, then the Hoare triple $\rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_1\ \{Q\}$ holds as well for the original program prog_1. This inference rule is denoted by

$$\frac{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_2 \qquad \rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_2\ \{Q\}}{\rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_1\ \{Q\}}$$

Many inference rules represent not only implications, but equivalences. Since the abstraction relation is reflexive, i. e. since $\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_1$ holds for all environments *env* and all programs prog_1, the program abstraction rule is really an equivalence. Instead of

$$\begin{array}{ll} \forall \mathit{env}, \mathit{prog}_1, \mathit{prog}_2, P, Q. & \big(\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_2 \ \wedge\ \rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_2\ \{Q\}\big) \Longrightarrow \\ & \rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_1\ \{Q\} \end{array}$$

it can be written as

$$\begin{array}{ll} \forall \mathit{env}, \mathit{prog}_1, P, Q. & \big(\exists \mathit{prog}_2.\ \mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_2 \ \wedge\ \rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_2\ \{Q\}\big) \Longleftrightarrow \\ & \rhd_{\mathit{env}}\ \{P\}\ \mathit{prog}_1\ \{Q\} \end{array}$$

The inference notation presented above always represents an implication. A double line is used to express that there is a similar equivalence.

$$\frac{prog_1 \sqsubseteq_{env} prog_2 \qquad \rhd_{env} \{P\}\ prog_2\ \{Q\}}{\rhd_{env} \{P\}\ prog_1\ \{Q\}}$$

As shown on this simple example, the inference rule has usually to be modified slightly in order to form a real equivalence. The double line notation is mainly used to alert the reader to the fact, that this inference rule can (with care) safely be applied without losing some information about the original problem. As such, the program abstraction rule is written with a single line, because it is usually used in an unsafe way, i. e. its application usally loses information.

3.2.5.1 Frame Rule

As motivated in Section 3.2.2.3 local reasoning is an important feature of separation logic. Thanks to the careful construction of the programming language of Abstract Separation Logic, the semantics of a program is a local action (Lemma 3.2.49) (HOL4-Thm 125). Therefore, the frame rule for local actions can be lifted to the program level:

Frame Rule *(HOL4-Thm 91)*

$$\frac{\rhd_{env} \{P\}\ prog\ \{Q\}}{\rhd_{env} \{P * R\}\ prog\ \{Q * R\}}$$

This frame rule captures the essence of Abstract Separation Logic's local reasoning. Whenever a Hoare triple $\rhd_{env} \{P\}\ prog\ \{Q\}$ holds for some environment *env*, some program *prog* and some pre- and postcondition P and Q, this triple can safely be extended by an arbitrary context R.

3.2.5.2 Structural Rules

Besides this high level frame rule, which depends on the semantics of Abstract Separation Logic being carefully constructed, there are some simple structural inference rules, that just follow from the definition of Hoare triples:

Strengthen Rule
(HOL4-Thm 109)

$$\frac{P_2 \subseteq P_1 \qquad Q_1 \subseteq Q_2 \qquad \vartriangleright_{env} \{P_1\}\ prog\ \{Q_1\}}{\vartriangleright_{env} \{P_2\}\ prog\ \{Q_2\}}$$

Program Abstraction Rule
(HOL4-Thm 140)

$$\frac{prog_1 \sqsubseteq_{env} prog_2 \qquad \vartriangleright_{env} \{P\}\ prog_2\ \{Q\}}{\vartriangleright_{env} \{P\}\ prog_1\ \{Q\}}$$

(HOL4-Thm 86)

$$\frac{\forall x.\ \vartriangleright_{env} \{P(x)\}\ prog\ \{Q\}}{\vartriangleright_{env} \{\exists x.\ P(x)\}\ prog\ \{Q\}}$$

(HOL4-Thm 86)

$$\frac{\exists x.\ \vartriangleright_{env} \{P(x)\}\ prog\ \{Q\}}{\vartriangleright_{env} \{\forall x.\ P(x)\}\ prog\ \{Q\}}$$

(HOL4-Thm 86)

$$\frac{\exists x.\ \vartriangleright_{env} \{P\}\ prog\ \{Q(x)\}}{\vartriangleright_{env} \{P\}\ prog\ \{\exists x.\ Q(x)\}}$$

(HOL4-Thm 86)

$$\frac{\forall x.\ \vartriangleright_{env} \{P\}\ prog\ \{Q(x)\}}{\vartriangleright_{env} \{P\}\ prog\ \{\forall x.\ Q(x)\}}$$

(HOL4-Thm 90)

$$\frac{\forall i.\ \vartriangleright_{env} \{P_i\}\ prog\ \{Q_i\}}{\vartriangleright_{env} \{\bigvee_i P_i\}\ prog\ \{\bigvee_i Q_i\}}$$

(HOL4-Thm 89)

$$\frac{\forall i.\ \vartriangleright_{env} \{P_i\}\ prog\ \{Q_i\}}{\vartriangleright_{env} \{\bigwedge_i P_i\}\ prog\ \{\bigwedge_i Q_i\}}$$

3.2.5.3 Basic commands

For the basic actions lifted to programs there are the following inference rules:

(HOL4-Thm 104)

$$\frac{}{\vartriangleright_{env} \{P\}\ skip\ \{P\}}$$

(HOL4-Thm 96)

$$\frac{}{\vartriangleright_{env} \{P\}\ diverge\ \{Q\}}$$

(HOL4-Thm 87)

$$\frac{B \text{ is decided in } P}{\vartriangleright_{env} \{P\}\ assume(B)\ \{P \wedge B\}}$$

(HOL4-Thm 194)

$$\frac{}{\vartriangleright_{env} \{P\}\ bla[P, Q]\ \{Q\}}$$

(HOL4-Thm 101)

$$\frac{}{\vartriangleright_{env} \{P_f(x)\}\ qbla[P_f, Q_f]\ \{Q_f(x)\}}$$

(HOL4-Thm 102)

$$\frac{\exists x.\ \vartriangleright_{env} \{P\}\ bla[P_f(x), Q_f(x)]\ \{Q\}}{\vartriangleright_{env} \{P\}\ qbla[P_f, Q_f]\ \{Q\}}$$

3.2.5.4 Basic Program Compositions

Sequential Composition Rule
(HOL4-Thm 103)

$$\frac{\rhd_{env} \{P\}\ prog_1\ \{Q\} \qquad \rhd_{env} \{Q\}\ prog_2\ \{R\}}{\rhd_{env} \{P\}\ prog_1\ ;\ prog_2\ \{R\}}$$

Parallel Composition Rule
(HOL4-Thm 99)

$$\frac{\rhd_{env} \{P_1\}\ prog_1\ \{Q_1\} \qquad \rhd_{env} \{P_2\}\ prog_2\ \{Q_2\}}{\rhd_{env} \{P_1 * P_2\}\ prog_1 \mid\mid prog_2\ \{Q_1 * Q_2\}}$$

Nondeterministic Choice Rule
(HOL4-Thm 170)

$$\frac{\forall prog \in prog\text{-}set.\ \ \rhd_{env} \{P\}\ prog\ \{Q\}}{\rhd_{env} \{P\}\ \bigcup prog\text{-}set\ \{Q\}}$$

(HOL4-Thm 92)

$$\frac{\rhd_{env} \{P\}\ prog_1\ \{Q\} \qquad \rhd_{env} \{P\}\ prog_2\ \{Q\}}{\rhd_{env} \{P\}\ prog_1\ +\ prog_2\ \{Q\}}$$

(HOL4-Thm 97)

$$\frac{\rhd_{env} \{P\}\ prog\ \{P\}}{\rhd_{env} \{P\}\ prog^*\ \{P\}}$$

Most of these inference rules for basic program compositions are straightforward and hold for arbitrary Hoare logics. However the parallel composition rule is specific to separation logic. It expresses that the local reasoning of Abstract Separation Logic extends to parallelism.

3.2.5.5 Control Structures

Conditional Execution Rule
(HOL4-Thm 93)

$$\frac{\rhd_{env} \{P\}\ assume[B]; prog_t\ \{Q\} \qquad \rhd_{env} \{P\}\ assume[\neg_i B]; prog_f\ \{Q\}}{\rhd_{env} \{P\}\ \text{if } B \text{ then } prog_t \text{ else } prog_f\ \{Q\}}$$

Simple Loop-Invariant Rule
(HOL4-Thm 105)

$$\frac{B \text{ is decided in } P \qquad \rhd_{env} \{P \wedge B\}\ prog\ \{P\}}{\rhd_{env} \{P\}\ \text{while } B \text{ do } prog\ \{P \wedge \neg_i B\}}$$

Simple Loop-Specification Rule
(HOL4-Thm 107)

$$\frac{B \text{ is decided in } P \qquad \forall x.\ \rhd_{env} \{P(x) \wedge \neg_i B\}\ prog_2\ \{Q(x)\} \qquad \forall x.\ \rhd_{env} \{P(x) \wedge B\}\ prog_1\ ;\ qbla[P,Q]\ \{Q(x)\}}{\forall x.\ \rhd_{env} \{P(x)\}\ \text{while } B \text{ do } prog_1;\ prog_2\ \{Q(x)\}}$$

(HOL4-Thm 95)

$$\frac{lenv(l) \text{ is precise} \qquad \rhd_{(penv,lenv)} \{P * lenv(l)\}\ prog\ \{Q * lenv(l)\}}{\rhd_{(penv,lenv)} \{P\}\ with\ l\ do\ prog\ \{Q\}}$$

(HOL4-Thm 94)

$$\frac{\mathit{lenv}(l) \text{ is precise} \qquad B \text{ is decided in } P \qquad \rhd_{(\mathit{penv},\mathit{lenv})} \{(P * \mathit{lenv}(l)) \wedge B\}\ \mathit{prog}\ \{Q * \mathit{lenv}(l)\}}{\rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \mathit{with}\ l\ \mathit{when}\ B\ \mathit{do}\ \mathit{prog}\ \{Q\}}$$

(HOL4-Thm 98)

$$\frac{\mathit{lenv}(l) \text{ is precise} \qquad \rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \mathit{prog}\ \{Q\}}{\rhd_{(\mathit{penv},\mathit{lenv})} \{P * \mathit{lenv}(l)\}\ l.\mathit{prog}\ \{Q * \mathit{lenv}(l)\}}$$

(HOL4-Thm 111)

$$\frac{\begin{array}{c}\forall 1 \leqslant i \leqslant n, s \in P.\ e_i(s) = v_i \\ \forall 1 \leqslant i \leqslant n, s_1, s_2.\ (e_i(s_1) = v_i) \wedge s_1 \preceq s_2 \Longrightarrow (e_i(s_2) = v_i) \\ \rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \mathit{prog}(v_1, \ldots, v_n)\ \{Q\}\end{array}}{\rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \text{choose-constants}(e_1, \ldots, e_n)(\lambda c_1, \ldots, c_n.\mathit{prog}(c_1, \ldots, c_n))\ \{Q\}}$$

(HOL4-Thm 100)

$$\frac{\mathit{name} \in \mathit{dom}(\mathit{penv}) \qquad \rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \mathit{penv}(\mathit{name})(\mathit{arg})\ \{Q\}}{\rhd_{(\mathit{penv},\mathit{lenv})} \{P\}\ \text{proccall}(\mathit{name}, \mathit{arg})\ \{Q\}}$$

3.2.5.6 Symbolic Execution

All inference rules are presented in as concise a way as possible. In order to use these rules to verify specifications of a larger program, they are usually combined with the frame rule and the sequential composition rule (HOL4-Thm 103). The frame rule allows local reasoning, i. e. it allows to extended the specification with an arbitrary context. The sequential composition rule is essential for lifting the inference rules on single commands to whole programs. As usual there are two directions for using the sequential composition rule: forward and backward analysis. A classic tool for forward and backward analysis are strongest post- and weakest preconditions:

Definition 3.2.60 (Weakest Liberal Precondition (HOL4-Thms 191, 190) / Strongest Postcondition (HOL4-Thms 179, 178))**.** Given an environment *env*, a program *prog* and a postcondition Q the *weakest liberal precondition* $\mathit{wlp}_{\mathit{env}}[\mathit{prog}, Q]$ is the weakest precondition such that $\rhd_{\mathit{env}} \{\mathit{wlp}_{\mathit{env}}[\mathit{prog}, Q]\}\ \mathit{prog}\ \{Q\}$ holds. This means that it can be characterised as follows:

$$\begin{array}{c}\rhd_{\mathit{env}} \{\mathit{wlp}_{\mathit{env}}[\mathit{prog}, Q]\}\ \mathit{prog}\ \{Q\} \\ \forall P.\ \rhd_{\mathit{env}} \{P\}\ \mathit{prog}\ \{Q\} \Longrightarrow P \subseteq \mathit{wlp}_{\mathit{env}}[\mathit{prog}, Q]\end{array}$$

Given *env*, *prog* and a precondition P, the *strongest postcondition* $\mathit{sp}_{\mathit{env}}[P, \mathit{prog}]$ is the strongest postcondition such that $\rhd_{\mathit{env}} \{P\}\ \mathit{prog}\ \{\mathit{sp}_{\mathit{env}}[P, \mathit{prog}]\}$ holds. This means that it can be characterised as follows:

$$\begin{array}{c}\rhd_{\mathit{env}} \{P\}\ \mathit{prog}\ \{\mathit{sp}_{\mathit{env}}[P, \mathit{prog}]\} \\ \forall Q.\ \rhd_{\mathit{env}} \{P\}\ \mathit{prog}\ \{Q\} \Longrightarrow \mathit{sp}_{\mathit{env}}[P, \mathit{prog}] \subseteq Q\end{array}$$

Notice that if the program *prog* may fail when started from a state in P, then $\mathit{sp}_{\mathit{env}}[P, \mathit{prog}]$ is not defined. In contrast $\mathit{wlp}_{\mathit{env}}[\mathit{prog}, Q]$ exists for all *prog*, Q.

Using these definitions there are the following inference rules that can be used for forward and backwards analysis.

(HOL4-Thm 108)

$$\frac{sp_{env}[P, prog_1] \text{ is defined} \qquad \rhd_{env} \{sp_{env}[P, prog_1]\}\ prog_2\ \{Q\}}{\rhd_{env} \{P\}\ prog_1\ ;\ prog_2\ \{Q\}}$$

(HOL4-Thm 110)

$$\frac{\rhd_{env} \{P\}\ prog_1\ \{wlp_{env}[prog_2, Q]\}}{\rhd_{env} \{P\}\ prog_1\ ;\ prog_2\ \{Q\}}$$

Besides these foundations and a few basic lemmata (HOL4-Thms 181, 182, 192), the framework hardly uses weakest pre- and strongest post-conditions. Using weakest preconditions introduces complicated constructs like separation logic's magic-wand operator as well as many quantifiers [17]. Therefore, it is difficult to build an automated reasoning tool for separation logic that is using backward analysis.

Most tools use forward analysis and implement some kind of symbolic execution [2]. Inference rules for this forward analysis can be derived using strongest postconditions. This is occasionally done in the HOL4-formalisation. An example is the inference rule for *assume*:

Lemma 3.2.61 ((HOL4-Thms 180, 88))**.**

$$\begin{array}{rcl} sp_{env}[P, assume[B]] & = & \begin{cases} P \wedge B & \text{iff } B \text{ is decided in } P \\ \mathit{undefined} & \text{otherwise} \end{cases} \\ \rhd_{env} \{P\}\ assume[B]\ ;\ prog\ \{Q\} & \Longleftrightarrow & B \text{ is decided in } P \wedge \rhd_{env} \{P \wedge B\}\ prog\ \{Q\} \end{array}$$

For the purposes of the framework, it is not important most of the time to derive equivalences; implications are sufficient. By not using strongest postconditions one can save the effort to prove that a given postcondition is the strongest one. More importantly, the framework supports arbitrary inferences, not just the ones resulting from strongest postconditions. This includes inferences that modify the program as well as inferences that are proper implications, i. e. inferences for which no equivalence could be proved.

3.2.5.7 *assume*

An example of an inference rule that modifies the program and represents a proper implication is a specialised inference rule for the command $assume[B_1 \wedge B_2]$. The inference rule for *assume* presented above (HOL4-Thm 88) results in

$$\begin{array}{l} \rhd_{env} \{P\}\ assume[B_1 \wedge B_2]\ ;\ prog\ \{Q\} \qquad \Longleftrightarrow \\ (B_1 \wedge B_2) \text{ is decided in } P \wedge \rhd_{env} \{P \wedge (B_1 \wedge B_2)\}\ prog\ \{Q\} \end{array}$$

Usually, $P \wedge (B_1 \wedge B_2)$ will be converted into some kind of normal form after applying the inference rule (see Sec. 3.3.4). For complicated predicates B_1 and B_2 one might wish to perform this conversion into a normal form stepwise. Similarly, it might be beneficial to prove that both B_1 and B_2 are decided in P, which is a strictly stronger statement than $B_1 \wedge B_2$ is decided in P. To this end, one can use the following inference rule

$$\frac{\rhd_{env} \{P\}\ assume[B_1]\ ;\ assume[B_2]\ ;\ prog\ \{Q\}}{\rhd_{env} \{P\}\ assume[B_1 \wedge B_2]\ ;\ prog\ \{Q\}}$$

Instead of assuming $B_1 \wedge B_2$ one first assumes B_1 and then B_2. If B_1 is decided in P and B_2 is decided in $P \wedge B_1$ this inference represents an equivalence (HOL4-Thm 143). Otherwise, $\mathit{assume}[B_1]$; $\mathit{assume}[B_2]$ will fail for a state $s \in P$ and therefore the triple $\rhd_{\mathit{env}} \{P\}\ \mathit{assume}[B_1]\ ;\ \mathit{assume}[B_2]\ ;\ \mathit{prog}\ \{Q\}$ does not hold. However, the original triple might hold, if $B_1 \wedge B_2$ is decided in P. Thus, this inference represents in general just an implication.

This inference is proven by combining the program abstraction rule (HOL4-Thm 140)

$$\frac{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_2 \qquad \rhd_{\mathit{env}} \{P\}\ \mathit{prog}_2\ \{Q\}}{\rhd_{\mathit{env}} \{P\}\ \mathit{prog}_1\ \{Q\}}$$

with an abstraction lemma for $\mathit{assume}[B_1 \wedge B_2]$ (HOL4-Thm 142). There are similar program abstraction lemmata that can be used to break assumptions into smaller parts:

(HOL4-Thm 142)
$$\mathit{assume}[B_1 \wedge B_2] \sqsubseteq_{\mathit{env}} \mathit{assume}[B_1]; \mathit{assume}[B_2]$$

(HOL4-Thm 147)
$$\mathit{assume}[B_1 \vee B_2] \sqsubseteq_{\mathit{env}} \mathit{assume}[B_1] + \mathit{assume}[B_2]$$

(HOL4-Thm 144)
$$\mathit{assume}[\neg_i(B_1 \wedge B_2)] \sqsubseteq_{\mathit{env}} \mathit{assume}[(\neg_i B_1) \vee (\neg_i B_2)]$$

(HOL4-Thm 146)
$$\mathit{assume}[\neg_i(B_1 \vee B_2)] \sqsubseteq_{\mathit{env}} \mathit{assume}[(\neg_i B_1) \wedge (\neg_i B_2)]$$

(HOL4-Thm 145)
$$\mathit{assume}[\neg_i(\neg_i B)] \sqsubseteq_{\mathit{env}} \mathit{assume}[B]$$

Using the inferences rules resulting from these program abstractions is an essential part of how the framework handles *assume* (see Sec. 3.3.6.1).

3.2.6 Program Abstraction

The example of program abstractions for *assume* shows that program abstractions can be very useful. In order to gain this usefulness, abstractions for single actions or single program statements need to be lifted to larger programs. There are a lot of inference rules that achieve this lifting. The most essential ones are:

(HOL4-Thm 155)
$$\frac{}{\mathit{prog} \sqsubseteq_{\mathit{env}} \mathit{prog}}$$

(HOL4-Thm 157)
$$\frac{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_2 \qquad \mathit{prog}_2 \sqsubseteq_{\mathit{env}} \mathit{prog}_3}{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_3}$$

(HOL4-Thm 156)
$$\frac{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_1' \qquad \mathit{prog}_2 \sqsubseteq_{\mathit{env}} \mathit{prog}_2'}{\mathit{prog}_1\ ; \mathit{prog}_2 \sqsubseteq_{\mathit{env}} \mathit{prog}_1'\ ;\ \mathit{prog}_2'}$$

(HOL4-Thm 149)
$$\frac{\mathit{prog}_1 \sqsubseteq_{\mathit{env}} \mathit{prog}_1' \qquad \mathit{prog}_2 \sqsubseteq_{\mathit{env}} \mathit{prog}_2'}{\mathit{prog}_1 + \mathit{prog}_2 \sqsubseteq_{\mathit{env}} \mathit{prog}_1' + \mathit{prog}_2'}$$

These inference rules can easily be combined to form inference rules for higher level program constructs:

$$\text{(HOL4-Thm 151)}\quad \frac{prog_1 \sqsubseteq_{env} prog_2}{prog_1^* \sqsubseteq_{env} prog_2^*}$$

$$\text{(HOL4-Thm 150)}\quad \frac{prog_t \sqsubseteq_{env} prog_t' \qquad prog_f \sqsubseteq_{env} prog_f'}{\text{if } B \text{ then } prog_t \text{ else } prog_f \sqsubseteq_{env} \text{if } B \text{ then } prog_t' \text{ else } prog_f'}$$

$$\text{(HOL4-Thm 158)}\quad \frac{prog_1 \sqsubseteq_{env} prog_2}{\text{while } B \text{ do } prog_1 \sqsubseteq_{env} \text{while } B \text{ do } prog_2}$$

$$\text{(HOL4-Thm 153)}\quad \frac{}{with\ l\ do\ prog \sqsubseteq_{(penv,lenv)} bla[emp, lenv(l)]\ ;\ prog\ ;\ bla[lenv(l), emp]}$$

$$\text{(HOL4-Thm 141)}\quad \frac{lenv(l) \text{ is precise}}{l.prog \sqsubseteq_{(penv,lenv)} bla[lenv(l), emp]\ ;\ prog\ ;\ bla[emp, lenv(l)]}$$

As motivated before, there is a close connection between best local actions and abstraction. The best local action $bla[P, Q]$ is an abstraction of a program $prog$, if and only if $prog$ satisfies $\rhd_{env} \{P\}\ prog\ \{Q\}$:

$$\text{(HOL4-Thm 154)}\quad \frac{\forall x.\ \rhd_{env} \{P_f(x)\}\ prog\ \{Q_f(x)\}}{prog \sqsubseteq_{env} qbla[P_f, Q_f]} \text{ (double line)} \qquad \text{(HOL4-Thm 148)}\quad \frac{\rhd_{env} \{P\}\ prog\ \{Q\}}{prog \sqsubseteq_{env} bla[P, Q]} \text{ (double line)}$$

Abstracting a program with a best local action is often useful. When proving a Hoare triple $\rhd_{env} \{P\}\ prog_1\ ;\ prog_2\ \{Q\}$ using forward analysis, it is common that some family of Hoare triples $\forall x.\ \rhd_{env} \{P_f(x)\}\ prog_1\ \{Q_f(x)\}$ is known. In this case one needs to find a member of the family of specifications, i. e. an argument x, and a frame R such that P implies $P_f(x) * R$. Using the frame rule as well as the strengthen rule, one can then derive $\rhd_{env} \{P\}\ prog_1\ \{Q_f(x) * R\}$. Finally, the sequential composition rule reduces the original goal to $\rhd_{env} \{Q_f(x) * R\}\ prog_2\ \{Q\}$. Using the connection between program abstractions and best local actions as well as the definition of program abstractions (Def. 3.2.51), all such cases can be reduced to the symbolic execution of best local actions:

$$\frac{\forall x.\ \rhd_{env} \{P_f(x)\}\ prog_2\ \{Q_f(x)\}}{prog_1\ ;\ prog_2\ ;\ prog_3 \sqsubseteq_{env} prog_1\ ;\ qbla[P_f, Q_f]\ ;\ prog_3}$$

$$\frac{\forall x.\ \rhd_{env} \{P_f(x)\}\ prog_1\ \{Q_f(x)\} \qquad \rhd_{env} \{P\}\ qbla[P_f, Q_f]\ ;\ prog_2\ \{Q\}}{\rhd_{env} \{P\}\ prog_1\ ;\ prog_2\ \{Q\}}$$

Remark 3.2.62. Using symbolic execution of best local actions whenever a frame needs to be calculated simplifies automation considerably. This symbolic evaluation step is one of the most important parts of the infrastructure. All program commands can be abstracted by best local actions. The verification of a Hoare triple then reduces to the symbolic execution of best local actions. For the sake of performance, the framework often uses specialised inference rules for program commands, though. These specialised rules, however, are proved using symbolic execution of best local actions.

One simple, but important example for such an abstraction by a best local action, is an inference rule that is used for the forward analysis of while-loops. A simple rule using loop-invariants has already been presented above:

(HOL4-Thm 105)

$$\frac{\begin{array}{c} B \text{ is decided in } P \\ \rhd_{env}\, \{P \wedge B\}\ \mathit{prog}\ \{P\} \end{array}}{\rhd_{env}\, \{P\}\ \text{while } B \text{ do } \mathit{prog}\ \{P \wedge \neg_i B\}}$$

In order to modularise the verification effort and in order to facilitate forward analysis, one can keep *assume* and abstract the while-loop by a best local action. This leads to the following inference rule for while-loops:

Loop-Invariant Rule
(HOL4-Thm 106)

$$\frac{\begin{array}{c} \forall x.\ \rhd_{env}\, \{I_f(x)\}\ \mathit{assume}[B]\ ;\ \mathit{prog}_1\ \{I_f(x)\} \\ \rhd_{env}\, \{P\}\ \mathit{qbla}[I_f, I_f]\ ;\ \mathit{assume}[\neg_i B]\ ;\ \mathit{prog}_2\ \{Q\} \end{array}}{\rhd_{env}\, \{P\}\ \text{while } B \text{ do } \mathit{prog}_1\ ;\ \mathit{prog}_2\ \{Q\}}$$

Similarly, this leads to the following rule for loop-specifications:

Loop-Specification Rule
(HOL4-Thm 107)

$$\frac{\begin{array}{c} \forall x.\ \rhd_{env}\, \{P_f(x)\}\ \mathit{assume}[\neg_i B]\ ;\ \mathit{prog}_2\ \{Q_f(x)\} \\ \forall x.\ \rhd_{env}\, \{P_f(x)\}\ \mathit{assume}[B]\ ;\ \mathit{prog}_1\ ;\ \mathit{qbla}[P_f, Q_f]\ \{Q_f(x)\} \\ \rhd_{env}\, \{P\}\ \mathit{qbla}[P_f, Q_f]\ ;\ \mathit{prog}_3\ \{Q\} \end{array}}{\rhd_{env}\, \{P\}\ \text{while } B \text{ do } \mathit{prog}_1\ ;\ \mathit{prog}_2\ ;\ \mathit{prog}_3\ \{Q\}}$$

A more interesting example might be program abstraction for parallel composition. For sequential composition and nondeterministic choice, program abstraction is nicely modular. Unfortunately, it is more complicated for parallel composition. The following parallel composition rule was presented before:

Parallel Composition Rule
(HOL4-Thm 99)

$$\frac{\begin{array}{c} \rhd_{env}\, \{P_1\}\ \mathit{prog}_1\ \{Q_1\} \\ \rhd_{env}\, \{P_2\}\ \mathit{prog}_2\ \{Q_2\} \end{array}}{\rhd_{env}\, \{P_1 * P_2\}\ \mathit{prog}_1 \mid\mid \mathit{prog}_2\ \{Q_1 * Q_2\}}$$

Using best local actions and program abstractions, this rule can be rewritten to

(HOL4-Thm 152)

$$\frac{\mathit{prog}_1 \sqsubseteq_{env} \mathit{qbla}[P_{f_1}, Q_{f_1}] \qquad \mathit{prog}_2 \sqsubseteq_{env} \mathit{qbla}[P_{f_2}, Q_{f_2}]}{\mathit{prog}_1 \mid\mid \mathit{prog}_2 \sqsubseteq_{env} \mathit{qbla}[\lambda(x_1, x_2).\ P_{f_1}(x_1) * P_{f_2}(x_2),\ \lambda(x_1, x_2).\ Q_{f_1}(x_1) * Q_{f_2}(x_2)]}$$

3.2.7 Recursive Procedures

Till now inference rules for program constructs have been presented and extended in order to perform forward analysis of programs. Moreover program abstractions have been introduced. However, one important concept is still missing: reasoning about recursive procedures. The following inference rule for procedure calls is presented above:

$$\frac{\textit{(HOL4-Thm 100)} \quad \mathit{name} \in \mathit{dom}(\mathit{penv}) \qquad \rhd_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \mathit{penv}(\mathit{name})(\mathit{arg})\ \{Q\}}{\rhd_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \mathrm{proccall}(\mathit{name}, \mathit{arg})\ \{Q\}}$$

This rule is sufficient for non-recursive procedures. For recursive ones, using this rule would lead to unrolling the body of the procedure over and over again. Instead, some inductive argument is needed for recursive procedures. In order to handle mutually recursive procedures, this inductive argument has to consider several procedures at once. Combining the inference rule with an induction on the maximum nesting-depth of procedure calls during execution leads to the following lemma. For its validity it is essential that Abstract Separation Logic is just concerned with partial correctness and that non-terminating executions can therefore be ignored.

Lemma 3.2.63 ((HOL4-Thm 112))**.** In order to show that in a given environment $(\mathit{lenv}, \mathit{penv})$ some set of procedures satisfies given specifications, i. e. in order to show statements of the form

$$\begin{array}{l} \forall \mathit{arg}_1,\ x_1.\ \rhd_{(\mathit{penv},\mathit{lenv})}\ \{P_1(\mathit{arg}_1, x_1)\}\ \mathrm{proccall}(\mathit{name}_1, \mathit{arg}_1)\ \{Q_1(\mathit{arg}_1, x_1)\} \quad \wedge \\ \qquad\qquad\vdots \\ \forall \mathit{arg}_n,\ x_n.\ \rhd_{(\mathit{penv},\mathit{lenv})}\ \{P_n(\mathit{arg}_n, x_n)\}\ \mathrm{proccall}(\mathit{name}_n, \mathit{arg}_n)\ \{Q_n(\mathit{arg}_n, x_n)\} \end{array}$$

it is sufficient to show that for any procedure environment penv', such that these specifications hold for penv', the procedure bodies in the original environment penv satisfy the specifications:

$$\begin{array}{l} \forall \mathit{penv}'. \\ \left(\begin{array}{l} \forall \mathit{arg}_1,\ x_1.\ \rhd_{(\mathit{lenv},\mathit{penv}')}\ \{P_1(\mathit{arg}_1, x_1)\}\ \mathrm{proccall}(\mathit{name}_1, \mathit{arg}_1)\ \{Q_1(\mathit{arg}_1, x_1)\} \wedge \\ \qquad\qquad\vdots \\ \forall \mathit{arg}_n,\ x_n.\ \rhd_{(\mathit{lenv},\mathit{penv}')}\ \{P_n(\mathit{arg}_n, x_n)\}\ \mathrm{proccall}(\mathit{name}_n, \mathit{arg}_n)\ \{Q_n(\mathit{arg}_n, x_n)\} \end{array}\right) \Longrightarrow \\ \left(\begin{array}{l} \forall \mathit{arg}_1,\ x_1.\ \rhd_{(\mathit{lenv},\mathit{penv}')}\ \{P_1(\mathit{arg}_1, x_1)\}\ \mathit{penv}(\mathit{name}_1)(\mathit{arg}_1)\ \{Q_1(\mathit{arg}_1, x_1)\} \wedge \\ \qquad\qquad\vdots \\ \forall \mathit{arg}_n,\ x_n.\ \rhd_{(\mathit{lenv},\mathit{penv}')}\ \{P_n(\mathit{arg}_n, x_n)\}\ \mathit{penv}(\mathit{name}_n)(\mathit{arg}_n)\ \{Q_n(\mathit{arg}_n, x_n)\} \end{array}\right) \end{array}$$

When verifying a set of procedures, this lemma is usually applied as a first step. It is however, slightly awkward to apply. After applying the lemma one might have to keep track of a lot of preconditions stating that procedures satisfy their specifications. In order to avoid this bookkeeping, the preconditions are used to abstract procedure calls. A procedure call $\mathrm{proccall}(\mathit{name}_i, \mathit{arg}_i)$ is abstracted by its specification $\mathit{qbla}[P_i, Q_i]$. After abstraction, the semantics of the programs do not depend on the procedure environment penv' any more. Therefore, the preconditions are unimportant and can be dropped. Similarly, the lock environment lenv is removed by abstracting lock operations. After these preprocessing steps it remains to verify a collection of Hoare triples that do not depend on the environment any more.

3.2.8 Summary

The most important concepts of Abstract Separation Logic have been introduced above. First abstract states and separation combinators are presented. Combined with predicates seen as sets of states, these are the foundation of Abstract Separation Logics specification language.

Next, actions are presented and extended to a programming language. Compared to the original work on Abstract Separation Logic [7] the programming language has been extended by procedure calls. Another minor, but important extension is the generalisation of best local actions to quantified best local actions. The semantics of this extended programming language is, however, still given in terms of traces as defined by Brookes [5] for concurrent separation logic.

Based on this definition of Abstract Separation Logic's specification and programming language high level inference rules are presented. Using the concepts of program abstraction and quantified best local actions these inference rules are used to perform forward analysis on Hoare triples. In order to handle recursive procedure calls, a preprocessing step is necessary, that eliminates procedure calls.

It remains to instantiate this Abstract Separation Logic framework with a concrete separation combinator and concrete states. This allows programming constructs to be added that operate on these concrete states.

3.3 Variables as Resource

In Section 3.2 Abstract Separation Logic is presented. This section presents its first instantiation. A stack with explicit read / write permissions is added. This instantiations follows ideas from *Variables as Resource in Separation Logics* by Parkinson, Bornat and Calcagno [31]. However, these ideas are adapted to an Abstract Separation Logic setting. Moreover, they are extended to be powerful enough to base Holfoot on top.

This section is structured as follows: In Sec. 3.3.1 the stack is introduced. Then, expressions and predicates on the resulting states are discussed in Sec. 3.3.2 and 3.3.3. Well-formedness of expressions and predicates is an important topic in these sections. Based on these concepts of well-formedness normal forms for predicates are introduced in Sec. 3.3.4. This section also introduces special Hoare triples that implicitly guarantee that their pre- and postcondition is well-formed and that the program does not modify variable permissions. Sec. 3.3.5 shows that inference rules for standard Hoare triples can easily be lifted to these new Hoare triples. Sec. 3.3.6 discusses program constructs. First, it is discussed, how program constructs already introduced in Sec. 3.2 are used. Especially, the assume statement is discussed in detail. Then, a construct for local variable declaration and a variable assignment statement are introduced. Additionally, for the assignment statement semantic substitutions are defined. Sec. 3.3.7 discusses frame inference calculations. A specialised frame inference predicate is introduced and inference rules for the predicate presented. Finally, Sec. 3.3.8 presents, how to extend predicates with information that is only implicitly present.

3.3.1 Stacks with Read / Write Permissions

This instantiation adds a stack with explicit read / write permissions to the Abstract Separation Logic framework. The stack is a finite map from variables to values and permissions. It is not yet defined, what exactly variables and values are. Further instantiations can concretise variables and values. However, the type of permissions is fixed.

Definition 3.3.1 (Permissions (HOL4-Thm 254))**.** Let *Perms* be a set of permissions, $\top \in \mathit{Perms}$ a special permission and $\circledast : \mathit{Perms} \times \mathit{Perms} \rightharpoonup \mathit{Perms}$ a partial function such that

- $\circledast$ is partially associative, i. e.
  $\forall p_1, p_2, p_3.\ \ \text{Defined}(p_1 \circledast (p_2 \circledast p_3)) \Leftrightarrow \text{Defined}((p_1 \circledast p_2) \circledast p_3) \quad \wedge$
  $\forall p_1, p_2, p_3.\ \ \text{Defined}(p_1 \circledast (p_2 \circledast p_3)) \Longrightarrow (p_1 \circledast (p_2 \circledast p_3) = (p_1 \circledast p_2) \circledast p_3)$
- $\circledast$ is partially commutative, i. e.
  $\forall p_1, p_2.\ \ \text{Defined}(p_1 \circledast p_2) \Leftrightarrow \text{Defined}(p_2 \circledast p_1) \quad \wedge$
  $\forall p_1, p_2.\ \ \text{Defined}(p_1 \circledast p_2) \Longrightarrow (p_1 \circledast p_2 = p_2 \circledast p_1)$
- $\circledast$ is partially cancellative, i. e.
  $\forall p_1, p_2, p_3.\ \ \text{Defined}(p_1 \circledast p_2) \wedge \text{Defined}(p_1 \circledast p_3) \wedge$
  $(p_1 \circledast p_2 = p_1 \circledast p_3) \Longrightarrow (p_2 = p_3)$
- each permission can be split into two subpermissions, i. e.
  $\forall p.\ \exists p_1, p_2.\ (p_1 \circledast p_2) = p$
- $\top$ cannot be combined with any other permission, i. e.
  $\forall p. \neg\text{Defined}(\top \circledast p)$
- there is no unit element, i. e.
  $\forall p_1, p_2.\ (p_1 \circledast p_2) \neq p_1$

The idea behind this abstract definition of permissions is that there are read and write permissions. $\top$ represents the write permission, all other permissions are read permissions. A permission can be split into arbitrary many read permissions that can be recombined using $\circledast$.

Example 3.3.2 (Model of Permissions)**.** The definition of permissions and $\circledast$ are abstract. One model is, for example, obtained by setting:

- $\mathit{Perms} := \{x \in \mathbb{R} \mid 0 < x \leqslant 1\}$
- $\top := 1$
- $p_1 \circledast p_2 := \begin{cases} p_1 + p_2 & \text{if } p_1 + p_2 \leqslant 1 \\ \textit{undefined} & \text{otherwise} \end{cases}$

Splitting permissions naturally leads to a concept of subpermissions:

Definition 3.3.3 (Subpermissions (HOL4-Thm 257))**.** A permission p_1 is a subpermission of a permission p_2 (denoted by $p_1 \preceq p_2$), iff either p_1 equals p_2 or p_2 can be split into p_1 and another permission. This means:

$$p_1 \preceq p_2 \quad := \quad (p_1 = p_2) \vee \exists p_3.\ p_1 \circledast p_3 = p_2$$

This concept of permissions allows stacks with explicit permissions and a separation combinator for these stacks to be introduced:

Definition 3.3.4 (Stacks)**.** A stack is a finite map from variables to values and permissions.

$$\mathit{Stacks} \stackrel{\text{def}}{=} \mathit{Variables} \stackrel{\text{fin}}{\rightharpoonup} (\mathit{Values} \times \mathit{Permissions})$$

When provided with a variable x in its domain a stack s returns a pair consisting of a value and a permission (v, p). The functions *perm* and *val* are used to denote just the permission or the value, i. e. $\mathit{val}(s, x) = v$ and $\mathit{perm}(s, x) = p$.

Notice, that the notions of variables and values are kept abstract. They can be concretised by further instantiations. The notion of stacks is concrete enough, however, to define a separation combinator for stacks.

Definition 3.3.5 (Combining Stacks (HOL4-Thms 395, 392))**.** Two stacks s_1 and s_2 are separate (denoted by $s_1 \#_{St} s_2$), if they agree on the values of the variables that are present in both stacks and if the permissions of these variables are compatible. If s_1 and s_2 are not separate, their combination (denoted by $s_1 \bullet_{St} s_2$) is undefined, otherwise it is defined as the combination of the values and permissions of the two stacks.

$$\begin{array}{rcl} s_1 \#_{St} s_2 & := & \forall x, p_1, v_1, p_2, v_2.\ \big(x \in \mathit{dom}(s_1) \wedge x \in \mathit{dom}(s_2) \quad \wedge \\ & & \quad s_1(x) = (v_1, p_1) \quad \wedge \\ & & \quad s_2(x) = (v_2, p_2) \big) \Longrightarrow \\ & & \big((v_1 = v_2) \ \wedge\ \mathrm{Defined}(p_1 \circledast p_2)\big) \end{array}$$

$$\mathit{stack_merge}(s_1, s_2)(x) \ := \ \begin{cases} s_1(x) & \text{if } x \in \mathit{dom}(s_1) \ \wedge\ x \notin \mathit{dom}(s_2) \\ s_2(x) & \text{if } x \notin \mathit{dom}(s_1) \ \wedge\ x \in \mathit{dom}(s_2) \\ (v_1, p_1 \circledast p_2) & \text{if } x \in \mathit{dom}(s_1) \ \wedge\ x \in \mathit{dom}(s_2) \ \wedge \\ & \quad s_1(x) = (v_1, p_1) \ \wedge\ s_2(x) = (v_2, p_2) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$(s_1 \bullet_{St} s_2) \ := \ \begin{cases} \mathit{stack_merge}(s_1, s_2) & \text{if } s_1 \#_{St} s_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Notice that *stack_merge* is only applied to separate stacks. This avoids problems in the case that a variable is in the domain of both stacks.

Lemma 3.3.6 (Separation Algebra for Stacks (HOL4-Thms 394, 393))**.** The operation $\bullet_{St}$ is a separation combinator on stacks. The empty stack $\varnothing$ is the neutral element with respect to $\bullet_{St}$, i. e. $\forall s.\ s \bullet_{St} \varnothing = s$ holds. This means that $(\mathit{Stacks}, \bullet_{St}, \varnothing)$ is a separation algebra.

Following Definition 3.2.3 this separation combinator induces a substate relation for stacks. This relation will be useful in the following. It can be nicely characterised:

Lemma 3.3.7 (Substacks (HOL4-Thm 396))**.** A stack s_1 is a substack of a stack s_2, i. e. $s_1 \preceq_{\bullet_{St}} s_2$ holds, iff s_2 contains for each variable of s_1 the same value and a stronger

permission.

$$\begin{array}{lll}\forall s_1, s_2.\ s_1 \preceq_{\bullet_{St}} s_2 & \iff & dom(s_1) \subseteq dom(s_2)\ \wedge \\ & & \forall x \in dom(s_1), v_1, p_1, v_2, p_2. \\ & & \quad \big(s_1(v) = (v_1, p_1)\ \wedge\ s_2(v) = (v_2, p_2)\big) \implies \\ & & \quad \big(v_1 = v_2\ \wedge\ p_1 \preceq p_2\big)\end{array}$$

States of common programming languages consist of more than just a stack. In the case of Holfoot, for example, there is a heap as well. Therefore, the separation algebra for stacks is normally used as part of a product separation combinator.

Definition 3.3.8 (Product Separation Combinator (HOL4-Thm 263))**.** Let $\circ$ be a separation combinator on some set of states Σ. Then $\odot_\circ$ is defined as the product of this separation combinator with the separation combinator for stacks:

$$\odot_\circ := \bullet_{St} \times \circ$$

$\odot_\circ$ is a separation combinator on $Stacks \times \Sigma$ (HOL4-Thm 256). If $\bullet$ is a separation algebra, then $\odot_\bullet$ is a separation algebra as well (HOL4-Thm 255).

3.3.2 Expressions

Definition 3.3.9 (Expressions)**.** An *expression* is a partial function that given a stack returns a value. The most basic expressions are constants (HOL4-Thm 301), variables (HOL4-Thm 307) and function expressions (HOL4-Thms 303, 300). Constants always return a constant value. Variables look-up a value in the stack. Function expressions evaluate other expressions and apply a function to their results.

$$\begin{array}{lll} Const(c)(s) & := & c \\ Var(x)(s) & := & \begin{cases} v & \text{if } x \in dom(s)\ \wedge s(x) = (v, p) \\ undefined & \text{otherwise} \end{cases} \\ FunExp(f, [e_1, \ldots, e_n]) & := & \begin{cases} f([v_1, \ldots, v_n]) & \text{if } \forall 1 \leqslant i \leqslant n.\ \text{Defined}(e_i(s))\ \wedge \\ & \qquad e_i(s) = v_i \\ undefined & \text{otherwise} \end{cases} \end{array}$$

Since expressions are arbitrary functions from stacks to values, they may show arbitrary behaviour. The intention is however, that expressions are used to look up the values of some variables in the stack and perform some computation on these values. An expression should therefore not be affected by permissions. Moreover, there should be a finite set of variables that are used by the expression. The expression is defined, iff all these variables are present in the stack. Additional variables do not effect the expression. This notion of well-formed expressions is formalised as follows:

Definition 3.3.10 (Well-formed Expressions (HOL4-Thms 338, 339))**.** An expression e is *strongly well-formed* with respect to a finite set of variables $\mathcal{V}$, if the value of e in a

stack s depends exactly on the value of the variables $\mathcal{V}$ in this stack.

$$\begin{array}{lll} \mathit{isWellFormedStrong}(e,\mathcal{V}) & := & \forall s.\ \mathrm{Defined}(e(s)) \Leftrightarrow \mathcal{V} \subseteq \mathit{dom}(s) \wedge \\ & & \forall s_1, s_2.\ \Big(\mathcal{V} \subseteq \mathit{dom}(s_1) \wedge \mathcal{V} \subseteq \mathit{dom}(s_2) \wedge \\ & & \quad \forall v \in \mathcal{V}.\ \mathit{val}(s_1,v) = \mathit{val}(s_2,v) \Big) \Longrightarrow \\ & & \quad e(s_1) = e(s_2) \end{array}$$

Usually, an overapproximation of the set of used variables is sufficient. This leads to the definition of *well-formed* expressions:

$$\mathit{isWellFormed}(e,\mathcal{V}) \quad := \quad \exists \mathcal{V}' \subseteq \mathcal{V}.\ \mathit{isWellFormedStrong}(e,\mathcal{V}')$$

All expressions presented so far are well formed. This can easily be shown using the following inference rules.

(HOL4-Thm 340)

$$\frac{}{\mathit{isWellFormed}\big(\mathit{Const}(c),\mathcal{V}\big)}$$

(HOL4-Thm 340)

$$\frac{x \in \mathcal{V}}{\mathit{isWellFormed}\big(\mathit{Var}(x),\mathcal{V}\big)}$$

(HOL4-Thm 341)

$$\frac{\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}\big(e_i,\mathcal{V}\big)}{\mathit{isWellFormed}\big(\mathit{FunExp}(f,[e_1,\ldots,e_n]),\mathcal{V}\big)}$$

In the following, only well-formed expressions are considered. This is not a big restriction. Notice, however, that a well-formed expression is defined, iff all its variables are present in a stack. This means that undefined values can not be used to model failing computations of *FunExp*.

3.3.3 Predicates

As usual predicates on extended states $\mathit{Stacks} \times \Sigma$ are subsets of $\mathit{Stacks} \times \Sigma$. However, predicates that use the stack in a restricted way are of special interest. These restrictions are closely related to well-formed expressions.

3.3.3.1 Stack-Imprecise Predicates

Given an extended state $(s,h) \in \mathit{Stacks} \times \Sigma$, a predicate is supposed to use the stack s only to evaluate a finite set of well-formed expressions. This concept is captured syntactically by the following definition of *expression predicates*:

Definition 3.3.11 (Expression Predicates (HOL4-Thm 302))**.** Given a state $(s,h) \in \mathit{Stacks} \times \Sigma$, a list of well-formed expressions $e_1,\ldots,e_n$ and a predicate p that given n values returns a subset of Σ, the *expression predicate* $\mathit{ExpPred}(p,[e_1,\ldots,e_n])$ is defined as:

$$(s,h) \in \mathit{ExpPred}(p,[e_1,\ldots,e_n]) := \begin{cases} h \in p([v_1,\ldots,v_n]) & \text{if } \forall 1 \leqslant i \leqslant n.\ \mathrm{Defined}(e_i(s)) \wedge \\ & \quad e_i(s) = v_i \\ \mathit{false} & \text{otherwise} \end{cases}$$

In principle all predicates used by this instantiation should be defineable using *ExpPred*. However, a semantic characterisation is often more useful:

Definition 3.3.12 (Stack Imprecise Predicates (HOL4-Thm 344))**.** A predicate P on $Stacks \times \Sigma$ is called *stack imprecise* with respect to a set of variables $\mathcal{V}$, iff

$$\begin{aligned}\forall s_1, s_2, h.\ \ & (s_1, h) \in P \ \wedge\ (dom(s_1) \cap \mathcal{V} \subseteq dom(s_2)) \ \wedge \\ & \big(\forall x \in dom(s_1) \cap \mathcal{V}.\ val(s_1, x) = val(s_2, x)\big) \implies \\ & (s_2, h) \in P\end{aligned}$$

P is simply called *stack imprecise* if such a $\mathcal{V}$ exists.

When discussing Holfoot's separating conjunction operator in Section 2.1.3 it was informally argued that the stack does not need to be split. This is the case, because all predicates used by Holfoot are stack imprecise:

Lemma 3.3.13 (Separating Conjunction of Stack Imprecise Predicates (HOL4-Thm 251))**.** Let P_1 and P_2 be predicates on extended states $Stacks \times \Sigma$. Then $P_1 *_{\odot_\circ} P_2$ evaluates to (HOL4-Thm 227):

$$\begin{aligned}\forall P_1, P_2.\ \ & P_1 *_{\odot_\circ} P_2 = P_1 *_{(\bullet_{St} \times \circ)} P_2 = \\ & \lambda(s, h).\ \exists s_1, s_2, h_1, h_2.\ \ (s_1 \bullet_{St} s_2 = s) \ \wedge\ (h_1 \circ h_2 = h) \ \wedge \\ & \qquad\qquad\qquad\qquad\quad\ (s_1, h_1) \in P_1 \ \wedge (s_2, h_2) \in P_2\end{aligned}$$

So in general, the stack and the remainder of the state have both to be split. If, however, P_1 and P_2 are stack imprecise, the stack does not need to be split (HOL4-Thm 251):

$$\begin{aligned}\forall P_1, P_2.\ \ & \textit{stack-imprecise}(P_1) \ \wedge\ \textit{stack-imprecise}(P_2) \implies \\ & P_1 *_{\odot_\circ} P_2 = \lambda(s, h).\ \exists h_1, h_2.\ (h_1 \circ h_2 = h) \ \wedge\ (s, h_1) \in P_1 \ \wedge (s, h_2) \in P_2\end{aligned}$$

Stack imprecise predicates are an important concept. As already motivated, expression predicates are stack imprecise:

(HOL4-Thm 355)

$$\frac{\forall 1 \leqslant i \leqslant n.\ \textit{isWellFormed}\big(e_i, \mathcal{V}\big)}{\textit{isStackImprecise}\big(\textit{ExpPred}(p, [e_1, \ldots, e_n]), \mathcal{V}\big)}$$

Moreover, some basic predicates that have already been introduced are stack-imprecise

and stack impreciseness is preserved by common operations on predicates:

(HOL4-Thm 352)
$$\frac{}{\mathit{isStackImprecise}\big(\mathit{true}, \mathcal{V}\big)}$$

(HOL4-Thm 347)
$$\frac{}{\mathit{isStackImprecise}\big(\mathit{false}, \mathcal{V}\big)}$$

(HOL4-Thm 345)
$$\frac{\mathit{isStackImprecise}\big(P_1, \mathcal{V}\big) \quad \mathit{isStackImprecise}\big(P_2, \mathcal{V}\big)}{\mathit{isStackImprecise}\big(P_1 \wedge P_2, \mathcal{V}\big)}$$

(HOL4-Thm 349)
$$\frac{\mathit{isStackImprecise}\big(P_1, \mathcal{V}\big) \quad \mathit{isStackImprecise}\big(P_2, \mathcal{V}\big)}{\mathit{isStackImprecise}\big(P_1 \vee P_2, \mathcal{V}\big)}$$

(HOL4-Thm 351)
$$\frac{c \Rightarrow \mathit{isStackImprecise}\big(P, \mathcal{V}\big)}{\mathit{isStackImprecise}\big(c \,\&\, P, \mathcal{V}\big)}$$

(HOL4-Thm 346)
$$\frac{\forall x.\ \mathit{isStackImprecise}\big(P(x), \mathcal{V}\big)}{\mathit{isStackImprecise}\big(\exists x.\ P(x), \mathcal{V}\big)}$$

(HOL4-Thm 348)
$$\frac{\forall x.\ \mathit{isStackImprecise}\big(P(x), \mathcal{V}\big)}{\mathit{isStackImprecise}\big(\forall x.\ P(x), \mathcal{V}\big)}$$

(HOL4-Thm 350)
$$\frac{\mathit{isStackImprecise}\big(P_1, \mathcal{V}\big) \quad \mathit{isStackImprecise}\big(P_2, \mathcal{V}\big)}{\mathit{isStackImprecise}\big(P_1 * P_2, \mathcal{V}\big)}$$

So, most connectives and most basic predicates are stack-imprecise. In the following, only stack-imprecise predicates are used. Notice, that negation does not preserve stack-impreciseness in general. Moreover, the predicate *emp* is not stack-imprecise. It demands that the stack is empty (HOL4-Thm 238): $\mathit{emp}_{\odot_\circ} = \{(\varnothing, h) \mid h \in \mathit{emp}_\circ\}$. Because *emp* is not stack-imprecise, it is not used in the following. Instead a predicate *stack-true* is used that demands that the second state-component is empty, but allows arbitrary stacks:

Definition 3.3.14 (*stack-true* (HOL4-Thm 376))**.**

$$\mathit{stack\text{-}true}_\circ := \{(s, h) \mid h \in \mathit{emp}_\circ\}$$

stack-true is stack-imprecise (HOL4-Thm 356), i. e. $\forall \circ, \mathcal{V}.\ \mathit{isStackImprecise}\big(\mathit{stack\text{-}true}_\circ, \mathcal{V}\big)$. In general, $\mathit{stack\text{-}true}_\circ$ is not the neutral element with respect to $*_{\odot_\circ}$. It is, however, for stack-imprecise predicates P (HOL4-Thm 252), i. e.

$$\forall P.\ \mathit{isStackImprecise}(P) \Longrightarrow (P \ *_{\odot_\circ}\ \mathit{stack\text{-}true}_\circ = P)$$

Notice, that `emp` in Holfoot's input language (see Section 2.1.3) is implemented by *stack-true*.

3.3.3.2 Pure Predicates

Stack imprecise predicates use the stack in a restricted way. A further restriction is that the second component of the state may not be considered. This leads to *pure predicates*. Weak pure predicates accept any second component of the state, strong pure predicates q demand that it is empty:

Definition 3.3.15 (Pure Predicates (HOL4-Thms 369, 384))**.**

$$\begin{array}{rcl} \mathit{PurePred}_{weak}(p, el) & := & \mathit{ExpPred}(\lambda vl, h.\ p(vl),\ el) \\ \mathit{PurePred}_{strong}(p, el) & := & \mathit{ExpPred}(\lambda vl, h.\ p(vl)\ \wedge\ h \in \mathit{emp},\ el) \end{array}$$

Strong pure predicates are useful to express side conditions in specifications. Weak pure predicates are intuitionistic (HOL4-Thm 240). They can therefore be used as conditions with assume or control structures like conditional execution and while-loops (see Sec. 3.3.6.1). As pure predicates are defined in terms of expression predicates, they are stack imprecise.

An important special case are Boolean predicates:

Definition 3.3.16 (Boolean Predicates (HOL4-Thm 262))**.** Boolean predicates are strong pure predicates that do not depend on any expression, i. e. they do not depend on the stack.

$$\mathit{BoolPred}(c) \quad := \quad \mathit{PurePred}_{strong}(\lambda vl.\ c, [])$$

Boolean predicates do not depend on the state at all. It will be demonstrated later, that they can therefore be removed from Hoare triples and frame calculations.

Constant arguments to pure predicates can be eliminated. This elimination frequently leads to Boolean predicates, which can then be removed:

Lemma 3.3.17 (Constant Argument Elimination (HOL4-Thms 370, 371))**.**

$$\begin{array}{rcl} \mathit{PurePred}_{strong}(p, [\mathit{Const}(c), e_1, \ldots, e_n]) & = & \mathit{PurePred}_{strong}(\lambda vl.\ p(c :: vl), [e_1, \ldots, e_n]) \\ \mathit{PurePred}_{strong}(p, []) & = & \mathit{BoolPred}(p[]) \end{array}$$

Other important special cases of pure predicates are predicates on two expressions and especially equality checks:

Definition 3.3.18 ((HOL4-Thm 367))**.**

$$\begin{array}{rcl} \mathit{BinPurePred}_{weak}(op, e_1, e_2) & := & \mathit{PurePred}_{weak}(\lambda vl.\ \big(el(0, vl)\ op\ el(1, vl)\big), [e_1, e_2]) \\ \mathit{BinPurePred}_{strong}(op, e_1, e_2) & := & \mathit{PurePred}_{strong}(\lambda vl.\ \big(el(0, vl)\ op\ el(1, vl)\big), [e_1, e_2]) \end{array}$$

Definition 3.3.19 (Equality Checks (HOL4-Thms 368, 377, 383, 386))**.**

$$\begin{array}{rcl} e_1 = e_2 & := & \mathit{BinPurePred}_{strong}(=, e_1, e_2) \\ e_1 \neq e_2 & := & \mathit{BinPurePred}_{strong}(\neq, e_1, e_2) \\ e_1 =_{weak} e_2 & := & \mathit{BinPurePred}_{weak}(=, e_1, e_2) \\ e_1 \neq_{weak} e_2 & := & \mathit{BinPurePred}_{weak}(\neq, e_1, e_2) \end{array}$$

3.3.3.3 Separating Conjunction on Lists

Common predicates have been presented above. These are usually combined using the separating conjunction operator $*$. In order to simplify the syntax, the definition of star is extended to lists:

Definition 3.3.20 (Separating Conjunction on Lists (HOL4-Thm 260))**.**

$$\begin{array}{rcl} \circledast[] & = & \textit{stack-true} \\ \circledast(P :: pl) & = & P * \circledast pl \end{array}$$

This operation is intended to combine stack imprecise predicates. Therefore, it uses *stack-true* as base case instead of *emp* as one might expect (compare (HOL4-Thm 79)). This guarantees that the following inference rule holds even for the empty list []:

$$\frac{\textit{(HOL4-Thm 354)} \quad \forall P \in pl.\ \textit{isStackImprecise}\big(P, \mathcal{V}\big)}{\textit{isStackImprecise}\big(\circledast pl, \mathcal{V}\big)}$$

Since the separating conjunction operator $*$ is associative and commutative, the order of the list elements does not matter. Therefore, the lists can be replaced by finite multisets:

Definition 3.3.21 ((HOL4-Thm 261))**.**

$$\begin{array}{rcl} \circledast\varnothing & = & \textit{stack-true} \\ \circledast(\{P\} \cup \mathcal{P}) & = & P * \circledast\mathcal{P} \end{array}$$

$$\frac{\textit{(HOL4-Thm 353)} \quad \forall P \in \mathcal{S}.\ \textit{isStackImprecise}\big(P, \mathcal{V}\big)}{\textit{isStackImprecise}\big(\circledast\mathcal{S}, \mathcal{V}\big)}$$

It is also often useful to map some function f over a list l before combining all the predicates in the resulting list $\mathit{map}(f, l)$. This leads to the separating map operator:

Definition 3.3.22 (Separating Map (HOL4-Thm 359))**.**

$$\circledast\text{-}\mathit{map}(f, l) := \circledast(\mathit{map}(f, l))$$

3.3.4 Normal Forms

Stack imprecise predicates demand implicitly that the variables they access are present in the stack, i. e. that they have read permissions for these variables. In order to specify write permissions as well, the following normal form is used:

Definition 3.3.23 ((HOL4-Thm 375))**.** For sets of variables $\mathcal{W}$, $\mathcal{R}$, a list of variables d and a predicate P, the predicate $\mathit{inputVRProp}(\mathcal{W}, \mathcal{R}, d, P)$ checks for a state $(s, h) \in \mathit{Stacks} \times \Sigma$ that

- the stack s contains write permissions for the variables in $\mathcal{W}$,
- s contains read permissions for the variables in $\mathcal{R}$,
- all variables in d are distinct from each other and
- P holds in (s, h).

$$\begin{array}{rcl} \mathit{inputVRProp}(\mathcal{W}, \mathcal{R}, d, P) & := & \lambda(s,h).\ \ \mathcal{W} \cup \mathcal{R} \subset \mathit{dom}(s) \wedge \\ & & \forall x \in \mathcal{W}.\ \mathit{perm}(x,s) = \top \wedge \\ & & \mathit{all\text{-}distinct}(d) \wedge \\ & & (s,h) \in P \end{array}$$

The sets $\mathcal{W}$ and $\mathcal{R}$ allow to specify the necessary read and write permissions. It is usually desirable to know that certain variables are distinct from each other; d can be used for this purpose. This is often used to specify that all the variables given as call-by-reference parameters to a procedure call are distinct. Finally, P is the main predicate.

P is usually a spatial conjunction of several predicates, i. e. P is of the form $P_1 * \ldots * P_n$. It is desirable that all predicates P_i are stack imprecise. Furthermore, the variables described by $\mathcal{W}$ and $\mathcal{R}$ should be the only ones needed. Combining this condition with distinctiveness of the variables leads to an additional normal form.

Definition 3.3.24 ((HOL4-Thms 389, 387, 390))**.** For finite multisets of variables $\mathcal{W}$, $\mathcal{R}$ and a finite multiset of predicates $\mathcal{P}$, the predicate $\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P})$ checks for a state $(s, h) \in \mathit{Stacks} \times \Sigma$ that

- the stack s contains write permissions for the variables in $\mathcal{W}$,
- read permissions for the variables in $\mathcal{R}$ and
- the separating conjunction of all the predicates in $\mathcal{P}$ holds in (s, h).

The function $\mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{P})$ checks that all the variables in $\mathcal{W} \cup \mathcal{R}$ are distinct and that all predicates $P \in \mathcal{P}$ are stack imprecise with respect to $\mathcal{W} \cup \mathcal{R}$. This leads to the following formal definitions:

$$\begin{array}{rcl} \mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P}) & := & \lambda(s,h).\ \ \mathcal{W} \cup \mathcal{R} \subseteq \mathit{dom}(s) \wedge \\ & & \forall x \in \mathcal{W}.\ \mathit{perm}(x,s) = \top \wedge \\ & & (s,h) \in \circledast\mathcal{P} \end{array}$$

$$\begin{array}{rcl} \mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{P}) & := & \text{all variables in the multiset } \mathcal{W} \cup \mathcal{R} \text{ are distinct} \wedge \\ & & \forall P \in \mathcal{P}.\ \mathit{isStackImprecise}(P, \mathcal{W} \cup \mathcal{R}) \end{array}$$

VRProp and *VRCond* are very important in the following. *VRProp* is usually used to describe the pre- and postconditions of Hoare triples. A typical Hoare triple is of the form $\{\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P}_{pre})\}\ \mathit{prog}\ \{\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P}_{post})\}$. In rare cases the postcondition may use different variable permissions than the precondition, though.

Unfortunately, these Hoare triples are not strong enough, yet. Consider the case of calling a procedure that needs read access to a variable x. If the caller has write permission for x, it should still have write permission after the procedure call. However, *VRProp* allows only to specify that if there is a read permission for x before the procedure call, then there is a read permission afterwards. It is not guaranteed that the permission is not modified.

In order to preserve permissions, it is first defined when two stacks are *equal with respect to permissions*.

Definition 3.3.25 ((HOL4-Thm 397))**.** A stack s_1 is *equal with respect to permissions* to a stack s_2 (denoted by $s_1 \overset{\text{perms}}{=} s_2$), iff all variables that occur in both stacks have the

same permissions and all other variables have write permissions.

$$\begin{array}{rcl} s_1 \overset{\text{perms}}{=} s_2 & := & \forall x.\ x \in dom(s1) \wedge x \in dom(s_2) \implies (perm(s_1, x) = perm(s_2, x)) \wedge \\ & & \forall x.\ x \in dom(s1) \wedge x \notin dom(s_2) \implies (perm(s_1, x) = \top) \wedge \\ & & \forall x.\ x \notin dom(s1) \wedge x \in dom(s_2) \implies (perm(s_2, x) = \top) \end{array}$$

This new definition is used to introduce *Variable as Resource Hoare Triples*:

Definition 3.3.26 (Variable as Resource Hoare Triples (HOL4-Thms 327))**.** Variable as resource Hoare triples $[P]\ prog\ [Q]$ are defined as follows:

$$\begin{array}{rl} [P]\ prog\ [Q] := \forall S. & \{P \ \wedge\ (\lambda(s,h).\ s = S)\} \\ & \quad prog \\ & \{Q \ \wedge\ (\lambda(s,h).\ s \overset{\text{perms}}{=} S)\} \end{array}$$

The idea of these Hoare triples is that programs cannot modify permissions. If they have write permission to a variable, they can change the value of that variable and even remove that variable from the stack. They can't, however, somehow remove only a part of the write permission. Similarly, if they introduce new stack variables these variables come with write permissions.

Integrating well-formedness conditions into variable as resource Hoare triples leads to *Conditional Variable as Resource Hoare Triples.*

Definition 3.3.27 (Conditional Variable as Resource Hoare Triples (HOL4-Thm 265))**.**

$$[\![(P_b, P_p)]\!]\ prog\ [\![(Q_b, Q_p)]\!] \ := \ (P_b \wedge Q_b) \implies [P_p]\ prog\ [Q_p]$$

Notation 3.3.28. Normally, variable as resource Hoare triples use *VRProp* for their conditions. Sometimes, additional existential quantification is required. Let therefore $\mathcal{W}; \mathcal{R} \mid \mathcal{P}$ denote either $VRProp(\mathcal{W}, \mathcal{R}, \mathcal{P})$ or $(VRCond(\mathcal{W}, \mathcal{R}, \mathcal{P}), VRProp(\mathcal{W}, \mathcal{R}, \mathcal{P}))$ depending on context. Similarly, let $\mathcal{W}; \mathcal{R} \mid \exists x.\ \mathcal{P}(x)$ denote either $\exists x. VRProp(\mathcal{W}, \mathcal{R}, \mathcal{P}(x))$ or $(\forall x.\ VRCond(\mathcal{W}, \mathcal{R}, \mathcal{P}(x)), \exists x.\ VRProp(\mathcal{W}, \mathcal{R}, \mathcal{P}(x)))$ depending on context. This means that for example the following notations are valid abbreviations:

$$\begin{array}{ll} [\mathcal{W}_1; \mathcal{R}_1 \mid \mathcal{P}_1]\ prog\ [\mathcal{W}_2; \mathcal{R}_2 \mid \mathcal{P}_2] & = \\ [VRProp(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1)]\ prog\ [VRProp(\mathcal{W}_2, \mathcal{R}_2, \mathcal{P}_2)] & \\ & \\ [\mathcal{W}_1; \mathcal{R}_1 \mid \mathcal{P}_1]\ prog\ [Q] & = \\ [VRProp(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1)]\ prog\ [Q] & \\ & \\ [\![\mathcal{W}_1; \mathcal{R}_1 \mid \mathcal{P}_1]\!]\ prog\ [\![\mathcal{W}_2; \mathcal{R}_2 \mid \exists x.\ \mathcal{P}_2(x)]\!] & = \\ [\![(VRCond(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1),\ VRProp(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1))]\!] & \\ \quad prog & \\ [\![(\forall x.\ VRCond(\mathcal{W}_2, \mathcal{R}_2, \mathcal{P}_2(x)), \exists x.\ VRProp(\mathcal{W}_2, \mathcal{R}_2, \mathcal{P}_2(x)))]\!] & \\ & \\ [\![\mathcal{W}_1; \mathcal{R}_1 \mid \mathcal{P}_1]\!]\ prog\ [\![Q]\!] & = \\ [\![(VRCond(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1),\ VRProp(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1))]\!]\ prog\ [\![Q]\!] & \end{array}$$

Remark 3.3.29. It might be surprising that conditional Hoare triples assume that both the pre- and the postcondition are well-formed. The idea of conditional Hoare triples is to carry as much information as possible. Holfoot uses general Hoare triples as input. The pre- and postconditions of these Hoare triples use *inputVRProp*. In a preprocessing step, these general Hoare triples are transformed into conditional Hoare triples that use *VRProp*. This transformation involves showing the well-formedness of predicates and the distinctiveness of variables. These properties are shown once and then recorded using conditional Hoare triples. This allows the verification to rely on these properties without proving them over and over again.

As discussed above, best local actions are closely related to Hoare triples. There is for example the following connection:

$$\begin{array}{rcll} prog \sqsubseteq bla[P,Q] & \iff & \{P\}\ prog\ \{Q\} & \text{(HOL4-Thm 148)} \\ prog \sqsubseteq qbla[P_f,Q_f] & \iff & \forall x.\ \{P(x)\}\ prog\ \{Q(x)\} & \text{(HOL4-Thm 154)} \end{array}$$

In order to establish similar relationships for variables as resource Hoare triples, best local actions corresponding to the Hoare triples are introduced:

Definition 3.3.30 (Best Local Action (HOL4-Thms 259, 391))**.**

$$vrbla[P,Q] \quad := \quad \begin{array}{l} qbla[\lambda t,(s,h).\ (s,h) \in P \ \wedge\ (s = t), \\ \quad \lambda t,(s,h).\ (s,h) \in Q \ \wedge\ (s \overset{\text{perms}}{=} t)] \end{array}$$

$$qvrbla[P_f,Q_f] \quad := \quad \begin{array}{l} qbla[\lambda(x,t),(s,h).\ (s,h) \in P(x) \ \wedge\ (s = t), \\ \quad \lambda(x,t),(s,h).\ (s,h) \in Q(x) \ \wedge\ (s \overset{\text{perms}}{=} t)] \end{array}$$

These new best local actions correspond to variables as resource Hoare triples:

Lemma 3.3.31.

$$\begin{array}{rcll} prog \sqsubseteq vrbla[P,Q] & \iff & [P]\ prog\ [Q] & \text{(HOL4-Thm 245)} \\ prog \sqsubseteq qvrbla[P_f,Q_f] & \iff & \forall x.\ [P(x)]\ prog\ [Q(x)] & \text{(HOL4-Thm 250)} \end{array}$$

There are conditional versions as well. Notice however, that these do not correspond directly to conditional Hoare triples. They serve the same purpose of encoding well-formedness side-conditions.

Definition 3.3.32 (Conditional Best Local Action (HOL4-Thms 264, 298))**.**

$$cvrbla[(P_b,P_p),(Q_b,Q_p)] \quad := \quad \text{if } (P_b \wedge Q_b) \text{ then } vrbla[P_p,Q_p] \text{ else } diverge$$

$$\begin{array}{l} qcvrbla[\lambda x.\ (P_{f_b}(x),P_{f_p}(x)), \\ \quad \lambda x.\ (Q_{f_b}(x),Q_{f_p}(x))] \end{array} := \begin{array}{l} \text{if } (\forall x.\ P_{f_b}(x) \wedge Q_{f_b}(x)) \text{ then } qvrbla[P_{f_p},Q_{f_p}] \\ \qquad \text{else } diverge \end{array}$$

3.3.5 Inference Rules

$\overset{\text{perms}}{=}$ is an equivalence relation (HOL4-Thms 398, 399, 400). Moreover, it is compatible with combining stacks:

Lemma 3.3.33 ((HOL4-Thm 401)).

$$\forall s, s_1, s_2, t, t_1, t_2.\ (s_1 \bullet_{St} s_2 = s) \ \wedge\ (t_1 \bullet_{St} t_2 = t) \ \wedge\ (s_1 \overset{\text{perms}}{=} t_1) \ \wedge\ (s_2 \overset{\text{perms}}{=} t_2) \implies s \overset{\text{perms}}{=} t$$

These properties of $\overset{\text{perms}}{=}$ guarantee that most of the inferences that hold for general Hoare triples (see Sec. 3.2.5), hold for variable as resource Hoare triples as well. This includes the following important inference rules:

Sequential Composition Rule *(HOL4-Thm 335)*

$$\frac{[P]\ prog_1\ [Q] \qquad [Q]\ prog_2\ [R]}{[P]\ prog_1\ ;\ prog_2\ [R]}$$

Parallel Composition Rule *(HOL4-Thm 334)*

$$\frac{[P_1]\ prog_1\ [Q_1] \qquad [P_2]\ prog_2\ [Q_2]}{[P_1 * P_2]\ prog_1 \mid\mid prog_2\ [Q_1 * Q_2]}$$

Frame Rule *(HOL4-Thm 332)*

$$\frac{[P]\ prog\ [Q]}{[P * R]\ prog\ [Q * R]}$$

Program Abstraction Rule *(HOL4-Thm 267)*

$$\frac{prog_1 \sqsubseteq prog_2 \qquad [P]\ prog_2\ [Q]}{[P]\ prog_1\ [Q]}$$

(HOL4-Thm 286)

$$\frac{[P]\ prog_1\ ;\ prog_3\ [Q] \qquad [P]\ prog_2\ ;\ prog_3\ [Q]}{[P]\ (prog_1\ +\ prog_2)\ ;\ prog_3\ [Q]}$$

(HOL4-Thm 333)

$$\frac{[P]\ prog\ [P]}{[P]\ prog^*\ [P]}$$

To restore the normal form after applying the parallel composition rule, the following lemma is useful.

Lemma 3.3.34 ((HOL4-Thm 253)).

$$\begin{array}{l}\forall \mathcal{W}_1, \mathcal{W}_2, \mathcal{R}_1, \mathcal{R}_2, \mathcal{P}_1, \mathcal{P}2. \\ \quad disjoint(\mathcal{W}_1, \mathcal{W}_2 \cup \mathcal{R}_2) \ \wedge\ disjoint(\mathcal{W}_2, \mathcal{W}_1 \cup \mathcal{R}_1) \ \wedge \\ \quad VRCond(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1) \ \wedge\ VRCond(\mathcal{W}_2, \mathcal{R}_2, \mathcal{P}_2) \implies \\ \quad \big(VRProp(\mathcal{W}_1, \mathcal{R}_1, \mathcal{P}_1) * VRProp(\mathcal{W}_2, \mathcal{R}_2, \mathcal{P}_2) = VRProp(\mathcal{W}_1 \cup \mathcal{W}_2, \mathcal{R}_1 \sqcup \mathcal{R}_2, \mathcal{P}_1 \cup \mathcal{P}_2)\big)\end{array}$$

Other inference rules exploit the structure of variable as resource Hoare triples. The strengthening rule can for example be instantiated to strengthen variable permissions:

(HOL4-Thm 292)

$$\frac{\mathcal{W}_1 \subseteq \mathcal{W}_2 \qquad (\mathcal{W}_1 \cup \mathcal{R}_1 = \mathcal{W}_2 \cup \mathcal{R}_2) \qquad (\mathcal{W}'_1 \cup \mathcal{R}_1 = \mathcal{W}'_2 \cup \mathcal{R}_2) \qquad [\![\mathcal{W}_1; \mathcal{R}_1 \mid \mathcal{P}]\!]\ prog\ [\![\mathcal{W}'_1; \mathcal{R}_1 \mid \mathcal{P}']\!]}{[\![\mathcal{W}_2; \mathcal{R}_2 \mid \mathcal{P}]\!]\ prog\ [\![\mathcal{W}'_2; \mathcal{R}_2 \mid \mathcal{P}']\!]}$$

There are many similar inference rules. Here just a few interesting ones are listed:

$$\textit{(HOL4-Thm 297)}\quad \dfrac{[\![\mathcal{W};\mathcal{R} \mid \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}{[\![\mathcal{W};\mathcal{R} \mid \{\textit{stack-true}\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

$$\textit{(HOL4-Thm 297)}\quad \dfrac{}{[\![\mathcal{W};\mathcal{R} \mid \{\textit{false}\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

$$\textit{(HOL4-Thm 269)}\quad \dfrac{\begin{array}{c}\forall x.\ \textit{isStackImprecise}\big(P(x), \mathcal{W} \cup \mathcal{R}\big)\\ \forall x.\ [\![\mathcal{W};\mathcal{R} \mid \{P(x)\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W};\mathcal{R} \mid \{\exists x.P(x)\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

$$\textit{(HOL4-Thm 270)}\quad \dfrac{\begin{array}{c}\textit{isStackImprecise}\big(P_1, \mathcal{W} \cup \mathcal{R}\big)\\ \textit{isStackImprecise}\big(P_2, \mathcal{W} \cup \mathcal{R}\big)\\ [\![\mathcal{W};\mathcal{R} \mid \{P_1\} \cup \{P_2\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W};\mathcal{R} \mid \{P_1 * P_2\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

$$\textit{(HOL4-Thm 271)}\quad \dfrac{c \Longrightarrow [\![\mathcal{W};\mathcal{R} \mid \{P\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}{[\![\mathcal{W};\mathcal{R} \mid \{c \,\&\, P\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

$$\textit{(HOL4-Thm 293)}\quad \dfrac{c \Longrightarrow [\![\mathcal{W};\mathcal{R} \mid \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}{[\![\mathcal{W};\mathcal{R} \mid \{\textit{BoolPred}(c)\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

The last two inference rules move Boolean conditions out of the Hoare triple. They exploit that Boolean depend only on context information like specification variables but not on the current state. In contrast, general pure predicates are frequently used to connect specification variables with the values in the stack. The following inference rule uses the pure predicate $e = \textit{Const}(c)$ to introduce a specification variable c that holds the value of expression e:

Constant Introduction Rule *(HOL4-Thm 272)*

$$\dfrac{\begin{array}{c}\textit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big)\\ \forall c.\ [\![\mathcal{W};\mathcal{R} \mid \{e = \textit{Const}(c)\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W};\mathcal{R} \mid \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

If e uses only variables with read permissions, its value is not going to be changed. Therefore, $e = \textit{Const}(c)$ can be added to the postcondition as well.

(HOL4-Thm 273)

$$\dfrac{\begin{array}{c}\textit{isWellFormed}\big(e, \mathcal{R}\big)\\ \forall c.\ [\![\mathcal{W}_1;\mathcal{R} \mid \{e = \textit{Const}(c)\} \cup \mathcal{P}_1]\!]\ \textit{prog}\ [\![\mathcal{W}_2;\mathcal{R} \mid \{e = \textit{Const}(c)\} \cup \mathcal{P}_2]\!]\end{array}}{[\![\mathcal{W}_1;\mathcal{R} \mid \mathcal{P}_1]\!]\ \textit{prog}\ [\![\mathcal{W}_2;\mathcal{R} \mid \mathcal{P}_2]\!]}$$

In general, there is a specialised frame rule that is aware of the normal forms. It allows predicates that do not use variables with write permissions to be added. This inference rule illustrates the semantics of read- and write-permissions:

(HOL4-Thm 276)

$$\dfrac{\begin{array}{c}\textit{isStackImprecise}\big(P, \mathcal{R}\big)\\ [\![\mathcal{W}_1;\mathcal{R} \mid \mathcal{P}_1]\!]\ \textit{prog}\ [\![\mathcal{W}_2;\mathcal{R} \mid \mathcal{P}_2]\!]\end{array}}{[\![\mathcal{W}_1;\mathcal{R} \mid \{P\} \cup \mathcal{P}_1]\!]\ \textit{prog}\ [\![\mathcal{W}_2;\mathcal{R} \mid \{P\} \cup \mathcal{P}_2]\!]}$$

3.3.6 Program Constructs

After discussing variable as resource Hoare triples and basic inference rules for these Hoare triples, program constructs can now be considered.

3.3.6.1 Assume

The *assume* statement is introduced in Sec. 3.2.2.8. It is stated there that *assume* is only used with intuitionistic predicates (see Definition 3.2.37) in order to guarantee its locality. However, no intuitionistic predicates are presented there. After extending the model with a stack, it is now possible to discuss an interesting class of intuitionistic predicates: weak pure predicates.

Lemma 3.3.35 ((HOL4-Thm 240))**.** Weak pure predicates are intuitionistic.

$$\forall p, e_1, \ldots e_n.\ \left(\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}(e_i)\right) \Longrightarrow \mathit{isIntuitionistic}(\mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n]))$$

There is also a simple sufficient condition for showing that a weak pure predicate is decided in a set of states.

Lemma 3.3.36 ((HOL4-Thm 244))**.** A weak pure predicate $\mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])$ is decided in a set of states P, if all expressions $e_1, \ldots, e_n$ are well-formed with respect to a set of variables $\mathcal{V}$ such that the variables in $\mathcal{V}$ are present in all states in P.

$$\forall p, e_1, \ldots e_n, P, \mathcal{V}.\ \left(\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}\big(e_i, \mathcal{V}\big)\ \wedge\ \forall (s,h) \in P.\ \mathcal{V} \subseteq \mathit{dom}(s)\right) \Longrightarrow \mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n]) \text{ is decided in } P$$

In Sec. 3.2.37, the following inference rule for *assume* is presented:

(HOL4-Thm 88)

$$\frac{B \text{ is decided in } P \qquad \rhd_{\mathit{env}} \{P \wedge B\}\ \mathit{prog}\ \{Q\}}{\rhd_{\mathit{env}} \{P\}\ \mathit{assume}[B]\ ;\ \mathit{prog}\ \{Q\}}$$

Using weak pure predicates and Lemma 3.3.36 it can be instantiated to

$$\frac{\begin{array}{c}\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}\big(e_i, \mathcal{W} \cup \mathcal{R}\big) \\ [\,\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P})\ \wedge\ \mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])]\ \mathit{prog}\ [Q]\end{array}}{[\,\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{P})]\ \mathit{assume}[\mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])]\ ;\ \mathit{prog}\ [Q]}$$

This inference rule destroys the normal form in the precondition. Replacing the weak pure predicate with a strong one allows the normal form to be preserved:

(HOL4-Thm 284)

$$\frac{\begin{array}{c}\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}\big(e_i, \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}; \mathcal{R} \mid \{\mathit{PurePred}_{\mathit{strong}}(p, [e_1, \ldots, e_n])\} \cup \mathcal{P})]\!]\ \mathit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{assume}[\mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

Pure predicates have nice properties with respect to intuitionistic negation as well. The intuitionistic negation of a decided weak pure predicate $\mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])$ can be achieved by negating p (HOL4-Thm 239). This leads to the following inference rule:

(HOL4-Thm 282)

$$\frac{\begin{array}{c}\forall 1 \leqslant i \leqslant n.\ \mathit{isWellFormed}\big(e_i, \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}; \mathcal{R} \mid \{\mathit{PurePred}_{\mathit{strong}}(\neg p, [e_1, \ldots, e_n])\} \cup \mathcal{P})]\!]\ \mathit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{assume}[\neg_i \mathit{PurePred}_{\mathit{weak}}(p, [e_1, \ldots, e_n])]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

These inference rules allow the simple handling of assumptions using pure predicates. Notice, that the list of expressions of a weak pure predicate might be empty. Therefore, *false* and *true* are weak pure predicates as well (HOL4-Thm 385). Boolean combinations of conditions can be handled using the abstractions presented in Sec. 3.2.5.7. They lead to the following inference rules:

(HOL4-Thm 278)

$$\frac{[\![P]\!]\ \mathit{assume}[B_1]\ ;\ \mathit{assume}[B_2]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{assume}[B_1 \wedge B_2]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

(HOL4-Thm 283)

$$\frac{[\![P]\!]\ \mathit{assume}[B_1]\ ;\ \mathit{prog}\ [\![Q]\!] \qquad [\![P]\!]\ \mathit{assume}[B_2]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{assume}[B_1 \vee B_2]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

(HOL4-Thm 279)

$$\frac{[\![P]\!]\ \mathit{assume}[(\neg_i B_1) \vee (\neg_i B_2)]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{assume}[\neg_i (B_1 \wedge B_2)]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

(HOL4-Thm 280)

$$\frac{[\![P]\!]\ \mathit{assume}[B]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{assume}[\neg_i (\neg_i B)]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

(HOL4-Thm 281)

$$\frac{[\![P]\!]\ \mathit{assume}[(\neg_i B_1) \wedge (\neg_i B_2)]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{assume}[\neg_i (B_1 \vee B_2)]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

3.3.6.2 Control Structures

Using these inference rules for *assume*, handling control structures is straightforward. The inference rules presented in Sec. 3.2 can be lifted:

(HOL4-Thm 287)

$$\frac{[\![P]\!]\ \mathit{assume}[B]; \mathit{prog}_t\ [\![Q]\!] \qquad [\![P]\!]\ \mathit{assume}[\neg_i B]; \mathit{prog}_f\ [\![Q]\!]}{[\![P]\!]\ \text{if } B \text{ then } \mathit{prog}_t \text{ else } \mathit{prog}_f\ [\![Q]\!]}$$

Loop-Invariant Rule *(HOL4-Thm 289)*

$$\frac{\begin{array}{c}\big(I_f = \lambda x.\ (I_{f_b}(x), I_{f_p}(x))\big) \\ P_b \Longrightarrow \forall x.\ I_{f_b}(x) \\ \forall x.\ [\![I_f(x)]\!]\ \mathit{assume}[B]\ ;\ \mathit{prog}_1\ [\![I_f(x)]\!] \\ [\![(P_b, P_p)]\!]\ \mathit{qcvrbla}[I_f, I_f]\ ;\ \mathit{assume}[\neg_i B]\ ;\ \mathit{prog}_2\ [\![Q]\!]\end{array}}{[\![(P_b, P_p)]\!]\ \text{while } B \text{ do } \mathit{prog}_1\ ;\ \mathit{prog}_2\ [\![Q]\!]}$$

Loop-Specification Rule *(HOL4-Thm 277)*

$$\frac{\begin{array}{c}\big(P_f = \lambda x.\ (P_{f_b}(x), P_{f_p}(x))\big) \qquad \big(Q_f = \lambda x.\ (Q_{f_b}(x), Q_{f_p}(x))\big) \\ P_b \Longrightarrow \forall x.\ P_{f_b}(x)\ \wedge Q_{f_b}(x) \\ \forall x.\ [\![P_f(x)]\!]\ \mathit{assume}[\neg_i B]\ ;\ \mathit{prog}_2\ [\![Q_f(x)]\!] \\ \forall x.\ [\![P_f(x)]\!]\ \mathit{assume}[B]\ ;\ \mathit{prog}_1\ ;\ \mathit{qcvrbla}[P_f, Q_f]\ [\![Q_f(x)]\!] \\ [\![(P_b, P_p)]\!]\ \mathit{qcvrbla}[P_f, Q_f]\ ;\ \mathit{prog}_3\ [\![Q]\!]\end{array}}{[\![(P_b, P_p)]\!]\ \text{while } B \text{ do } \mathit{prog}_1\ ;\ \mathit{prog}_2\ ;\ \mathit{prog}_3\ [\![Q]\!]}$$

3.3.6.3 Semaphore Operations

Semaphore operations require a little bit more attention. As motivated in Sec. 3.2.2.6 only precise predicates are used as lock invariants. Therefore, a new normal form for lock invariants is used:

Definition 3.3.37 ((HOL4-Thm 358))**.** For a set of variables $\mathcal{W}$ and a predicate P, the predicate $\mathit{VRLockInv}(\mathcal{W}, P)$ is defined by:

$$\mathit{VRLockInv}(\mathcal{W}, P) \;:=\; \lambda(s,h).\;(\mathit{dom}(s) = \mathcal{W}) \;\wedge\; (\forall x \in \mathcal{W}.\; \mathit{perm}(x,s) = \top) \;\wedge\; (s,h) \in P$$

These lock invariants fix the set of variables. Moreover, they require write permission for all the variables mentioned by the invariant. Using lock invariants of the given form, the abstractions presented in Sec. 3.2.6 become (HOL4-Thms 153, 141, 246, 249):

$$\begin{array}{l}
\forall \mathit{lenv}, \mathit{penv}, l, \mathcal{W}, \mathit{prog}, P.\\
\quad \big(\mathit{lenv}(l) = \mathit{VRLockInv}(\mathcal{W}, P) \;\wedge\; \mathit{VRCond}(\mathcal{W}, \emptyset, \{P\})\big) \Longrightarrow\\
\quad \mathit{with}\ l\ \mathit{do}\ \mathit{prog}\ \sqsubseteq_{(\mathit{penv},\mathit{lenv})}\\
\qquad \mathit{cvrbla}[(\emptyset;\emptyset \mid \emptyset)\ ,\ (\mathcal{W};\emptyset \mid \{P\})]\ ;\ \mathit{prog}\ ;\\
\qquad \mathit{cvrbla}[(\mathcal{W};\emptyset \mid \{P\})\ ,\ (\emptyset;\emptyset \mid \emptyset)]
\end{array}$$

$$\begin{array}{l}
\forall \mathit{lenv}, \mathit{penv}, l, B, \mathcal{W}, \mathit{prog}, P.\\
\quad \big(\mathit{lenv}(l) = \mathit{VRLockInv}(\mathcal{W}, P) \;\wedge\; \mathit{VRCond}(\mathcal{W}, \emptyset, \{P\})\big) \Longrightarrow\\
\quad \mathit{with}\ l\ \mathit{when}\ B\ \mathit{do}\ \mathit{prog}\ \sqsubseteq_{(\mathit{penv},\mathit{lenv})}\\
\qquad \mathit{cvrbla}[(\emptyset;\emptyset \mid \emptyset)\ ,\ (\mathcal{W};\emptyset \mid \{P\})]\ ;\ \mathit{assume}[B]\ ;\ \mathit{prog}\ ;\\
\qquad \mathit{cvrbla}[(\mathcal{W};\emptyset \mid \{P\})\ ,\ (\emptyset;\emptyset \mid \emptyset)]
\end{array}$$

$$\begin{array}{l}
\forall \mathit{lenv}, \mathit{penv}, l, \mathcal{W}, \mathit{prog}, P.\\
\quad \big(\mathit{lenv}(l) = \mathit{VRLockInv}(\mathcal{W}, P) \;\wedge\; \mathit{VRCond}(\mathcal{W}, \emptyset, \{P\})\big) \Longrightarrow\\
\quad l.\mathit{prog}\ \sqsubseteq_{(\mathit{penv},\mathit{lenv})}\\
\qquad \mathit{cvrbla}[(\mathcal{W};\emptyset \mid \{P\})\ ,\ (\emptyset;\emptyset \mid \emptyset)]\ ;\ \mathit{prog}\ ;\\
\qquad \mathit{cvrbla}[(\emptyset;\emptyset \mid \emptyset)\ ,\ (\mathcal{W};\emptyset \mid \{P\})]
\end{array}$$

Using these abstractions, semaphore operations are removed from the program during a preprocessing step. It remains to symbolically execute *cvrbla*.

3.3.6.4 Procedure Calls

In Sec. 3.2.4.5 *choose-constants* is introduced for the purpose of defining procedure calls with call-by-value parameters. This setting is instantiated here. Procedure calls are introduced that get a list of variables as call-by-value arguments and expressions as call-by-value arguments.

A slight complication is that *choose-constants* evaluates its arguments on the whole state, while expressions are just evaluated on the stack. Therefore, a wrapper is introduced first:

Definition 3.3.38 (*eval-expressions* (HOL4-Thm 363))**.**

$$\begin{array}{lll}
\mathit{eval\text{-}expressions} & \mathit{expr\text{-}list} & \mathit{prog}_f \;:=\\
\mathit{choose\text{-}constants} & (\mathit{map}\ (\lambda e, (s,h).\ e(s))\ \mathit{expr\text{-}list}) & \mathit{prog}_f
\end{array}$$

Definition 3.3.39 (Procedure Calls with call-by-value Arguments (HOL4-Thms 366, 167))**.**

$$\begin{array}{l}\text{ext-proccall}(\mathit{name}, \mathit{refArgs}, \mathit{valArgs}) := \\ \mathit{eval\text{-}expressions}\ \mathit{valArgs}\ (\lambda \mathit{values}.\ \text{proccall}(\mathit{name}, (\mathit{refArgs}, \mathit{values})))\end{array}$$

As described in Sec. 3.2.7 procedures are abstracted in a preprocessing step in order to handle recursive procedure calls. *eval-expressions* does not interfere with this abstraction, because it is compatible with program abstraction:

(HOL4-Thm 247)
$$\frac{\forall \mathit{values}.\ \mathit{prog}_f(\mathit{values})\ \sqsubseteq_{env}\ \mathit{prog}'_f(\mathit{values})}{\mathit{eval\text{-}expressions}\ \mathit{expr\text{-}list}\ \mathit{prog}_f\ \sqsubseteq_{env}\ \mathit{eval\text{-}expressions}\ \mathit{expr\text{-}list}\ \mathit{prog}'_f}$$

eval-expressions is also used for parallel procedure calls:

Definition 3.3.40 (Parallel Procedure Calls (HOL4-Thm 365))**.**

$$\begin{array}{l}\text{ext-parallel-proccall}(\mathit{name}_1, \mathit{refArgs}_1, \mathit{valArgs}_1, \mathit{name}_2, \mathit{refArgs}_2, \mathit{valArgs}_2) := \\ \mathit{eval\text{-}expressions}\ \mathit{valArgs}_1\ (\lambda \mathit{values}_1. \\ \quad \mathit{eval\text{-}expressions}\ \mathit{valArgs}_2\ (\lambda \mathit{values}_2. \\ \quad\quad \text{proccall}(\mathit{name}_1, (\mathit{refArgs}_1, \mathit{values}_1))\ ||\ \text{proccall}(\mathit{name}_2, (\mathit{refArgs}_2, \mathit{values}_2))))\end{array}$$

Notice, that all call-by-value arguments are evaluated before either of the procedures is executed. Parallel procedure calls can be abstracted exploiting the compatibility of *eval-expressions* with program abstraction and the parallel composition rule (HOL4-Thm 248).

It remains to symbolically evaluate *eval-expressions*. If the list of expressions is empty, *eval-expressions* can be dropped. Expressions, whose value is known, can be removed. This leads to the following inference rules:

(HOL4-Thm 274)
$$\frac{[\![P]\!]\ \mathit{eval\text{-}expressions}([])(\mathit{prog}_f)\ [\![\mathit{prog}_f]\!]Q}{[\![P]\!]\ \mathit{prog}_f([])\ [\![Q]\!]}$$

(HOL4-Thm 275)
$$\frac{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{eval\text{-}expressions}(\mathit{Const}(c) :: \mathit{el})(\mathit{prog}_f)\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{eval\text{-}expressions}(\mathit{el})(\lambda \mathit{vl}.\ \mathit{prog}_f(c :: \mathit{vl}))\ [\![Q]\!]}$$

(HOL4-Thm 275)
$$\frac{[\![\mathcal{W}; \mathcal{R} \mid \{e = \mathit{Const}(c)\} \cup \mathcal{P}]\!]\ \mathit{eval\text{-}expressions}(e :: \mathit{el})(\mathit{prog}_f)\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \{e = \mathit{Const}(c)\} \cup \mathcal{P}]\!]\ \mathit{eval\text{-}expressions}(\mathit{el})(\lambda \mathit{vl}.\ \mathit{prog}_f(c :: \mathit{vl}))\ [\![Q]\!]}$$

Expressions like $e = \mathit{Const}(c)$ can be introduced into the precondition using the constant introduction rule (see page 102). Combined with the inference rules above, this allows the symbolic evaluation of *eval-expressions*.

3.3.6.5 Assignments

Until now, only already presented program constructs have been instantiated. Now the first new construct is presented: assigning the value of an expression to a stack variable. Before it can be defined, variable updates on states need to be discussed.

Definition 3.3.41 (Variable Updates (HOL4-Thms 402, 309))**.** $\textit{stack-var-update}[v,c](s)$ updates the value of variable v in stack s with c. If v is not present in the stack, it is inserted. Similarly, $\textit{state-var-update}[v,c](s,h)$ updates the value of v in the stack s belonging to the extended state (s,h). These functions are defined by:

$$\begin{aligned}\textit{stack-var-update}[v,c](s)(x) &:= \begin{cases}(c,\top) & \text{if } x = v\\ s(x) & \text{otherwise}\end{cases}\\ \textit{state-var-update}[v,c](s,h) &:= \big(\textit{stack-var-update}[v,c](s), h\big)\end{aligned}$$

Definition 3.3.42 (Assigment (HOL4-Thm 258))**.** For a variable v and a expression e the assignment action $\textit{assign}[v,e]$ is defined by

$$(\textit{assign}[v,e])(s,h) := \begin{cases}\textit{state-var-update}[v,c](s,h) & \text{if } \text{Defined}(e(s)) \;\wedge\; e(s) = c \;\wedge\\ & \qquad v \in \textit{dom}(s) \;\wedge\; \textit{perm}(v,s) = \top\\ \textit{undefined} & \text{otherwise}\end{cases}$$

This means that $\textit{assign}[v,e](s,h)$ will fail, if s does not hold write permission for the variable v or if the expression e cannot be evaluated. $\textit{assign}[v,e]$ is a local action (HOL4-Thm 241) for well-formed e.

Inference rules for $\textit{assign}[v,e]$ need to refer to the value of e before the update of v. Usually this uses some kind of syntactic substitution. Here, however, the syntax of expressions and predicates is not fixed. They are just functions with certain properties. Therefore, semantic substitution functions are defined:

Definition 3.3.43 (Semantic Substitution (HOL4-Thms 382, 308))**.** For a variable v, a value c, an expression e and a predicate P, the semantic substitution operations $\textit{exp-var-update}[v,c](e)$ and $\textit{pred-var-update}[v,c](P)$ that evaluate e and P on the state that results from updating v with c are defined by:

$$\begin{aligned}\textit{exp-var-update}[v,c](e) &:= \lambda s.\ e(\textit{stack-var-update}[v,c](s))\\ \textit{pred-var-update}[v,c](P) &:= \lambda(s,h).\ \textit{state-var-update}[v,c](s,h) \in P\end{aligned}$$

exp-var-update can easily be evaluated on common expressions:

$$\textit{exp-var-update}[v,c](\textit{Const}(c_2)) = \textit{Const}(c_2) \qquad \text{(HOL4-Thm 304)}$$

$$\textit{exp-var-update}[v,c](\textit{Var}(v_2)) = \begin{cases}\textit{Const}(c) & \text{if } v = v_2\\ \textit{Var}(v_2) & \text{otherwise}\end{cases} \qquad \text{(HOL4-Thm 305)}$$

$$\begin{aligned}&\textit{exp-var-update}[v,c](\textit{FunExp}(f,[e_1,\ldots,e_n])) =\\ &\textit{FunExp}(f, \text{map } \textit{exp-var-update}[v,c]\ [e_1,\ldots,e_n])\end{aligned} \qquad \text{(HOL4-Thm 306)}$$

$$\begin{aligned}&\textit{isWellFormedStrong}(e,\mathcal{V}) \Longrightarrow\\ &\textit{isWellFormedStrong}(\textit{exp-var-update}[v,c](e),\ \mathcal{V}\backslash\{v\})\end{aligned} \qquad \text{(HOL4-Thm 343)}$$

$$\begin{aligned}&\textit{isWellFormed}(e,\mathcal{V}) \Longrightarrow\\ &\textit{isWellFormed}(\textit{exp-var-update}[v,c](e),\ \mathcal{V})\end{aligned} \qquad \text{(HOL4-Thm 342)}$$

Similarly, *pred-var-update* can easily be evaluated for common predicates:

$$\textit{pred-var-update}[v,c](\textit{ExpPred}(p,[e_1,\ldots,e_n])) = \textit{ExpPred}(P, \text{map } \textit{exp-var-update}[v,c]\ [e_1,\ldots,e_n]) \qquad \text{(HOL4-Thm 380)}$$

$$\textit{isStackImprecise}(P,\{v\}\cup\mathcal{V}) \Longrightarrow \textit{isStackImprecise}(P,\mathcal{V}) \qquad \text{(HOL4-Thm 357)}$$

$$\textit{isStackImprecise}(P_1) \wedge \textit{isStackImprecise}(P_2) \Longrightarrow \textit{pred-var-update}[v,c](P_1 * P_2) = \textit{pred-var-update}[v,c](P_1) * \textit{pred-var-update}[v,c](P_2) \qquad \text{(HOL4-Thm 378)}$$

$$\textit{pred-var-update}[v,c](P_1 \wedge P_2) = \textit{pred-var-update}[v,c](P_1) \wedge \textit{pred-var-update}[v,c](P_2) \qquad \text{(HOL4-Thm 379)}$$

$$\textit{pred-var-update}[v,c](\exists x.\ P(x)) = \exists x.\ \textit{pred-var-update}[v,c](P(x)) \qquad \text{(HOL4-Thm 379)}$$

Since pure and Boolean predicates are defined in terms of expression predicates, semantic substitutions can easily be evaluated for these as well (HOL4-Thms 381, 379).

Using these semantic substitutions, it can be shown that assignments can be abstracted as follows (HOL4-Thm 361):

$$\forall e,\mathcal{V},v,c.\ \ \textit{isWellFormedStrong}(e,\mathcal{V}) \Longrightarrow \textit{assign}[v,e] \sqsubseteq \textit{cvrbla}[\{v\}\ ;\ \mathcal{V}\backslash\{v\}\ \mid\ \{\textit{Var}(v) = \textit{Const}(c)\},\ \{v\}\ ;\ \mathcal{V}\backslash\{v\}\ \mid\ \{\textit{Var}(v) = \textit{exp-var-update}[v,c](e)\}]$$

Combining this program abstraction rule with the inference rule for sequential composition and the frame rule, leads to the following inference rule for assignments:

Variable Assignment Rule *(HOL4-Thm 294)*

$$\frac{\textit{isWellFormed}\big(e,\mathcal{W}\cup\mathcal{R}\big) \qquad v\in\mathcal{W} \\ [\![\mathcal{W};\mathcal{R}\mid\{\textit{Var}(v) = \textit{exp-var-update}[v,c](e)\}\cup \textit{image}\ (\textit{pred-var-update}[v,c])\ \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}{[\![\mathcal{W};\mathcal{R}\mid\{\textit{Var}(v) = \textit{Const}(c)\}\cup\mathcal{P}]\!]\ \textit{assign}[v,e]\ ;\ \textit{prog}\ [\![Q]\!]}$$

Notice, that this inference rule heavily relies on the well-formedness of the precondition. Because all elements of $\mathcal{P}$ are stack imprecise, the semantic substitution can be mapped over the multiset $\mathcal{P}$. This guarantees that no predicate in the resulting multiset depends on the variable v. Therefore, this multiset of predicates can be added using the frame rule. Notice moreover, that the predicate $\textit{Var}(v) = \textit{Const}(c)$ can be introduced into the precondition using the constant introduction rule (see page 102).

In general it is useful to propagate equality information. The following inference rule allows the precondition of Hoare triples to be normalised:

Equality Propagation Rule *(HOL4-Thms 388, 266)*

$$\frac{[\![\mathcal{W};\mathcal{R}\mid\{\textit{Var}(v) = \textit{Const}(c)\}\cup \textit{image}\ (\textit{pred-var-update}[v,c])\ \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}{[\![\mathcal{W};\mathcal{R}\mid\{\textit{Var}(v) = \textit{Const}(c)\}\cup\mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!]}$$

Both inference rules introduce semantic substitutions into the precondition. Their evaluation usually requires equality checks between stack variables. The well-formedness of the precondition can be used for these equality checks. Exploiting that all variables in $\mathcal{W} \cup \mathcal{R}$ are distinct, is essential for the evaluation of the semantic substitutions.

3.3.6.6 Local Variables

Local variables are implemented using an action for introducing a new stack variable and one for disposing stack variables.

Definition 3.3.44 (Initialising Stack Variables (HOL4-Thm 360))**.** Let e be an expression and v a stack variable. Then the action $\mathit{var\text{-}init}(v, e)$ tries to add the variable v with a write-permission to the stack and initialise it with e. If e cannot be evaluated, the action fails. If v is already in the stack, it diverges.

$$\mathit{var\text{-}init}(v,e)(s,h) := \begin{cases} \top & \text{if } \neg\text{Defined}(e(s)) \\ \varnothing & \text{if Defined}(e(s)) \ \wedge \ v \in \mathit{dom}(s) \\ \mathit{stack\text{-}var\text{-}update}[v, e(s)](s,h) & \mathit{otherwise} \end{cases}$$

Definition 3.3.45 (Disposing Stack Variables (HOL4-Thm 299))**.** If the stack contains write permission for a variable v, the action $\mathit{var\text{-}dispose}(v)$ removes v from the stack. Otherwise, it fails.

$$\mathit{var\text{-}dispose}(v)(s,h) := \begin{cases} (s\backslash\{v\}, h) & \text{if } v \in \mathit{dom}(s) \ \wedge \ \mathit{perm}(s,v) = \top \\ \top & \mathit{otherwise} \end{cases}$$

Lemma 3.3.46 ((HOL4-Thms 242, 243))**.** $\mathit{varDisp}(v)$ and $\mathit{varInit}(v, e)$ are local actions for well-formed expressions e.

Remark 3.3.47. One might expect that initialising a stack variable masks an existing definition. Disposing this variable could then restore the original definition. These initialise and dispose actions, however, would not be local.

Combining initialising and disposing of a stack variable with nondeterministic choice leads to a local variable declaration action:

Definition 3.3.48 (Local Variables (HOL4-Thms 362, 364))**.** A local variable declaration of a variable v that is initialised by some value c consists of nondeterministically choosing a variable, initialising it with c, executing the body of the local variable declaration and then disposing the variable.

$$\mathit{local\text{-}var\text{-}init}_c\ v.\ \mathit{prog}_f(v) \ := \ \bigcup_v \left\{ \mathit{var\text{-}init}(v, \mathit{Const}(c)) \ ; \ \mathit{prog}_f(v) \ ; \ \mathit{var\text{-}dispose}(v) \right\}$$

Local variable declarations without initialisation are defined as the nondeterministic choice between all possible initial values:

$$\mathit{local\text{-}var}\ v.\ \mathit{prog}_f(v) \ := \ \bigcup_c \left\{ \mathit{local\text{-}var\text{-}init}_c\ v.\ \mathit{prog}_f(v) \} \right\}$$

So, local variable declarations nondeterministically choose a variable. If this variable is already present in the stack, its initialisation diverges. Since Abstract Separation Logic is only interested in partial correctness, this has the effect of not considering variables that are already in the stack. Therefore, local variable declarations have the intended semantics. This semantics is captured by the following inference rules:

$$\frac{\textit{(HOL4-Thms 285)} \atop \forall v.\ [\![\{v\} \cup \mathcal{W}; \mathcal{R} \mid \{\mathit{Var}(v) = \mathit{Const}(c)\} \cup \mathcal{P}_1]\!]\ \mathit{prog}_f(v)\ [\![\{v\} \cup \mathcal{W}; \mathcal{R} \mid \mathcal{P}_2]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}_1]\!]\ \mathit{local\text{-}var\text{-}init}_c\ v.\ \mathit{prog}_f(v)\ [\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}_2]\!]}$$

$$\frac{\textit{(HOL4-Thms 288)} \atop \forall v.\ [\![\{v\} \cup \mathcal{W}; \mathcal{R} \mid \mathcal{P}_1]\!]\ \mathit{prog}_f(v)\ [\![\{v\} \cup \mathcal{W}; \mathcal{R} \mid \mathcal{P}_2]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}_1]\!]\ \mathit{local\text{-}var}\ v.\ \mathit{prog}_f(v)\ [\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}_2]\!]}$$

3.3.6.7 Quantified Best Local Actions

Inference rules for constructs like procedure calls, critical regions or while loops reduce the symbolic evaluation of these constructs to the symbolic evaluation of quantified best local actions. A quantified best local action $\mathit{qcvrbla}[P_f, Q_f]$ satisfies the family of specifications $\forall x.\ [\![P_f(x)]\!]\ \mathit{qcvrbla}[P_f, Q_f]\ [\![Q_f(x)]\!]$. A first step to evaluate such a quantified best local action is choosing a member x of this family of specifications:

$$\frac{\textit{(HOL4-Thms 296)} \atop \exists x.\ [\![P]\!]\ \mathit{cvrbla}[P_f(x), Q_f(x)]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![P]\!]\ \mathit{qcvrbla}[P_f, Q_f]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

This inference rule represents a real implication and has to be applied carefully. One has to be sure that the precondition P contains enough information to choose x. As an example consider a procedure that increments a variable v. In this case the parameter x might be used to hold the value of the variable before the procedure is called. If the inference rule is applied too early, one might need to show that there exists x such that forall possible values c of v the value of v is x. Thus, one would need to show $\exists x.\ \forall c.\ x = c$ which does not hold in general. If however, the value c is introduced into the precondition before applying the inference rule, one needs to show $\forall c.\ \exists x.\ (x = c)$, which is easy.

3.3.7 Frame Inference

In the previous section inference rules for the forward analysis of common program constructs are presented. One important construct is, however, still missing: best local actions. In order to symbolically evaluate a best local action a frame needs to be infered.

3.3.7.1 Informal Discussion

Let's first consider the frame inference problem informally. Assume there is some Hoare triple $\{P\}\ \mathit{bla}[P_2, Q_2]\ ;\ \mathit{prog}\ \{Q\}$. In order to symbolically evaluate $\mathit{bla}[P_2, Q_2]$ one needs

to find a frame F such that P implies $P_2 * F$. Given such a frame F the following reasoning is possible: By definition $\{P_2\}$ $bla[P_2, Q_2]$ $\{Q_2\}$ holds. Using the frame rule, this can be extended to $\{P_2 * F\}$ $bla[P_2, Q_2]$ $\{Q_2 * F\}$. Since P implies $P_2 * F$, the Hoare triple $\{P\}$ $bla[P_2, Q_2]$ $\{Q_2 * F\}$ holds as well. Finally, the sequential composition rule can be used to show that $\{P\}$ $bla[P_2, Q_2]$; *prog* $\{Q\}$ is implied by $\{Q_2 * F\}$ *prog* $\{Q\}$.

So, the frame inference problem can be described as follows: given P and Q, a frame F is searched such that $P \vdash Q * F$ holds. One of the most important rules separation logic tools like Smallfoot [2] use for such entailments is the separating conjunction introduction rule: $\forall P_1, P_2, P_3.\ P_2 \vdash P_3 \implies P_1 * P_2 \vdash P_1 * P_3$. One has to be careful when to apply this rule. Consider the following example in Holfoot syntax: `x |-> [] * x |-> []` $\vdash$ `x |-> []` does hold, because `x |-> [] * x |-> []` is unsatisfiable. However, applying the separating conjunction introduction rule results in `x |-> []` $\vdash$ `emp`, which does not hold. In order to avoid such problems, I extend the frame inference problem with a context C. The problem then becomes $C * P \vdash C * Q * F$. Instead of removing common parts of P and Q they can be moved to C. Thus, this context C allows storing information and removes the need to pay close attention when to apply certain rules for entailments.

Once a frame F is found, it is used in some way. In the case of symbolically executing a best local action, F is for example used in the precondition of a Hoare triple. It is useful to include this further usage into the frame inference as a predicate *framePred*. The problem then becomes: $\exists F.\ C * P \vdash C * Q * F \ \wedge \mathit{framePred}(F)$. This predicate *framePred* should satisfy some properties that allow proving interesting inference rules. It should be satisfiable and it should be compatible with existential quantification in some sense that will be explained below.

3.3.7.2 Basic Definitions

This informal discussion of a frame inference leads to the following definition.

Definition 3.3.49 (Frame Inference Predicate (HOL4-Thms 310, 322)). Given multisets of variables $\mathcal{W}$, $\mathcal{R}$, $\mathcal{W}'$, multisets of predicate $\mathcal{C}$, $\mathcal{P}$, $\mathcal{Q}$ and a set of multisets of predicates *fP*, the frame inference predicate is defined by:

$$\begin{aligned} &[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!] := \\ &\quad \mathit{isFramePred}(\mathcal{W}\backslash\mathcal{W}', \mathcal{R}\backslash\mathcal{W}', \mathit{fP}) \Longrightarrow \\ &\quad \exists \mathcal{F}.\ \big(\mathit{fP}(\mathcal{F})\ \wedge\ \mathit{VRCond}(\mathcal{W}\backslash\mathcal{W}', \mathcal{R}\backslash\mathcal{W}', \mathcal{F})\big) \wedge \\ &\quad\quad \big(\mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P} \cup \mathcal{Q}) \Longrightarrow \\ &\quad\quad\quad \mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}) \subseteq \mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{Q} \cup \mathcal{F})\big) \end{aligned}$$

The predicate *isFramePred* checks, whether *fP* is satisfiable and compatible with existential quantification.

$$\begin{aligned} &\mathit{isFramePred}(\mathcal{W}, \mathcal{R}, \mathit{fP}) := \\ &\quad \big(\exists \mathcal{F}.\ \mathit{fP}(\mathcal{F})\ \wedge\ \mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{F})\big) \wedge \\ &\quad \big(\forall F.\ \big(\forall \mathcal{F} \in F.\ \mathit{fP}(\mathcal{F})\ \wedge\ \mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{F})\big) \Longrightarrow \\ &\quad\quad \mathit{fP}\{\lambda(s,h).\ \exists \mathcal{F} \in F.\ (s,h) \in \circledast\mathcal{F}\}\big) \end{aligned}$$

As intended, this frame inference predicate can be used for symbolically executing best local actions:

(HOL4-Thm 295)

$$\frac{\mathcal{W}' \subseteq \mathcal{W} \qquad \mathcal{R}' \subseteq \mathcal{W} \cup \mathcal{R} \qquad [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \varnothing \mid \mathcal{P} \mid \mathcal{P}' \mid \lambda \mathcal{F}.\ [\![(\mathcal{W} \backslash \mathcal{W}') \cup \mathcal{W}''; \mathcal{R} \mid \mathcal{P}'' \cup \mathcal{F}]\!]\ \mathit{prog}\ [\![Q]\!]]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{cvrbla}[(\mathcal{W}'; \mathcal{R}' \mid \mathcal{P}')\ ,\ (\mathcal{W}''; \mathcal{R}'' \mid \mathcal{P}'')]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

If *fP* demands that the frame is empty, the frame inference predicate checks entailments. One has to be careful, though, how to express that the frame is empty. Demanding $\mathcal{F} = \varnothing$ causes problems, because *isFramePred* does not hold for this predicate. Instead of this syntactic definition, a semantic one is needed:

Definition 3.3.50 (Pure Predicate Check (HOL4-Thm 336))**.** The following function checks, whether a predicate P is a strong pure predicate.

$$\mathit{isStrongPurePred}(P) \quad := \quad \forall (s, h) \in P.\ h \in \mathit{emp}$$

Notice, that $\mathit{isStrongPurePred}(\mathit{PurePred}_{\mathit{strong}}(p, \mathit{el}))$ holds trivially (HOL4-Thm 337).

This definition leads to the following inference rule for Hoare triples with empty body:

(HOL4-Thm 268)

$$\frac{\mathcal{W}' \subseteq \mathcal{W} \qquad \mathcal{R}' \subseteq \mathcal{W} \cup \mathcal{R} \qquad [\![\mathcal{W}, \mathcal{R}, \varnothing;\ \varnothing \mid \mathcal{P} \mid \mathcal{P}' \mid \lambda \mathcal{F}.\ \forall P \in \mathcal{F}.\ \mathit{isStrongPurePred}(P)]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{skip}\ [\![\mathcal{W}'; \mathcal{R}' \mid \mathcal{P}']\!]}$$

The definition of *isFramePred*(*fP*) consists of two parts. The first part demands that a frame exists that satisfies *fP*. Therefore, $[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]$ holds, if $\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P})$ is unsatisfiable. This property allows inference rules like:

(HOL4-Thm 315)

$$\frac{c \Longrightarrow [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\mathit{BoolPred}(c)\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

(HOL4-Thm 317)

$$\frac{c \Longrightarrow [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\mathit{BoolPred}(c)\} \cup \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

The second part of *isFramePred*(*fP*) demands that *fP* is compatible with existential quantification. It is a technical property designed to allow proving the following inference rules:

(HOL4-Thm 313)

$$\frac{\forall x.\ \mathit{isStackImprecise}\big(P(x), \mathcal{W} \cup \mathcal{R}\big) \qquad \forall x.\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P(x)\} \cup \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\exists x.\ P(x)\} \cup \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

(HOL4-Thm 311)

$$\frac{\forall x.\ \mathit{isStackImprecise}\big(P(x), \mathcal{W} \cup \mathcal{R}\big) \qquad \forall x.\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P(x)\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\exists x.\ P(x)\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

3.3.7.3 Inference Rules

Other important inference rules for frame inferences include:

(HOL4-Thm 312)

$$\frac{\begin{array}{c}\forall x.\ \mathit{isStackImprecise}\Big(P(x), \mathcal{W} \cup \mathcal{R}\Big) \\ \exists x.\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \{P(x)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \{\exists x.\ P(x)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

***∗-Introduction** (HOL4-Thm 319)*

$$\frac{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P\} \cup \mathcal{P} \mid \{P\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

Strong pure predicates are idempotent. This allows simplified separating conjunction introduction inference rules for strong pure predicates.

(HOL4-Thm 321)

$$\frac{\begin{array}{c}\mathit{isStrongPurePred}(P) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P\} \cup \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

(HOL4-Thm 320)

$$\frac{\begin{array}{c}\mathit{isStrongPurePred}(P) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P\} \cup \mathcal{C} \mid \mathcal{P} \mid \{P\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

Equations are strong pure predicates. Therefore, these inference rule allow moving equations to the context. It is useful to combine moving an equation to the context with equality propagation. This allows using the context $\mathcal{C}$ to record which equalities have already been propagated.

(HOL4-Thm 318)

$$\frac{\begin{array}{c}\mathit{eqprop} = \lambda \mathcal{S}.\ \mathit{image}\ (\mathit{pred\text{-}var\text{-}update}[v, c])\ \mathcal{S} \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\mathit{Var}(v) = \mathit{Const}(c)\} \cup \mathit{eqprop}(\mathcal{C}) \mid \mathit{eqprop}(\mathcal{P}) \mid \mathit{eqprop}(\mathcal{Q}) \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\mathit{Var}(v) = \mathit{Const}(c)\} \cup \mathcal{P} \mid \{P\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

There are many other inference rules for frame inference predicates. Here, only a few exemplary structural rules are listed:

(HOL4-Thm 314)

$$\frac{\begin{array}{c}\mathit{isStackImprecise}\Big(P_1, \mathcal{W} \cup \mathcal{R}\Big) \\ \mathit{isStackImprecise}\Big(P_2, \mathcal{W} \cup \mathcal{R}\Big) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P_1\} \cup \{P_2\} \cup \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P_1 * P_2\}\mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

(HOL4-Thm 316)

$$\frac{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \{\mathit{BoolPred}(c_1 \wedge c_2)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \{\mathit{BoolPred}(c_1)\} \cup \{\mathit{BoolPred}(c_2)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

3.3.7.4 Solving Frame Inference Predicates

The inference rules for frame inference predicates can be used to simplify a frame inference predicate $[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid fP]\!]$ stepwise until $\mathcal{Q}$ becomes empty. Informally, a frame $\mathcal{F}$ has to be found then, such that $\mathcal{P}$ implies $\mathcal{F}$ and such that $\mathcal{F}$ does not use any variables from $\mathcal{W}'$. An obvious choice for $\mathcal{F}$ is $\mathcal{P}$:

$$\frac{\textit{(HOL4-Thm 323)}\quad \forall P \in \mathcal{P}.\ \textit{isStackImprecise}\big(P, (\mathcal{W} \cup \mathcal{R})\backslash\mathcal{W}'\big) \qquad fP(\mathcal{P})}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \varnothing \mid fP]\!]}$$

$\mathcal{Q}$ may even contain a Boolean predicate.

$$\frac{\textit{(HOL4-Thm 324)}\quad \forall P \in \mathcal{P}.\ \textit{isStackImprecise}\big(P, (\mathcal{W} \cup \mathcal{R})\backslash\mathcal{W}'\big) \qquad c \qquad fP(\mathcal{P})}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \textit{BoolPred}(c) \mid fP]\!]}$$

Usually, one is interested in strong frames, i. e. frames that contain as much information as possible. An inference rule is presented above to move strong pure predicates to the context. Before solving a frame inference predicate it is sensible to move as many strong pure predicates from the $\mathcal{C}$ into $\mathcal{P}$ as possible. One should be careful though to move only predicates that do not use any variable from $\mathcal{W}'$.

$$\frac{\frac{\textit{(HOL4-Thm 321)}\quad \textit{isStrongPurePred}(P) \qquad \textit{isStackImprecise}\big(P, (\mathcal{W} \cup \mathcal{R})\backslash\mathcal{W}'\big) \qquad [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P\} \cup \mathcal{P} \mid \varnothing \mid fP]\!]}{}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P\} \cup \mathcal{C} \mid \mathcal{P} \mid \varnothing \mid fP]\!]}$$

3.3.7.5 Frame Inference Algorithm

Above inference rules are presented that allow introducing, simplifying and finally solving frame inference predicates. Compared to the frame inference used by tools like Smallfoot, one does not need to be particularly careful about the order of applying inference rules, because the context can be used to store additional information. However, one needs to be careful about quantification.

Consider for example the frame inference predicate $[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\exists x.P(x)\} \cup \mathcal{P} \mid \{\exists y.Q(y)\} \cup \mathcal{Q} \mid fP]\!]$. Depending on the order the quantifiers are removed, one ends up with either $\exists y.\forall x.[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P(x)\} \cup \mathcal{P} \mid \{Q(y)\} \cup \mathcal{Q} \mid fP]\!]$ or $\forall x.\exists y.[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P(x)\} \cup \mathcal{P} \mid \{Q(y)\} \cup \mathcal{Q} \mid fP]\!]$. The latter is preferable, because it is the weaker statement. Therefore, existential quantification in $\mathcal{P}$ and $\mathcal{C}$ should be removed before removing it from $\mathcal{Q}$. Notice, that the existential quantification may be implicit. It might for example be introduced by rewriting some predicates or by applying inference rules. A simple example of such an inference rule is a rule similar to the constant introduction rule:

$$\frac{\frac{v \in \mathcal{W} \cup \mathcal{R} \qquad [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\exists c.\ \textit{Var}(v) = \textit{Const}(c)\} \cup \mathcal{P} \mid \mathcal{Q} \mid fP]\!]}{}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid fP]\!]}$$

A similar problem occurs when reducing quantified best local actions to best local actions. Whenever, a frame inference predicate is introduced, one should try to move quantifiers that lead to universal quantification out, before moving those that lead to existential quantification. Frame inference predicates are usually introduced if there is a Hoare triple, with empty body, with a quantified best local action as first statement or with a best local action action as first statement. In these situations, the following steps should be taken to introduce, simplify and solve frame inference predicates:

- Use the constant introduction rule to introduce constants for all variables into the precondition of the Hoare triple. In addition, remove existential quantification from the precondition.
- Reduce a quantified best local action to a best local action if applicable.
- Introduce a frame inference predicate $[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]$.
- Remove quantification from $\mathcal{Q}$.
- Simplify the frame inference predicate until $\mathcal{Q}$ becomes empty or contains only a single Boolean predicate.
- Move strong pure predicates from $\mathcal{C}$ to $\mathcal{P}$.
- Solve the frame inference predicate.

3.3.8 Implicit Information

Often, it is important to make implicitly contained information explicit. For this purpose the following definition is introduced that states that two normal forms are equivalent:

Definition 3.3.51 (Normal Form Equivalent (HOL4-Thm 373))**.**

$$\begin{array}{rl} \mathit{VREquiv}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}, \mathcal{P}') := & \Big(\mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}) \;\Leftrightarrow\; \mathit{VRCond}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}')\Big) \;\wedge \\ & \Big(\mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}) \;=\; \mathit{VRProp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}')\Big) \end{array}$$

Using this definition Hoare triples and frame inference predicates can be rewritten:

$$\textit{(HOL4-Thm 291)}\quad \frac{\begin{array}{c}\mathit{VREquiv}(\mathcal{W}, \mathcal{R}, \emptyset, \mathcal{P}, \mathcal{P}') \\ [\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{prog}\ [\![Q]\!]\end{array}}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}']\!]\ \mathit{prog}\ [\![Q]\!]} \qquad \textit{(HOL4-Thm 325)}\quad \frac{\begin{array}{c}\mathit{VREquiv}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}, \mathcal{P}') \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]\end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P}' \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

Usually, these inference rules are used to just add predicates that were previously only implicitly contained in $\mathcal{P}$. An important special case is adding pure strong predicates that state that some expressions evaluate to different values. This leads to the following definitions:

Definition 3.3.52. (HOL4-Thms 372, 328)

$$\begin{array}{rcl} \mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}) & := & \mathit{VREquiv}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \emptyset, \mathcal{P}) \\[1em] \mathit{VRImpUnequal}(\mathcal{C}, e_1, e_2) & := & \mathit{isWellFormed}(e_1) \;\wedge\; \mathit{isWellFormed}(e_2) \;\Longrightarrow \\ & & (\ast\mathcal{C} \subseteq \mathit{PurePred}_{weak}(\neq, e_1, e_2)) \end{array}$$

There are several inference rules for these definitions. However, most these definitions are mainly used with the Holfoot formalisation.

(HOL4-Thm 329)
$$\frac{\mathit{isWellFormed}\big(e_1, \mathcal{W} \cup \mathcal{R}\big) \quad \mathit{isWellFormed}\big(e_2, \mathcal{W} \cup \mathcal{R}\big) \quad \mathit{VRImpUnequal}(\mathcal{C}, e_1, e_2)}{\mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, e_1 \neq e_2)}$$

(HOL4-Thm 374)
$$\frac{\mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_1) \quad \mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_2)}{\mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_1 \cup \mathcal{P}_2)}$$

(HOL4-Thm 330)
$$\frac{c_1 \neq c_2}{\mathit{VRImpUnequal}(\mathcal{C}, \mathit{Const}(c_1), \mathit{Const}(c_2))}$$

(HOL4-Thm 331)
$$\frac{(e_1 \neq e_2) \in \mathcal{C}}{\mathit{VRImpUnequal}(\mathcal{C}, e_1, e_2)}$$

(HOL4-Thm 290)
$$\frac{\mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{P}, \mathcal{P}') \quad [\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{prog}\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \mathcal{P} \cup \mathcal{P}']\!]\ \mathit{prog}\ [\![Q]\!]}$$

(HOL4-Thm 326)
$$\frac{\mathit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C} \cup \mathcal{P}, \mathcal{P}') \quad [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q} \mid \mathit{fP}]\!]}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \mathcal{P} \cup \mathcal{P}' \mid \mathcal{Q} \mid \mathit{fP}]\!]}$$

3.4 Holfoot

In Section 3.3 a first instantiation of Abstract Separation Logic is presented. It introduces a stack as a finite map from some variable type to some value type and permissions. Moreover, there is a second component of the state that is still abstract. In this section, this instantiation is further instantiated in order to build a formalisation of Smallfoot [3]. Variables are represented as strings, values are instantiated as natural numbers. Most importantly though, the second component of the state is instantiated to become a heap.

All important concepts like normal forms or frame inference predicates are already introduced in Section 3.3. Here, first the states used by Holfoot are discussed in Sec. 3.4.1. Then predicates on these states are defined in Sec. 3.4.2. Many of these predicates describe data-structures in the heap. There are, for example, predicates describing singly-linked lists, trees or arrays. Sec. 3.4.3 introduces new program constructs. There are program statements for explicit memory allocation or deallocation as well as statements that lookup or store a value in the heap. Sec. 3.4.4 discusses how information that is implicitly contained in predicates can be made explicit. Finally, Sec. 3.4.5 presents inference rules for frame inference predicates. These inference rules are specific to the predicates defined for Holfoot.

3.4.1 States

As described in Sec. 2.1 Holfoot uses heaps that are finite maps from locations to named records of values. Locations are natural numbers excluding 0. Named record are maps from tags to values. Tags are identifiers used to index the entry in the record. They are represented as strings. Values are natural numbers.

Definition 3.4.1 (Heaps)**.** A heap is a finite map from locations to a map from tags to values.

$$\begin{aligned} \mathit{Values} &\stackrel{\text{def}}{=} \mathbb{N}_0 \\ \mathit{Locations} &\stackrel{\text{def}}{=} \mathit{Values}\setminus\{0\} = \mathbb{N} \\ \mathit{Tags} &\stackrel{\text{def}}{=} \mathit{Strings} \\ \mathit{Heaps} &\stackrel{\text{def}}{=} \mathit{Locations} \stackrel{\text{fin}}{\rightharpoonup} (\mathit{Tags} \to \mathit{Values}) \end{aligned}$$

Notice, that named records are not finite maps. They are defined for every tag! This means that if a location is in the domain of a heap h, then h contains values for all tags at this location.

Definition 3.4.2 (Combining Heaps (HOL4-Thm 196))**.** Two heaps h_1 and h_2 are separate if their domains are separate. The combination of two separate heaps h_1 and h_2 is defined as their disjoint union.

$$h_1 \bullet_H h_2 \quad = \quad \begin{cases} h_1 \uplus h_2 & \text{iff } \mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Remark 3.4.3. $\bullet_H$ is one of the initial examples for separation combinators. It is discussed in Example 3.2.8.

Lemma 3.4.4 ((HOL4-Thms 216, 220))**.** $\bullet_H$ is a separation combinator for heaps. Moreover, the empty heap $\emptyset$ is the neutral element for all states, i. e. $\mathit{emp}_\circ = \{\emptyset\}$ (HOL4-Thm 82). Therefore, $(\mathit{Heaps}, \bullet_H, \emptyset)$ is a separation algebra.

Definition 3.4.5 (Stacks)**.** Holfoot instantiates the stacks presented in Sec. 3.3. Variables are represented by strings, values become natural numbers.

$$\begin{aligned} \mathit{Values} &\stackrel{\text{def}}{=} \mathbb{N}_0 \\ \mathit{Vars} &\stackrel{\text{def}}{=} \mathit{Strings} \\ \mathit{Stacks} &\stackrel{\text{def}}{=} \mathit{Vars} \stackrel{\text{fin}}{\rightharpoonup} (\mathit{Values} \times \mathit{Perms}) \end{aligned}$$

Definition 3.4.6 (Holfoot Separation Combinator (HOL4-Thms 49, 50, 51))**.** Holfoot uses states that consist of a stack and a heap. The function $\odot_{\bullet_H} = (\bullet_{St} \times \bullet_H)$ is a separation combinator on these states. Moreover, $(\mathit{Stacks} \times \mathit{Heaps}, \odot_{\bullet_H}, (\emptyset, \emptyset))$ is a separation algebra (HOL4-Thm 50).

3.4.2 Predicates

Expressions and pure predicates as introduced in Section 3.3 are important concepts in Holfoot as well. With values being instantiated to natural numbers, function expressions can lift operations on natural numbers to expressions. Typical examples are addition and subtraction on natural numbers (monus) expressions:

$$\begin{aligned} e_1 + e_2 &:= \mathit{FunExp}(+, e_1, e_2) \\ e_1 - e_2 &:= \mathit{FunExp}(-, e_1, e_2) \end{aligned}$$

Besides pure predicates, Holfoot uses predicates that describe data-structures in the heap. There are predicates describing single heap-cells, non-cyclic singly-linked lists, trees and arrays. These predicates are informally described in Sec. 2.1.3. Their formal definition is presented here.

3.4.2.1 Points-To

Definition 3.4.7 (Single Heap Cell (HOL4-Thm 20))**.** Given an expression e and a finite map from tags to expressions T, the predicate $\mathit{pointsTo}(e, T)$ describes a single heap cell at location e that contains the values described by T.

$$\begin{array}{lcl}(s, h) \in \mathit{pointsTo}(e, T) & := & \text{Defined}(e(s)) \;\wedge\; e(s) \neq 0 \;\wedge\; \mathit{dom}(h) = \{e(s)\} \;\wedge \\ & & \forall t \in \mathit{dom}(T).\; \text{Defined}(T(t)(s)) \;\wedge\; h(e(s))(t) = T(t)(s)\end{array}$$

This predicate is stack imprecise and compatible with semantic substitution.

Lemma 3.4.8 ((HOL4-Thm 62))**.**

$$\frac{\mathit{isWellFormed}\big(e, \mathcal{V}\big) \qquad \forall t \in \mathit{dom}(T).\; \mathit{isWellFormed}\big(T(t), \mathcal{V}\big)}{\mathit{isStackImprecise}\big(\mathit{pointsTo}(e, T), \mathcal{V}\big)}$$

Lemma 3.4.9 ((HOL4-Thm 66))**.**

$$\begin{array}{l}\mathit{pred\text{-}var\text{-}update}[v, c]\Big(\mathit{pointsTo}(e, T)\Big) = \\ \mathit{pointsTo}\Big(\mathit{exp\text{-}var\text{-}update}[v, c](e), \lambda t.\; \mathit{exp\text{-}var\text{-}update}[v, c](T(t))\Big)\end{array}$$

3.4.2.2 Singly-Linked Lists

Definition 3.4.10 (Singly-Linked Lists (HOL4-Thm 11))**.** Given two expressions e_1, e_2, a tag tl, a natural number n and a list data of pairs consisting of tags and lists of natural numbers, the predicate $\mathit{data\text{-}lseg}_n(\mathit{tl}, e_1, \mathit{data}, e_2)$ describes a non-cyclic singly-linked list of length n starting at e_1, ending at e_2, containing the data described by data and using the tag tl for linking.

$$\begin{array}{lcl}\mathit{isWellFormed}(n, \mathit{tl}, \mathit{data}) & := & \forall (t, l) \in \mathit{data}.\; \mathit{length}(l) = n \;\wedge\; t \neq \mathit{tl} \;\wedge \\ & & \text{all tags in } \mathit{data} \text{ are pairwise distinct}\end{array}$$

$$\begin{array}{l}\neg \mathit{isWellFormed}(n, \mathit{tl}, \mathit{data}) \Longrightarrow \\ \Big(\mathit{data\text{-}lseg}_n(\mathit{tl}, e_1, \mathit{data}, e_2) := \mathit{false}\Big)\end{array}$$

$$\begin{array}{l}\mathit{isWellFormed}(0, \mathit{tl}, \mathit{data}) \Longrightarrow \\ \Big(\mathit{data\text{-}lseg}_0(\mathit{tl}, e_1, \mathit{data}, e_2) := (e_1 = e_2)\Big)\end{array}$$

$$\begin{array}{l}\mathit{isWellFormed}(n+1, \mathit{tl}, \mathit{data}) \Longrightarrow \\ \Big(\mathit{data\text{-}lseg}_{n+1}(\mathit{tl}, e_1, \mathit{data}, e_2) := (e_1 \neq e_2) * \exists c.\; \Big(\\ \qquad \mathit{pointsTo}(e_1, (\mathit{tl}, \mathit{Const}(c)) :: (\mathit{HD}(\mathit{data}))) * \\ \qquad \mathit{data\text{-}lseg}_n(\mathit{tl}, \mathit{Const}(c), \mathit{TL}(\mathit{data}), e_2)\Big)\end{array}$$

If *data* is not well-formed, i. e. if not all data-lists have the correct length or if there are multiple data-entries for one tag, then the list-predicate is false for all states. Otherwise, the empty list demands that the expressions e_1 and e_2 evaluate to the same value and the heap is empty. This is achieved using the strong pure predicate $e_1 = e_2$. On the other hand, if the list is not empty then e_1 is not allowed to be equal to e_2, because the list should not contain a cycle. Moreover, e_1 has to be a valid location in the heap such that the first elements of the data lists ($HD(data)$) are stored at this location. Moreover, the value for tag tl at this location is some constant c. At this location c the tail of the list has to start, containing the rest of the data lists ($TL(data)$).

Notice, that the notations $HD(data)$ and $TL(data)$ are informal. *data* is a list of pairs consisting of tags and lists. $HD(data)$ informally represents the finite map that maps these tags to the head of the lists. Similarly, $TL(data)$ represents the list that contains the pairs consisting of the tags and the tails of the lists.

Usually, the length of a singly-linked-list is not expressed explicitly. Moreover, as discussed in Sec. 2.1.3, several common list variants are defined:

$$\begin{array}{rcll} \mathit{data\text{-}lseg}(tl, e_1, data, e_2) & := & \exists n.\ \mathit{data\text{-}lseg}_n(tl, e_1, data, e_2) & \text{(HOL4-Thm 10)} \\ \mathit{data\text{-}list}(tl, e, data) & := & \mathit{data\text{-}lseg}(tl, e, data, \mathit{Const}(0)) & \text{(HOL4-Thm 9)} \\ \mathit{lseg}(tl, e_1, e_2) & := & \mathit{data\text{-}lseg}(tl, e_1, [], e_2) & \text{(HOL4-Thm 19)} \\ \mathit{list}(tl, e) & := & \mathit{lseg}(tl, e, \mathit{Const}(0)) & \text{(HOL4-Thm 18)} \end{array}$$

These list predicates are stack imprecise and compatible with semantic substitution.

Lemma 3.4.11 ((HOL4-Thm 59))**.**

$$\frac{\mathit{isWellFormed}\Big(e_1, \mathcal{V}\Big) \qquad \mathit{isWellFormed}\Big(e_2, \mathcal{V}\Big)}{\mathit{isStackImprecise}\Big(\mathit{data\text{-}lseg}(tl, e_1, data, e_2), \mathcal{V}\Big)}$$

Lemma 3.4.12 ((HOL4-Thm 64))**.**

$$\begin{array}{l} \mathit{isWellFormed}(e_1) \ \wedge \ \mathit{isWellFormed}(e_2) \Longrightarrow \\ \Big(\mathit{pred\text{-}var\text{-}update}[v, c](\mathit{data\text{-}lseg}(tl, e_1, data, e_2)) = \\ \mathit{data\text{-}lseg}(tl, \mathit{exp\text{-}var\text{-}update}[v, c](e_1), data, \mathit{exp\text{-}var\text{-}update}[v, c](e_2))\Big) \end{array}$$

3.4.2.3 Trees

Trees are a bit harder to define. The data-content of singly-linked lists in the heap can be described using lists of natural numbers. For trees, some kind of tree structure is needed to represent the data content. To this end, I defined an algebraic data-type for trees in HOL4. A tree in this definition is either a leaf *leaf* or a node *node*(*data-list*, *subtree-list*) containing a list of natural numbers *data-list* as data-content and a list of subtrees *subtree-list*. Using this data-type for trees, a predicate for trees can be defined.

Definition 3.4.13 (Tree Predicates (HOL4-Thms 15, 16, 17))**.** Let e be an expression, *tagL* and *dtagL* lists of tags and *data* a tree. Then *data-tree*(*tagL*, e, *dtagL*, *data*) describes a tree with root at location e that uses the tags in *tagL* to point to subtrees and the tags in *dtagL* for data-entries. This tree corresponds to the tree described by *data*.

$$\neg \mathit{isWellFormed}(\mathit{tagL}, \mathit{dtagL}, \mathit{data}) \implies \Big(\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data}) := \mathit{false}\Big)$$

$$\mathit{isWellFormed}(\mathit{tagL}, \mathit{dtagL}, \mathit{leaf}) \implies \Big(\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{leaf}) := (e = \mathit{Const}(0))\Big)$$

$$\begin{array}{l}\mathit{isWellFormed}([t_1, \ldots, t_n], [u_1, \ldots, u_m], \mathit{node}([c_1, \ldots, c_m], [s_1, \ldots, s_n])) \implies \\ \Big(\mathit{data\text{-}tree}([t_1, \ldots, t_n], e, [u_1, \ldots, u_m], \mathit{node}([c_1, \ldots, c_m], [s_1, \ldots, s_n])) := \\ \exists d_1, \ldots, d_n.\;\; \mathit{pointsTo}(e, [(t_1, \mathit{Const}(d_1)), \ldots, (t_n, \mathit{Const}(d_n)), (u_1, c_1), \ldots (u_m, c_m)]) * \\ \qquad \mathit{data\text{-}tree}([t_1, \ldots, t_n], \mathit{Const}(d_1), [u_1, \ldots, u_m], s_1) * \ldots * \\ \qquad \mathit{data\text{-}tree}([t_1, \ldots, t_n], \mathit{Const}(d_n), [u_1, \ldots, u_m], s_n)\Big)\end{array}$$

It is lengthy to describe $\mathit{isWellFormed}(\mathit{tagL}, \mathit{dtagL}, \mathit{data})$ formally (HOL4-Thm 17). Informally it means that the tree described by *data* has the right number of data-entries and subtrees in each node and that moreover all tags in `tagL` and `dtagL` are distinct to each other.

For well-formed *data*, there are two cases. If the predicate $\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data})$ represents the empty tree ($\mathit{data} = \mathit{leaf}$), then the root has to be NULL. Otherwise, the root has to point to the subtrees and the correct data-entries have to be stored in the heap. The locations of the subtrees are described by $d_1, \ldots, d_n$; their data-content $s_1, \ldots, s_n$ is already present in *data*.

Smallfoot uses binary trees without data. These can easily be defined:

$$\begin{array}{rcll}\mathit{tree}(\mathit{tagL}, e) & := & \exists \mathit{data}.\; \mathit{data\text{-}tree}(\mathit{tagL}, e, [], \mathit{data}) & \text{(HOL4-Thm 22)} \\ \mathit{bintree}(l, r, e) & := & \mathit{tree}([l, r], e) & \text{(HOL4-Thm 6)}\end{array}$$

These tree predicates are stack imprecise and compatible with semantic substitution.

Lemma 3.4.14 ((HOL4-Thm 61))**.**

$$\frac{\mathit{isWellFormed}\big(e, \mathcal{V}\big)}{\mathit{isStackImprecise}\big(\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data}), \mathcal{V}\big)}$$

Lemma 3.4.15 ((HOL4-Thm 65))**.**

$$\begin{array}{l}\mathit{isWellFormed}(e) \implies \\ \Big(\mathit{pred\text{-}var\text{-}update}[v, c](\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data})) = \\ \mathit{data\text{-}tree}(\mathit{tagL}, \mathit{exp\text{-}var\text{-}update}[v, c](e), \mathit{dtagL}, \mathit{data})\Big)\end{array}$$

3.4.2.4 Arrays

Arrays describe blocks of allocated heap locations. They can easily be defined using the points-to predicate:

Definition 3.4.16 (Array Predicate (HOL4-Thm 7))**.** Let e_b and e_l be expressions and *data* be a list of pairs consisting of tags and lists of natural numbers. Then the predicate $array(e_b, e_n, data)$ describes an array starting at e_b of length e_n containing *data*.

$$\begin{array}{l} isWellFormed(n, [(t_1, l_1), \ldots, (t_m, l_m)]) \;:= \\ \forall 1 \leqslant i < j \leqslant m.\ t_i \neq t_j \;\wedge\; \forall 1 \leqslant i \leqslant m.\ length(l_i) = n \end{array}$$

$$(s,h) \in array(e_b, e_n, data) \;:=$$
$$\begin{cases} (s,h) \in \Big(& \text{if Defined}(e_n(s)) \wedge (e_n(s) = n) \wedge \\ pointsTo(Const(b+0), \text{EL } 0\ data) * \ldots * & \quad \text{Defined}(e_b(s)) \wedge (e_b(s) = b) \wedge \\ pointsTo(Const(b+(n-1)), \text{EL } (n-1)\ data)\Big) & \quad isWellFormed(n, data) \\ & \\ false & \text{otherwise} \end{cases}$$

Notice, that EL n *data* is a informal notation, similar to the ones used by the definition of singly-linked list predicates. It denotes the finite map that for all $(t, l) \in data$ maps the tag t to $Const(el(n, l))$.

As discussed in Sec. 2.1.3 it is sometimes convenient to describe arrays by providing there first and last location instead of their length. This leads to the following definition:

$$\begin{array}{rcll} interval(e_1, e_2, data) & := & array(e_1, (e_2+1) - e_1, data) & \text{(HOL4-Thm 8)} \\ array(e_1, e_2) & := & array(e_1, e_2, []) & \text{(HOL4-Thm 5)} \end{array}$$

These array predicates are stack imprecise and compatible with semantic substitution.

Lemma 3.4.17 ((HOL4-Thm 60))**.**

$$\frac{isWellFormed\big(e_b, \mathcal{V}\big) \qquad isWellFormed\big(e_n, \mathcal{V}\big)}{isStackImprecise\big(array(e_b, e_n, data), \mathcal{V}\big)}$$

Lemma 3.4.18 ((HOL4-Thm 63))**.**

$$\begin{array}{l} isWellFormed(e_b) \;\wedge\; isWellFormed(e_n) \Longrightarrow \\ \Big(pred\text{-}var\text{-}update[v,c](array(e_b, e_n, data)) = \\ array(exp\text{-}var\text{-}update[v,c](e_b), exp\text{-}var\text{-}update[v,c](e_n), data)\Big) \end{array}$$

3.4.3 Program Constructs

In Section 3.3 program constructs like assignments, procedure calls, local variable declarations and conditional critical regions have been discussed. Moreover, conditional execution and while-loops were discussed and it was demonstrated that weak pure predicates can be used as conditions with assume and control structures. Holfoot can easily use comparison operators like $<$, $>$, $\geqslant$, $\leqslant$ in these conditions, because values are instantiated to natural numbers by Holfoot.

It remains to introduce program constructs that operate on the heap. This includes explicit memory allocation and deallocation as well as heap-lookup and heap-assignment operations.

3.4.3.1 Memory Allocation

Definition 3.4.19 (Memory Allocation (HOL4-Thm 48))**.** Given a variable v and an expression e, the action $new[e, v]$ tries to allocate a new consecutive portion of the heap of size e and stores the first location in the stack-variable v. The action fails, if e cannot be evaluated or if there is no write permission for the variable v. Otherwise, memory allocation always succeeds. This means that this action does not fail because there is insufficient free memory.

$$
\mathit{new}[e,v](s,h) := \begin{cases} \left\{ \begin{array}{l} (\mathit{stack\text{-}var\text{-}update}[v,l](s), h') \mid \\ l \neq 0 \;\wedge\; \forall l \leqslant l' < l+n.\; l' \notin \mathit{dom}(h) \;\wedge \\ \mathit{dom}(h') = \{l,\ldots,l+(n-1)\} \cup \mathit{dom}(h) \;\wedge \\ \forall l' \in \mathit{dom}(h).\; h'(l') = h(l') \end{array} \right\} & \begin{array}{l} \text{if Defined}(e(s)) \;\wedge\; e(s) = n \;\wedge \\ \quad v \in \mathit{dom}(s) \;\wedge\; \mathit{perm}(s,v) = \top \end{array} \\ \top & \text{otherwise} \end{cases}
$$

This memory allocation action is a local action (HOL4-Thm 4). The following inference rule can be used for its symbolic evaluation:

$$
\frac{\begin{array}{c} \textit{(HOL4-Thm 31)} \\ \mathit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big) \qquad v \in \mathcal{W} \\ \mathcal{P}' = \mathit{image}\ (\mathit{pred\text{-}var\text{-}update}[v,c])\ \mathcal{P} \qquad e' = \mathit{exp\text{-}var\text{-}update}[v,c](e) \\ [\![\mathcal{W};\mathcal{R} \mid \{\mathit{array}(\mathit{Var}(v), e', [])\} \cup \mathcal{P}']\!]\ \mathit{prog}\ [\![Q]\!] \end{array}}{[\![\mathcal{W};\mathcal{R} \mid \{\mathit{Var}(v) = \mathit{Const}(c)\} \cup \mathcal{P}]\!]\ \mathit{new}[e,v]\ ;\ \mathit{prog}\ [\![Q]\!]}
$$

Notice, that this rule is very similar to the assignment rule presented in Sec. 3.3.6.5. This rule also updates the value of a stack variable. Therefore, the old value of this variable has to be propagated first. In the common case, that a single heap cell is allocated, i. e. in the case $e = \mathit{Const}(1)$, the array becomes a point-to predicate (HOL4-Thm 32).

3.4.3.2 Memory Deallocation

Definition 3.4.20 (Memory Deallocation (HOL4-Thm 33))**.** Given two expressions e_b and e_l, the action $\mathit{dispose}[e_l, e_b]$ tries to deallocate a consecutive portion of the heap of size e_l starting at location e_b. The action fails, if e_l or e_b cannot be evaluated or if one of the locations is not allocated.

$$
\mathit{dispose}[e_l,e_b](s,h) := \begin{cases} \top & \text{if } \neg\text{Defined}(e_l(s)) \\ (s,h) & \text{if Defined}(e_l(s)) \;\wedge\; e_l(s) = 0 \\ (s, h \backslash \{l,\ldots,l+(n-1)\}) & \begin{array}{l} \text{if Defined}(e_l(s)) \;\wedge\; e_l(s) = n \;\wedge\; n \neq 0 \;\wedge \\ \quad \text{Defined}(e_b(s)) \;\wedge\; e_b(s) = l \;\wedge \\ \quad \{l,\ldots,l+(n-1)\} \subseteq \mathit{dom}(h) \end{array} \\ \top & \text{otherwise} \end{cases}
$$

dispose$[e_l, e_b]$ is a local action (HOL4-Thm 1) for well-formed expressions e_b and e_l. The following inference rule can be used for its symbolic evaluation:

(HOL4-Thm 23)

$$\frac{\mathit{isWellFormed}\big(e_l, \mathcal{W} \cup \mathcal{R}\big) \qquad \mathit{isWellFormed}\big(e_b, \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}; \mathcal{R} \mid \mathcal{P}]\!]\ \mathit{prog}\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \{\mathit{array}(e_b, e_l, \mathit{data})\} \cup \mathcal{P}]\!]\ \mathit{dispose}[e_l, e_b]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

In the common case, that a single heap cell is deallocated, i. e. in the case $e_l = \mathit{Const}(1)$, the array becomes a point-to predicate (HOL4-Thm 24).

3.4.3.3 Heap Lookup

Definition 3.4.21 (Heap Lookup (HOL4-Thm 35))**.** For an expression e, a stack variable v and a tag t, the action *heap-lookup*(v, e, t) tries to lookup the value stored in the heap at location e indexed by the tag t and store it in the variable v. The action fails, if e cannot be evaluated, the location e is not allocated in the heap or if there is no write permission for v.

$$\mathit{heap\text{-}lookup}[v, e, t](s, h) \ := \begin{cases} (\mathit{stack\text{-}var\text{-}update}[v, c](s), h) & \text{if Defined}(e(s)) \ \wedge\ e(s) = l\ \wedge \\ & \quad l \in \mathit{dom}(h)\ \wedge\ h(l)(t) = c\ \wedge \\ & \quad v \in \mathit{dom}(s)\ \wedge\ \mathit{perm}(s, v) = \top \\ \top & \text{otherwise} \end{cases}$$

heap-lookup$[v, e, t]$ is a local action (HOL4-Thm 3) for well-formed expressions e. The following inference rule can be used for its symbolic evaluation:

(HOL4-Thm 28)

$$\frac{t \in \mathit{dom}(L) \qquad v \in \mathcal{W} \\ \mathit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big) \qquad \mathit{isWellFormed}\big(L(t), \mathcal{W} \cup \mathcal{R}\big) \\ \mathcal{P}' = \mathit{image}\ (\mathit{pred\text{-}var\text{-}update}[v, c])\ (\{\mathit{pointsTo}(e, L)\} \cup \mathcal{P}) \\ e' = \mathit{exp\text{-}var\text{-}update}[v, c](L(t)) \\ [\![\mathcal{W}; \mathcal{R} \mid \{\mathit{Var}(v) = e'\} \cup \mathcal{P}']\!]\ \mathit{prog}\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \{\mathit{Var}(v) = \mathit{Const}(c)\} \cup \{\mathit{pointsTo}(e, L)\} \cup \mathcal{P}]\!]\ \mathit{heap\text{-}lookup}[v, e, t]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

Arrays might need to be split such that the precondition of a Hoare triple contains *pointsTo*(e, L). Since this is tedious, there is a specialised inference rule for arrays (HOL4-Thm 29). Another minor problem might be, that the inference rule requires the expression e to occur in the precondition. Often this is problematic, because expressions in the precondition in contrast to those in the program get normalised. Therefore, the following inference rule is useful. It rewrites e in the command using the precondition:

(HOL4-Thm 30)

$$\frac{\mathit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big) \qquad \mathit{isWellFormed}\big(e', \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}; \mathcal{R} \mid \{e = e'\} \cup \mathcal{P}]\!]\ \mathit{heap\text{-}lookup}[v, e', t]\ ;\ \mathit{prog}\ [\![Q]\!]}{[\![\mathcal{W}; \mathcal{R} \mid \{e = e'\} \cup \mathcal{P}]\!]\ \mathit{heap\text{-}lookup}[v, e, t]\ ;\ \mathit{prog}\ [\![Q]\!]}$$

3.4.3.4 Heap Assignment

Definition 3.4.22 (Heap Assignment (HOL4-Thm 34))**.** For two expressions e_l, e_v and a tag t, the action *heap-assign*(e_l, t, e_v) tries to store the value of expression e_v in the heap at location e_l indexed by tag t. The action fails, if e_l or e_v cannot be evaluated or if the location e_l is not allocated in the heap.

$$\textit{heap-assign}[e_l, t, e_v](s,h) \;:=\; \begin{cases} (s, \textit{update}[l, \lambda t_2.\ \text{if } t_2 = t \text{ then } v \text{ else } h(l)(t)]h) & \text{if Defined}(e_l(s)) \;\wedge\; e_l(s) = l \;\wedge \\ & \quad \text{Defined}(e_v(s)) \;\wedge\; e_v(s) = v \;\wedge \\ & \quad l \in \textit{dom}(h) \\ \top & \text{otherwise} \end{cases}$$

heap-assign$[e_l, t, e_v]$ is a local action (HOL4-Thm 2) for well-formed expressions e_l, e_v. The following inference rule can be used for its symbolic evaluation:

$$\frac{\begin{array}{c}\textit{(HOL4-Thm 25)} \\ \textit{isWellFormed}\big(e_l, \mathcal{W} \cup \mathcal{R}\big) \qquad \textit{isWellFormed}\big(e_v, \mathcal{W} \cup \mathcal{R}\big) \\ \textit{isStackImprecise}\big(\textit{pointsTo}(e_l, \textit{update}[t, e_v](L)), \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}; \mathcal{R} \mid \{\textit{pointsTo}(e_l, \textit{update}[t, e_v](L))\} \cup \mathcal{P}]\!]\ \textit{prog}\ [\![Q]\!] \end{array}}{[\![\mathcal{W}; \mathcal{R} \mid \{\textit{pointsTo}(e_l, L)\} \cup \mathcal{P}]\!]\ \textit{heap-assign}[e_l, t, e_v]\ ;\ \textit{prog}\ [\![Q]\!]}$$

Similar to heap lookups there are specialised inference rules for arrays (HOL4-Thm 26) as well as an inference rule that allows rewriting the expression e_l in the program (HOL4-Thm 27).

3.4.4 Implicit Information

As discussed in Sec. 3.3.8 *VRImp* becomes more interesting with heaps present. If for example the precondition of a Hoare triple contains the predicates *pointsTo*(e_1, T_1) and *pointsTo*(e_2, T_2), one can safely add the predicate $e_1 \neq e_2$ to the precondition as well. The idea is, that if two location l_1 and l_2 are present in separate parts of a heap, one can conclude $l_1 \neq l_2$. Exploiting this as well as the fact that no heap contains location 0, leads to the following definitions:

Definition 3.4.23 (Expressions in Heap (HOL4-Thms 42, 36, 37))**.** Given two multisets of predicates $\mathcal{C}$ and $\mathcal{P}$ and an expression e, the predicates *in-heap*$(\mathcal{C}, \mathcal{P}, e)$ and *in-heap*$_0(\mathcal{C}, \mathcal{P}, e)$ are defined as follows:

$$\begin{array}{rcl} \textit{in-heap}(\mathcal{C}, \mathcal{P}, e) & := & \forall s, s', h, h'.\ \ (s', h') \in \circledast\mathcal{C} \;\wedge\; (s, h) \in \circledast\mathcal{P} \;\wedge\; s \preceq s' \implies \\ & & \Big(\text{Defined}(e(s)) \;\wedge\; \big(e(s) \in \textit{dom}(h) \;\wedge\; e(s) \neq 0\big)\Big) \\[1em] \textit{in-heap}_0(\mathcal{C}, \mathcal{P}, e) & := & \forall s, s', h, h'.\ \ (s', h') \in \circledast\mathcal{C} \;\wedge\; (s, h) \in \circledast\mathcal{P} \;\wedge\; s \preceq s' \implies \\ & & \Big(\text{Defined}(e(s)) \;\wedge\; \big(e(s) \in \textit{dom}(h) \;\vee\; e(s) = 0\big)\Big) \end{array}$$

Informally, *in-heap*$(\mathcal{C}, \mathcal{P}, e)$ states that if some state satisfies the separating conjunction of all predicates in $\mathcal{P}$, then the value of e in this state is in the domain of the heap.

$\textit{in-heap}_0(\mathcal{C}, \mathcal{P}, e)$ is slightly weaker ($\forall \mathcal{C}, \mathcal{P}, e.\ \textit{in-heap}(\mathcal{C}, \mathcal{P}, e) \Rightarrow \textit{in-heap}_0(\mathcal{C}, \mathcal{P}, e)$ (HOL4-Thm 45)). It allows e to evaluate to 0 as well. The multiset of predicates $\mathcal{C}$ can be used to restrict the set of stacks under consideration.

These definitions have the intended effect: if two expressions are present in separate parts of a heap, they do not evaluate to the same value.

(HOL4-Thm 43)

$$\frac{\mathcal{P}_1 \cup \mathcal{P}_2 \subseteq \mathcal{C} \qquad \textit{in-heap}(\mathcal{C}, \mathcal{P}_1, e_1) \qquad \textit{in-heap}(\mathcal{C}, \mathcal{P}_2, e_2)}{\textit{VRImpUnequal}(\mathcal{C}, e_1, e_2)}$$

Notice, that $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{C}$ are multisets and that therefore the union and subset operations of multisets are used.

Other interesting inference rules hold as well:

(HOL4-Thm 46)

$$\frac{\mathcal{P}_1 \cup \mathcal{P}_2 \subseteq \mathcal{C} \qquad \textit{in-heap}(\mathcal{C}, \mathcal{P}_1, e_1) \qquad \textit{in-heap}_0(\mathcal{C}, \mathcal{P}_2, e_2)}{\textit{VRImpUnequal}(\mathcal{C}, e_1, e_2)}$$

(HOL4-Thm 44)

$$\frac{\mathcal{P} \subseteq \mathcal{C} \qquad \textit{in-heap}(\mathcal{C}, \mathcal{P}, e)}{\textit{VRImpUnequal}(\mathcal{C}, e, \textit{Const}(0))}$$

(HOL4-Thm 40)

$$\frac{\mathcal{P}_1 \cup \mathcal{P}_2 \subseteq \mathcal{C} \qquad \textit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big) \qquad \textit{in-heap}_0(\mathcal{C}, \mathcal{P}_1, e) \qquad \textit{in-heap}_0(\mathcal{C}, \mathcal{P}_2, e)}{\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \{e = \textit{Const}(0)\})}$$

(HOL4-Thm 329)

$$\frac{\textit{isWellFormed}\big(e_1, \mathcal{W} \cup \mathcal{R}\big) \qquad \textit{isWellFormed}\big(e_2, \mathcal{W} \cup \mathcal{R}\big) \qquad \textit{VRImpUnequal}(\mathcal{C}, e_1, e_2)}{\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, e_1 \neq e_2)}$$

In order to add implicit information explicitly to a Hoare triple or a frame inference predicate, one has to find $\mathcal{P}$ such that for given $\mathcal{W}$, $\mathcal{R}$ and $\mathcal{C}$ the predicate $\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P})$ holds. Holfoot searches for $\mathcal{P}' \subseteq \mathcal{C}$ and e' such that $\textit{in-heap}(\mathcal{C}, \mathcal{P}', e')$ or $\textit{in-heap}_0(\mathcal{C}, \mathcal{P}', e')$ holds. Then, predicates of the form $\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P})$ are derived using the inference rules presented above. Two such predicates $\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_1)$ and $\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_2)$ can be combined to $\textit{VRImp}(\mathcal{W}, \mathcal{R}, \mathcal{C}, \mathcal{P}_1 \cup \mathcal{P}_2)$ (HOL4-Thm 374).

It remains to present some inference rules for *in-heap* and $\textit{in-heap}_0$. There are the following structural inference rules:

(HOL4-Thm 47)

$$\frac{\mathcal{P} \subseteq \mathcal{P}' \qquad \textit{in-heap}(\mathcal{C}, \mathcal{P}, e)}{\textit{in-heap}(\mathcal{C}, \mathcal{P}', e)}$$

(HOL4-Thm 41)

$$\frac{\mathcal{P} \subseteq \mathcal{P}' \qquad \textit{in-heap}_0(\mathcal{C}, \mathcal{P}, e)}{\textit{in-heap}_0(\mathcal{C}, \mathcal{P}', e)}$$

(HOL4-Thm 38)

$$\frac{}{\textit{in-heap}_0(\mathcal{C}, \mathcal{P}, \textit{Const}(0))}$$

(HOL4-Thm 39)

$$\frac{}{\textit{in-heap}_0(\mathcal{C}, \{e = 0\}, e)}$$

The predicate for single heap cells demands that its location is in the heap. Lists and trees require that their root location is either in the heap or 0.

(HOL4-Thm 21)

$$\frac{}{\textit{in-heap}(\mathcal{C}, \{\textit{pointsTo}(e, T)\}, e)}$$

(HOL4-Thm 14)

$$\frac{\mathit{isWellFormed}(e)}{\mathit{in\text{-}heap}_0(\mathcal{C}, \{\mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data})\}, e)}$$

(HOL4-Thm 13)

$$\frac{\mathit{isWellFormed}(e)}{\mathit{in\text{-}heap}_0(\mathcal{C}, \{\mathit{data\text{-}list}(\mathit{tl}, e, \mathit{data})\}, e)}$$

For list-segments it is slightly more complicated. If the start and end location are unequal, then the start location is in the heap.

(HOL4-Thm 12)

$$\frac{\mathit{isWellFormed}(e_1) \qquad \mathit{isWellFormed}(e_2) \\ \mathit{VRImpUnequal}(\mathcal{C}, e_1, e_2) \qquad \mathcal{C} \neq \varnothing}{\mathit{in\text{-}heap}(\mathcal{C}, \{\mathit{data\text{-}lseg}(\mathit{tl}, e_1, \mathit{data}, e_2)\}, e_1)}$$

3.4.5 Frame Inference

It remains to present specialised inference rules to handle frame inference predicates that utilise the newly introduced predicates. These inference rules are lengthy and complicated. One problem is that singly-linked list and array predicates represent their data-contents as lists of pairs consisting of tags and lists of natural numbers. These lists of pairs represent finite maps from tags to lists of natural numbers. In the following finite map notations are used for these lists. This informal notation allows expressing some inference rules more concisely.

(HOL4-Thm 58)

$$\frac{\begin{array}{c} \mathcal{L} \subseteq \mathit{dom}(L') \subseteq \mathit{dom}(L) \\ \forall t \in \mathit{dom}(L') \backslash \mathcal{L}.\ L(t) = L'(t) \\ \mathit{isWellFormed}\big(e, \mathcal{W} \cup \mathcal{R}\big) \\ \forall t \in \mathit{dom}(L).\ \mathit{isWellFormed}\big(L(t), \mathcal{W} \cup \mathcal{R}\big) \\ \forall t \in \mathcal{L}.\ \mathit{isWellFormed}\big(L'(t), \mathcal{W} \cup \mathcal{R}\big) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\mathit{pointsTo}(e, L)\} \cup \mathcal{C} \mid \mathcal{P} \mid (\mathit{image}\ (\lambda t.\ L(t) = L'(t))\ \mathcal{L}) \cup \mathcal{Q} \mid \mathit{fP}]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\mathit{pointsTo}(e, L)\} \cup \mathcal{P} \mid \{\mathit{pointsTo}(e, L')\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

This rule states that $\mathit{pointsTo}(e, L)$ can be moved to the context, if $\mathit{pointsTo}(e, L)$ occurs in $\mathcal{P}$ and $\mathit{pointsTo}(e, L')$ in $\mathcal{Q}$. Besides checking that all relevant expressions are well-formed, one has to be careful about L and L'. All the tags mentioned by L' have to be present in L as well. If for a tag $t \in \mathit{dom}(L')$ the expressions $L(t)$ and $L'(t)$ are not equal, then an equality check has to be added to $\mathcal{Q}$. The set of tags $\mathcal{L}$ is used for this purpose. It contains all tags $t \in \mathit{dom}(L')$ for which $L(t)$ and $L'(t)$ are not equal.

(HOL4-Thm 56)

$$\frac{\begin{array}{c} tl \in dom(L) \qquad dom(data) \subseteq dom(L) \\ isWellFormed\Big(e_1, \mathcal{W} \cup \mathcal{R}\Big) \\ isWellFormed\Big(e_2, \mathcal{W} \cup \mathcal{R}\Big) \\ VRImpUnequal(\mathcal{C} \cup \{pointsTo(e_1, L)\} \cup \mathcal{P}, e_1, e_2) \\ \forall t \in dom(L).\ isWellFormed\Big(L(t), \mathcal{W} \cup \mathcal{R}\Big) \\ \mathcal{Q}_{eq} = image\ (\lambda t.\ L(t) = Const(hd(data(t))))\ dom(data) \\ P_{wf} = BoolPred(\text{not empty } data\ \wedge\ \text{all distinct } (\{tl\} \cup dom(data))) \\ P_{tl} = data\text{-}lseg(tl, L(tl), TL(data), e_2) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{pointsTo(e_1, L)\} \cup \mathcal{C} \mid \mathcal{P} \mid \mathcal{Q}_{eq} \cup \{P_{wf}, P_{tl}\} \cup \mathcal{Q} \mid fP]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{pointsTo(e_1, L)\} \cup \mathcal{P} \mid \{data\text{-}lseg(tl, e_1, data, e_2)\} \cup \mathcal{Q} \mid fP]\!]}$$

If $pointsTo(e_1, L)$ is in $\mathcal{P}$ and $data\text{-}lseg(tl, e_1, data, e_2)$ in $\mathcal{Q}$, one can try to move the first node of the list-segment to the context. In order to do this, one has to be sure, that the list segment is not empty. This is ensured by $VRImpUnequal(\mathcal{C} \cup \{pointsTo(e_1, L)\} \cup \mathcal{P}, e_1, e_2)$. Notice, that this check succeeds trivially for lists, i. e. in case $e_2 = Const(0)$. Additionally, all involved expressions have to be well-formed and L needs to contain all the tags needed by the list-segment. If these conditions are satisfied, the first node of the list-segment is moved to the context. A predicate describing the tail of the list (P_{tl}) remains in $\mathcal{Q}$. Moreover, a well-formedness check for the original list (P_{wf}) is added to $\mathcal{Q}$ and it has to be shown that L contains the proper data-entries ($\mathcal{Q}_{eq}$) described by *data*.

(HOL4-Thm 54)

$$\frac{\begin{array}{c} dom(data_2) \subseteq dom(data_1) \qquad \text{all distinct } (dom(data_2)) \\ isWellFormed\Big(e_1, \mathcal{W} \cup \mathcal{R}\Big) \qquad isWellFormed\Big(e_2, \mathcal{W} \cup \mathcal{R}\Big) \\ P_{eq} = BoolPred(\forall t \in dom(data_2).\ data_1(t) = data_2(t)) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{data\text{-}lseg(tl, e_1, data_1, e_2)\} \cup \mathcal{C} \mid \mathcal{P} \mid \{P_{eq}\} \cup \mathcal{Q} \mid fP]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{data\text{-}lseg(tl, e_1, data_1, e_2)\} \cup \mathcal{P} \mid \{data\text{-}lseg(tl, e_1, data_2, e_2)\} \cup \mathcal{Q} \mid fP]\!]}$$

This rule tries to move a list-segment $data\text{-}lseg(tl, e_1, data_1, e_2)$ to the context, if it is in $\mathcal{P}$ and $data\text{-}lseg(tl, e_1, data_2, e_2)$ in $\mathcal{Q}$, i. e. if a list-segment predicate is in $\mathcal{Q}$ that differs only in its data-content. Besides checking some well-formedness conditions, one has to be careful that $data_1$ contains an entry for all tags used by $data_2$. If these conditions are satisfied, $data\text{-}lseg(tl, e_1, data_1, e_2)$ can be moved to the context. A check that $data_2$ uses the same data as $data_1$ remains in $\mathcal{Q}$.

This inference rule becomes much more complicated, if different end-points of the list-

segments are considered:

(HOL4-Thm 53)

$$\frac{\begin{array}{c} \textit{in-heap}_0(\mathcal{C} \cup \mathcal{P}, \mathcal{C} \cup \mathcal{P}, e_3) \\ \textit{dom}(\textit{data}_2) \subseteq \textit{dom}(\textit{data}_1) \\ \text{all distinct } (\{tl\} \cup \textit{dom}(\textit{data}_1)) \Longrightarrow \text{all distinct } (\textit{dom}(\textit{data}_2)) \\ \textit{isWellFormed}\Big(e_1, \mathcal{W} \cup \mathcal{R}\Big) \\ \textit{isWellFormed}\Big(e_2, \mathcal{W} \cup \mathcal{R}\Big) \\ \textit{isWellFormed}\Big(e_3, \mathcal{W} \cup \mathcal{R}\Big) \\ P_{eq} = \textit{BoolPred}(\forall t \in \textit{dom}(\textit{data}_2).\ \textit{data}_1(t) = \text{TAKE (LENGTH } \textit{data}_1)\ \textit{data}_2(t)) \\ P_{rest} = \textit{data-lseg}(tl, e_2, \text{DROP (LENGTH } \textit{data}_1)\ \textit{data}_2, e_3) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\textit{data-lseg}(tl, e_1, \textit{data}_1, e_2)\} \cup \mathcal{C} \mid \mathcal{P} \mid \{P_{eq}, P_{rest}\} \cup \mathcal{Q} \mid fP]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\textit{data-lseg}(tl, e_1, \textit{data}_1, e_2)\} \cup \mathcal{P} \mid \{\textit{data-lseg}(tl, e_1, \textit{data}_2, e_3)\} \cup \mathcal{Q} \mid fP]\!]}$$

Now, one has to check, whether the list-segment described by $\textit{data-lseg}(tl, e_1, \textit{data}_1, e_2)$ or the one described by $\textit{data-lseg}(tl, e_1, \textit{data}_2, e_3)$ is longer. The condition $\textit{in-heap}_0(\mathcal{C} \cup \mathcal{P}, \mathcal{C} \cup \mathcal{P}, e_3)$ guarantees that e_3 is not present in the list-segment from e_1 to e_2. Therefore, the list-segment from e_1 to e_3 is at least as long as the one ending with e_2. Notice, that the condition $\textit{in-heap}_0(\mathcal{C} \cup \mathcal{P}, \mathcal{C} \cup \mathcal{P}, e_3)$ holds trivially for lists, i. e. in case $e_3 = \textit{Const}(0)$. After moving $\textit{data-lseg}(tl, e_1, \textit{data}_1, e_2)$ to the context, a list-segment from e_2 to e_3 remains in $\mathcal{Q}$. Moreover, $\mathcal{Q}$ contains the equality check on data known already from the previous inference rule.

Inference rules similar to the ones presented for list-segments exist for trees and arrays as well. The following rule, for example, moves the root node of a tree to the context. It is similar to the inference rule that removes the first element of a list-segment.

(HOL4-Thm 57)

$$\frac{\begin{array}{c} \textit{tagL} \cup \textit{dtagL} \subseteq \textit{dom}(L) \qquad \textit{isWellFormed}\Big(e, \mathcal{W} \cup \mathcal{R}\Big) \\ \forall t \in \textit{dom}(L).\ \textit{isWellFormed}\Big(L(t), \mathcal{W} \cup \mathcal{R}\Big) \\ P = \exists tl, dl, cl.\ \textit{BoolPred}(\textit{data} = \textit{node}(dl, tl)) * P_{eq}(dl, tl, cl) * P_{rest}(tl, cl) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\textit{pointsTo}(e, L)\} \cup \mathcal{C} \mid \mathcal{P} \mid \{P\} \cup \mathcal{Q} \mid fP]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\textit{pointsTo}(e, L)\} \cup \mathcal{P} \mid \{\textit{data-tree}(\textit{tagL}, e, \textit{dtagL}, \textit{data})\} \cup \mathcal{Q} \mid fP]\!]}$$

Again, it has to be shown that some expressions are well-formed and that L contains all the tags needed by the tree. In contrast to the corresponding inference rule for list-segments, one does not need to prove that the tree is not empty though. This corresponds to the case of lists instead of list-segments. Similar to the rule for list-segments, a predicate describing well-formedness conditions, one describing the remainder of the tree (P_{rest}) and one describing that L points to the proper values (P_{eq}) remain in $\mathcal{Q}$. Because a formal definition of these predicates is lengthy and complicated, they are only informally discussed here. The original symbolic tree *data* has to be a node containing some data-entries *dl* and some subtrees *tl*. The root-nodes of these subtrees are *cl*. P_{eq} describes that the values in *dl* and *cl* are stored in the heap at location e. P_{rest} describes the subtrees *tl* with root nodes *cl*.

Inference rules that move predicates to the context that differ only in their data-content

are much easier. Only a check that the data is equivalent remains in $\mathcal{Q}$:

(HOL4-Thm 55)

$$\frac{\begin{array}{c} \mathit{isWellFormed}\Big(e, \mathcal{W} \cup \mathcal{R}\Big) \\ P_{\mathit{tree}}(\mathit{data}) = \mathit{data\text{-}tree}(\mathit{tagL}, e, \mathit{dtagL}, \mathit{data}) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{P_{\mathit{tree}}(\mathit{data}_1)\} \cup \mathcal{C} \mid \mathcal{P} \mid \{\mathit{BoolPred}(\mathit{data}_1 = \mathit{data}_2)\} \cup \mathcal{Q} \mid \mathit{fP}]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{P_{\mathit{tree}}(\mathit{data}_1)\} \cup \mathcal{P} \mid \{P_{\mathit{tree}}(\mathit{data}_2)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

(HOL4-Thm 52)

$$\frac{\begin{array}{c} \mathit{dom}(\mathit{data}_2) \subseteq \mathit{dom}(\mathit{data}_1) \qquad \text{all distinct } (\mathit{dom}(\mathit{data}_2)) \\ \mathit{isWellFormed}\Big(e, \mathcal{W} \cup \mathcal{R}\Big) \\ P_{\mathit{eq}} = \mathit{BoolPred}(\forall t \in \mathit{dom}(\mathit{data}_2).\ \mathit{data}_1(t) = \mathit{data}_2(t)) \\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \{\mathit{array}(e_b, e_n, \mathit{data}_1)\} \cup \mathcal{C} \mid \mathcal{P} \mid \{P_{\mathit{eq}}\} \cup \mathcal{Q} \mid \mathit{fP}]\!] \end{array}}{[\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C} \mid \{\mathit{array}(e_b, e_n, \mathit{data}_1)\} \cup \mathcal{P} \mid \{\mathit{array}(e_b, e_n, \mathit{data}_2)\} \cup \mathcal{Q} \mid \mathit{fP}]\!]}$$

There are many more similar inference rules. In particular, there are further inference rules for handling arrays. Thanks to Holfoot's architecture, it is very easy to add additional inference rules.

3.5 Holfoot Implementation

3.5.1 Overview

Holfoot can handle inputs in the language described in Sec. 2.1. These input specifications are processed in three steps. First, the input is parsed. In a preprocessing step the resulting term is then transformed into a conjunction of conditional variable as resource Hoare triples. Finally, these triples are verified. The main method for verification is applying inference rules for the forward analysis of programs and for frame inference predicates.

During parsing, the program is analysed and additional information added. For example, the necessary read and write permissions are determined by inspecting the bodies of procedures. Moreover, the parsing step uses heuristics to distinguish between program variables and specification variables or to determine the scope of specification variables. Some concepts, like referring to the old value of call-by-reference arguments in the postcondition of a procedure, are removed by reducing them to other concepts. The result of parsing is a single HOL4 term describing a list of resources with their invariants and a list of specified procedures (see Sec. 3.2.7). The semantics of this term is formally defined in HOL4. However, neither the input language, nor the parser are handled formally.

A preprocessing step then abstracts procedure calls with their specifications as described in Sec. 3.2.7. Similarly, conditional critical regions are abstracted using resource invariants. The result is a conjunction of Hoare triples that do not depend on the environment any more. Next, well-formedness conditions of the pre- and postconditions are proved. This transforms the Hoare triples into conditional variable as resource Hoare triples.

In order to verify these conditional Hoare triples, inference rules are applied. Simplifying slightly, there is a set of inference rules, containing the rules presented above for the

forward analysis of problems, plus frame inference rules and some additional rules not mentioned above. There are calls of HOL4 tools in this set as well. These include calls of HOL4's simplifier with specialised rewrite rules as well as calling certain decision procedures for list expressions and arithmetic problems. As discussed in Sec. 3.3.7.5 one has to be careful, when introducing frame inference predicates. Otherwise, the inference rules in this set can be applied in an arbitrary order. Holfoot's automation tries to find an inference rule to apply. If the problem is not solved yet, but no more inference rules can be applied, the automation stops and the user can interactively work on the remaining problem.

Holfoot's automation heavily relies on consequence conversions and quantifier heuristics. I implemented these concepts in HOL4 for Holfoot.

3.5.2 Consequence Conversions

In general, inference rules are implemented as *consequence conversions*, i. e. ML functions that given a term P return a theorem of the form $Q \Longrightarrow P$. HOL4 provides infrastructure for *conversions*, i. e. ML function that given P return theorems of the form $P = Q$. However, there was no infrastructure for consequence conversions.

Similarly to the infrastructure for conversions, I implemented functions that sequentially compose consequence conversions or that try a replacement consequence conversion in case the first one failed. There are also consequence conversions that allow theorems to be used as implicational rewrite rules, a reflexive consequence conversion and one that always fails. Most importantly, though, I provide infrastructure for applying consequence conversions repeatedly at subpositions.

Imagine, you need to show that $\mathit{isStackImprecise}\Big(\mathit{data\text{-}lseg}(\mathit{tl}, \mathit{Var}(x), \mathit{data}, \mathit{Const}(c)), \{x\}\Big)$ holds. The following steps are performed to show this:

$$
\begin{array}{lll}
\mathit{isStackImprecise}\Big(\mathit{data\text{-}lseg}(\mathit{tl}, \mathit{Var}(x), \mathit{data}, \mathit{Const}(c)), \{x\}\Big) & \Longleftarrow & \text{(HOL4-Thm 59)} \\
\mathit{isWellFormed}\Big(\mathit{Var}(x), \{x\}\Big) \;\wedge\; \mathit{isWellFormed}\Big(\mathit{Const}(c), \{x\}\Big) & \Longleftarrow & \text{(HOL4-Thm 340)} \\
\mathit{isWellFormed}\Big(\mathit{Var}(x), \{x\}\Big) \;\wedge\; \mathit{true} & \Longleftarrow & \text{(HOL4-Thm 340)} \\
x \in \{x\} \;\wedge\; \mathit{true} & \Longleftarrow & \\
\mathit{true} & &
\end{array}
$$

The first application of an inference rule leads to a conjunction. Then, inference rules need to be applied to the conjuncts. This application of inference rules at subpositions exploits that $P \Longrightarrow Q$ implies $P \wedge R \Longrightarrow Q \wedge R$ and $R \wedge P \Longrightarrow R \wedge Q$.

Similar congruence rules exist for other operations like disjunction, negation, implication or existential and universal quantification as well. I implemented infrastructure for applying consequence conversions at subpositions using these congruence rules. The resulting *depth consequence conversion* is highly configurable. It supports top-down as well as bottom-up search for subpositions. Moreover, there is support for caching results and counting the number of applied consequence conversions.

The infrastructure for consequence conversions is an essential part of Holfoot. It is used to prove well-formedness properties as demonstrated above. More importantly, it the used to apply Holfoot's inference rules for symbolic execution and frame inference predicates.

3.5.3 Quantifier Heuristics

Holfoot's automation introduces many quantifiers, especially when reasoning about data-content. A common source of quantifiers are procedure specifications that contain free specification variables. These introduce universally quantified variables when verifying the procedure specification and existential quantification for procedure calls. Quantifiers might also be introduced by existential quantification in specifications or by expanding definitions of predicates.

Whatever the cause, Holfoot frequently has to handle both universal and existential quantification. HOL4 provides tools for instantiating simple cases. It can for example simplify $\forall x.\ (x = y) \Longrightarrow P(x)$ to $P(y)$ by instantiating x with y. Such simple instantiations are unfortunately not sufficient for Holfoot's automation. Therefore, I implemented *quantifier heuristics* that are more powerful.

My tools can handle more complicated Boolean connectives. Their main advantage, however, is that they can utilise knowledge about data types. They can, for example, simplify $\forall l.\ \neg(l = []) \Longrightarrow P(l)$ to $\forall l_h, l_t.\ P(l_h :: l_t)$ using the fact that the list l is not empty, iff a head l_h and a tail l_t of l exist. Another advantage of my quantifier heuristics is that they do not need to prove equality. Instead of finding an instantiation y and prove $(\exists x.P(x)) = P(y)$, it is often sufficient to come up with a reasonable, but not formally justified guess y and use $P(y) \Longrightarrow \exists x.P(x)$. A common situation, where such guesses are used are frame inference predicates of the form $\exists x.\ [\![\mathcal{W}, \mathcal{R}, \mathcal{W}';\ \mathcal{C}(x) \mid \mathcal{P}(x) \mid \{\mathit{BoolPred}(x = y)\} \cup \mathcal{Q}(x) \mid \mathit{fP}]\!]$. In this case, the guess y is used, which is usually sensible. If $\mathcal{C}$ and $\mathcal{P}$ do not depend on x, it can be formally justified.

Chapter 4

Conclusion

4.1 Summary

In this work, a separation logic framework inside HOL4 is presented. This framework is based on Abstract Separation Logic. The formalisation of Abstract Separation Logic (see Sec. 3.2) follows the original work [7] closely. However, the original work is extended by adding procedures, which may be mutually recursive. Moreover, concepts like nondeterministic choice between an infinite number of choices and quantified best local actions are added. There are additional inference rules like loop specifications (see Sec. 2.3.2) as well.

This formalisation of Abstract Separation Logic is instantiated by adding a stack with explicit read / write permissions (see Sec. 3.3). This follows ideas presented in *Variables as Resource in Hoare Logics* by Parkinson, Bornat and Calcagno [31]. These ideas have, however, been adapted to an Abstract Separation Logic context. Moreover, a lot of effort is spent defining well-formedness conditions and normal forms. The concept of stack impreciseness is, for example, extended in this work to restrict the set of variables used by a predicate. Moreover, Hoare triples are extended to carry well-formedness information and to guarantee that the programs do not modify permissions.

Based on these normal forms and well-formedness properties a frame inference predicate is introduced (see Sec. 3.3.7). In contrast to the frame inferences used by tools like Smallfoot, I extended the frame problem with a context. This context can store additional information and thereby reduce the need to be careful about the order in which inference rules are applied. This idea has meanwhile been adapted by JStar [10]. Besides adding a context, predicates that describe how the frame is used are added to the frame inference predicate as well. By demanding certain properties of these predicates, interesting additional inference rules can be proved for frame inference predicates. For example, it allows moving quantifiers out of the frame calculation.

In an additional instantiation step, this model is extended by a heap (see Sec. 3.4). This allows actions for explicit memory management, heap assignments and heap look-ups to be defined. Moreover, predicates are added that describe datastructures in the heap like singly-linked lists, trees or arrays.

This last instantiation is a formalisation of Smallfoot [2, 3] in HOL4. A parser as well as specialised tactics are implemented. The resulting tool Holfoot (see Chapter 2) can

reason about the partial correctness of programs written in a simple imperative language similar to the one used by Smallfoot (see Sec. 2.1). In contrast to Smallfoot, Holfoot can reason about the content of data structures instead of just their shape. This allows verifying fully functional specifications. Simple specifications (see Appx. B.1) like the original Smallfoot specifications can be verified automatically. More complicated specifications (see Appx. B.2 and Appx. B.3) can be verified interactively using all of HOL4's infrastructure.

Holfoot can for example verify fully functional specifications of sorting algorithms like mergesort (see Appx. B.2.13) or quicksort (see Appx. B.2.15). Another interesting example is the fully functional specification of insertion into a red / black tree (see Appx. B.2.17).

4.2 Conclusion

The framework developed in this work is to my knowledge the first formalisation of Abstract Separation Logic. Building Holfoot as an instantiation of this framework demonstrates the flexibility and power of the framework and thereby the flexibility and power of Abstract Separation Logic.

Formalising Abstract Separation Logic itself was straightforward. Since large parts of the instantiations originate in this work, the instantiations took more effort. Apparently simple concepts sometimes caused trouble. Defining the semantics of local variable declarations in terms of local actions was, for example, surprisingly difficult. Similarly, the definition of procedure calls with call-by-value arguments turned out to be tricky. Some technical problems do not even exist in the high level presentation. An example is expressing Lemma 3.2.63 in HOL4. This lemma allows handling mutually recursive procedures by abstracting procedure calls with their specification. These procedure specifications usually use free specification variables. Because the number and type of these free specification variables differs between the procedures, it is hard to express Lemma 3.2.63 in HOL4 without typing problems. After first introducing a fixed type for specification variables, which is a very significant restriction, I finally use program abstractions and quantified best local actions to solve the typing problem. The free specification variables are hidden inside the quantified best local actions. Other concepts, that feature prominently in Holfoot, turned out to be easily implementable. Loop specifications, for example, are very easy to add.

The most significant theoretical contribution is probably the definition of frame inference predicates (see Sec. 3.3.7). These predicates hide the existential quantification of the frame by integrating a frame predicate. Moreover, a context has been added that allows additional information to be preserved. It was tricky to define, which frame predicates should be allowed. The well-formedness condition *isFramePred* is carefully designed such that Boolean conditions as well as quantifiers can be moved out of the frame inference predicate.

Implementing the framework resulted in contributions to the HOL4 system, particularly the addition of libraries for consequence conversions and quantifier heuristics. Moreover, HOL4's list and pair libraries were extended. As purely technical contributions, HOL4's pretty printer was extended to allow Holfoot's syntax highlighting. Additionally, a HTML-backend was added to HOL4's pretty-printer in order to implement Holfoot's web-interface.

This work was started with the goal to build a formalisation of Smallfoot as an instantiation of a general separation logic framework based on Abstract Separation Logic. This goal has been achieved. Holfoot is able to parse Smallfoot specifications and verify most of them automatically. Moreover, Holfoot can reason about the data-content of datastructures instead of just their shape. This allows fully functional specifications. Moreover, Holfoot can reason about arrays and pointer arithmetic as well. It combines the automation of separation logic with the power of HOL4.

Despite this success, I would do many things differently, if I started again. Abstract Separation Logic is powerful and flexible. However, its semantics is far from intuitive, especially with respect to concurrency. However, even the definitions of conditional execution and while-loops are not intuitive. Therefore, I would formalise intuitive semantics as the foundation of the framework. These semantics should not be concerned with local actions, race freedom, resource invariants or similar high level concepts. Semantics similar to the ones used by Abstract Separation Logic that discuss these concepts should be introduced as a sound abstraction of the intuitive semantics. Both semantics should consider termination. Moreover, synchronisation primitives like fork / join parallelism and storeable locks are worth considering.

Besides these changes to the semantic foundation, I would not implement a tool similar to Smallfoot as a case study again. Smallfoot is a well known, relatively simple separation logic tool. That makes it a good choice for a case study. Moreover, it comes with many example specifications. However, I would try to implement a programming language that is closer to real world languages. In particular, I would consider reasoning about errors caused by expressions like arithmetic overflow or division by zero problems. The heap used by this programming language should not contain entries for all tags. Similar to locations, there should only be a finite number of tags allocated at each heap location. With this change, memory allocations would need to explicitly allocate certain tags. In general, memory allocation and deallocation operations should become more realistic. Allocations should nondeterministically not be able to allocate memory. The deallocation action should – following C conventions – not require an explicit argument of how much memory to free.

4.3 Future Work

Holfoot uses a very simple C-like programming language. This language is powerful enough to reason about interesting problems, though. All problems from the VSTTE'10 competition can, for example, be solved using Holfoot (see Appx. B.3). However, the language is very restricted. Not even for-loops are available. Moreover, the language is untyped. All values and locations are natural numbers. Since these are unbounded in size, no arithmetic overflows occur. Similarly other real world problems like division by zero are ignored. It would be interesting to use the framework with a more realistic programming language. A good candidate might be a subset of C like CMinor [21]. I do not anticipate any major problem when using the framework with a language closer to C. The essential concepts are already there and many definitions and much of the automation should be reusable. However, formalising an interesting subset of C is very time consuming [28, 29].

Another interesting future extension might be guessing specifications. Currently, Holfoot requires all procedures and loops to be annotated. External tools could be used to guess

loop invariants and specifications of auxiliary procedures. There is no need to trust these external tools. The loop invariants or procedure specifications that these tools provide can be verified formally using Holfoot.

Using the separation logic framework with real world programming languages as well as automatically inferring loop annotations can be achieved relatively simply and quickly by using the separation logic framework as a backend for existing tools. JStar [10] for example parses Java programs and preprocesses them. This results essentially in an annotated control flow graph with best local actions as the only operations. I'm currently working on formalising these control flow graphs in the separation logic framework. Verifying these annotated control flow graphs would be a compromise between building a trustworthy tool and building a tool for real world programs quickly. On the one hand side, the semantics of Java are not formalised, on the other a large part of the verification and especially some tricky separation logic problems are handled formally in HOL4.

Bibliography

[1] A.W. Appel and S. Blazy. Separation logic for small-step Cminor. In K. Schneider and J. Brandt, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 4732 of *LNCS*, pages 5–21, Kaiserslautern, Germany, 2007. Springer.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic, 2005. URL `citeseer.ist.psu.edu/berdine05symbolic.html`.

[3] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40(1):259–270, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1047659.1040327.

[5] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2006.12.034.

[6] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

[7] C. Calcagno, P.W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2908-9. doi: http://dx.doi.org/10.1109/LICS.2007.30.

[8] Cristiano Calcagno, Dino Distefano, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *In Proceedings of POPL-36*, 2009.

[9] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596565.

[10] Dino Distefano and Matthew J. Parkinson J. jStar: towards practical verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: http://doi.acm.org/10.1145/1449764.1449782.

[11] Dino Distefano, Peter W. O'Hearn, Peter W. Ohearn, and Hongseok Yang. A local shape analysis based on separation logic. In *In TACAS*, pages 287–302. Springer, 2006.

[12] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *In ESOP09: European Symposium on Programming, volume 5502 of LNCS*, pages 363–377. Springer, 2009.

[13] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University, 1993.

[14] Alexey Gotsman. Logics and analyses for concurrent heap-manipulating programs. Technical Report UCAM-CL-TR-758, University of Cambridge, Computer Laboratory, October 2009. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-758.pdf`.

[15] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS07)*, 2007.

[16] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005. ISBN 978-3-540-69147-1.

[17] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/373243.375719.

[18] Bart Jacobs and Frank Piessens. The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008. URL `https://lirias.kuleuven.be/handle/123456789/197789`.

[19] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems*. Springer-Verlag, November 2010. URL `https://lirias.kuleuven.be/handle/123456789/275140`.

[20] Rafal Kolanski and Gerwin Klein. Types, maps and separation logic. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 276–292, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: http://dx.doi.org/10.1007/978-3-642-03359-9_20.

[21] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1111320.1111042.

[22] Stephen Magill and et al. Inferring invariants in separation logic for imperative list-processing programs. In *3RD SPACE WORKSHOP*, 2006.

[23] N. Marti, R. Affeldt, and A. Yonezawa. Towards formal verification of memory properties using separation logic. In *22nd Workshop of the Japan Society for Software Science and Technology, Tohoku University, Sendai, Japan, September 13–15, 2005.* Japan Society for Software Science and Technology, Sep. 2005.

[24] The Coq development team. *The Coq proof assistant reference manual*, 2009. URL `http://coq.inria.fr`. Version 8.3.

[25] Andrew McCreight. Practical tactics for separation logic. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 343–358, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: http://dx.doi.org/10.1007/978-3-642-03359-9_24.

[26] Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 355–369, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70543-7. doi: http://dx.doi.org/10.1007/978-3-540-70545-1_34.

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[28] Michael Norrish. A formal semantics for C++. Technical report, NICTA, 2008.

[29] Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf`.

[30] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, September 2001. ISBN 3-540-42554-3.

[31] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logics. In *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2631-4. doi: http://dx.doi.org/10.1109/LICS.2006.52.

[32] Lawrence C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-56543-X.

[33] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

[34] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. ISBN 978-3-540-71065-3.

[35] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: http://doi.acm.org/10.1145/1190216.1190234.

[36] Thomas Tuerk. Local reasoning about while-loops. In R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, editors, *VSTTE 2010 Workshop Proceedings*, pages 29–39. ETH Zurich, 2010.

[37] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In Lus Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. ISBN 978-3-540-74406-1. URL `http://dblp.uni-trier.de/db/conf/concur/concur2007.html#VafeiadisP07`.

[38] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004. ISBN 3-540-23024-6.

Appendix A

Holfoot Installation

Holfoot is distributed as an example inside the HOL4 distribution. HOL4 is an open source project with a BSD-style licence that allows its free use in commercial products. Its available from Sourceforge (`http://hol.sourceforge.net`). The separation logic framework and its instantiation Holfoot can be found in the `examples/separationLogic` subdirectory.

Holfoot needs to be run from within HOL4 in order to allow interactive proofs. Automated proofs are available through a command line version as well. At Holfoot's webpage (`http://holfoot.heap-of-problems.org`) there are precompiled versions of this command line tool available. Moreover, a web-interface of Holfoot can be found at this site. This web-interface might be sufficient to get a first impression of Holfoot or run Holfoot on just a few selected examples.

A.1 Installation of HOL4

I recommend using HOL4 with Poly/ML 5.4 or newer (`http://www.polyml.org`) and the experimental kernel. Holfoot works with Moscow ML as well, but there is trouble building binaries and it is much slower. Holfoot works with the standard and the experimental kernel of HOL4. However, the kernels differ slightly when introducing new variable names. Holfoot's interactive examples are written for the experimental kernel. In order to get them working with the standard kernel, one needs to rename variables. Usually this involves adding or removing priming, i. e. replacing a variable `x` with `x'` or vice versa.

The current version of Holfoot is just available via the subversion repository of HOL4. This documentation is written with respect to revision 8816. I recommend getting the newest version, though. Documentation on how to install HOL4 is available from its webpage (`http://hol.sourceforge.net`). In the following a short description is given that covers the standard case. If you have problems, please refer to HOL4's documentation.

First install Poly/ML 5.4 or newer (see `http://www.polyml.org`). Then download HOL4's sources from the subversion repository:

```
svn co https://hol.svn.sourceforge.net/svnroot/hol/HOL
```

You should now have the sources of HOL4 in a new directory called `HOL`. These sources include Holfoot in subdirectory `examples/separationLogic/src/holfoot`. Documentation can be found in subdirectory `Manual`. Especially the tutorial is good for beginners. Its first chapter contains a detailed description of how to install HOL4.

To build HOL4 from this new source directory, change to the `HOL` directory and use the following command to configure it:

```
poly < tools/smart-configure.sml
```

One common problem with smart-configure is, that it is sometimes not able to figure out the correct library directory. In this case, please create a file `tools-poly/poly-includes.ML` to specify the library directory. This file should contain a line of the following form:

```
val polymllibdir = "path-to-dir-containing-libpolymain.a";
```

After successful configuration, HOL4 needs to be build with the experimental kernel:

```
./bin/build -expk
```

On old machines, this build process might take several hours. Afterwards, the HOL4 binaries should have been created in the directory `HOL/bin`. I recommend adding this directory the `PATH` environment variable, such that the HOL4 binaries, in particular `Holmake`, can easily be used.

There are several different methods to use HOL4. I recommend using HOL4's Emacs-mode. Documentation of this mode can be found in `Manual/Interaction`.

A.2 Installation of Holfoot

After installing HOL4 successfully, run `Holmake` in directory `examples/separationLogic/src/holfoot/poly`. This should build Holfoot. As part of this process, several executables are created:

- `holfoot` is the command-line version of Holfoot. It can be used for examples that can be handled automatically.
- `holfoot-full` is an extended command-line version. It allows interactive proof scripts to be replayed.
- `holfoot-web` is used for step-wise proofs on Holfoot's web-interface.

Precompiled versions of `holfoot-full` and `holfoot` are available at Holfoot's web-page (`http://holfoot.heap-of-problems.org`). This site also contains a web-interface to Holfoot.

Holfoot is able to use the SMT-solver Yices through HOL4's libraries for external SMT solvers. In order to allow the command line version to use Yices, simply add the location of the `yices` executable to the `PATH` environment variable. Holfoot has been tested with Yices 1.0.27.

A.3 Testing Holfoot

Holfoot comes with a collection of example specifications. These can be found in the directory `examples/separationLogic/src/holfoot/EXAMPLES` (or just `EXAMPLES` when using a precompiled version). There are four subdirectories:

- `automatic` contains examples that can be verified automatically
- `interactive` contains examples that need interactive proof scripts
- `not_solvable` contains examples that contain errors
- `vstte` contains solutions to the VSTTE'10 competition problems[1]

There are several different types of files in these directories:

- `.sf`-files are Smallfoot specifications that can be verified with Smallfoot and Holfoot.
- `.sf-orig`-files are Smallfoot specifications that cannot be verified with Holfoot, usually because they contain errors that are not detected by Smallfoot.
- `.dsf`- and `.dsf2`-files are Holfoot specifications.
- `.hol`-files contain HOL4 proof scripts.

The examples in directory `automatic` can be verified with the command line version of Holfoot. For a first test, call

```
./holfoot ../EXAMPLES/automatic/list_length.dsf
```

This verifies the list-length example. A call of `holfoot` without any parameters or with the parameter `-h` prints all the available command line options. If you have trouble with displaying Unicode, it can be turned off by the parameter `-nu`; the parameter `-r` disables VT100-terminal specials. When processing multiple specifications, Holfoot's quiet mode (parameter `-q`) is useful. For a more extensive test, you can for example run

```
./holfoot -q ../EXAMPLES/automatic/*
```

All examples except `binary_search-shape.dsf` should be successfully verified. The binary search example requires arithmetic reasoning beyond the automatic capabilities of HOL4. The external SMT-solver Yices can be used to verify it:

```
./holfoot --yices ../EXAMPLES/automatic/binary_search-shape.dsf
```

If you want to understand how Holfoot works or more interestingly, why certain examples fail to be verified, you can use the Holfoot's interactive mode. This interactive mode allows stepping through the verification process. It does not provide real interaction. To explore the full interactive capabilities of Holfoot, you have to run it inside HOL4. To test the interactive mode, run

```
./holfoot -i ../EXAMPLES/automatic/list_length.dsf
```

After parsing the input specification, Holfoot stops and asks for commands. Pressing `?` (followed by enter) prints the list of available commands.

`holfoot-full` can be used to replay proof-scripts, i.e. especially the `.hol`-files in the directories `interactive` and `vstte`. `holfoot-full` is just intended for quickly replaying

[1] `http://www.macs.hw.ac.uk/vstte10/Competition.html`

proof scripts. The full interactive capabilities of Holfoot can only be used inside an interactive HOL4 session.

Proof scripts usually read several specification files from disc. Therefore, it is important to start `holfoot-full` in directory `examples/separationLogic/src/holfoot/EXAMPLES`. An example call is

```
../poly/holfoot-full -f interactive/mergesort.hol
```

Appendix B

Example Specifications

B.1 Automatic Examples

B.1.1 General List Example

This example is copied from Smallfoot. It contains several standard operations on singly-linked lists. This example specifies just the shape of data-structures. Fully functional specifications of several of the operations are provided as separate examples.

Listing B.1: automatic/list.sf

```
list_traverse(x) [list(x)] {
  local t;
  t = x;
  /* lseg(x,t) should be framed */
  while(t != NULL) [lseg(x,t) * list (t)] {
    t = t->tl;
  }
} [ list (x)]

lseg_traverse(x,y) [lseg(x,y)] {
  local t;
  t = x;
  if(t != y) {
    t = t->tl;
    lseg_traverse(t,y);
  } else {}
} [lseg(x,y)]

list_copy(p) [list(p)] {
  local t;
  t = p; q = NULL;
  while(t != NULL) [list(q) * lseg(p,t) * list (t)] {
    sq = q; q = new(); q->tl = sq;
    t = t -> tl;
  }
} [ list (p) * list (q)]

list_reverse(o;i) [list(i)] {
  local t;
  o = NULL;
  while (i != NULL) [list(i) * list (o)] {
    t = i->tl; i->tl = o; o = i; i = t;
  }
} [ list (o)]

list_deallocate(x) [lseg(x,0)] {
  local t;
  while(x != NULL) [lseg(x,0)] {
    t = x; x = x->tl; dispose t;
  }
} [emp]

list_append(x;y) [list(x) * list (y)] {
  local t, n;
  if (x == NULL) {
    x = y;
  } else {
    t = x; n = t->tl;
    while (n != NULL) [lseg(x,t) * t |-> n * list (n)] {
      t = n; n = t->tl;
    }
    t->tl = y;
  }
} [ list (x)]

list_insert(l;x) [x|-> * list(l)] {
  local s, t, u;
  if (l == NULL) {
    x->tl = NULL; l = x;
  } else {
    s = x->hd; t = l->hd;
    if (s > t) {
      u = l->tl; list_insert(u;x); l->tl = u;
    } else {
      x->tl = l; l = x;
    }
  }
} [ list (l)]

list_remove(l;x) [list(l)] {
  local t;
  if (l != NULL) {
    if (l == x) {
      l = l->tl; dispose(x);
    } else {
      t = l->tl;
```

```
    list_remove(t;x);
    l->tl = t;
  }
```

```
  }
} [ list (l)]
```

B.1.2 List Length

One of the introductory examples is calculating the length of a list recursively. A fully functional specification of an iterative implementation needs a complicated loop invariant. Using a loop-specification simplifies the specification considerably.

Listing B.2: automatic/list_length.dsf

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [ data_list (c,cdata) * (r == ``LENGTH cdata``)]
```

Listing B.3: automatic/list_length-iter.dsf

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  r = 0; t = c;
  while (t != NULL)
      [data_lseg (c, _cdata1, t) * data_list (t, _cdata2) *
       (r == ``LENGTH _cdata1``) * ``cdata = _cdata1 ++ _cdata2``] {
    t = t->tl; r = r + 1;
  }
} [ data_list (c,cdata) * (r == ``LENGTH cdata``)]
```

Listing B.4: automatic/list_length-iter.dsf2

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  r = 0; t = c;
  loop_spec [ data_list (t, data)] {
    while (t != NULL) {
      t = t->tl; r = r + 1;
    }
  } [ data_list (old(t), data) * (r == ``LENGTH data + old(r)``)]
} [ data_list (c,cdata) * (r == ``LENGTH cdata``)]
```

B.1.3 List Reverse

Another introductory example is reversal of a singly-linked list. Again, there are two specifications, one is using a loop-invariant and the other a loop-specification:

Listing B.5: automatic/reverse.dsf

```
list_reverse(i;) [data_list(i,data)] {
  local p, x;
  p = NULL;
  while (i != NULL)
      [ data_list (i, _idata ) * data_list (p,_pdata) *
        ''data = (REVERSE _pdata) ++ _idata''] {
    x = i->tl; i->tl = p; p = i; i = x;
  }
  i = p;
} [ data_list (i,'' REVERSE data'')]
```

Listing B.6: automatic/reverse.dsf2

```
list_reverse(i;) [data_list(i,data)] {
  local p, x;
  p = NULL;
  loop_spec [ data_list (i,data) * data_list (p, data2)] {
    while (i != NULL) {
      x = i->tl; i->tl = p; p = i; i = x;
    }
  } [ data_list (p, ''(REVERSE data)++data2'')]
  i = p;
} [ data_list (i,'' REVERSE data'')]
```

B.1.4 List Copy

Copying a singly-linked list is another example that benefits from using loop-specifications:

Listing B.7: automatic/copy.dsf

```
list_copy(z;c) [data_list(c,data)] {
  local x,y,w,d;
  if (c == NULL) {
    z=NULL;
  } else {
    z=new(); z->tl=NULL; x = c->dta; z->dta = x;
    w=z; y=c->tl;
    while (y!=NULL)
        [data_lseg (c,'' _data1++[_cdate]'',y) *
         data_list(y,''_data2'') *
         data_lseg(z,_data1,w) *
         w |-> tl:0,dta:_cdate *
         ''data = _data1 ++ _cdate::_data2''] {
      d=new(); d->tl=NULL; x=y->dta; d->dta=x;
      w->tl=d; w=d; y=y->tl;
    }
  }
} [ data_list (c,data) * data_list (z,data)]
```

Listing B.8: automatic/copy.dsf2

```
list_copy(z;c) [data_list(c,data)] {
  local x,y,w,d;
  if (c == NULL) {
    z=NULL;
  } else {
    z=new(); z->tl=NULL; x = c->dta; z->dta = x;
    w=z; y=c->tl;
    loop_spec [w |-> [tl:0,dta:#date] *
                     data_list (y, data2)] {
      while (y != NULL) {
        d=new(); d->tl=NULL; x=y->dta; d->dta=x;
        w->tl=d; w=d; y=y->tl;
      }
    } [ data_list (old(w), ''date::data2'') *
        data_list (old(y), data2)]
  }
} [ data_list (c,data) * data_list (z,data)]
```

B.1.5 List Append

Appending two singly-linked lists is another example that uses loop-specifications.

Listing B.9: automatic/append.dsf

```
list_append(x;y) [data_list(x,xdata) * data_list (y,ydata)] {
  local n,t;
  if (x == NULL) {
    x = y;
  } else {
    t = x; n = t->tl;
    while (n != NULL) [data_lseg(x,_xdata1,t) * t |-> [tl:n,dta:_tdate] *
                       data_list(n,_xdata2) * ``xdata = _xdata1 ++ _tdate::_xdata2``] {
      t = n; n = t->tl;
    }
    t->tl = y;
  }
} [ data_list (x, ``xdata++ydata``)]
```

Listing B.10: automatic/append.dsf2

```
list_append(x;y) [data_list(x,xdata) * data_list (y,ydata)] {
  local n,t;
  if (x == NULL) {
    x = y;
  } else {
    t = x; n = t->tl;
    loop_spec [t |-> [tl:n,dta:#tdate] * data_list(n,data2) * data_list(y, data3)] {
       while (n != NULL) {
         t = n; n = t->tl;
       }
       t->tl = y;
    } [ data_list (old(t ),`` tdate ::( data2++data3)``)]
  }
} [ data_list (x, ``xdata++ydata``)]
```

It can also be used to demonstrate unrolling loops. The loop that moves to the end of the list starting at `n` has been modified in this example compared to the previous one. The first look-up of the tail has been moved inside the loop. In order to get a nice loop-specification, unrolling is used.

Listing B.11: automatic/append-unroll.dsf

```
list_append(x;y) [data_list(x,xdata) * data_list (y,ydata)] {
  local n,t;
  if (x == NULL) {
    x = y;
  } else {
    n = x;
    loop_spec [unroll 1] [(t == #tc) * (t |-> tl:n,dta:#tdate) * data_list (n,data2) * data_list (y, data3)] {
       while (n != NULL) {
         t = n; n = t->tl;
       }
       t->tl = y;
    } [ data_list (#tc,`` tdate ::( data2++data3)``)]
  }
} [ data_list (x, ``xdata++ydata``)]
```

B.1.6 List Allocation and Deallocation by Length

If the specification is restricted to shape descriptions, most Holfoot examples can be handled by Smallfoot. This is an example that demonstrates that even for very simple specifications, using the content of data-structures might be essential. This example allocates a list of length `n` and then deallocates it again. The interesting point is that

the deallocation is using the length of the list. Therefore, the loop-specification has to be able to reason about the length.

Listing B.12: automatic/list_alloc_dealloc_length.dsf

```
list_alloc_delete(;n) [emp] {
  local t,i,c;
  i = 0; c = NULL;
  loop_spec [i <= n * data_list(c,_data) * ``LENGTH data = i``] {
     while (i < n) {
        t=new() [dta]; t->tl=c; c=t;
        i=i+1;
     }
  } [ data_list (c,_data) * ``LENGTH data = n`` * (n == i)]

  loop_spec [ data_list (c,_data) * ``LENGTH data = i``] {
     while (i != 0) {
        t=c->tl; dispose c; c=t;
        i=i-1;
     }
  } [emp]
} [emp]
```

B.1.7 List Filter

Removing all occurrences of an element from a singly-linked list, i. e. filtering is one of Smallfoot's examples:

Listing B.13: automatic/filter.sf

```
list_filter(l;x) [list(l)] {
  local y, z, e;
  y = l;
  z = NULL;
  while (y != NULL) [if (y==l) then list(l) else lseg(l,z) * z |-> tl:y * list (y)] {
    e = y->dta;
    if(e == x) { /* need to remove y */
      if(y == l) { /* first link */
        l = y->tl; dispose y; y = l;
      } else { /* not first link */
        e = y->tl; z->tl = e; dispose y; y = z->tl;
      }
    } else { /* don't need to remove y */
      z = y; y = y->tl;
    }
  }
} [ list (l)]
```

Verifying a fully-functional specification of this procedure requires interaction and is presented in Appx. B.2.5. A recursive implementation can, however, be verified automatically. Filtering is also a good example for demonstrating global specification variables. Holfoot can verify filtering with respect to an arbitrary predicate P.

Listing B.14: automatic/filter_rec.dsf

```
list_filter(l;x) [data_list(l,data)] {
  local e, m;
  if (l == NULL) {
  } else {
    e = l->dta; m = l->tl;
    list_filter(m;x);
    if (e == x) {
      dispose l; l = m;
    } else {
      l->tl = m;
    }
  }
} [ data_list (l, ``FILTER (\e:num. ~(e = x)) data``)]
```

Listing B.15: automatic/filter_rec-gen.dsf

```
global P;

list_filter(l;) [data_list(l,data)] {
  local e, m;
  if (l == NULL) {
  } else {
    e = l->dta; m = l->tl;
    list_filter(m;);
    if (``~(P e)``) {
      dispose l; l = m;
    } else {
      l->tl = m;
    }
  }
} [ data_list (l, ``FILTER P data``)]
```

B.1.8 Queue

Smallfoot provides an example about queues. Holfoot can even handle a fully-functional specifications of this example automatically.

Listing B.16: automatic/queue.dsf

```
/* queues represented as a linked list with front and back pointers
 * queue(f,r) iff if f==NULL then emp else lseg(f,r) * r|->NULL) */

/* insert new node at rear; without pointers into the stack, have to pass in f */
insert(f,r;d) [if (f == NULL) then ``data = []`` else
                ``~(data = [])`` * data_lseg(f,``FRONT data``,r) *
                r |-> [tl:NULL,dta:``LAST data``]] {
  local t;
  t = new(); t->tl = NULL; t->dta = d;
  if(f == NULL) {
    f = t; r = t;
  } else {
    r->tl = t; r = t;
  }
} [f != NULL * data_lseg(f,``data``, r) * r|->[tl:NULL,dta:d]]

/* delete node from front */
delete(f;r) [data_lseg(f, data, r) * r|->[tl:NULL,dta:#data_last]] {
  local t;
  t = f; f = f->tl; dispose t;
} [if f==NULL then emp else data_lseg(f,``(TL data):num list``, r) * r|->[tl:NULL,dta:#data_last]]
```

B.1.9 Binary Tree Copy / Deallocate

Smallfoot provides examples of copying a binary tree and deallocating it.

Listing B.17: automatic/tree.sf

```
tree_copy(s;t) [tree(t)] {
  local i, j, ii, jj;
  if (t == NULL) {
    s = t;
  } else {
    i = t->l; j = t->r;
    tree_copy(ii;i); tree_copy(jj;j);
    s = new(); s->l = ii; s->r = jj;
  }
} [tree(s) * tree(t)]

tree_deallocate(t) [tree(t)] {
  local i, j;
  if (t == NULL) {
  } else {
    i = t->l; j = t->r;
    tree_deallocate(i); tree_deallocate(j);
    dispose(t);
  }
} [emp]
```

Listing B.18: automatic/parallel_tree_deallocate.dsf

```
tree_deallocate(t) [tree(t)] {
  local i, j;
  if(t == NULL) {
  } else {
    i = t->l; j = t->r;
    tree_deallocate(i) || tree_deallocate(j);
    dispose t;
  }
} [emp]
```

Holfoot can verify a fully-functional specification of copying a binary tree as well:

Listing B.19: automatic/tree_copy.dsf

```
tree_copy(s;t) [data_tree(t, data)] {
  local i, j, k, ii, jj;
  if(t == NULL) s = t;
  else {
    i = t->l; j = t->r; k = t->dta;
    tree_copy(ii;i); tree_copy(jj;j);
    s = new(); s->l = ii; s->r = jj; s->dta = k;
  }
} [data_tree ([l,r];s,[dta]:data) * data_tree(t,[dta]:data)]
```

B.1.10 Races

The following examples are intended to check, whether Smallfoot detects races. The first example calls two procedures in parallel that both access the heap. As these accesses happen at different heap locations, the program is race-free.

Listing B.20: automatic/business1.sf

```
proc(x,y) [x|-> ] {
  x->tl = y;
} [x|-> tl:y]

main(x,z;)[emp] {
  x = new(); z= new(); x->tl=3; z->tl=3;
  proc(x,4) || proc(z,5);
} [x|->tl:4 * z|-> tl:5]
```

The next example is very similar. However, the parallel calls now try to access the same heap location. This race is detected by Holfoot (and Smallfoot).

Listing B.21: not_solvable/business1.sf

```
proc(x,y) [x|-> ] {
  x->tl = y;
```

```
} [x|-> tl:y]

main(x;)[emp] {
  x = new(); x->tl=3;
  proc(x,4) || proc(x,5);
} [x|->tl:4]
```

A similar problem is calling two functions with the same call-by-reference argument. The following example uses the variable `x` as a call-by-reference argument of two parallel calls of a function that needs write access to `x`. It causes a stack-race.

Listing B.22: not_solvable/stack_race.sf

```
assign(x;y) {
  x = y;
}

stack_race() {
  local x;
  assign(x;42) || assign(x;13);
}
```

If one of arguments is a call-by-value one, it is fine. This behaviour is debatable. It depends on the exact semantics of parallel procedure calls. Holfoot evaluates the call-by-value arguments before evaluating either of the parallel procedure calls. Therefore, Holfoot accepts the following specification. Smallfoot in contrast, rejects it, because it evaluates the parallel procedure calls separately.

Listing B.23: automatic/passive_stack_race.sf

```
assign(x;y) {
  x = y;
}

stack_race() {
  local x,y;
  assign(x;42) || assign(y;x);
}
```

B.1.11 Buffers

This Smallfoot example demonstrates how resources and conditional critical regions can be used to transfer ownership of parts of the state between threads running in parallel.

Listing B.24: automatic/pointer_transfering_buffer.sf

```
resource buf (c) [if c==NULL then emp else c|->]

init() { c = NULL; }
put(x) [x|->] { with buf when (c==NULL) { c = x; } } [emp]
get(y;) [emp] { with buf when (c!=NULL) { y = c; c = NULL; } } [y|->]
putter() [emp] { local x; x = new(); put(x); putter(); } [emp]
getter() [emp] { local y; get(y;); dispose(y); getter(); } [emp]
main() [emp] { putter() || getter(); } [emp]
```

Listing B.25: automatic/pointer_non_transfering_buffer.sf

```
init() { c = NULL; }

resource buf (c) [emp]

put(x) [x|->] {
  with buf when (c==NULL) { c = x; }
} [x|->]

get(y;) [emp] {
  with buf when (c!=NULL) { y = c; }
} [emp]

putter() {
  local x;
  x = new(); put(x); dispose x;
}

getter() {
  local y;
  get(y;);
}

main() {
  putter() || getter();
}
```

Split binary semaphores can also be implemented.

Listing B.26: automatic/split_binary_semaphore.sf

```
c;

init() { b_free = new(); free = 1; busy = 0; }

resource free (free,b_free)
  [ if  free==0 then emp else b_free|->]
resource busy (busy,b_busy)
  [ if  busy==0 then emp else b_busy|->]

produce(m;) {}

producer() {
  local m,b;
  produce(m;);
  with free when (free == 1) {
    free = 0; b = b_free;
  }
  b->c = m;
  with busy when (busy == 0) {
    busy = 1; b_busy = b;
  }
  producer();
}

consume(n) {}
consumer() {
  local n,b;
  with busy when (busy == 1) {
    busy = 0; b = b_busy;
  }
  n = b->c;
  with free when (free == 0) {
    free = 1; b_free = b;
  }
  consume(n);
  consumer();
}

main() { producer() || consumer(); }
```

B.1.12 Memory Manager

Smallfoot comes with several memory manager examples, which can be used with Holfoot as well.

Listing B.27: automatic/memory_manager.sf

```
init() { f = NULL; }

resource mm (f) [list(f)]
```

```
alloc(x;) {
  with mm when(true) {
    if (f == NULL) {x = new();} else
               {x = f; f = x->tl;}
  }
} [x|->]

dealloc(y) [y|->] {
  with mm when(true) { y->tl = f; f = y; }
}

proc(;y) {
  local x;
  alloc(x;); x->tl = y; dealloc(x);
}

main() {
  proc(42) || proc(13);
}
```

Listing B.28: automatic/mm_buf.sf

```
init() { f = NULL; c = NULL; }

resource mm (f) [list(f)]

alloc(x;) {
  with mm when(true) {
    if(f==NULL) x = new();
    else { x = f; f = x->tl; }}
} [x|->]

dealloc(y) [y|->] {
  with mm when(true) { y->tl = f; f = y; }
}

resource buf (c) [if c==NULL then emp else c|->]

put(x) [x|->] {
  with buf when (c==NULL) { c = x; }
} [emp]

get(y;) [emp] {
  with buf when (c!=NULL) { y = c; c = NULL; }
} [y|->]

putter() {
  local x;
  alloc(x;); put(x); putter();
}

getter() {
  local y;
  get(y;); dealloc(y); getter();
}

main() {
  putter() || getter();
}
```

Listing B.29: automatic/mm_non_blocking.sf

```
/****
 * This implements a version of malloc and free. Malloc uses a semi DCAS instruction
 * ccr to ensure it is correct, while free is only uses atomic ccrs.
 *                                                   mjp
 ****/

init() { TOP = NULL; }

resource freelist1 (TOP) [ list(TOP) ]

cas(status,location;original,o,nw) [location==original] {
  if (location == o) {
    location = nw;
    status = 1;
  } else {
    status = 0;
  }
} [if ( original == o) then (location == nw) * (status == 1) else (status == 0) * (location == original)]

malloc1(i;) [emp] {
  local n,status,top,next;
  status=0;
  while (status == 0) [(if status == 0 then emp else i |->)] {
    with freelist1 when (true) {
        i = TOP;
    }

    if (i != NULL) {
      with freelist1 when (true) {
        if(TOP == i) {
          n = i->tl;
        } else {
          /* n = i->tl; Can't read as don't have permission need emp read rule */
        }
      }

      with freelist1 when (true) {
        /* Couldn't be bothered to write a DCAS instruction, so hacked a CAS one. */
```

```
        top = TOP;
        cas(status,top;top,i,n);
        if (status == 1) {
          next = i->tl;
          if(next == n) {
            TOP = top;
          } else {
            status = 0;
          }
        }
      }
    }
  }
} [i |->]
```

```
free1(;b) [b |-> ] {
  local t,status,top;
  status = 0;
  while (status == 0) [(if status==0 then b |-> else emp)] {
    with freelist1 when (true) {
      t = TOP;
    }
    b->tl = t;
    with freelist1 when (true) {
      cas(status, TOP;TOP, t, b);
    }
  }
} [emp]
```

B.1.13 Shape Property Versions of Interactive Examples

Many interactive examples can be solved automatically, if they are restricted to shape properties. Examples include mergesort, quicksort, copying an array and binary search. Binary search is special in so far as it requires the external SMT solver Yices for automatic verification.

Listing B.30: automatic/mergesort.sf

```
merge(r;p,q) [list(p) * list (q)] {
  local t;
  if (q == NULL) r = p;
  else if (p == NULL) r = q;
  else {
    if(q < p) {
      t = q; q = q->tl;
    } else {
      t = p; p = p->tl;
    }
    merge(r;p,q);
    t->tl = r; r = t;
  }
} [ list (r)]

split(r;p) [list(p)] {
  local t1,t2;
  if (p == NULL) r = NULL;
  else {
    t1 = p->tl;
    if (t1 == NULL) r = NULL;
    else {
      t2 = t1->tl; split(r;t2);
      p->tl = t2; t1->tl = r; r = t1;
    }
  }
} [ list (p) * list (r)]

mergesort(r;p) [list(p)] {
  local q,q1,p1;
  if (p == NULL) r = p;
  else {
    split(q;p);
    mergesort(q1;q);
    mergesort(p1;p);
    merge(r;p1,q1);
  }
} [ list (r)]
```

Listing B.31: automatic/parallel_mergesort.sf

```
merge(r;p,q) [list(p) * list (q)] {...} [list (r)]

split(r;p) [list(p)] {...} [list (p) * list (r)]

mergesort(r;p) [list(p)] {
  local q,q1,p1;
```

```
  if (p == NULL) r = p;
  else {
    split(q;p);
    mergesort(q1;q) || mergesort(p1;p);
    merge(r;p1,q1);
  }
} [ list (r)]
```

Listing B.32: automatic/quicksort-shape.dsf

```
quicksort(;b,e) [interval(b, e)] {
  local piv, l, r;
  if (e > b) {
    piv = b->dta;
    l = b + 1; r = e;
    while (l <= r) [b < l * l <= r + 1 * r <= e * interval (b,e)] {
      c = l->dta;
      if (c <= piv) {
        l = l + 1;
      } else {
        tmp1=l->dta; tmp2=r->dta; l->dta = tmp2; r->dta = tmp1;
        r = r - 1;
      }
    }
    tmp1=r->dta; tmp2=b->dta; r->dta = tmp2; b->dta = tmp1;
    quicksort (;b, r); quicksort (;l, e);
  }
} [ interval (b, e)]
```

Listing B.33: automatic/array_copy-shape.dsf

```
copy(r;a,n) [array(a,n)] {
   local i, tmp;
   i = 0;
   r = new(n);
   while (i < n) [array(a,n) * array(r,n)] {
      tmp = (a + i) -> dta;
      (r + i) -> dta = tmp;
      i = i + 1;
   }
} [array(a,n) * array(r, n)]
```

Listing B.34: automatic/binary_search-shape.dsf

```
binsearch(f;a,n,e) [array(a,n)] {
   local l, r, m, tmp;
   l = 0; r = n; f = 0;
   while ((f == 0) and (l < r)) [array(a,n) * (r <= n)] {
      block_spec [l < r] {
         m = l + ((r - l) / 2);
      } [l <= m * m < r]
      tmp = (a+m)->dta;
      if (tmp < e) { l = m+1; } else
         if (e < tmp) { r = m; } else { f = 1; }
   }
} [array(a,n)]
```

B.2 Interactive Examples

B.2.1 Tree Map

Applying a function on all data-nodes of a tree is a simple operation. Unfortunately, Holfoot can't verify it automatically, because by default, it does not know about the function `TREE_MAP`. The proof-script for this example consists of just calling Holfoot's automation with a suitable rewrite for `TREE_MAP`.

Listing B.35: interactive/tree_map.dsf

```
tree_map(;t) [data_tree(t, data)] {
  local i;
  if (t != NULL) {
    i = t->dta; i = i+1; t->dta = i;
    i = t->l; tree_map(;i);
    i = t->r; tree_map(;i);
 }
} [data_tree(t,``TREE_MAP (\l. [SUC (HD l)]) data``)]
```

Listing B.36: interactive/tree_map.hol

```
val file = concat [examplesDir, "/interactive/tree_map.dsf"];
val _ = holfoot_verify_spec file [ add_rewrites [ TREE_MAP_THM ] ];
```

B.2.2 Tree Depth

Determining the minimal and maximal depth of a binary tree is simple as well. Holfoot can verify this problem automatically, if the right rewrite rules are provided. This example is also used for showing the benefits of block-specifications and using HOL4-functions in expressions. Implementing `MIN` and `MAX` via conditional execution is appropriate, but leads to many case splits and therefore a slowdown during verification. Block-specifications can

be used to restrict these case-splits to the region of the program where they are really needed. Finally, the HOL4 functions are used to replace the conditional execution.

Listing B.37: interactive/tree_depth.dsf

```
tree_depth(r1,r2;t) [data_tree(t, data)] {
  local i, j, di1, di2, dj1, dj2;
  if (t == NULL) { r1 = 0; r2 = 0; } else {
    i = t->l; j = t->r;
    tree_depth(di1, di2; i); tree_depth(dj1, dj2; j);
    if (di1 < dj1) r1 = dj1 + 1; else r1 = di1 + 1;
    if (di2 < dj2) r2 = di2 + 1; else r2 = dj2 + 1;
  }
} [data_tree(t,data) * (r1 == ``MAX_DEPTH data``) * (r2 == ``MIN_DEPTH data``)]
```

Listing B.38: interactive/tree_depth.dsf2

```
tree_depth(r1,r2;t) [data_tree(t, data)] {
  local i, j, di1, di2, dj1, dj2;
  if (t == NULL) { r1 = 0; r2 = 0; } else {
    i = t->l; j = t->r;
    tree_depth(di1, di2; i);
    tree_depth(dj1, dj2; j);
    block_spec [] {
      if (di1 < dj1) r1 = dj1 + 1; else r1 = di1 + 1;
    } [r1 == ``(MAX di1 dj1) + 1``]
    block_spec [] {
      if (di2 < dj2) r2 = di2 + 1; else r2 = dj2 + 1;
    } [r2 == ``(MIN di2 dj2) + 1``]
  }
} [data_tree(t,data) * (r1 == ``MAX_DEPTH data``) * (r2 == ``MIN_DEPTH data``)]
```

Listing B.39: interactive/tree_depth-holexp.dsf2

```
tree_depth(r1,r2;t) [data_tree(t,data)] {
  local i, j, di1, di2, dj1, dj2;
  if (t == NULL) { r1 = 0; r2 = 0; } else {
    i = t->l; j = t->r;
    tree_depth(di1, di2; i); tree_depth(dj1, dj2; j);
    r1 = ``MAX di1 dj1`` + 1;
    r2 = ``MIN di2 dj2`` + 1;
  }
} [data_tree(t,data) * (r1 == ``MAX_DEPTH data``) * (r2 == ``MIN_DEPTH data``)]
```

Listing B.40: interactive/tree_depth.hol

```
val file = concat [examplesDir, "/interactive/tree_depth.dsf"];
val file2 = concat [examplesDir, "/interactive/tree_depth.dsf2"];
val file3 = concat [examplesDir, "/interactive/tree_depth-holexp.dsf2"];

val rewriteL = [MIN_MAX_DEPTH_THM, arithmeticTheory.MIN_DEF,
   MIN_MAX_LIST_THM, arithmeticTheory.MAX_DEF, MIN_MAX_DEPTH_THM];

val _ = holfoot_verify_spec file [ add_rewrites rewriteL ];
val _ = holfoot_verify_spec file2 [ add_rewrites rewriteL ];
val _ = holfoot_verify_spec file3 [ add_rewrites rewriteL ];
```

B.2.3 List Remove

Removing the first occurrence of an element is a simple algorithm on singly-linked lists. Many similar programs can be verified automatically. Remove needs user-interaction, though, because no `REMOVE` function is defined in the HOL4 list libraries. Holfoot allows defining `REMOVE` using HOL4's infrastructure:

```
val REMOVE_def = Define `
  (REMOVE x [] = []) /\
  (REMOVE x (y::ys) = if (x = y) then ys else (y::REMOVE x ys))`;
```

Then Holfoot's automation can prove the following specification automatically:

Listing B.41: interactive/remove.dsf

```
list_remove(l;x) [data_list(l,data)] {
  local v,t;
  if (l != NULL) {
    v = l->dta;
    if (v == x) {
      t = l; l = l->tl; dispose(t);
    } else {
      t = l->tl; list_remove(t;x); l->tl = t;
    }
  }
} [ data_list (l, ``REMOVE x data``)]
```

An iterative implementation is much more complicated and harder to verify. The necessary loop-invariant is lengthy and even a loop-specification is still complicated. However, Holfoot can verify the recursive as well as the interactive implementation automatically, if it is provided with the definition of `REMOVE` and some rewrite rules.

Listing B.42: interactive/remove-iter.dsf

```
list_remove(l;x) [data_list(l,data)] {
  local p, t, f, v;
  p = NULL; t = l; f = 0;
  while (t != NULL and (f == 0)) [
       data_list (t, _data2) *
      (if p == 0 then ((t == l) * ``_data2 = data``) else
        (data_lseg (l, _data1, p) * (p != t) *
         p |-> [tl:t,dta:_pdate] *
         ``~(MEM x data1) /\ ~(x = pdate) /\
         (data = _data1 ++ (_pdate::_data2))``)) *
       ``((~(t = 0)) /\ (~(f = 0))) ==> (HD data2 = x)``] {
    v = t->dta;
    if (v==x) {
      f = 1;
    } else {
      p = t; t = t->tl;
    }
  }
  if (t != NULL) {
     v = t->tl; dispose(t);
     if (p == NULL) {
        l = v;
     } else {
        p->tl = v;
     }
  }
} [ data_list (l, ``REMOVE x data``)]
```

Listing B.43: interactive/remove-iter-loopspec.dsf

```
list_remove(l;x) [data_list(l,data)] {
  local p, t, f, v;
  p = NULL; t = l; f = 0;
  loop_spec [ data_list (t, tdata) *
              ``((~( t = 0)) /\ (~(f = 0))) ==> (HD tdata = x)`` *
             (if p != 0 then (p |-> [tl:t,dta:#pdate]) else (t == l))] {
    while (t != NULL and (f == 0)) {
      v = t->dta;
      if (v==x) {
        f = 1;
      } else {
        p = t;
        t = t->tl;
      }
    }
    if (t != NULL) {
      v = t->tl;
      dispose(t);
      if (p == NULL) {
        l = v;
      } else {
        p->tl = v;
      }
    }
  } [if old(p) == 0 then data_list(l, ``REMOVE x tdata``) else
        ((l == old(l)) * data_list (old(p), ``pdate::(REMOVE x tdata)``))]
} [ data_list (l, ``REMOVE x data``)]
```

Listing B.44: interactive/remove.hol

```
(******************************************************************************)
(* New Definition for the Specification                                      *)
(******************************************************************************)
val REMOVE_def = Define `
   (REMOVE x [] = []) /\
   (REMOVE x (x'::xs) = if (x = x') then xs else (x'::REMOVE x xs))`;

val REMOVE_ID = prove (
``!x l. (REMOVE x l = l) = ~(MEM x l)``,
Induct_on `l` THEN
ASM_SIMP_TAC list_ss [REMOVE_def, COND_RAND, COND_RATOR]);

val REMOVE_APPEND = prove (
``!x l1 l2. REMOVE x (l1 ++ l2) =
   if (MEM x l1) then
      (REMOVE x l1) ++ l2
   else
      l1 ++ (REMOVE x l2)``,
Induct_on `l1` THEN
ASM_SIMP_TAC list_ss [REMOVE_def] THEN
REPEAT STRIP_TAC THEN
Cases_on `x = h` THEN ASM_SIMP_TAC std_ss [] THEN
ASM_SIMP_TAC list_ss [COND_RAND, COND_RATOR]);

(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/remove.dsf"];
val _ = holfoot_verify_spec file [ add_rewrites [REMOVE_def] ];

val file2 = concat [examplesDir, "/interactive/remove-iter.dsf"];
val _ = holfoot_verify_spec file2 [ add_rewrites [REMOVE_def, REMOVE_APPEND, REMOVE_ID] ];

val file3 = concat [examplesDir, "/interactive/remove-iter-loopspec.dsf"];
val _ = holfoot_verify_spec file3 [ add_rewrites [REMOVE_def] ];
```

B.2.4 Circular List

The circular list example can be verified automatically by Smallfoot. However, it can not automatically be verified by Holfoot. The reason is that Holfoot is less aggressive with case-splits and guessing instantiations. To verify the circular list example inside HOL4 a little bit of user-interaction is needed to perform the necessary case-splits and instantiations of existential quantifiers.

Listing B.45: interactive/circular-list.sf

```
push(r) [r|->_tf * lseg(_tf,r)] {
  local t, u;
  t = new();
  u = r->tl;
  t->tl = u;
  r->tl = t;
} [r|->_b * _b|->_tf * lseg(_tf,r)]

enqueue(r;) [r|->_tf * lseg(_tf,r)] {
  push(r);
  r = r->tl;
} [r|->_tf * lseg(_tf,_b) * _b|->r]

pop_dequeue(r) [r!=_tf * r|->_tf * lseg(_tf,r)] {
  local t, u;
  t = r->tl;
  u = t->tl;
  r->tl = u;
  dispose t;
} [r|->_b * lseg(_b,r)]

test(r;) [r|->_tf * lseg(_tf,r)] {
  push(r);
  pop_dequeue(r);
  enqueue(r;);
  pop_dequeue(r);
} [r|->_a * lseg(_a,r)]
```

Listing B.46: interactive/circular-list.hol

```
(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/circular_list.sf"];

val enqueue_TAC =
   HF_CONTINUE_TAC THEN REPEAT STRIP_TAC THEN
   Q.EXISTS_TAC `r_const'` THEN HF_CONTINUE_TAC;

val test_TAC =
   HF_SOLVE_TAC THEN REPEAT STRIP_TAC THEN
   Cases_on `b' = tf` THEN HF_SOLVE_TAC;

val _ = holfoot_tac_verify_spec file (SOME []) [
    ("enqueue", enqueue_TAC),
    ("test", test_TAC)];
```

B.2.5 List Filter

A recursive implementation of filtering a list can be verified automatically (see example B.1.7). An iterative implementation requires a complicated loop invariant. While Holfoot is able to verify a specification of an iterative implementation with a shape specification automatically, the verification of a fully-functional implementation needs user guidance. The user has to provide several instantiations of existential quantifiers. Using a loop-specification instead of an invariant simplifies the specification and its proof.

Listing B.47: interactive/filter.dsf

```
list_filter(l;x) [data_list(l,data)] {
  local y, z, e;
  y = l; z = NULL;
  while(y != NULL) [if (y == l) then
                       data_list (l,_data1) *
                       ``? data_fc . (EVERY (\n. n = x) data_fc) /\
                              (data:num list = data_fc ++ _data1)``
                    else
                      ( data_lseg (l ,`` FILTER (\n:num. ~(n = x)) _data1``,z) *
                      z |-> [tl:y, dta:_date] *
                      data_list(y,_data2) *
                      ``?data_fc. (EVERY (\n. n = x) data_fc) /\
                              (data:num list = _data1 ++ _date::(data_fc++_data2)) /\
                        (~(_date = x))``)] {
    e = y->dta;
    if (e == x) { /* need to remove y */
      if (y == l) { /* first link */
        l = y->tl; dispose y; y = l;
      } else { /* not first link */
        e = y->tl; z->tl = e; dispose y; y = z->tl;
      }
    } else { /* don't need to remove y */
      z = y; y = y->tl;
    }
  }
} [ data_list (l, ``FILTER (\n:num. ~(n = x)) data``)]
```

Listing B.48: interactive/filter.dsf2

```
list_filter(l;x) [data_list(l,data)] {
  local y, z, e;
  y = l;
  z = NULL;
  loop_spec [ data_list (y, data2) *
             ( if (y != l) then data_lseg (l, data, z) * (z |-> tl:y,dta:#zdata))] {
    while (y != NULL) {
      e = y->dta;
      if(e == x) { /* need to remove y */
        if(y == l) { /* first link */
          l = y->tl; dispose y; y = l;
        } else { /* not first link */
          e = y->tl; z->tl = e; dispose y; y = z->tl;
        }
      } else { /* don't need to remove y */
        z = y; y = y->tl;
      }
    }
  } [ if (old(y) == old(l)) then
        data_list (l, ``FILTER (\n. ~(n = x)) data2``)
      else
        ( data_list (l, ``data ++ [zdata] ++ (FILTER (\n. ~(n = x)) data2)``))]
} [ data_list (l, ``FILTER (\n. ~(n = x)) data``) ]
```

Listing B.49: interactive/filter.hol

```
(******************************************************************************)
(* Recursive implementation                                                   *)
(******************************************************************************)
val file_rec = concat [examplesDir, "/automatic/filter_rec.dsf"];
val _ = holfoot_auto_verify_spec file_rec;


(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/filter.dsf"];

val filter_TAC =
   xHF_SOLVE_TAC [ add_rewrites [NULL_EQ, FILTER_EQ_NIL] ] THEN
   REPEAT STRIP_TAC THENL [
      Q.EXISTS_TAC `data_fc` THEN
      HF_SOLVE_TAC,

      Q.EXISTS_TAC `data1` THEN
      HF_SOLVE_TAC,

      Q.EXISTS_TAC `data1 ++ [date] ++ data_fc` THEN
      xHF_SOLVE_TAC [add_rewrites [FILTER_APPEND, FILTER_EQ_NIL]],

      Q.EXISTS_TAC `[]` THEN
      HF_SOLVE_TAC THEN
      SIMP_TAC list_ss [GSYM RIGHT_EXISTS_AND_THM, GSYM LEFT_EXISTS_AND_THM,
        GSYM LEFT_FORALL_IMP_THM, FILTER_APPEND, NULL_EQ, FILTER_EQ_NIL] THEN
      xHF_SOLVE_TAC [add_rewrites[FILTER_EQ_NIL]]
   ];

val _ = holfoot_tac_verify_spec file NONE [("list_filter", filter_TAC)];

(******************************************************************************)
(* Using loop specs                                                           *)
(******************************************************************************)
val file2 = concat [examplesDir, "/interactive/filter.dsf2"];

val filter2_TAC =
   HF_SOLVE_TAC THEN
   REPEAT STRIP_TAC THEN
   Q.EXISTS_TAC `data++[zdata]` THEN
   xHF_SOLVE_TAC [no_case_splits];

val _ = holfoot_tac_verify_spec file2 NONE [("list_filter", filter2_TAC)]
```

B.2.6 List Rotating

Verifying a fully functional specification of rotating a list requires defining the ROTATE function and proving rewrite rules.

Listing B.50: interactive/rotate.dsf

```
list_replace_last(i;k) [data_list(i,data) * k |-> [tl:#n, dta:#d]] {
   local l;
   if (i == 0) {
     i = k;
   } else {
     l=i->tl; list_replace_last(l;k); i->tl=l;
  }
} [data_lseg(i,data,k) * k |-> [tl:#n, dta:#d]]

list_rotate(i;n) [data_list(i,data) * (i != 0)] {
  local k, c;
  c = 0;
  while (c < n) [data_list(i,''ROTATE c data'') * (i != 0) * (c <= n)] {
    k = i->tl; i->tl = 0;
    list_replace_last(k;i);
    i = k; c = c + 1;
  }
} [ data_list (i,''ROTATE n data'')]
```

Listing B.51: interactive/rotate.hol

```
(******************************************************************************)
(* Some definitions                                                          *)
(******************************************************************************)
val SINGLE_ROTATE_def = Define `
  (SINGLE_ROTATE [] = []) /\
  (SINGLE_ROTATE (x::xs) = SNOC x xs)`

val SINGLE_ROTATE_REWRITE = prove (
``0 < LENGTH l ==> (SINGLE_ROTATE l = SNOC (HD l) (TL l))``,
Cases_on `l` THEN SIMP_TAC list_ss [SINGLE_ROTATE_def]);

val ROTATE_def = Define `
   (ROTATE 0 l = l) /\
   (ROTATE (SUC n) l = SINGLE_ROTATE (ROTATE n l))`

val LENGTH_SINGLE_ROTATE = prove (
``LENGTH (SINGLE_ROTATE l) = LENGTH l``,
Cases_on `l` THEN SIMP_TAC list_ss [SINGLE_ROTATE_def])

val LENGTH_ROTATE = prove (
``LENGTH (ROTATE n l) = LENGTH l``,
Induct_on `n` THEN ASM_SIMP_TAC std_ss [ROTATE_def, LENGTH_SINGLE_ROTATE]);

val NULL_ROTATE = prove (
``NULL (ROTATE n l) = NULL l``,
SIMP_TAC std_ss [NULL_LENGTH, LENGTH_ROTATE]);
```

```
(***************************************************************************)
(* Verify specification                                                    *)
(***************************************************************************)
val file = concat [examplesDir, "/interactive/rotate.dsf"];
val list_rotate_TAC =
   xHF_CONTINUE_TAC [add_rewrites [ROTATE_def]] THEN
   REPEAT STRIP_TAC THENL [
   SIMP_TAC list_ss [NULL_DROP, LENGTH_ROTATE, LENGTH_TL,
      GSYM arithmeticTheory.ADD1, ROTATE_def,
      SINGLE_ROTATE_REWRITE, SNOC_APPEND,
      TAKE_APPEND1, FIRSTN_LENGTH_ID_EVAL,
      BUTFIRSTN_APPEND2] THEN
   Cases_on `i'_const' = 0` THEN HF_SOLVE_TAC,

   `c_lc = n_const` by DECIDE_TAC THEN HF_SOLVE_TAC
];

val _ = holfoot_tac_verify_spec file (SOME []) [("list_rotate", list_rotate_TAC)];
```

B.2.7 Factorial

The following example does not manipulate any dynamic data-structures. Instead it demonstrates the different possibilities of specifying a while-loop.

Listing B.52: interactive/fact.dsf

```
fact_recursive(r;n) {
  if (n > 1) {
    fact_recursive(r;n-1); r = r * n;
  } else {
    r = 1;
  }
} [r == ``FACT n``]

fact_invariant(r;n) {
  local i;
  r = 1; i = n;
  while (i > 1) [``r * FACT i = FACT n``] {
    r = r * i; i = i - 1;
  }
} [r == ``FACT n``]

fact_loopspec(r;n) {
  local i;
  r = 1; i = n;
  loop_spec [emp] {
    while (i > 1) {
      r = r * i; i = i - 1;
    }
  } [r == ``old(r) * FACT (old i)``]
} [r == ``FACT n``]

fact_invariant2 (r;n) {
  local i;
  r = 1; i = 1;
  while (i < n)
      [(r == ``FACT i``) *
        ``( i <= n) \/ (i = 1)``] {
    i = i + 1; r = r * i;
  }
} [r == ``FACT n``]

fact_loopspec2(r;n) {
  local i;
  r = 1; i = 1;
  loop_spec [(r == ``FACT i``) *
             ``( i <= n) \/ (i = 1)``] {
    while (i < n) {
      i = i + 1; r = r * i;
    }
  } [r == ``FACT n``]
} [r == ``FACT n``]

fact_invariant3 (r;n) {
  local i;
  r = 1; i = 1;
  while (i < n) [unroll 1]
      [r == ``FACT i`` * (i <= n)] {
    i = i + 1; r = r * i;
  }
} [r == ``FACT n``]

fact_loopspec3(r;n) {
  local i;
  r = 1; i = 1;
  loop_spec [ unroll 1]
      [(r == ``FACT i``) * (i <= n)] {
    while (i < n) {
      i = i + 1; r = r * i;
    }
  } [r == ``FACT n``]
} [r == ``FACT n``]
```

Listing B.53: interactive/fact.hol

```
val GREATER_1 = prove (``!n. (1 < n) ==> ?m. n = SUC m``,
  Cases_on `n` THEN SIMP_TAC arith_ss []);
val LESSEQ_1 = prove (``!n:num. (n <= 1) ==> ((n = 0) \/ (n = 1))``, DECIDE_TAC);
val FACT_DEF = CONJ FACT numeral_fact;

(******************************************************************************)
(* Verify specification                                                     *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/fact.dsf"];

val fact_TAC =
HF_CONTINUE_TAC THEN
HF_VC_TAC THEN
REPEAT STRIP_TAC THEN (
  MAP_EVERY IMP_RES_TAC [GREATER_1, LESSEQ_1] THEN
  FULL_SIMP_TAC arith_ss [FACT_DEF]
);

val fact2_TAC =
HF_CONTINUE_TAC THEN
SIMP_TAC arith_ss [FACT_DEF, GSYM ADD1] THEN
HF_CONTINUE_TAC THEN
HF_VC_TAC THEN
REPEAT STRIP_TAC THENL [
   METIS_TAC[LESS_EQUAL_ANTISYM],
   `(n_const = 1) \/ (n_const = 0)` by DECIDE_TAC THEN
   FULL_SIMP_TAC arith_ss [FACT_DEF]
];

val fact3_TAC =
HF_CONTINUE_TAC THEN
SIMP_TAC arith_ss [FACT_DEF, GSYM ADD1] THEN
HF_CONTINUE_TAC THEN
HF_VC_TAC THEN
REPEAT STRIP_TAC THENL [
   IMP_RES_TAC LESSEQ_1 THEN
   FULL_SIMP_TAC arith_ss [FACT_DEF],

   METIS_TAC[LESS_EQUAL_ANTISYM]
];

val thm = holfoot_tac_verify_spec file NONE
              [("fact_loopspec", fact_TAC),
               ("fact_invariant", fact_TAC),
               ("fact_recursive", fact_TAC),
               ("fact_invariant2", fact2_TAC),
               ("fact_loopspec2", fact2_TAC),
               ("fact_invariant3", fact3_TAC),
               ("fact_loopspec3", fact3_TAC)]
```

B.2.8 Tree Sum

Verifying a recursive implementation of summing all the nodes of a tree is easy. A function `TREE_SUM` is defined and a rewrite rule proven. With this rewrite rule, the recursive implementation can be automatically verified by Holfoot.

Listing B.54: interactive/tree_sum.dsf

```
tree_sum(r;t) [data_tree(t,data)] {
  local i;
  if (t == NULL) { r = 0; } else {
    r = t->dta;
    i = t->l; tree_sum(i;i); r = r + i;
    i = t->r; tree_sum(i;i); r = r + i;
  }
} [data_tree(t,data) * (r == ``TREE_SUM data``)]
```

However, an iterative implementation is much more interesting, because it needs to explicitly maintain a stack of all the parts of the tree that still need processing. It is tricky to reason about this stack. Most tools use complicated constructs like the magic-wand operator. Holfoot can avoid this by using a loop-specification:

Listing B.55: interactive/tree_sum-iter.dsf

```
assume pop(sp,r;) [w/r: sp,r;]
   [ data_list  (sp,``v::vs``)] [ data_list  (sp, vs) * (r == #v)]
assume push(sp;v) [w/r: sp;]
   [ data_list  (sp,data)] [ data_list  (sp,  ``v::data``)]

tree_sum_depth (r;t) [data_tree(t, data)] {
  local sp, c, i;
  r = 0;
  if (t != 0) {
    sp = 0; push(sp;t);
    loop_spec [ data_list (sp, trees) *  ``~(MEM 0 trees)`` *
                ``LENGTH trees_data = LENGTH trees`` *
               map (\t d. data_tree(t,d)) ``ZIP (trees, trees_data)``] {
      while (sp != 0) {
        pop(sp,c;);
        i = c->l; if (i != 0) push(sp;i);
        i = c->r; if (i != 0) push(sp;i);
        i = c->dta; r = r + i;
      }
    } [map (\t d. data_tree(t,d)) ``ZIP (trees, trees_data)`` *
       (r == ``old(r) + SUM (MAP TREE_SUM trees_data)``)]
  }
} [data_tree(t,data) * (r == ``TREE_SUM data``)]
```

Listing B.56: interactive/tree_sum.hol

```
(******************************************************************************)
(* Some useful REWRITES                                                       *)
(******************************************************************************)
val TREE_SUM_def = Define `TREE_SUM = TREE_FOLD (0:num, \v vL. (FOLDL (\a b. a + b) 0 ((HD v)::vL)))`

val TREE_SUM_REWRITE = prove (``
   (TREE_SUM leaf = 0) /\ (TREE_SUM (node v tL) = SUM ((HD v)::(MAP TREE_SUM tL)))``,
SIMP_TAC (std_ss++boolSimps.ETA_ss) [TREE_SUM_def, TREE_FOLD_def, SUM_FOLDL]);

(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/tree_sum.dsf"];
val _ = holfoot_verify_spec file [ add_rewrites [TREE_SUM_REWRITE] ];

val file2 = concat [examplesDir, "/interactive/tree_sum_iter.dsf"];
val tree_sum_depth_TAC =
   xHF_CONTINUE_TAC [add_rewrites [TREE_SUM_REWRITE, LENGTH_EQ_NUM_compute]] THEN
   REPEAT GEN_TAC THEN
   Cases_on `NULL trees` THEN1 HF_SOLVE_TAC THEN
   xHF_CONTINUE_TAC [use_asms, add_rewrites[LENGTH_EQ_ADD_CONST_compute]] THEN
   REPEAT STRIP_TAC THEN (
      Q.EXISTS_TAC `1'` THEN
      xHF_CONTINUE_TAC [use_asms, add_rewrites [TREE_SUM_REWRITE]]
   );
val _ = holfoot_tac_verify_spec file2 (SOME []) [("tree_sum_depth", tree_sum_depth_TAC)]
```

B.2.9 Array Increment

Another example illustrating loop-specifications is incrementing each element of an array. This is one of Eric Hehner's examples of *specified blocks* [16]. For comparison, there are three specifications: `inc1` uses a loop-invariant, `inc2` a loop-specification similar to the work of Hehner and `inc3` uses a loop-specification and exploits local reasoning.

Listing B.57: interactive/array-inc.dsf

```
inc1(;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   while (i < n) [data_array(a, n, _data2) *
         ``(! id . id < i ==>              (EL id data2 = SUC (EL id data))) /\
          (! id . i <= id /\ id < n ==> (EL id data2 = EL id data))``] {
      tmp = (a + i) -> dta; (a + i) -> dta = tmp + 1;
      i = i + 1;
   }
} [data_array(a,n,``MAP SUC data``)]

inc2(;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   loop_spec [data_array(a,n,data)] {
      while (i < n) {
         tmp = (a + i) -> dta; (a + i) -> dta = tmp + 1;
         i = i + 1;
      }
   } [data_array(a, n, _data2) *
       ``(! id . id < old(i) ==>              (EL id data2 = EL id data)) /\
        (! id . old(i) <= id /\ id < n ==> (EL id data2 = SUC (EL id data)))``]
} [data_array(a,n,``MAP SUC data``)]

inc3(;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   loop_spec [data_array(a+i,n-i,data)] {
      while (i < n) {
         tmp = (a + i) -> dta; (a + i) -> dta = tmp + 1;
         i = i + 1;
      }
   } [data_array(a+old(i), n-old(i), ``MAP SUC data``)]
} [data_array(a,n,``MAP SUC data``)]
```

Listing B.58: interactive/array-inc.hol

```
val inc1_TAC =
  HF_VC_SOLVE_TAC THEN HF_VC_TAC THEN
  REPEAT STRIP_TAC THENL [
     Cases_on `i_const = id` THEN ASM_SIMP_TAC arith_ss [],

     Q.EXISTS_TAC `data` THEN
     SIMP_TAC list_ss [] THEN
     REPEAT STRIP_TAC THEN
     MATCH_MP_TAC LIST_EQ THEN
     ASM_SIMP_TAC list_ss [EL_MAP]
  ];

val inc2_TAC =
  HF_SOLVE_TAC THEN HF_VC_TAC THEN
  REPEAT STRIP_TAC THENL [
     Cases_on `old_i = id` THEN ASM_SIMP_TAC arith_ss [],
     MATCH_MP_TAC LIST_EQ THEN ASM_SIMP_TAC list_ss [EL_MAP]
  ];

val inc3_TAC =
  HF_SOLVE_TAC THEN
  REPEAT STRIP_TAC THENL [
     `n_const - old_i = 0` by DECIDE_TAC THEN HF_SOLVE_TAC,
     Cases_on `data = []` THEN HF_SOLVE_TAC
  ];

val file = concat [examplesDir, "/interactive/array-inc.dsf"];
val _ = holfoot_tac_verify_spec file NONE
              [("inc1", inc1_TAC),
               ("inc2", inc2_TAC),
               ("inc3", inc3_TAC)];
```

B.2.10 Array Copy

This example program copies an array. While its shape-specification can be verified automatically, the fully-functional one needs a short interactive proof. The user needs to add some reasoning about lists and a manual case-split.

Listing B.59: interactive/array_copy-full.dsf

```
copy(r;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0;
   r = new(n) [dta];
   while (i < n) [data_array(a,n,data) * data_array(r,n,_data_new) * (i <= n) *
                  ``!x. x < i ==> (EL x data = EL x _data_new)``] {
      tmp = (a + i) -> dta; (r + i) -> dta = tmp;
      i = i + 1;
   }
} [data_array(a,n,data) * data_array(r, n, data)]
```

If a loop specification instead of an invariant is used, the specification becomes slightly simpler. However, the interactive effort increases, because the array-boundaries are now changing.

Listing B.60: interactive/array_copy-full-loopspec.dsf

```
copy(r;a,n) [data_array(a,n,data)] {
   local i, tmp;
   i = 0; r = new(n) [dta];
   loop_spec [(i == #ic) * data_array(a+#ic,n-#ic,data) *
              array(r+#ic,n-#ic)] {
      while (i < n) {
         tmp = (a + i) -> dta; (r + i) -> dta = tmp;
         i = i + 1;
      }
   } [data_array(a+#ic,n-#ic,data) *
      data_array(r+#ic,n-#ic,data)]
} [data_array(a,n,data) * data_array(r, n, data)]
```

Listing B.61: interactive/array_copy.hol

```
(****************************************************************************)
(* Just the shape works automatically                                      *)
(****************************************************************************)
val file = concat [examplesDir, "/automatic/array_copy-shape.dsf"];
val thm = holfoot_auto_verify_spec file;

(****************************************************************************)
(* Verify specification / Manual Case split and reasoning about list needed *)
(****************************************************************************)
val file = concat [examplesDir, "/interactive/array_copy-full.dsf"];

val copy_TAC =
HF_SOLVE_TAC THEN
HF_VC_TAC THEN
REPEAT STRIP_TAC THENL [
   Cases_on `x'' = ic` THEN ASM_REWRITE_TAC[] THEN
   FULL_SIMP_TAC arith_ss [],

   ASM_SIMP_TAC arith_ss [LIST_EQ_REWRITE]
];
val _ = holfoot_tac_verify_spec file NONE [("copy", copy_TAC)];


(****************************************************************************)
(* with loop-spec                                                           *)
(****************************************************************************)
val file2 = concat [examplesDir, "/interactive/array_copy-full-loopspec.dsf"];

val copy2_TAC =
HF_CONTINUE_TAC THEN
REPEAT STRIP_TAC THENL [
   `n_const - ic = 0` by DECIDE_TAC THEN
   ASM_REWRITE_TAC[] THEN
   HF_SOLVE_TAC,

   Cases_on `data` THEN1 HF_SOLVE_TAC THEN
   Cases_on `data_dta` THEN1 HF_SOLVE_TAC THEN
   SIMP_TAC list_ss [REPLACE_ELEMENT_compute] THEN
   HF_CONTINUE_TAC
];
val _ = holfoot_tac_verify_spec file2 NONE [("copy", copy2_TAC)];
```

B.2.11 Array Reverse

Another array-example is reversing the data-content of an array. Again using a loop specification leads to a simpler specification but increases the interactive effort needed.

Listing B.62: interactive/array_reverse.dsf

```
reverse2(;a,n) [data_array(a,n+1,data)] {
   local i, j, tmp_i, tmp_j;
   i = 0; j = n;
   while (i < j) [data_array(a, n+1, _data2) * (j == n − i) *
        ‘‘! x. x <= n ==> (EL x _data2 = EL (
             if i <= x /\ x <= j then x else (n−x)) data)‘‘] {
      tmp_i = (a + i) -> dta; tmp_j = (a + j) -> dta;
      (a + i) -> dta = tmp_j; (a + j) -> dta = tmp_i;
      i = i + 1; j = j - 1;
   }
} [data_array(a,n+1,‘‘REVERSE data‘‘)]

reverse5(;a,n) [data_array(a,n+1,data)] {
   local i, j, tmp_i, tmp_j;
   i = 0; j = n;
   loop_spec [ data_interval (a+i, a+j, data)] {
   while (i < j) {
      tmp_i = (a + i) -> dta; tmp_j = (a + j) -> dta;
      (a + i) -> dta = tmp_j; (a + j) -> dta = tmp_i;
      i = i + 1; j = j - 1;
   }} [ data_interval (a+old(i), a+old(j), ‘‘REVERSE data‘‘)]
} [data_array(a,n+1,‘‘REVERSE data‘‘)]
```

Listing B.63: interactive/array_reverse.hol

```
val file = concat [examplesDir, "/interactive/array_reverse.dsf"];

(* holfoot_set_goal_procedures file ["reverse2"] *)
val reverse2_TAC =
  HF_SOLVE_TAC THEN
  REPEAT STRIP_TAC THENL [
     HF_VC_TAC THEN
     ASM_SIMP_TAC (arith_ss++boolSimps.LIFT_COND_ss) [COND_EXPAND_IMP],

     Q.EXISTS_TAC ‘data‘ THEN
     HF_SOLVE_TAC THEN HF_VC_TAC THEN
     REPEAT STRIP_TAC THEN
     MATCH_MP_TAC LIST_EQ THEN
     ASM_SIMP_TAC list_ss [] THEN
     REPEAT STRIP_TAC THEN
     ‘PRE (n_const + 1   x) = n_const - x‘ by DECIDE_TAC THEN
     ASM_SIMP_TAC (list_ss++boolSimps.LIFT_COND_ss) [EL_REVERSE, COND_EXPAND_IMP] THEN
     REPEAT STRIP_TAC THEN
     ‘(n_const = 2*i_const) /\ (x = i_const)‘ by DECIDE_TAC THEN
     ASM_SIMP_TAC arith_ss []
  ];

(* holfoot_set_goal_procedures file ["reverse5"] *)
val reverse5_TAC =
  xHF_SOLVE_TAC [simple_prop_simps] THEN
  REPEAT STRIP_TAC THENL [
     HF_VC_TAC THEN
     ‘(LENGTH data = 0) \/ (LENGTH data = 1)‘ by DECIDE_TAC THEN
     FULL_SIMP_TAC list_ss [LENGTH_EQ_NUM_compute],

     HF_VC_TAC THEN
     ‘old_j    (old_i + 1) = LENGTH data - 2‘ by DECIDE_TAC THEN
     ‘old_j    old_i = LENGTH data - 1‘ by DECIDE_TAC THEN
     ASM_REWRITE_TAC[] THEN
     Q.PAT_ASSUM ‘MIN 1 X = 1‘ MP_TAC THEN
     REPEAT (POP_ASSUM (K ALL_TAC)) THEN
     SIMP_TAC arith_ss [MIN_EQ, BUTFIRSTN_BUTFIRSTN, LENGTH_REPLACE_ELEMENT,
        LENGTH_REVERSE] THEN
     CONSEQ_REWRITE_TAC ([LIST_EQ], [], []) THEN
     ASM_SIMP_TAC arith_ss [LENGTH_DROP, LENGTH_REVERSE, LENGTH_REPLACE_ELEMENT,
```

```
      EL_BUTFIRSTN, EL_REVERSE, LENGTH_TAKE, EL_REPLACE_ELEMENT,
      EL_FIRSTN, MIN_EQ] THEN
    REPEAT STRIP_TAC THENL [
      `PRE (1 - x) = 0` by DECIDE_TAC THEN
      ASM_SIMP_TAC list_ss [],

      AP_THM_TAC THEN AP_TERM_TAC THEN DECIDE_TAC,

      `x = 0` by DECIDE_TAC THEN
      ASM_SIMP_TAC arith_ss [COND_RAND, COND_RATOR] THEN
      REPEAT STRIP_TAC THEN
      `PRE (LENGTH data) = 0` by DECIDE_TAC THEN
      ASM_SIMP_TAC list_ss []
    ],

    HF_VC_TAC THEN
    FULL_SIMP_TAC std_ss [MIN_EQ, NULL_DROP] THEN
    `n_const + 1 = LENGTH data` by DECIDE_TAC THEN
    ASM_SIMP_TAC list_ss [FIRSTN_LENGTH_ID_EVAL]
  ];

val _ = holfoot_tac_verify_spec file NONE
   [("reverse2", reverse2_TAC), ("reverse5", reverse5_TAC)];
```

B.2.12 Binary Search

Binary search is used to demonstrate the benefits of using an external SMT solver. The shape specification can be verified automatically using Yices.

Listing B.64: automatic/binary_search-shape.dsf

```
binsearch(f;a,n,e) [array(a,n)] {
   local l, r, m, tmp;
   l = 0; r = n; f = 0;
   while ((f == 0) and (l < r)) [array(a,n) * (r <= n)] {
      block_spec [l < r] {
         m = l + ((r - l) / 2);
      } [l <= m * m < r]
      tmp = (a+m)->dta;
      if (tmp < e) { l = m+1; } else
         if (e < tmp) { r = m; } else { f = 1; }
   }
} [array(a,n)]
```

A fully functional specification needs interactive effort though.

Listing B.65: interactive/binary_search-full.dsf

```
binsearch(f;a,n,e) [data_array(a,n,data) * ``SORTED $<= data``] {
   local l, r, m, tmp;
   l = 0; r = n; f = 0;
   while ((f == 0) and (l < r)) [
         data_array(a,n,data) * (r <= n) *
         ``IS_BOOL_TO_NUM f /\ SORTED $<= data /\
           (MEM e data = ((f = 1) \/ (?i. l <= i /\ i < r /\ (EL i data = e))))``] {
      block_spec [l < r] {
         m = l + ((r - l) / 2);
      } [l <= m * m < r]
      tmp = (a+m)->dta;
      if (tmp < e) {
         l = m+1;
      } else if (e < tmp) {
         r = m;
      } else {
         f = 1;
      }
   }
} [data_array(a,n,data) * (f == ``BOOL_TO_NUM (MEM e data)``)]
```

Listing B.66: interactive/binary_search.hol

```
(******************************************************************************)
(* Just the shape works automatically (with Yices)                  *)
(******************************************************************************)
(*turn yices on*) set_trace "holfoot use Yices" 1;
val file = concat [examplesDir, "/automatic/binary_search-shape.dsf"];
val _ = holfoot_auto_verify_spec file;

(******************************************************************************)
(* Verify the fully functional spec now                             *)
(******************************************************************************)
val _ = set_trace "holfoot use Yices" 0; (*turn yices off again*)
val file_full = concat [examplesDir, "/interactive/binary_search-full.dsf"];

(* holfoot_set_goal file_full *)
val binsearch_full_TAC =
HF_SOLVE_TAC THEN
REPEAT STRIP_TAC THENL [
   HF_VC_TAC THEN YICES_TAC,

   HF_VC_TAC THEN
   DEPTH_CONSEQ_CONV_TAC (K EXISTS_EQ___CONSEQ_CONV) THEN
   SIMP_TAC (std_ss++EQUIV_EXTRACT_ss) [] THEN
   REPEAT STRIP_TAC THEN
   EQ_TAC THEN ASM_SIMP_TAC arith_ss [] THEN
   REPEAT STRIP_TAC THEN
   MP_TAC (Q.SPECL [`data'`, `m_const`, `i`] SORTED_EL_LESS_EQ) THEN
   ASM_SIMP_TAC arith_ss [],

   HF_VC_TAC THEN
   DEPTH_CONSEQ_CONV_TAC (K EXISTS_EQ___CONSEQ_CONV) THEN
   SIMP_TAC (std_ss++EQUIV_EXTRACT_ss) [] THEN
   REPEAT STRIP_TAC THEN
   EQ_TAC THEN ASM_SIMP_TAC arith_ss [] THEN
   REPEAT STRIP_TAC THEN
   MP_TAC (Q.SPECL [`data'`, `i`, `m_const`] SORTED_EL_LESS_EQ) THEN
   ASM_SIMP_TAC arith_ss [],

   HF_VC_TAC THEN
   Q.EXISTS_TAC `m_const` THEN
   ASM_SIMP_TAC arith_ss [],

   HF_VC_TAC THEN CONJ_TAC THEN1 METIS_TAC[MEM_EL] THEN
   HF_CONTINUE_TAC THEN HF_VC_TAC THEN
   REPEAT STRIP_TAC THEN (
      FULL_SIMP_TAC arith_ss [IS_BOOL_TO_NUM_def, BOOL_TO_NUM_REWRITE]
   ) THEN
```

```
  CCONTR_TAC THEN
  FULL_SIMP_TAC arith_ss []
];

val _ = holfoot_tac_verify_spec file_full NONE [("binsearch", binsearch_full_TAC)];
```

B.2.13 Mergesort

Mergesort is one of Smallfoot's examples. Its shape specification can be verified automatically (see Appx. B.1.13). Verifying a fully functional specification is also straightforward using HOL4 libraries for permutations and orders.

Listing B.67: interactive/mergesort.dsf

```
merge(r;p,q) [data_list(p,pdata) * data_list (q,qdata) *
             ``SORTED $<= pdata /\ SORTED $<= qdata``] {
  local t, q_date, p_date;
  if (q == NULL) { r = p; } else
  if (p == NULL) { r = q; } else {
    q_date = q->dta; p_date = p->dta;
    if (q_date < p_date) { t = q; q = q->tl; } else
                         { t = p; p = p->tl; }
    merge(r;p,q);
    t->tl = r; r = t;
  }
} [ data_list (r,_rdata) * ``(SORTED $<= _rdata) /\ (PERM (pdata ++ qdata) _rdata)``]

split(r;p) [data_list(p,data)] {
  local t1,t2;
  if (p == NULL) { r = NULL; } else {
    t1 = p->tl;
    if (t1 == NULL) { r = NULL; } else {
      t2 = t1->tl; split(r;t2);
      p->tl = t2; t1->tl = r; r = t1;
    }
  }
} [ data_list (p,_pdata) * data_list (r,_rdata) * ``PERM (_pdata ++ _rdata) data``]

mergesort(r;p) [data_list(p,data)] {
  local q,q1,p1;
  if (p == NULL) { r = p; } else {
    split(q;p);
    mergesort(q1;q); mergesort(p1;p);
    merge(r;p1,q1);
  }
} [ data_list (r,_rdata) * ``(SORTED $<= _rdata) /\ (PERM data _rdata)``]
```

Listing B.68: interactive/mergesort.hol

```
val file = concat [examplesDir, "/interactive/mergesort.dsf"];

val merge_TAC_0 =
HF_ELIM_COMMENTS_TAC THEN
REPEAT STRIP_TAC THENL [
   IMP_RES_TAC SORTED_CONS_IMP,

   FULL_SIMP_TAC arith_ss [SORTED_EQ, transitive_LE] THEN
   `!y. MEM y rdata = MEM y (pdata_h::(pdata_t ++ qdata_t))` by
      METIS_TAC[PERM_MEM_EQ] THEN
   ASM_SIMP_TAC list_ss [DISJ_IMP_THM, FORALL_AND_THM] THEN
   REPEAT STRIP_TAC THEN
   RES_TAC THEN
   DECIDE_TAC,

   IMP_RES_TAC SORTED_CONS_IMP,

   FULL_SIMP_TAC arith_ss [SORTED_EQ, transitive_LE] THEN
   `!y. MEM y rdata = MEM y (qdata_h::(pdata_t ++ qdata_t))` by
      METIS_TAC[PERM_MEM_EQ] THEN
   ASM_SIMP_TAC list_ss [DISJ_IMP_THM, FORALL_AND_THM] THEN
   REPEAT STRIP_TAC THEN
   RES_TAC THEN
   DECIDE_TAC
];

val mergesort_gen_step_opt = combined_gen_step_tac_opt [
    add_rewrites [SORTED_DEF, PERM_REFL],
    add_ssfrags [permLib.PERM_ss] ];

val merge_TAC =
   xHF_CONTINUE_TAC [mergesort_gen_step_opt, generate_vcs] THEN
   merge_TAC_0;

val _ = holfoot_tac_verify_spec file
              (SOME [careful, generate_vcs, mergesort_gen_step_opt])
              [("merge", merge_TAC)]
```

B.2.14 Insertion Sort

Another straightforward sorting example is insertion sort:

Listing B.69: interactive/insertionsort.dsf

```
min(m;i) [data_list(i,data)] {
  local ih, it;
  if (i == NULL) {} else {
    ih = i->dta;
    it = i->tl;
    if (ih < m) {
      m = ih;
    }
    min (m;it);
  }
} [ data_list (i,data) * ``( MEM m (old(m)::data)) /\ (EVERY (\n. m <= n) (old(m)::data))``]

delete(i,j;m) [data_list(i,data) * `` MEM m data``] {
  local ih, it;
  ih = i->dta; it = i->tl;
  if (ih == m) {
     j = i; i = it;
  } else {
     delete(it,j;m); i->tl = it;
  }
} [ data_list (i,``REMOVE m data``) * (j |-> dta:m)]

sortlist(i;) [data_list(i,data)] {
   local m, j;
```

```
   if (i == NULL) {} else {
      m = i->dta;
      min(m;i);
      delete(i,j;m);
      sortlist(i;);
      j->tl = i;
      i = j;
   }
} [ data_list (i, _idata) * ``(SORTED $<= _idata) /\ (PERM data _idata)``]
```

Listing B.70: interactive/insertionsort.hol

```
val SORTED_CONS_IMP = prove (``!R x xs. (SORTED R (x::xs) ==> SORTED R xs)``,
   Cases_on `xs` THEN SIMP_TAC list_ss [SORTED_DEF])

val transitive_LE = prove (``transitive (($<=): num -> num -> bool)``,
   SIMP_TAC arith_ss [relationTheory.transitive_def]);

val REMOVE_def = Define `
   (REMOVE x [] = []) /\
   (REMOVE x (x'::xs) = if (x = x') then xs else (x'::REMOVE x xs))`

val MEM_REMOVE_IMP = prove (
``!y x l. MEM y (REMOVE x l) ==> MEM y l``,
   Induct_on `l` THEN SIMP_TAC list_ss [REMOVE_def, COND_RAND, COND_RATOR] THEN METIS_TAC[]);


val PERM_REMOVE = prove (
``!x xs. MEM x xs ==> (PERM (x::REMOVE x xs) xs)``,
Induct_on `xs` THEN
SIMP_TAC list_ss [REMOVE_def] THEN
REPEAT GEN_TAC THEN
Cases_on `x = h` THEN (
   ASM_SIMP_TAC (std_ss++permLib.PERM_ss) []
));

(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/insertionsort.dsf"];

(* holfoot_set_goal_procedures file ["delete"] *)
val delete_TAC =
   HF_CONTINUE_TAC THEN REPEAT STRIP_TAC THEN
   Cases_on `i'_const = 0` THEN1 xHF_CONTINUE_TAC [use_asms] THEN
   xHF_CONTINUE_TAC [use_asms, add_rewrites [REMOVE_def], generate_vcs] THEN
   HF_VC_TAC THEN FULL_SIMP_TAC list_ss []

(* holfoot_set_goal_procedures file ["min"] *)
val min_TAC =
   HF_CONTINUE_TAC THEN HF_VC_TAC THEN
   REPEAT STRIP_TAC THEN ASM_SIMP_TAC arith_ss []

(* holfoot_set_goal_procedures file ["sortlist"] *)
val sortlist_TAC =
   xHF_CONTINUE_TAC [add_rewrites [SORTED_DEF, SORTED_DEF],
                    add_ssfrags [permLib.PERM_ss]] THEN
   xHF_SOLVE_TAC [generate_vcs,
       add_rewrites [SORTED_EQ, transitive_LE, EVERY_MEM, SORTED_DEF],
       add_ssfrags [permLib.PERM_ss]] THEN
   SIMP_TAC (std_ss++boolSimps.CONJ_ss) [GSYM FORALL_AND_THM] THEN
   REPEAT GEN_TAC THEN HF_ELIM_COMMENTS_TAC THEN
   Cases_on `m'_const = data_h` THEN ASM_SIMP_TAC (std_ss++permLib.PERM_ss) [REMOVE_def] THEN
   REPEAT STRIP_TAC THENL [
      METIS_TAC [PERM_MEM_EQ],

     `MEM y (data_h::REMOVE m'_const data_t)` by METIS_TAC [PERM_MEM_EQ] THEN
      FULL_SIMP_TAC list_ss [] THEN
      METIS_TAC[MEM_REMOVE_IMP],

      Q.PAT_ASSUM `PERM X Y` (ASSUME_TAC o ONCE_REWRITE_RULE [PERM_SYM]) THEN
      ASM_SIMP_TAC (std_ss++permLib.PERM_SIMPLE_ss) [] THEN
      METIS_TAC[PERM_REMOVE]
```

```
  ];

val _ = holfoot_tac_verify_spec file NONE
              [("sortlist", sortlist_TAC),
               ("delete",  delete_TAC),
               ("min",     min_TAC)];
```

B.2.15 Quicksort

Quicksort is harder to verify, because it operates on arrays. Again, there are two specifications, one using a loop-invariant and one using a loop-specification.

Listing B.71: interactive/quicksort-full.dsf

```
quicksort(;b,e) [data_interval(b, e, data)] {
  local piv, l, r;
  if (e > b) {
    piv = b->dta; l = b + 1; r = e;
    while (l <= r) [data_interval(b,e, _data) *
                    (b < l) * (l <= r + 1) * (r <= e) *
                    ``PERM org_data _data`` * ``HD org_data = HD _data`` *
                    ``!n. (0 < n) /\ (n < l - b) ==> (EL n _data <= piv)`` *
                    ``!n. (r - b < n) /\ (n <= e - b) ==> (piv < EL n _data)``] {
      c = l->dta;
      if (c <= piv) { l = l + 1; } else {
        tmp1=l->dta; tmp2=r->dta; l->dta = tmp2; r->dta = tmp1;
        r = r - 1;
      }
    }
    tmp1=r->dta; tmp2=b->dta; r->dta = tmp2; b->dta = tmp1;
    quicksort (;b, r);
    quicksort (;l, e);
  }
} [ data_interval (b, e, _rdata) *
    ``(SORTED $<= _rdata) /\ (PERM data _rdata)``]
```

Listing B.72: interactive/quicksort-full-loopspec.dsf

```
quicksort(;b,e) [data_interval(b, e, data)] {
  local piv, l, r;
  if (e > b) {
    piv = b->dta;
    l = b + 1; r = e;
    loop_spec [ data_interval (l,r,data) * (l <= r + 1)] {
      while (l <= r) {
        c = l->dta;
        if (c <= piv) { l = l + 1; } else {
          tmp1=l->dta; tmp2=r->dta; l->dta = tmp2; r->dta = tmp1;
          r = r - 1;
        }
      }
    } [ data_interval (old(l),old(r), _data2) *
        (l >= old(l)) * (r <= old(r)) * (l == r + 1) *
        ``PERM data data2 /\ (!n. (n < LENGTH data2) ==> ((piv < EL n data2) = (l - old(l) <= n)))``]
    assert [data_interval (b, e, data3)];
    tmp1=r->dta; tmp2=b->dta; r->dta = tmp2; b->dta = tmp1;
    quicksort (;b, r);
    quicksort (;l, e);
  }
} [ data_interval (b, e, _rdata) * ``(SORTED $<= _rdata) /\ (PERM data _rdata)``]
```

Listing B.73: interactive/quicksort.hol

```
val quicksort_opt = combined_gen_step_tac_opt [
    add_rewrites [SORTED_DEF, PERM_REFL],
    add_ssfrags [permLib.PERM_ss] ];
```

```
(****************************************************************************)
(* Verify specification / loop invariant                                    *)
(****************************************************************************)

val file2 = concat [examplesDir, "/interactive/quicksort-full.dsf"];

val quicksort_TAC =
xHF_SOLVE_TAC [quicksort_opt] THEN
REPEAT STRIP_TAC THENL [
   HF_VC_TAC THEN REPEAT STRIP_TAC THEN
   Cases_on `n = l_const - b_const` THEN ASM_SIMP_TAC arith_ss [],

   HF_VC_TAC THEN
   REWRITE_TAC [GSYM SWAP_ELEMENTS_def] THEN
   MATCH_MP_TAC (ONCE_REWRITE_RULE [PERM_SYM] PERM_SWAP_ELEMENTS) THEN
   DECIDE_TAC,

   Q.EXISTS_TAC `data` THEN
   xHF_CONTINUE_TAC [quicksort_opt] THEN
   REPEAT STRIP_TAC THEN
   Q.EXISTS_TAC `rdata ++ rdata'` THEN
   HF_SOLVE_TAC THEN HF_VC_TAC THEN
   Q.ABBREV_TAC `rdata_len = l_const - b_const'` THEN
   `rdata_len > 0 /\
    (r_const + 1 - b_const' = rdata_len) /\
    (r_const - b_const' = PRE rdata_len) /\
    (!n. ((r_const < b_const' + n /\ 0 < n) /\ n <= (e_const' - b_const')) =
         (PRE rdata_len < n /\ n < LENGTH data'))` by ALL_TAC THEN1 (
      UNABBREV_ALL_TAC THEN
      `LENGTH data' = LENGTH data` by METIS_TAC[PERM_LENGTH] THEN
      POP_ASSUM MP_TAC THEN
      ASM_SIMP_TAC arith_ss []
   ) THEN
   FULL_SIMP_TAC std_ss [GSYM SWAP_ELEMENTS_def, GSYM EL] THEN
   NTAC 3 (POP_ASSUM (K ALL_TAC)) THEN
   REPEAT STRIP_TAC THENL [
      MATCH_MP_TAC sortingTheory.SORTED_APPEND THEN
      FULL_SIMP_TAC arith_ss [relationTheory.transitive_def, EL] THEN
      REPEAT STRIP_TAC THEN
      Tactical.REVERSE (`(x <= HD data') /\ (HD data' < y)` by ALL_TAC) THEN1 (
        ASM_SIMP_TAC arith_ss []
      ) THEN
      `MEM y (DROP rdata_len (SWAP_ELEMENTS (PRE rdata_len) 0 data')) /\
       MEM x (TAKE rdata_len (SWAP_ELEMENTS (PRE rdata_len) 0 data'))` by
         METIS_TAC[PERM_MEM_EQ] THEN
      NTAC 2 (POP_ASSUM MP_TAC) THEN
      SIMP_TAC list_ss [MEM_EL,
           SWAP_ELEMENTS_def, REPLACE_ELEMENT_SEM, GSYM LEFT_FORALL_IMP_THM,
           EL_REPLACE_ELEMENT, EL_FIRSTN, EL_BUTFIRSTN] THEN
      REPEAT STRIP_TAC THENL [
         Cases_on `PRE rdata_len = 0` THEN1 (
            `n' = 0` by DECIDE_TAC THEN
            ASM_SIMP_TAC list_ss []
         ) THEN
         Cases_on `n' = 0` THEN1 ASM_SIMP_TAC arith_ss [] THEN
         ASM_SIMP_TAC arith_ss [COND_RAND, COND_RATOR],

         Q.PAT_ASSUM `!n. X n ==> (HD data < EL n data)` MATCH_MP_TAC THEN
         ASM_SIMP_TAC arith_ss []
      ],

      MAP_EVERY (fn x => Q.PAT_ASSUM (`PERM X` @ x) (ASSUME_TAC o
         ONCE_REWRITE_RULE [PERM_SYM])) [`data`, `rdata'`, `rdata`] THEN
      ASM_SIMP_TAC (std_ss++permLib.PERM_SIMPLE_ss) [] THEN
      ONCE_REWRITE_TAC [PERM_FUN_APPEND] THEN
      SIMP_TAC list_ss [] THEN
      MATCH_MP_TAC (ONCE_REWRITE_RULE [PERM_SYM] PERM_SWAP_ELEMENTS) THEN
      `LENGTH data' = LENGTH data` by
         METIS_TAC[PERM_LENGTH] THEN
      UNABBREV_ALL_TAC THEN
      ASM_SIMP_TAC arith_ss []
   ],
```

```
   HF_VC_TAC THEN
   Q.ABBREV_TAC `len = e_const' + 1 - b_const'` THEN
   `(len = 0) \/ (len = 1)` by (UNABBREV_ALL_TAC THEN DECIDE_TAC) THEN (
      FULL_SIMP_TAC std_ss [LENGTH_EQ_NUM_compute, SORTED_DEF]
   )
];

val _ = holfoot_tac_verify_spec file2 NONE [("quicksort", quicksort_TAC)];


(******************************************************************************)
(* Verify specification - loop spec                                         *)
(******************************************************************************)
val file3 = concat [examplesDir, "/interactive/quicksort-full-loopspec.dsf"];

val quicksort_loopspec_TAC =
xHF_SOLVE_TAC [quicksort_opt, no_expands, simple_prop_simps] THEN
REPEAT STRIP_TAC THENL [
   HF_SOLVE_TAC,

   Q.EXISTS_TAC `(HD data)::data2` THEN
   HF_SOLVE_TAC THEN HF_VC_TAC THEN
   Cases_on `data` THEN (
      FULL_SIMP_TAC list_ss [PERM_CONS_IFF]
   ) THEN
   REPEAT STRIP_TAC THENL [
     METIS_TAC[PERM_SYM],
     Cases_on `n` THEN FULL_SIMP_TAC list_ss []
   ],

   ASM_SIMP_TAC arith_ss [] THEN
   xHF_SOLVE_TAC [simple_prop_simps, no_expands, quicksort_opt] THEN
   Cases_on `old_l = 0` THEN1 xHF_SOLVE_TAC [quicksort_opt, simple_prop_simps] THEN
   Cases_on `old_l = old_r` THEN1 (
      xHF_SOLVE_TAC [quicksort_opt] THEN HF_VC_TAC THEN
      ASM_SIMP_TAC std_ss [GSYM EL, REPLACE_ELEMENT___REPLACE_ID, PERM_REFL]
   ) THEN
   `old_l < old_r` by DECIDE_TAC THEN FULL_SIMP_TAC arith_ss [] THEN
   REPEAT GEN_TAC THEN
   Q.EXISTS_TAC `data2 ++ DROP (old_r - old_l) (SWAP_ELEMENTS 0 (old_r - old_l) data)` THEN
   ASM_SIMP_TAC list_ss [SWAP_ELEMENTS_INTRO] THEN
   HF_SOLVE_TAC THEN
   HF_VC_TAC THEN REPEAT STRIP_TAC THENL [
       Q.PAT_ASSUM `PERM X data2` (ASSUME_TAC o ONCE_REWRITE_RULE [PERM_SYM]) THEN
       ASM_SIMP_TAC (std_ss++permLib.PERM_SIMPLE_ss) [] THEN
       MATCH_MP_TAC (ONCE_REWRITE_RULE [PERM_SYM] PERM_SWAP_ELEMENTS) THEN
       FULL_SIMP_TAC arith_ss [LENGTH_SWAP_ELEMENTS],

       Cases_on `n < LENGTH data2` THEN (
          FULL_SIMP_TAC list_ss [EL_APPEND1, EL_APPEND2, LENGTH_SWAP_ELEMENTS,
           EL_BUTFIRSTN, EL_SWAP_ELEMENTS, LENGTH_SWAP_ELEMENTS]
       )
   ],

   HF_SOLVE_TAC THEN
   REPEAT STRIP_TAC THEN
   Q.EXISTS_TAC `HD (data)::data2` THEN
   REPEAT STRIP_TAC THEN
   `?dh dtl. data = dh::dtl` by (Cases_on `data` THEN FULL_SIMP_TAC list_ss []) THEN
   FULL_SIMP_TAC list_ss [] THEN
   HF_SOLVE_TAC THEN
   REPEAT STRIP_TAC THEN
   Q.EXISTS_TAC `rdata ++ rdata'` THEN
   FULL_SIMP_TAC list_ss [SWAP_ELEMENTS_INTRO] THEN
   HF_SOLVE_TAC THEN
   STRIP_TAC THEN
   REPEAT (Q.PAT_ASSUM `LENGTH Y = X` (ASSUME_TAC o GSYM)) THEN
   `r_const    b_const' = PRE (LENGTH rdata)` by DECIDE_TAC THEN
   FULL_SIMP_TAC list_ss [MIN_EQ] THEN
   HF_VC_TAC THEN
   REPEAT STRIP_TAC THENL [
      MATCH_MP_TAC sortingTheory.SORTED_APPEND THEN
      FULL_SIMP_TAC arith_ss [relationTheory.transitive_def, EL] THEN
```

```
    REPEAT STRIP_TAC THEN
    Tactical.REVERSE (`~(dh < x) /\ (dh < y)` by ALL_TAC) THEN1 DECIDE_TAC THEN
    `MEM x (TAKE (LENGTH rdata) (SWAP_ELEMENTS (PRE (LENGTH rdata)) 0 (dh::data2))) /\
     MEM y (DROP (LENGTH rdata) (SWAP_ELEMENTS (PRE (LENGTH rdata)) 0 (dh::data2)))` by
        METIS_TAC[PERM_MEM_EQ] THEN
    NTAC 2 (POP_ASSUM MP_TAC) THEN
    Q.SUBGOAL_THEN `LENGTH rdata <= SUC (LENGTH data2)` MP_TAC THEN1 DECIDE_TAC THEN
    SIMP_TAC list_ss [MEM_EL, NOT_LESS, GSYM LEFT_FORALL_IMP_THM,
         SWAP_ELEMENTS_def, REPLACE_ELEMENT_SEM, LENGTH_REPLACE_ELEMENT,
         EL_FIRSTN, EL_BUTFIRSTN] THEN
    REPEAT STRIP_TAC THENL [
       Cases_on `PRE (LENGTH rdata)` THEN1 ASM_SIMP_TAC list_ss [] THEN
       Cases_on `n` THEN ASM_SIMP_TAC list_ss [GSYM NOT_LESS] THEN
       ASM_SIMP_TAC arith_ss [COND_RAND, COND_RATOR],

       Cases_on `n' + LENGTH rdata` THEN ASM_SIMP_TAC list_ss []
    ],

    ASM_SIMP_TAC (std_ss++permLib.PERM_SIMPLE_ss) [] THEN
    MATCH_MP_TAC PERM_SWAP_ELEMENTS THEN
    ASM_SIMP_TAC list_ss []
  ],

  HF_SOLVE_TAC THEN HF_VC_TAC THEN
  REPEAT STRIP_TAC THEN
  Q.ABBREV_TAC `len = e_const' + 1 - b_const'` THEN
  `(len = 0) \/ (len = 1)` by (UNABBREV_ALL_TAC THEN DECIDE_TAC) THEN (
    FULL_SIMP_TAC std_ss [LENGTH_EQ_NUM_compute, SORTED_DEF]
  )
]

val _ = holfoot_tac_verify_spec file3 NONE [("quicksort", quicksort_loopspec_TAC)]
```

B.2.16 Binary Search Tree

This is an example about binary search trees seen as sets. It contains procedures for membership test, inserting elements and deleting elements as well as a procedure for copying the content of a search tree into a sorted singly-linked list.

Listing B.74: interactive/binary_search_tree.dsf

```
search_tree_init(r;) {
  r = 0;
} [data_tree(r,_data) * ``BIN_SEARCH_TREE_SET data EMPTY``]

search_tree_insert(t;k) [data_tree(t,data) * ``BIN_SEARCH_TREE_SET data keys``] {
  local k0, tt;
  if (t == NULL) {
    t = new(); t->l = 0; t->r = 0; t->dta = k;
  } else {
    k0 = t->dta;
    if (k0 == k) { } else {
      if (k < k0) {
        tt = t->l;
        search_tree_insert(tt;k);
        t->l = tt;
      } else {
        tt = t->r;
        search_tree_insert(tt;k);
        t->r = tt;
      }
    }
  }
} [data_tree(t,_data) * ``BIN_SEARCH_TREE_SET data (k INSERT keys)``]

search_tree_delete_min (t,m;) [data_tree(t,data) *
      ``BIN_SEARCH_TREE_SET data keys /\ ~(keys = EMPTY)``] {
  local tt;
  tt = t->l;
```

```
   if (tt != NULL) {
      search_tree_delete_min (tt,m;);
      t->l = tt;
   } else {
      m = t->dta; tt = t->r; dispose (t); t = tt;
   }
} [data_tree (t, _data) * (m == _mk) *
    ''BIN_SEARCH_TREE_SET data (keys DELETE mk) /\
      (mk IN keys) /\ (!k. k IN keys ==> mk <= k)'']

search_tree_delete(t;k) [data_tree(t,data) * ''BIN_SEARCH_TREE_SET data keys''] {
   local k0, tt_l, tt_r;
   if (t == NULL) { } else {
      k0 = t->dta; tt_l = t->l; tt_r = t->r;
      if (k < k0) {
         search_tree_delete(tt_l;k);
         t->l = tt_l;
      } else if (k > k0) {
         search_tree_delete(tt_r;k);
         t->r = tt_r;
      } else {
        if (tt_l == 0) {
           dispose(t); t = tt_r;
        } else if (tt_r == 0) {
           dispose(t); t = tt_l;
        } else {
           search_tree_delete_min(tt_r,k0;);
           t->dta = k0; t->r = tt_r;
        }
      }
   }
} [data_tree (t, _data) * ''BIN_SEARCH_TREE_SET _data (keys DELETE k)'']

search_tree_lookup(r;t,k) [data_tree(t,data) * ''BIN_SEARCH_TREE_SET data keys''] {
   local k0, tt;
   if (t == NULL) { r = 0; } else {
      k0 = t->dta;
      if (k == k0) { r = 1; } else
         if (k < k0) { tt = t->l; search_tree_lookup (r;tt,k); } else
                     { tt = t->r; search_tree_lookup (r;tt,k); }
   }
} [data_tree (t, data) * ''BIN_SEARCH_TREE_SET data keys'' *
   (r == ''BOOL_TO_NUM (k IN keys)'')]

search_tree_to_list___rec (r;t) [data_tree(t,data_t) *
      data_list (r, data_l) * ''BIN_SEARCH_TREE_SET data_t keys''] {
   local n, tt;
   if (t == NULL) { } else {
      tt = t->r; search_tree_to_list___rec (r;tt);
      n = new(); n->tl = r; r = n;
      tt = t->dta; n->dta = tt;
      tt = t->l; search_tree_to_list___rec (r;tt);
   }
} [data_tree (t, data_t) * data_list (r,'' _data_lt ++ data_l'') *
    ''( BIN_SEARCH_TREE_SET data_t keys) /\ (LIST_TO_SET data_lt = keys) /\
      (SORTED $< data_lt)'']

search_tree_to_list (r;t) [data_tree(t,data_t) *
      ''BIN_SEARCH_TREE_SET data_t keys''] {
   r = 0; search_tree_to_list___rec (r;t);
} [data_tree (t, data_t) * data_list (r, _data_lt ) *
    ''( BIN_SEARCH_TREE_SET data_t keys) /\ (LIST_TO_SET data_lt = keys) /\
      (SORTED $< data_lt)'']
```

For lookup and deleting the minimal node, there are iterative versions as well:

Listing B.75: interactive/binary_search_tree.dsf2

```
search_tree_delete_min (t,m;) [data_tree(t,data) *
      ''BIN_SEARCH_TREE_SET data keys /\ ~(keys = EMPTY)''] {
   local tt, pp, p;
   p = t->l;
   if (p == 0) {
```

```
      m = t->dta; tt = t->r;
      dispose (t); t = tt;
   } else {
      pp = t; tt = p->l;
      loop_spec [
            (pp |-> [l:p,  r:#rc2,dta:#dc2]) *
            (p  |-> [l:tt,r:#rc, dta:#dc]) * (pp == #ppc) *
            data_tree(tt ,data_l) * data_tree(#rc,data_r) *
            ``BIN_SEARCH_TREE_SET (node [dc] [data_l;data_r]) keys``] {
         while (tt != NULL) {
            pp = p; p = tt; tt = p->l;
         }
         m = p->dta; tt = p->r; dispose (p); pp->l = tt;
      } [(m == _mk) * (#ppc |-> [l:_new_p,r:#rc2,dta:#dc2]) *
         data_tree(_new_p,_data) *
         ``BIN_SEARCH_TREE_SET _data (keys DELETE _mk) /\
          (_mk IN keys) /\ (!k. k IN keys ==> _mk <= k)``]
   }
} [data_tree(t,_data) * (m == _mk) *
    ``BIN_SEARCH_TREE_SET data (keys DELETE mk) /\
      (mk IN keys) /\ (!k. k IN keys ==> mk <= k)``]

search_tree_lookup(r;t,k) [data_tree(t,data) * ``BIN_SEARCH_TREE_SET data keys``] {
   local k0, tt;
   tt = t; r = 0;
   loop_spec [(k == #kv) * (r == #rc) * (tt == #tc) *
               data_tree(tt,data) * ``BIN_SEARCH_TREE_SET data keys /\ (rc IN {0;1:num})``] {
     while (``~(tt = 0) /\ (r = 0)``) {
       k0 = tt->dta;
       if (k == k0) { r = 1; } else
          if (k < k0) { tt = tt->l; } else { tt = tt->r; }
     }
   } [(k == #kv) * data_tree(#tc,data) *
       (r == ``BOOL_TO_NUM ((rc = 1:num) \/ (kv IN keys))``)]
} [data_tree(t,data) * ``BIN_SEARCH_TREE_SET data keys`` *
   (r == ``BOOL_TO_NUM (k IN keys)``)]
```

Due to the large number of procedures, the proof-script is lengthy. Therefore, just an excerpt is shown here:

Listing B.76: interactive/binary_search_tree.hol

```
(******************************************************************************)
(* Definitions of search trees                                               *)
(******************************************************************************)
val BIN_SEARCH_TREE_SET_def = Define
  `(BIN_SEARCH_TREE_SET leaf keys = (keys = EMPTY)) /\
   (BIN_SEARCH_TREE_SET (node [k] [t1; t2]) keys =
       ?k1 k2. (keys = k INSERT (k1 UNION k2)) /\
               (!k':num. k' IN k1 ==> k' < k) /\
               (!k':num. k' IN k2 ==> k' > k) /\
               (BIN_SEARCH_TREE_SET t1 k1) /\
               (BIN_SEARCH_TREE_SET t2 k2)) /\
   (BIN_SEARCH_TREE_SET _ _ = F)`;
...

(******************************************************************************)
(* Verify specification / here just insert                                   *)
(******************************************************************************)
val file = concat [examplesDir, "/interactive/binary_search_tree.dsf"];

val search_tree_insert_TAC =
   (* search_tree_insert *)
   xHF_CONTINUE_TAC [generate_vcs] THEN
   REPEAT STRIP_TAC THENL [
      HF_ELIM_COMMENTS_TAC THEN
      FULL_SIMP_TAC std_ss [BIN_SEARCH_TREE_SET_BIN_THM, IS_LEAF_REWRITE, UNION_EMPTY, NOT_IN_EMPTY],

      HF_ELIM_COMMENTS_TAC THEN
      Tactical.REVERSE (`k_const' IN keys` by ALL_TAC) THEN1 (
         `k_const' INSERT keys = keys` by ALL_TAC THEN1 (
            ASM_SIMP_TAC (std_ss++boolSimps.EQUIV_EXTRACT_ss) [EXTENSION, IN_INSERT]
         ) THEN
         FULL_SIMP_TAC std_ss []
      ) THEN
      FULL_SIMP_TAC std_ss [BIN_SEARCH_TREE_SET_BIN_THM, IN_INSERT],

      HF_ELIM_COMMENTS_TAC THEN
      FULL_SIMP_TAC std_ss [BIN_SEARCH_TREE_SET_BIN_THM] THEN
      Q.EXISTS_TAC `k1` THEN
      ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC THEN
      Q.EXISTS_TAC `k_const' INSERT k1` THEN Q.EXISTS_TAC `k2` THEN
      ASM_SIMP_TAC (std_ss ++ boolSimps.EQUIV_EXTRACT_ss) [IN_INSERT, EXTENSION, IN_UNION, DISJ_IMP_THM],

      HF_ELIM_COMMENTS_TAC THEN
      FULL_SIMP_TAC std_ss [BIN_SEARCH_TREE_SET_BIN_THM] THEN
      Q.EXISTS_TAC `k2` THEN
      ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC THEN
      Q.EXISTS_TAC `k1` THEN Q.EXISTS_TAC `k_const' INSERT k2` THEN
      ASM_SIMP_TAC (std_ss ++ boolSimps.EQUIV_EXTRACT_ss) [IN_INSERT, EXTENSION, IN_UNION, DISJ_IMP_THM] THEN
      DECIDE_TAC
   ];
...
```

B.2.17 Red-Black Tree

The red-black tree example is similar to the binary search tree example. First a predicate is defined that captures the relation of the abstract data structure with the representation in the heap. In comparison to the binary search tree example, the algorithm for inserting an element into a red-black tree is much more complicated, because the invariants of the data structure need to be maintained.

Listing B.77: interactive/red_black_tree.dsf

```
rb_tree_init(r;) {
```

```
   r = 0;
} [data_tree(r,[k,v,c]:_data) * ``RED_BLACK_TREE _data FEMPTY``]

rb_tree_mk_node (r; k, v) {
   r = new(); r->k = k; r->v = v; r->c = 1; r->l = 0; r->r = 0;
} [data_tree(r,[k,v,c]:``PROGRAM_FUN___mk_node k v``)]

rb_tree_is_red (r;t) [data_tree(t,[k,v,c]:data)] {
   local x;
   if (t == 0) { r = 0; } else {
     x = t->c;
     if (x == 1) { r = 1; } else { r = 0; }
   }
} [data_tree(t,[k,v,c]:data) * (r == ``BOOL_TO_NUM (RED_BLACK_TREE___IS_RED data)``)]

rb_tree_left_rotate (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_left_rotate data``] {
   local s, x;
   s = r->r; x = s->l; r->r = x; s->l = r; r->c = 1; s->c = 0; r = s;
} [data_tree(r,[k,v,c]:``PROGRAM_FUN___left_rotate data``)]

rb_tree_left_double_rotate (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_left_double_rotate data``] {
   local x;
   x = r->r;
   rb_tree_right_rotate (x;);
   r->r = x;
   rb_tree_left_rotate (r;);
}  [data_tree(r,[k,v,c]:``PROGRAM_FUN___left_double_rotate data``)]

rb_tree_right_rotate (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_right_rotate data``] {
   local s, x;
   s = r->l; x = s->r; r->l = x; s->r = r; r->c = 1; s->c = 0; r = s;
} [data_tree(r,[k,v,c]:``PROGRAM_FUN___right_rotate data``)]

rb_tree_right_double_rotate (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_right_double_rotate data``] {
   local x;
   x = r->l;
   rb_tree_left_rotate (x;);
   r->l = x;
   rb_tree_right_rotate (r;);
}  [data_tree(r,[k,v,c]:``PROGRAM_FUN___right_double_rotate data``)]

rb_tree_color_flip (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_color_flip data``] {
   local x;
   r->c = 1;
   x = r->l; x->c = 0;
   x = r->r; x->c = 0;
} [data_tree(r,[k,v,c]:``PROGRAM_FUN___color_flip data``)]
```

```
rb_tree_left_balance (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_left_balance data``] {
   local rl, rr, x, y;
   x  = r->c;
   if (x == 0) {
      rl = r->l;
      rr = r->r;
      rb_tree_is_red (x; rl);
      if (x == 1) {
         rb_tree_is_red (x; rr);
         if (x == 1) {
            rb_tree_color_flip (r;);
         } else {
            y = rl->l;
            rb_tree_is_red (x; y);
            if (x == 1) {
               rb_tree_right_rotate (r;);
            } else {
               y = rl->r;
               rb_tree_is_red (x; y);
               if (x == 1) {
                  rb_tree_right_double_rotate (r;);
               }
            }
         }
      }
   }
} [data_tree (r ,[k,v,c]:``PROGRAM_FUN___left_balance data``)]

rb_tree_right_balance (r;) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_right_balance data``] {
   local rl, rr, x, y;
   x  = r->c;
   if (x == 0) {
      rl = r->l;
      rr = r->r;
      rb_tree_is_red (x; rr);
      if (x == 1) {
         rb_tree_is_red (x; rl);
         if (x == 1) {
            rb_tree_color_flip (r;);
         } else {
            y = rr->r;
            rb_tree_is_red (x; y);
            if (x == 1) {
               rb_tree_left_rotate (r;);
            } else {
               y = rr->l;
               rb_tree_is_red (x; y);
               if (x == 1) {
                  rb_tree_left_double_rotate (r;);
               }
           }
         }
      }
  }
} [data_tree (r ,[k,v,c]:``PROGRAM_FUN___right_balance data``)]
```

```
rb_tree_insert_r (r; k, v) [data_tree(r,[k,v,c]:data) * ``PROGRAM_PRED___can_insert_r data``] {
   local rk, rl, rr, rlc, rrc;
   if (r == NULL) {
     rb_tree_mk_node (r;k,v);
   } else {
      rk = r->k;
      if (rk == k) {
        r->v = v;
      } else {
        rl = r->l;
        rr = r->r;
        if (k < rk) {
           rb_tree_insert_r (rl;k,v);
           r->l = rl;
           rb_tree_left_balance (r;);
        } else {
           rb_tree_insert_r (rr;k,v);
           r->r = rr;
           rb_tree_right_balance (r;);
        }
      }
   }
} [data_tree(r,[k,v,c]:``PROGRAM_FUN___insert_r data k v``)]

rb_tree_insert (r; k, v) [data_tree(r,[k,v,c]:data) * ``RED_BLACK_TREE data f``] {
   rb_tree_insert_r (r; k, v);
   r->c = 0;
} [data_tree(r,[k,v,c]:_data) * ``RED_BLACK_TREE _data (f |+ (k,v))``]

rb_tree_lookup(r,v;t,k) [data_tree(t,[k,v,c]:data) * ``BIN_SEARCH_TREE data f`` * (v == #vc)] {
   local k0, tt;
   if (t == NULL) {
     r = 0;
   } else {
      k0 = t->k;
      if (k == k0) {
         r = 1;
         v = t->v;
      } else if (k < k0) {
         tt = t->l;
         rb_tree_lookup (r,v;tt,k);
      } else {
         tt = t->r;
         rb_tree_lookup (r,v;tt,k);
      }
   }
} [data_tree(t,[k,v,c]:data) *
   (r == ``BOOL_TO_NUM ((k:num) IN FDOM (f:num |-> num))``) *
   (v == ``if (k IN FDOM (f:num |-> num)) then f ' k else vc``)]
```

The proof of this specification introduces specific predicates for auxiliary procedures. This translates the algorithm implemented in Holfoot's imperative programming language into a functional representation inside HOL4. Verifying the correctness of this translation is straightforward. Proving the correctness of the functional representation is, however, complicated. Thanks to the translation into a functional representation, these complicated, lengthy proofs are, however, only concerned with the essence of the algorithm. Low level implementation details are handled automatically during the translation.

Listing B.78: interactive/red_black_tree.hol

```
(******************************************************************************)
(* First, get the necessary HOL definitions in place                          *)
(******************************************************************************)
val BIN_SEARCH_TREE_def = Define
  `(BIN_SEARCH_TREE leaf f = (f = FEMPTY)) /\
   (BIN_SEARCH_TREE (node [k;v;c] [t1; t2]) f =
      ?f1 f2. (f = (FUNION f1 f2)|+(k,v)) /\
              (!k':num. k' IN FDOM f1 ==> k' < k) /\
              (!k':num. k' IN FDOM f2 ==> k' > k) /\
              (BIN_SEARCH_TREE t1 f1) /\
```

```
            (BIN_SEARCH_TREE t2 f2)) /\
   (BIN_SEARCH_TREE _ _ = F)`;

val RED_BLACK_TREE___IS_BLACK_def = Define `
   (RED_BLACK_TREE___IS_BLACK leaf = T) /\
   (RED_BLACK_TREE___IS_BLACK (node [k;v;c] [t1;t2]) = (c = 0:num)) /\
   (RED_BLACK_TREE___IS_BLACK _ = F)`;

val RED_BLACK_TREE___IS_RED_def = Define `
   (RED_BLACK_TREE___IS_RED leaf = F) /\
   (RED_BLACK_TREE___IS_RED (node [k;v;c] [t1;t2]) = (c=1:num)) /\
   (RED_BLACK_TREE___IS_RED _ = F)`;
...

val RED_BLACK_TREE___PROP_NO_RED_RED_def =
Define `RED_BLACK_TREE___PROP_NO_RED_RED t =
!t' t''. (t' IN SUBTREES t /\ t'' IN DIRECT_SUBTREES t') ==>
    ~(RED_BLACK_TREE___IS_RED t' /\ RED_BLACK_TREE___IS_RED t'')`

val RED_BLACK_TREE___PROP_BLACK_BALANCED_def =
Define `RED_BLACK_TREE___PROP_BLACK_BALANCED n t =
!p. p IN TREE_PATHS t ==> (LENGTH (FILTER (\t. (EL 2 t) = 0:num) p) = n)`
...

val IS_RED_BLACK_TREE_NODE_THM = prove (
``IS_RED_BLACK_TREE_NODE t =
  ?k v c t1 t2. (t = node [k;v;c] [t1;t2]) /\ (c IN {0;1})``,
Cases_on `t` THEN
SIMP_TAC (list_ss++CONJ_ss) [IS_RED_BLACK_TREE_NODE_def, tree_distinct,
   tree_11, LENGTH_EQ_NUM_compute, GSYM RIGHT_EXISTS_AND_THM,
   GSYM LEFT_EXISTS_AND_THM] THEN
METIS_TAC[]);

val RED_BLACK_TREE___NODES_OK_def =
Define `RED_BLACK_TREE___NODES_OK t =
!t'. t' IN SUBTREES t /\ ~(IS_LEAF t') ==> IS_RED_BLACK_TREE_NODE t'`;

val RED_BLACK_TREE_def = Define `
   (* a red-black tree representing the finite map f, *)
   RED_BLACK_TREE t f =
   ((* is a binary search tree containing f *)
    (BIN_SEARCH_TREE t f) /\
    (* has well-formed nodes all containing key, value and color *)
    (RED_BLACK_TREE___NODES_OK t) /\
    (* has a black root *)
    (RED_BLACK_TREE___IS_BLACK t) /\
    (* no red node has a red child *)
    (RED_BLACK_TREE___PROP_NO_RED_RED t) /\
    (* all paths through the tree have the same number of black nodes (n ones) *)
    (?n. RED_BLACK_TREE___PROP_BLACK_BALANCED n t))`
...

(* ----------------------------------------------------------------------- *)
(* Predicates and functions that describe exactly the behavior of the code *)
(* with the tactics that proof the correspondence with the code       *)
(* ----------------------------------------------------------------------- *)
val file = concat [examplesDir, "/interactive/red_black_tree.dsf"];

...

(* holfoot_set_goal_procedures file ["rb_tree_is_red"] *)
val rb_tree_is_red_TAC =
   xHF_CONTINUE_TAC [add_rewrites [RED_BLACK_TREE___IS_RED_BLACK___REWRITE] ]

(* ----------------------------------------------------------------------- *)

val PROGRAM_PRED___can_left_rotate_def = Define
  `PROGRAM_PRED___can_left_rotate t =
     IS_RED_BLACK_TREE_NODE t /\
     IS_RED_BLACK_TREE_NODE (RED_BLACK_TREE___RIGHT_SUBTREE t)`

val PROGRAM_PRED___can_left_rotate___REWRITE = prove (
``PROGRAM_PRED___can_left_rotate t =
```

```
  ?k v c k2 v2 c2 t1 t2 t3. (t = node [k;v;c] [t1; node [k2;v2;c2] [t2;t3]]) /\
     (c IN {0;1}) /\ (c2 IN {0;1})``,
SIMP_TAC (list_ss++CONJ_ss) [PROGRAM_PRED___can_left_rotate_def, tree_11, tree_distinct,
   IS_RED_BLACK_TREE_NODE_THM, GSYM RIGHT_EXISTS_AND_THM,
   GSYM LEFT_EXISTS_AND_THM, RED_BLACK_TREE___RIGHT_SUBTREE_def] THEN
EQ_TAC THEN REPEAT STRIP_TAC THEN
ASM_SIMP_TAC list_ss [tree_11]);

val PROGRAM_FUN___left_rotate_def = Define `
   PROGRAM_FUN___left_rotate
      (node [k1:num;v1;c1] [a; node [k2;v2;c2] [b;c]]) =
      (node [k2;v2;0] [node [k1;v1;1] [a;b]; c])`

(* holfoot_set_goal_procedures file ["rb_tree_left_rotate"] *)
val rb_tree_left_rotate_TAC =
   HF_CONTINUE_TAC THEN
   REPEAT STRIP_TAC THEN
   FULL_SIMP_TAC std_ss [PROGRAM_PRED___can_left_rotate___REWRITE] THEN
   xHF_CONTINUE_TAC [add_rewrites [PROGRAM_FUN___left_rotate_def]]
...

(* ------------------------------------------------------------------------- *)
(* After translating the program into a functional spec, the reasoning about *)
(* red black trees happens inside HOL4 using this functional representation. *)
(* It results in the following lemma, which captures the essense of why *)
(* the procedure rb_tree_insert is correct                            *)
(* ------------------------------------------------------------------------- *)
val RED_BLACK_TREE___WEAK___insert_r = prove (
``!k v t f n.
RED_BLACK_TREE___WEAK n T t f ==>
RED_BLACK_TREE___WEAK n (~(RED_BLACK_TREE___IS_RED t)) (PROGRAM_FUN___insert_r t k v) (f |+ (k,v))``,
...
```

B.3 VSTTE'10 Competition

During the *Verified Software: Theories, Tools and Experiments* conference in August 2010 in Edinburgh there was an informal verification competition[1] organised by Natarajan Shankar and Peter Mueller. I participated with Holfoot in this competition. Afterwards, I cleaned up my solutions and completed the missing problems. Here are the solutions to the competition problems:

B.3.1 Problem 1

This problem determines the sum and the maximum of all elements in an array l. It is to show that $sum(l) \leqslant length(l) * max(l)$ holds. A short interactive proof is needed to reason about the arithmetic properties of a straightforward implementation.

Listing B.79: vstte/vscomp1-simple.dsf

```
vscomp1(sum,max;a,n) [data_array(a,n,data)] {
   local i, tmp;
   sum = 0; max = 0; i = 0;
   while (i < n) [data_array(a,n,data) * i <= n * (sum <= (i * max))] {
      tmp = (a + i) -> dta;
      if (max < tmp) { max = tmp; }
      sum = sum + tmp;
      i = i + 1;
   }
} [data_array(a,n,data) * (sum <= (n * max))]

val file1 = concat [examplesDir, "/vstte/vscomp1-simple.dsf"];
val vscomp1_simple_TAC =
    (*run automation and then remove comments *)
    HF_CONTINUE_TAC THEN HF_VC_TAC THEN

    (* only some arithmetic verification conditions remain proof them interactively *)
    SIMP_TAC arith_ss [GSYM ADD1, MULT_CLAUSES] THEN
    REPEAT STRIP_TAC THENL [
       MATCH_MP_TAC LESS_EQ_TRANS THEN
       Q.EXISTS_TAC `i_const * max'_const` THEN
       ASM_SIMP_TAC arith_ss [],

       `n_const = i_const` by DECIDE_TAC THEN
       ASM_SIMP_TAC arith_ss []
    ];

val thm1 = holfoot_tac_verify_spec file1 NONE [("vscomp1", vscomp1_simple_TAC)];
```

Using HOL4 one can define functions for the maximum element and the sum of all elements. These definitions allow a simple, but strong specification of the problem. Moreover, the interesting part of the problem $sum(l) \leqslant length(l) * max(l)$ can be shown independently from the implementation. As usual, there is a specification using a loop-invariant and one using a loop-specification.

Listing B.80: vstte/vscomp1-invariant.dsf

```
vscomp1(sum,max;a,n) [data_array(a,n,data)] {
   local i, tmp;
   sum = 0; max = 0; i = 0;
   while (i < n) [data_array(a,n,data) * i <= n *
                  (max == ``LIST_MAX (FIRSTN i data)``) *
                  (sum == ``LIST_SUM (FIRSTN i data)``)] {
      tmp = (a + i) -> dta;
```

[1] http://www.macs.hw.ac.uk/vstte10/Competition.html

```
      if (max < tmp) { max = tmp; }
      sum = sum + tmp;
      i = i + 1;
   }
} [data_array(a,n,data) * (sum <= (n * max)) *
   (max == ``LIST_MAX data``) * (sum == ``LIST_SUM data``)]
```

Listing B.81: vstte/vscomp1-loopspec.dsf

```
vscomp1(sum,max;a,n) [data_array(a,n,data)] {
   local i, tmp;
   sum = 0; max = 0; i = 0;
   loop_spec [data_array(a,n,data)] {
      while (i < n) {
         tmp = (a + i) -> dta;
         if (max < tmp) { max = tmp; }
         sum = sum + tmp;
         i = i + 1;
      }
   } [data_array(a,n,data) *
      (max == ``MAX (old(max)) (LIST_MAX (BUTFIRSTN (old(i)) data))``) *
      (sum == old(sum) + ``LIST_SUM (BUTFIRSTN (old(i)) data)``)]
} [data_array(a,n,data) * (sum <= (n * max)) *
   (max == ``LIST_MAX data``) * (sum == ``LIST_SUM data``)]
```

Listing B.82: vstte/vscomp1.hol

```
val LIST_SUM_def = Define `
   (LIST_SUM ([]:num list) = 0) /\
   (LIST_SUM (n::ns)     = n + LIST_SUM ns)`

val LIST_MAX_def = Define `
   (LIST_MAX ([]:num list) = 0) /\
   (LIST_MAX (n::ns)     = MAX n (LIST_MAX ns))`

(* Proof the goal as a lemma *)
val LIST_MAX_SUM_THM = prove (
``!l. LIST_SUM l <= (LENGTH l) * LIST_MAX l``,
Induct_on `l` THENL [
   SIMP_TAC list_ss [LIST_SUM_def],

   ASM_SIMP_TAC list_ss [LIST_SUM_def, LIST_MAX_def,
      MULT_CLAUSES, MAX_DEF] THEN
   REPEAT STRIP_TAC THEN
   Cases_on `h < LIST_MAX l` THEN (
      ASM_SIMP_TAC arith_ss []
   ) THEN
   `LIST_MAX l <= h` by DECIDE_TAC THEN
   METIS_TAC[MULT_SYM,LESS_EQ_TRANS,LESS_MONO_MULT]
]);

val LIST_SUM_SNOC = prove (``!n ns. LIST_SUM (SNOC n ns) = LIST_SUM (n::ns)``,
Induct_on `ns` THEN ASM_SIMP_TAC list_ss [LIST_SUM_def]);

val LIST_MAX_SNOC = prove (``!n ns. LIST_MAX (SNOC n ns) = LIST_MAX (n::ns)``,
Induct_on `ns` THEN ASM_SIMP_TAC list_ss [LIST_MAX_def] THEN
SIMP_TAC (arith_ss++boolSimps.COND_elim_ss) [MAX_DEF]);

(******************************************************************************)
(* Verify specification using these definitions and a loop invariant  *)
(******************************************************************************)
val file2 = concat [examplesDir, "/vstte/vscomp1-invariant.dsf"];
val vscomp1_invariant_TAC =
    (* run automation and concentrate on remaining VCs i.e. remove comments *)
    HF_VC_SOLVE_TAC THEN HF_VC_TAC THEN

    (* solve vcs using the simplifier *)
    SIMP_TAC arith_ss [GSYM ADD1,
       LIST_SUM_def, LIST_MAX_def,
       LIST_MAX_SUM_THM,
       GSYM SNOC_EL_FIRSTN, MAX_DEF,
       LIST_SUM_SNOC, LIST_MAX_SNOC]
```

```
val thm2 = holfoot_tac_verify_spec file2 NONE [("vscomp1", vscomp1_invariant_TAC)];


(***************************************************************************)
(* Verify specification using these definitions and a loop spec        *)
(***************************************************************************)
val file3 = concat [examplesDir, "/vstte/vscomp1-loopspec.dsf"];
val vscomp1_loopspec_TAC =
    (*run automation and concentrate on remaining VCs i.e. remove comments *)
    HF_SOLVE_TAC THEN HF_VC_TAC THEN

    (* solve vcs using the simplifier *)
    SIMP_TAC arith_ss [BUTFIRSTN_LENGTH_LESS, GSYM ADD1,
       BUTFIRSTN_CONS_EL, LIST_SUM_def, LIST_MAX_def,
       LIST_MAX_SUM_THM, MAX_DEF]

val thm3 = holfoot_tac_verify_spec file3 NONE [("vscomp1", vscomp1_loopspec_TAC)];
```

B.3.2 Problem 2

This problem is about inverting an array. It is again solved using a new HOL4 definition that captures the semantics of the procedure. Then the interesting properties are shown for this newly introduced function instead directly for the implementation.

Listing B.83: vstte/vscomp2.dsf

```
vscomp2(;a,b,n) [data_array(a,n,data) * data_array(b,m,data2) * ``EVERY (\x. x < m) data``] {
   local i;
   i = 0;
   loop_spec [data_array(a,n,data) * data_array(b,m,data2) * ``EVERY (\x. x < m) data``] {
      while (i < n) {
         tmp = (a + i) -> dta;
         (b + tmp) -> dta = i;
         i = i + 1;
      }
   } [data_array(a,n,data) * data_array(b,m,
         ``VSCOMP2_FUN data2 (old(i)) (BUTFIRSTN (old(i)) data)``)]
} [data_array(a,n,data) * data_array(b,m,``VSCOMP2_FUN data2 0 data``)]
```

Listing B.84: vstte/vscomp2.hol

```
val VSCOMP2_FUN_def = Define `
  (VSCOMP2_FUN l i [] = l) /\
  (VSCOMP2_FUN l i (n::ns) =
     VSCOMP2_FUN (REPLACE_ELEMENT i n l) (SUC i) ns)`

val LENGTH_VSCOMP2_FUN = prove (
``!l i ns. LENGTH (VSCOMP2_FUN l i ns) = LENGTH l``,
Induct_on `ns` THEN
ASM_SIMP_TAC std_ss [VSCOMP2_FUN_def, LENGTH_REPLACE_ELEMENT]);

(***************************************************************************)
(* Verify the spec that the program implements VSCOMP2_FUN             *)
(***************************************************************************)
val file = concat [examplesDir, "/vstte/vscomp2.dsf"];

(* holfoot_set_goal_procedures file ["vscomp2"] *)
val vscomp2_TAC =
    (*simplify the EVERY into something easier for the automation *)
    SIMP_TAC std_ss [EVERY_MEM, MEM_EL,GSYM LEFT_FORALL_IMP_THM] THEN
    (*run automation *)
    HF_SOLVE_TAC THEN
    (* clean up the goal a bit and the prove VCs on VSCOMP2_FUN *)
    REPEAT STRIP_TAC THEN HF_VC_TAC THENL [
       ASM_SIMP_TAC std_ss [BUTFIRSTN_LENGTH_LESS, VSCOMP2_FUN_def],
       FULL_SIMP_TAC list_ss [LENGTH_VSCOMP2_FUN, LENGTH_REPLACE_ELEMENT,
                              GSYM ADD1, BUTFIRSTN_CONS_EL, VSCOMP2_FUN_def]
    ];
```

```
val final_thm = holfoot_tac_verify_spec file NONE [("vscomp2", vscomp2_TAC)];

(****************************************************************************)
(* So far, the program was reduced to a functional implementation as a HOL 4 *)
(* function VSCOMP2_FUN. Now show some interesting properties of VSCOMP2_FUN *)
(****************************************************************************)

val EL_VSCOMP2_FUN___NOT_IN = prove (``!l1 i l2 n.
   n < LENGTH l1 /\ ~(MEM n l2) ==>
   (EL n (VSCOMP2_FUN l1 i l2) = EL n l1)``, ...);

val EL_VSCOMP2_FUN___IN = prove (``!l1 i l2 n.
   (n < LENGTH l1 /\ (MEM n l2)) ==> (EL n (VSCOMP2_FUN l1 i l2) >= i)``, ...)

(* proving B[A[i]] = i *)
val VSCOMP2_FUN___EL = prove (``!l1 i l2.
  EVERY (\x. x < LENGTH l2) l1 /\ ALL_DISTINCT l1 ==>
  (!n. n < LENGTH l1 ==> (EL (EL n l1) (VSCOMP2_FUN l2 i l1) = n+i))``, ...);

(* proving B is injective *)
val VSCOMP2_FUN___INJ1 = prove (``!l1 i l2 n m.
  (EVERY (\x. x < LENGTH l2) l1 /\
  (EL n (VSCOMP2_FUN l2 i l1) = EL m (VSCOMP2_FUN l2 i l1)) /\
  MEM n l1 /\ MEM m l1) ==> (n = m)``, ...)

val VSCOMP2_FUN___INJ = prove (
``!l1 i l2 n m. (EVERY (\x. x < LENGTH l2) l1 /\ (!n. n < LENGTH l2 ==> MEM n l1)) ==>
  ALL_DISTINCT (VSCOMP2_FUN l2 i l1)``, ...)
```

B.3.3 Problem 3

This example searches for the first occurrence of 0 in a singly-linked list.

Listing B.85: vstte/vscomp3-loopspec.dsf

```
vscomp3(i;l1) [data_list(ll,data)] {
   local found, jj, tmp;
   jj = l1; found = 0; i = 0;
   loop_spec [ data_list (jj,data2) * ``(~(found = 0)) ==> (HD data2 = 0)``] {
      while ((jj != NULL) and (found == 0)) {
         tmp = jj -> dta;
         if (tmp == 0) { found = 1; } else { jj = jj -> tl; i = i + 1; }
      }
   } [ data_list (old(jj),data2) * (old(i) <= i) *
       (i <= old(i) + ``LENGTH data2``) *
       ``!n. n < (i - old(i)) ==> ~(EL n data2 = 0)`` *
       ``((i - old(i)) < LENGTH data2) ==> (EL (i - old(i)) data2 = 0)``]
} [ data_list (ll,data) *
    ``(i <= LENGTH data) /\ (!n. n < i ==> ~(EL n data = 0)) /\
     (i < LENGTH data ==> (EL i data = 0))``]
```

This specification looks complicated, because of the lengthy characterisation of the first index of 0. Introducing a new definition and generalising the search to find an element that satisfies some predicate P leads to the following, simpler specification.

Listing B.86: vstte/vscomp3-loopspec2.dsf

```
global P;

vscomp3(i;l1) [data_list(ll,data)] {
   local found, jj, tmp;
   jj = l1; found = 0; i = 0;
   loop_spec [ data_list (jj,data2) * ``(~(found = 0)) ==> (P (HD data2))``] {
      while ((jj != NULL) and (found == 0)) {
         tmp = jj -> dta;
         if (``P tmp``) { found = 1; } else { jj = jj -> tl; i = i + 1; }
      }
```

```
   } [ data_list (old(jj),data2) * (i == ‘‘old(i) + (FIRST_INDEX P data2)‘‘)]
} [ data_list (ll,data) * (i == ‘‘FIRST_INDEX P data‘‘)]
```

The verification of both specifications is straightforward.

Listing B.87: vstte/vscomp3.hol

```
val file = concat [examplesDir, "/vstte/vscomp3-loopspec.dsf"];

(* holfoot_set_goal_procedures file ["vscomp3"] *)
val vscomp3_loopspec_TAC =
    (*run automation and clean up *)
    HF_CONTINUE_TAC THEN REPEAT STRIP_TAC THEN HF_VC_TAC THEN

    (* a bit of arithmetic reasoning and a case split *)
    ‘i_const - old_i = SUC ( i_const  (old_i + 1))‘ by DECIDE_TAC THEN
    ASM_SIMP_TAC list_ss [] THEN
    Cases_on ‘n‘ THEN FULL_SIMP_TAC list_ss []

val thm1 = holfoot_tac_verify_spec file NONE [("vscomp3", vscomp3_loopspec_TAC)];


(**************************************************************************)
(* Introduce special search predicates                                    *)
(**************************************************************************)

val FIRST_INDEX_def = Define ‘
  FIRST_INDEX P l = LEAST n. (n = LENGTH l) \/ P (EL n l)‘

val FIRST_INDEX_THM = prove (
‘‘!P l n. (FIRST_INDEX P l = n) =
        (n <= LENGTH l) /\ (!i. i < n ==> ~(P (EL i l))) /\
        ((n < LENGTH l) ==> P (EL n l))‘‘,
...)

val FIRST_INDEX_REWRITE = prove (
‘‘(FIRST_INDEX P [] = 0) /\
  (FIRST_INDEX P (e::es) =
   if (P e) then 0 else SUC (FIRST_INDEX P es))‘‘, ...)


val file4 = concat [examplesDir, "/vstte/vscomp3-loopspec2.dsf"];
val vscomp3_loopspec2_TAC =
    (*run automation *)
    HF_SOLVE_TAC THEN HF_VC_TAC THEN

    (* use definition of FIRST_INDEX *)
    SIMP_TAC (list_ss++boolSimps.CONJ_ss) [FIRST_INDEX_REWRITE] THEN
    Cases_on ‘data2‘ THEN SIMP_TAC list_ss [FIRST_INDEX_REWRITE]

val thm4 = holfoot_tac_verify_spec file4 NONE [("vscomp3", vscomp3_loopspec2_TAC)];
```

B.3.4 Problem 4

Problem 4 searches for a solution of the n-queens problem, i. e. of the problem of placing n queens on a chess-board of size $n \times n$.

Listing B.88: vstte/vscomp4.dsf

```
isConsistent(r; board, p) [data_array(board,#m,data) * (p < #m)] {
   local q, b_q, b_p;
   r = 1; q = 0; b_p = (board + p) -> dta;
   while ((q < p) and (r == 1)) [
       data_array(board,#m,data) * (q <= p) * (q < #m) * (p < #m) *
       ‘‘( r = BOOL_TO_NUM (IS_CONSISTENT_BOARD_REC q p data)) /\ (b_p = EL p data)‘‘] {
      b_q = (board + q) -> dta;
      if (‘‘(b_q = b_p) \/
            (b_q - b_p = p - q) \/
            (b_p - b_q = p - q)‘‘) {
```

```
        r = 0;
      }
      q = q + 1;
    }
} [data_array (board,#m,data) *
    `` r = BOOL_TO_NUM (IS_CONSISTENT_BOARD_REC p p data)``]

search(r; board, p, m) [data_array(board,m,``data1++data2``) *
    ``(p = LENGTH data1) /\ (m = LENGTH data1 + LENGTH data2)``] {
  local i, c;
  r = 0;
  if (p == m) { r = 1; } else {
    i = 0;
    while ((i < m) and (r == 0)) [
        data_array (board,m,``data1++_data2``) *
        ``IS_BOOL_TO_NUM r`` * (i <= m) * (p < m) * (p == ``LENGTH data1``) *
        `` if (r = 1) then
            ((EVERY (\x. x < m) _data2) /\
             (!pp. (p <= pp /\ pp < m) ==> IS_CONSISTENT_BOARD_REC pp pp
                       (data1 ++ _data2)))
          else
             (!i' data3.
                 (i' < i) /\
                 (SUC (LENGTH data3) = LENGTH _data2) /\
                 (EVERY (\x. x < m) data3) ==> ?pp.
                 ((p <= pp /\ pp < m) /\
                  ~(IS_CONSISTENT_BOARD_REC pp pp (data1 ++ i'::data3))))``] {
        (board + p) -> dta = i;
         isConsistent(c; board, p);
         if (c == 1) { search (r;board, p+1, m); }
         i = i + 1;
      }
  }
} [data_array (board,m,``data1++_data2``) *
    ``IS_BOOL_TO_NUM r`` *
    `` if (r = 1) then
          ((EVERY (\x. x < m) _data2) /\
           (!pp. (p <= pp /\ pp < m) ==> IS_CONSISTENT_BOARD_REC pp pp
                       (data1 ++ _data2)))
       else
          (!data3.
               (EVERY (\x. x < m) data3) /\
               (LENGTH data1 + LENGTH data3 = m) ==>
               (?pp. (p <= pp /\ pp < m) /\
                     ~(IS_CONSISTENT_BOARD_REC pp pp (data1 ++ data3))))``]

find(r, b; m) [] {
  b = new(m) [dta];
  search(r; b, 0, m);
} [data_array (b, m, _data) *
    if (r == 1) then
       ``IS_CONSISTENT_BOARD _data``
    else
       ``! data. (LENGTH data = m) ==> ~(IS_CONSISTENT_BOARD data)``]
```

Listing B.89: vstte/vscomp4.hol

```
(******************************************************************************)
(* Define a predicates for boards                                             *)
(******************************************************************************)
val IS_CONSISTENT_BOARD_REC_def = Define `
  IS_CONSISTENT_BOARD_REC (n:num) (p:num) l =
  !q. q < n ==>
     (~(EL q l = EL p l) /\
      ~((EL q l - EL p l) = (p - q)) /\
      ~((EL p l - EL q l) = (p - q)))`;

val IS_CONSISTENT_BOARD_def = Define `
  IS_CONSISTENT_BOARD l =
  ((!p. p < (LENGTH l) ==> IS_CONSISTENT_BOARD_REC p p l) /\
  (EVERY (\x. x < LENGTH l) l))`;

val IS_CONSISTENT_BOARD_REC___REWRITE = prove (
```

```
``(IS_CONSISTENT_BOARD_REC 0 p l) /\
  (IS_CONSISTENT_BOARD_REC (SUC n) p l =
  (IS_CONSISTENT_BOARD_REC n p l /\
    (~(EL n l = EL p l) /\
     ~((EL n l - EL p l) = (p - n)) /\
     ~((EL p l - EL n l) = (p - n)))))``,
SIMP_TAC std_ss [IS_CONSISTENT_BOARD_REC_def] THEN
`!n m. n < SUC m = ((n < m) \/ (n = m))` by DECIDE_TAC THEN
ASM_SIMP_TAC std_ss [DISJ_IMP_THM, FORALL_AND_THM]);

val IS_CONSISTENT_BOARD_REC___JUST_FIRSTN = prove (
``!n p l. (n <= p) /\ (p < LENGTH l) ==>
         (IS_CONSISTENT_BOARD_REC n p l =
          IS_CONSISTENT_BOARD_REC n p (FIRSTN (SUC p) l))``,

SIMP_TAC arith_ss [IS_CONSISTENT_BOARD_REC_def,
   EL_FIRSTN]);

val IS_CONSISTENT_BOARD_REC___JUST_FIRSTN_MP = prove (
``!n p l l'. (n <= p) /\ (p < LENGTH l) /\ (p < LENGTH l') /\
            IS_CONSISTENT_BOARD_REC n p l /\
            (FIRSTN (SUC p) l = FIRSTN (SUC p) l') /\
            IS_CONSISTENT_BOARD_REC n p l ==>
            IS_CONSISTENT_BOARD_REC n p l'``,
METIS_TAC[IS_CONSISTENT_BOARD_REC___JUST_FIRSTN]);

val IS_CONSISTENT_BOARD___REWRITE = prove (
``IS_CONSISTENT_BOARD l =
  ((EVERY (\x. x < LENGTH l) l) /\
   (!i1 i2. (i1 < i2 /\ i2 < (LENGTH l)) ==>
       (~(EL i1 l = EL i2 l) /\
        ~((EL i1 l - EL i2 l) = (i2 - i1)) /\
        ~((EL i2 l - EL i1 l) = (i2 - i1)))))``,

SIMP_TAC std_ss [IS_CONSISTENT_BOARD_def,
  IS_CONSISTENT_BOARD_REC_def,
  FORALL_AND_THM, IMP_CONJ_THM,
  AND_IMP_INTRO,
  GSYM RIGHT_FORALL_IMP_THM,
  EVERY_MEM, MEM_EL, GSYM LEFT_FORALL_IMP_THM] THEN
REPEAT STRIP_TAC THEN EQ_TAC THEN STRIP_TAC THEN (
   ASM_SIMP_TAC std_ss []
));

(******************************************************************************)
(* Verify specification                                                       *)
(******************************************************************************)
val file = concat [examplesDir, "/vstte/vscomp4.dsf"];

(* holfoot_set_goal_procedures file ["isConsistent"] *)
val isConsistent_TAC =
    (*run automation *)
    HF_VC_SOLVE_TAC THEN HF_VC_TAC THEN

    (* simplify and instantiate loop invariant*)
    SIMP_TAC std_ss [GSYM ADD1, IS_CONSISTENT_BOARD_REC___REWRITE] THEN
    REPEAT STRIP_TAC THEN
    Q.EXISTS_TAC `data` THEN Q.EXISTS_TAC `LENGTH data` THEN

    (* generate VCs *)
    HF_VC_SOLVE_TAC THEN HF_VC_TAC THEN
    REPEAT STRIP_TAC THENL [
       `p_const = q_const` by DECIDE_TAC THEN
       ASM_REWRITE_TAC[],

       Q.PAT_ASSUM `~(IS_CONSISTENT_BOARD_REC q_const p_const data)` MP_TAC THEN
       FULL_SIMP_TAC std_ss [IS_CONSISTENT_BOARD_REC_def] THEN
       GEN_TAC THEN STRIP_TAC THEN
       ASM_SIMP_TAC list_ss []
    ]


(* holfoot_set_goal_procedures file ["search"] *)
```

```
val search_TAC =
   (*run automation *)
   HF_SOLVE_TAC THEN
   REPEAT STRIP_TAC THENL [
      (* while loop *)
      CONV_TAC SWAP_EXISTS_CONV THEN
      Q.EXISTS_TAC `LENGTH data1 + LENGTH data2` THEN
      xHF_SOLVE_TAC [add_rewrites [REPLACE_ELEMENT_APPEND2]] THEN
      REPEAT STRIP_TAC THENL [
         (* is consistent *)
         `?data2_hd data2_tl. data2 = data2_hd::data2_tl` by
             (Cases_on `data2` THEN FULL_SIMP_TAC list_ss []) THEN
         Q.EXISTS_TAC `data1 ++ [i_const]` THEN
         Q.EXISTS_TAC `data2_tl` THEN
         ASM_SIMP_TAC list_ss [REPLACE_ELEMENT_DEF] THEN
         HF_VC_SOLVE_TAC THEN HF_VC_TAC THEN
         CONJ_TAC THENL [
            FULL_SIMP_TAC (list_ss++CONJ_ss) [GSYM ADD1] THEN
            REPEAT STRIP_TAC THEN
            Cases_on `pp = LENGTH data1` THENL [
               MATCH_MP_TAC IS_CONSISTENT_BOARD_REC___JUST_FIRSTN_MP THEN
               Q.EXISTS_TAC `data1 ++ i_const::data2_tl` THEN
               ASM_SIMP_TAC list_ss [FIRSTN_APPEND2, GSYM ADD1],

               FULL_SIMP_TAC arith_ss [GSYM APPEND_ASSOC, APPEND]
            ],


            FULL_SIMP_TAC (list_ss++CONJ_ss) [GSYM ADD1] THEN
            REPEAT STRIP_TAC THEN
            Cases_on `i' < i_const` THEN1 (
               METIS_TAC[]
            ) THEN
            `i' = i_const` by DECIDE_TAC THEN
            FULL_SIMP_TAC arith_ss [GSYM APPEND_ASSOC, APPEND] THEN
            Q.PAT_ASSUM `!data3'. X` (MP_TAC o Q.SPECL [`data3`]) THEN
            Q.PAT_ASSUM `LENGTH data2' = X` ASSUME_TAC THEN
            FULL_SIMP_TAC arith_ss [] THEN
            STRIP_TAC THEN
            Q.EXISTS_TAC `pp` THEN
            ASM_SIMP_TAC arith_ss []
         ],

         (* is not consistent *)
         HF_VC_TAC THEN
         REPEAT STRIP_TAC THEN
         Cases_on `i' < i_const` THEN1 (
            METIS_TAC[]
         ) THEN
         `i' = i_const` by DECIDE_TAC THEN
         Q.EXISTS_TAC `LENGTH data1` THEN
         ASM_SIMP_TAC arith_ss [] THEN
         REPEAT STRIP_TAC THEN
         Q.PAT_ASSUM `~(IS_CONSISTENT_BOARD_REC X X Y)` MP_TAC THEN
         SIMP_TAC std_ss [] THEN
         MATCH_MP_TAC IS_CONSISTENT_BOARD_REC___JUST_FIRSTN_MP THEN
         Q.EXISTS_TAC `data1 ++ i_const::data3` THEN
         Cases_on `data2` THEN (
            FULL_SIMP_TAC list_ss [REPLACE_ELEMENT_DEF,
              FIRSTN_APPEND2, GSYM ADD1]
         )
   ],

   (* at the very end *)
   Q.EXISTS_TAC `data1` THEN
   Q.EXISTS_TAC `data2_h::data2_t` THEN
   HF_SOLVE_TAC THEN HF_VC_TAC THEN
   REPEAT STRIP_TAC THEN
   FULL_SIMP_TAC std_ss [] THEN
   `?data3_hd data3_tl. data3 = data3_hd :: data3_tl` by ALL_TAC THEN1 (
      Cases_on `data3` THEN FULL_SIMP_TAC list_ss []
   ) THEN
   Q.PAT_ASSUM `!i' data3'. X i' data3'` (MP_TAC o Q.SPECL [
```

```
      `data3_hd`, `data3_tl`]) THEN
    FULL_SIMP_TAC list_ss [GSYM ADD1]
]

(* holfoot_set_goal_procedures file ["find"] *)
val find_TAC =
    HF_SOLVE_TAC THEN HF_VC_TAC THEN
    SIMP_TAC (std_ss++CONJ_ss) [IS_CONSISTENT_BOARD_def] THEN
    METIS_TAC[]

(* put everything together *)
val final_thm = holfoot_tac_verify_spec file NONE
                  [("isConsistent", isConsistent_TAC),
                   ("search",       search_TAC),
                   ("find",         find_TAC)];
```

B.3.5 Problem 5

This problem is about amortised queues. In this implementation, an amortised queue consists of a front and a rear singly-linked list. Enqueing an element inserts the element as head of the rear list. Dequeing removes the first element of the front list. Moreover, the invariant is maintained that the rear list is at most as long as the front list.

The verification of this problem is simple. In order to provide readable specifications, new predicates for amortised queues are introduced. The interactive proof mainly consists of expanding this definition and calling Holfoot's automation.

Listing B.90: vstte/vscomp5.dsf

```
list_create(l;) [] {
  l = NULL;
} [ data_list (l ,''[ ]'')]

list_cons(l;d) [data_list(l,data)] {
  local t;
  t = new(); t->tl = l; t->dta = d; l = t;
} [ data_list (l ,'' d::data '') ]

list_dest(re,l;) [data_list(l ,'' d::data '') ] {
  local t;
  re = l->dta; t = l->tl; dispose(l); l = t;
} [ data_list (l,data) * (re == #d)]

list_concat(x;y) [data_list(x,xdata) * data_list (y,ydata)] {
  local n,t;
  if (x == NULL) { x = y; } else {
    t = x; n = t->tl;
    loop_spec [(t |-> tl:n,dta:#tdate) * data_list (n,data2) * data_list (y, data3)] {
       while (n != NULL) {
         t = n; n = t->tl;
       }
       t->tl = y;
    } [ data_list (old(t ),'' tdate ::( data2++data3)'')]
  }
} [ data_list (x, ''xdata++ydata'')]

list_reverse(i;) [data_list(i,data)] {
  local p, x;
  p = NULL;
  loop_spec [ data_list (i,data) * data_list (p, data2)] {
     while (i != NULL) {
        x = i->tl; i->tl = p; p = i; i = x;
     }
  } [ data_list (p, ''( REVERSE data)++data2'')]
  i = p;
} [ data_list (i ,'' REVERSE data'')]

queue_create(q;) [] {
  q = new();
  q->front = NULL; q->front_length = 0; q->rear = NULL; q->rear_length = 0;
} [amortized_queue(q, ''[ ]'')]

queue_length(re;q) [amortized_queue(q, data)] {
   local rl,fl;
   rl = q->rear_length; fl = q->front_length;
   re = rl + fl;
} [amortized_queue(q, data) * (re == ''LENGTH data'')]

queue_normalise(;q) [weak_amortized_queue(q, data)] {
   local r,rl,f,fl;
   r  = q->rear; rl = q->rear_length; f = q->front; fl = q->front_length;
   if (fl < rl) {
      list_reverse(r;); list_concat(f;r);
      q->rear = NULL; q->rear_length = 0;
      q->front = f; q->front_length = fl + rl;
   }
} [amortized_queue(q, data)]

queue_front(re;q) [amortized_queue(q, ''d::data '') ] {
   local f;
   f  = q->front; re = f->dta;
} [amortized_queue(q, '' d::data '') * (re == #d)]

queue_dequeue(re;q) [amortized_queue(q, ''d::data '') ] {
   local rl,f,fl;
   rl = q->rear_length; f = q->front; fl = q->front_length;
   list_dest(re,f;);
   fl = fl - 1; q->front = f; q->front_length = fl;
   if (fl < rl) { queue_normalise(;q); }
} [amortized_queue(q, data) * (re == #d)]
```

```
queue_enqueue(;q,d) [amortized_queue(q, data)] {
   local r,rl,fl;
   r  = q->rear; rl = q->rear_length; fl = q->front_length;
   list_cons(r;d); rl = rl + 1;
   q->rear = r; q->rear_length = rl;
   if (fl < rl) { queue_normalise(;q); }
} [amortized_queue(q, ``SNOC d data``)]
```

Listing B.91: vstte/vscomp5.hol

```
(******************************************************************************)
(* Define a predicate for amortized queues                                    *)
(******************************************************************************)
val holfoot_ap_amortized_queue_def = Define `
   holfoot_ap_amortized_queue strong tl q dta data =
   asl_exists f r f_data r_data. asl_bigstar_list holfoot_separation_combinator
      [holfoot_ap_points_to q (LIST_TO_FMAP [
           (holfoot_tag "front", var_res_exp_const f);
           (holfoot_tag "rear", var_res_exp_const r);
           (holfoot_tag "front_length", var_res_exp_const (LENGTH f_data));
           (holfoot_tag "rear_length", var_res_exp_const (LENGTH r_data))]);
       holfoot_ap_data_list tl (var_res_exp_const f) [(dta, f_data)];
       holfoot_ap_data_list tl (var_res_exp_const r) [(dta, r_data)];
       var_res_bool_proposition DISJOINT_FMAP_UNION
           ((data = f_data ++ (REVERSE r_data)) /\
            (strong ==> (LENGTH r_data <= LENGTH f_data)))]`

val holfoot_ap_amortized_queue_REWRITE = save_thm ("holfoot_ap_amortized_queue_REWRITE",
SIMP_RULE std_ss [asl_bigstar_list_REWRITE,
   asl_star_holfoot_THM] holfoot_ap_amortized_queue_def);

val holfoot_ap_amortized_queue_REWRITE2 = save_thm ("holfoot_ap_amortized_queue_REWRITE2",
SIMP_RULE list_ss [LIST_TO_FMAP_THM, holfoot_separation_combinator_def]
   holfoot_ap_amortized_queue_REWRITE);

(***************************************)
(* add the new predicate to the parser *)
(***************************************)
val holfoot_ap_amortized_queue_term = Term `holfoot_ap_amortized_queue`;
fun mk_holfoot_ap_amoritized_queue_absyn (strong, tag, exp, dtag, data) =
      Absyn.list_mk_app (Absyn.mk_AQ holfoot_ap_amortized_queue_term, [
         Absyn.mk_AQ (if strong then T else F), tag, exp, dtag, data]);

(* amortized_queue (q,data) *)
val _ = add_genpred ("amortized_queue", [Aspred_arg_ty_exp, Aspred_arg_ty_comma, Aspred_arg_ty_hol],
         fn [exp1,data] => mk_holfoot_ap_amoritized_queue_absyn (true,
            Absyn.mk_AQ (string2holfoot_tag (!list_link_tag)), exp1,
            Absyn.mk_AQ (string2holfoot_tag (!data_list_tag)), data));

(* weak_amortized_queue (q,data) *)
val _ = add_genpred ("weak_amortized_queue", [Aspred_arg_ty_exp, Aspred_arg_ty_comma, Aspred_arg_ty_hol],
         fn [exp1,data] => mk_holfoot_ap_amoritized_queue_absyn (false,
            Absyn.mk_AQ (string2holfoot_tag (!list_link_tag)), exp1,
            Absyn.mk_AQ (string2holfoot_tag (!data_list_tag)), data));

(***************************************)
(* add it to the pretty printer    *)
(***************************************)
fun amortized_queue_printer Gs sys (ppfns:term_pp_types.ppstream_funs) gravs d pps t = let
    open Portable term_pp_types
    val {add_string,add_break,begin_block,end_block,
         add_ann_string,add_newline,begin_style,end_style,...} = ppfns
    val (op_term,args) = strip_comb t;
  in
    if (same_const op_term holfoot_ap_amortized_queue_term) then (
      let
         val is_strong = same_const (el 1 args) T;
         val desc = if is_strong then "amortized_queue" else "weak_amortized_queue";
      in
         begin_block INCONSISTENT 0;
         add_string desc; add_string "(";
         add_break (0,!holfoot_pretty_printer_block_indent);
         sys (Top, Top, Top) (d - 1) (el 2 args);
```

```
        add_string ";";
        add_break (1,!holfoot_pretty_printer_block_indent);
        sys (Top, Top, Top) (d - 1) (el 3 args);
        add_string ",";add_break (1,!holfoot_pretty_printer_block_indent);
        sys (Top, Top, Top) (d - 1) (el 4 args);
        add_string ":";
        sys (Top, Top, Top) (d - 1) (el 5 args);
        add_string ")";
        end_block ()
      end
    ) else (
      raise term_pp_types.UserPP_Failed
    )
  end;

val _ = add_user_printer ("amortized_list_printer", ``x:'a set``, amortized_queue_printer);

(*****************************************)
(* prove thms needed for basic automation *)
(*****************************************)

val VAR_RES_IS_STACK_IMPRECISE___USED_VARS___holfoot_ap_amortized_queue = prove (
``!tl st q dta data vs.
     VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs q ==>
     VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
       (holfoot_ap_amortized_queue st tl q dta data)``, ...);

val var_res_prop_varlist_update___holfoot_ap_amortized_queue = prove (
``!vcL st tl q dta data.
     IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS q) ==>
     (var_res_prop_varlist_update vcL (holfoot_ap_amortized_queue st tl q dta data) =
      holfoot_ap_amortized_queue st tl (var_res_exp_varlist_update vcL q) dta data)``, ...);

holfoot_prover_extras_2 := [VAR_RES_IS_STACK_IMPRECISE___USED_VARS___holfoot_ap_amortized_queue];
holfoot_varlist_rwts := [var_res_prop_varlist_update___holfoot_ap_amortized_queue];
update_var_res_param();

(****************************************************************************)
(* Verify specification                                                    *)
(****************************************************************************)
val file = concat [examplesDir, "/vstte/vscomp5.dsf"];
val thm1 = prove (parse_holfoot_file file,
    (* use the definition for rewriting *)
    REWRITE_TAC [holfoot_ap_amortized_queue_REWRITE2] THEN
    (* Call automation *)
    HF_SOLVE_TAC THEN
    (* one manual case split needed *)
    REPEAT STRIP_TAC THEN
    Cases_on `f_data = []` THEN (
       HF_SOLVE_TAC
    ));
```

Appendix C

HOL4-Theorem Index

C.1 holfootTheory

(1) ASL_IS_LOCAL_ACTION___holfoot_dispose_action 123

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS ne) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  ASL_IS_LOCAL_ACTION holfoot_separation_combinator
    (holfoot_dispose_action ne e)
```

(2) ASL_IS_LOCAL_ACTION___holfoot_field_assign_action 124

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e2) ⇒
  ASL_IS_LOCAL_ACTION holfoot_separation_combinator
    (holfoot_field_assign_action e1 t e2)
```

(3) ASL_IS_LOCAL_ACTION___holfoot_field_lookup_action 123

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  ASL_IS_LOCAL_ACTION holfoot_separation_combinator
    (holfoot_field_lookup_action v e t)
```

(4) ASL_IS_LOCAL_ACTION___holfoot_new_action 122

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS ne) ⇒
  ASL_IS_LOCAL_ACTION holfoot_separation_combinator
    (holfoot_new_action ne v tL)
```

(5) holfoot_ap_array_def 121

```
⊢ holfoot_ap_array e n = holfoot_ap_data_array e n []
```

(6) holfoot_ap_bintree_def 120

```
⊢ holfoot_ap_bintree (lt,rt) startExp =
  holfoot_ap_tree [lt; rt] startExp
```

(7) holfoot_ap_data_array_def 121

```
⊢ holfoot_ap_data_array e ne data =
  var_res_exp_prop ne
    (λ n.
       asl_trivial_cond
         (EVERY (λ tl. LENGTH (SND tl) = n) data ∧
          ALL_DISTINCT (MAP FST data))
         (var_res_map DISJOINT_FMAP_UNION
            (λ el. holfoot_ap_points_to (FST el) (SND el))
            (holfoot_ap_data_array_MAP_LIST e n data)))
```

(8) holfoot_ap_data_interval_def 121

```
⊢ holfoot_ap_data_interval e1 e2 data =
  holfoot_ap_data_array e1
    (var_res_exp_binop (-) (var_res_exp_add e2 1) e1) data
```

(9) holfoot_ap_data_list_def 119

```
⊢ holfoot_ap_data_list tl startExp data =
  holfoot_ap_data_list_seg tl startExp data (var_res_exp_const 0)
```

(10) holfoot_ap_data_list_seg_def 119

```
⊢ holfoot_ap_data_list_seg tl startExp data endExp =
  asl_exists n.
    holfoot_ap_data_list_seg_num n tl startExp data endExp
```

(11) holfoot_ap_data_list_seg_num_REWRITE 118

```
⊢ (holfoot_ap_data_list_seg_num 0 tl startExp data endExp =
   if
     EVERY (λx. NULL (SND x)) data ∧ ALL_DISTINCT (tl::MAP FST data)
   then
     var_res_prop_equal DISJOINT_FMAP_UNION startExp endExp
   else
     asl_false) ∧
  (holfoot_ap_data_list_seg_num (SUC n) tl startExp data endExp =
   if
     EVERY (λx. ¬NULL (SND x)) data ∧ ALL_DISTINCT (tl::MAP FST data)
   then
     asl_and (var_res_prop_weak_unequal startExp endExp)
       (asl_exists n'.
          asl_star holfoot_separation_combinator
            (holfoot_ap_points_to startExp
               (LIST_TO_FMAP
                  (ZIP
                     (tl::MAP FST data,
                      MAP var_res_exp_const
                        (n'::MAP (λx. HD (SND x)) data)))))
            (holfoot_ap_data_list_seg_num n tl (var_res_exp_const n')
               (MAP (λ(t,l). (t,TL l)) data) endExp))
   else
     asl_false)
```

(12) holfoot_ap_data_list_seg___implies_in_heap___COMPUTE 126

```
⊢ var_res_implies_unequal DISJOINT_FMAP_UNION B e1 e2 ⇒
  B ≠ {||} ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e2) ⇒
  holfoot_implies_in_heap B {|holfoot_ap_data_list_seg tl e1 data e2|}
    e1
```

(13) holfoot_ap_data_list___implies_in_heap_or_null___COMPUTE 126

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1) ⇒
  holfoot_implies_in_heap_or_null B
    {|holfoot_ap_data_list_seg tl e1 data (var_res_exp_const 0)|} e1
```

(14) holfoot_ap_data_tree___implies_in_heap_or_null___COMPUTE 126

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  holfoot_implies_in_heap_or_null B
    {|holfoot_ap_data_tree tagL e data|} e
```

(15) holfoot_ap_data_tree___REWRITE 119

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  (holfoot_ap_data_tree tagL e (dtagL,data) =
   asl_or
     (asl_trivial_cond (ALL_DISTINCT (tagL ++ dtagL) ∧ IS_LEAF data)
        (var_res_prop_equal DISJOINT_FMAP_UNION e
           (var_res_exp_const 0)))
     (asl_exists_list dtagL
        (λ v.
           asl_exists_list tagL
             (λ lL.
                asl_exists_list tagL
                  (λ tL.
                     asl_trivial_cond
                       ((NULL tagL ⇒ ALL_DISTINCT dtagL) ∧
                        (data = node v tL))
                       (asl_bigstar_list
                          holfoot_separation_combinator
                          (holfoot_ap_points_to e
                             (LIST_TO_FMAP
                                (ZIP
                                   (tagL ++ dtagL,
                                    MAP var_res_exp_const
                                      (lL ++ v)))))::
                           MAP
                             (λ lt.
                                holfoot_ap_data_tree tagL
                                  (var_res_exp_const (FST lt))
                                  (dtagL,SND lt))
                             (ZIP (lL,tL))))))))
```

(16) holfoot_ap_data_tree___TREE_PROPS 119

```
⊢ ¬holfoot_ap_data_tree___WELL_FORMED_DATA tagL (dtagL,t) ⇒
  (holfoot_ap_data_tree tagL startExp (dtagL,t) = asl_false)
```

(17) holfoot_ap_data_tree___WELL_FORMED_DATA_def 119, 120

```
⊢ holfoot_ap_data_tree___WELL_FORMED_DATA tagL data ⇔
  TREE_EVERY (λ v. LENGTH v = LENGTH (FST data)) (SND data) ∧
  NARY (SND data) (LENGTH tagL) ∧ ALL_DISTINCT (tagL ++ FST data)
```

(18) holfoot_ap_list_def 119

```
⊢ holfoot_ap_list tl startExp =
  holfoot_ap_list_seg tl startExp (var_res_exp_const 0)
```

(19) holfoot_ap_list_seg_def 119

```
⊢ holfoot_ap_list_seg tl startExp endExp =
  holfoot_ap_data_list_seg tl startExp [] endExp
```

(20) holfoot_ap_points_to_def 118

```
⊢ holfoot_ap_points_to e1 L =
  (λ state.
     (let stack = FST state in
      let heap = SND state in
      let loc_opt = e1 stack
      in
        IS_SOME loc_opt ∧
        (let loc = THE loc_opt
         in
           loc ≠ 0 ∧ (FDOM heap = {loc}) ∧
           FEVERY
             (λ (tag,exp).
                IS_SOME (exp stack) ∧
                (THE (exp stack) = heap ' loc tag)) L)))
```

(21) holfoot_ap_points_to___implies_in_heap___COMPUTE 125

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  holfoot_implies_in_heap B {|holfoot_ap_points_to e L|} e
```

(22) holfoot_ap_tree_def 120

```
⊢ holfoot_ap_tree tagL startExp =
  asl_exists dataTree.
    holfoot_ap_data_tree tagL startExp ([],dataTree)
```

(23) HOLFOOT_COND_INFERENCE___prog_dispose 123

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) n ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb) sfb)
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT (holfoot_ap_data_array e n data) sfb))
    (asl_prog_block (holfoot_prog_dispose n e::progL)) Q
```

(24) HOLFOOT_COND_INFERENCE___prog_dispose_1 123

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb) sfb)
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT (holfoot_ap_points_to e L) sfb))
    (asl_prog_block
       (holfoot_prog_dispose (var_res_exp_const 1) e::progL)) Q
```

(25) HOLFOOT_COND_INFERENCE___prog_field_assign 124

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e1 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e2 ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
    (holfoot_ap_points_to e1 (L |+ (t,e2))) ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT (holfoot_ap_points_to e1 (L |+ (t,e2))) sfb))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT (holfoot_ap_points_to e1 L) sfb))
    (asl_prog_block (holfoot_prog_field_assign e1 t e2::progL)) Q
```

(26) HOLFOOT_COND_INFERENCE___prog_field_assign___array 124

```
⊢ ds ≤ e ∧ e < ds + dl ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (holfoot_ap_data_array (var_res_exp_const ds)
             (var_res_exp_const dl)
             ((t,REPLACE_ELEMENT c (e − ds) tdata)::data)) sfb))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (holfoot_ap_data_array (var_res_exp_const ds)
             (var_res_exp_const dl) ((t,tdata)::data)) sfb))
    (asl_prog_block
       (holfoot_prog_field_assign (var_res_exp_const e) t
          (var_res_exp_const c)::progL)) Q
```

(27) HOLFOOT_COND_INFERENCE___prog_field_assign___exp_rewrite 124

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1') ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e2) ⇒
  (VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
     (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
        (BAG_INSERT (var_res_prop_equal DISJOINT_FMAP_UNION e1 e1')
           sfb))
     (asl_prog_block (holfoot_prog_field_assign e1 t e2::progL)) Q ⟺
   VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
     (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
        (BAG_INSERT (var_res_prop_equal DISJOINT_FMAP_UNION e1 e1')
           sfb))
     (asl_prog_block (holfoot_prog_field_assign e1' t e2::progL)) Q)
```

(28) HOLFOOT_COND_INFERENCE___prog_field_lookup 123

```
⊢ v ∈: wpb ∧ t ∈ FDOM L ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) (L ' t) ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_varlist_update [(v,c)] (L ' t)))
          (BAG_IMAGE (var_res_prop_varlist_update [(v,c)])
             (BAG_INSERT (holfoot_ap_points_to e L) sfb))))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_const c))
          (BAG_INSERT (holfoot_ap_points_to e L) sfb)))
    (asl_prog_block (holfoot_prog_field_lookup v e t::progL)) Q
```

(29) HOLFOOT_COND_INFERENCE___prog_field_lookup___array 123

```
⊢ ds ⩽ e ∧ e < ds + dl ⇒
  v ∈: wpb ∧ MEM (t,tdata) data ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_const (EL (e − ds) tdata)))
          (BAG_IMAGE (var_res_prop_varlist_update [(v,c)])
             (BAG_INSERT
                (holfoot_ap_data_array (var_res_exp_const ds)
                   (var_res_exp_const dl) data) sfb))))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_const c))
          (BAG_INSERT
             (holfoot_ap_data_array (var_res_exp_const ds)
                (var_res_exp_const dl) data) sfb)))
    (asl_prog_block
       (holfoot_prog_field_lookup v (var_res_exp_const e) t::progL))
    Q
```

(30) HOLFOOT_COND_INFERENCE___prog_field_lookup___exp_rewrite 123

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e') ⇒
  (VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
     (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
        (BAG_INSERT (var_res_prop_equal DISJOINT_FMAP_UNION e e')
           sfb))
     (asl_prog_block (holfoot_prog_field_lookup v e t::progL)) Q ⟺
   VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
     (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
        (BAG_INSERT (var_res_prop_equal DISJOINT_FMAP_UNION e e')
           sfb))
     (asl_prog_block (holfoot_prog_field_lookup v e' t::progL)) Q)
```

(31) HOLFOOT_COND_INFERENCE___prog_new 122

```
⊢ v ∈: wpb ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) n ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (holfoot_ap_data_array (var_res_exp_var v)
             (var_res_exp_varlist_update [(v,c)] n) [])
          (BAG_IMAGE (var_res_prop_varlist_update [(v,c)]) sfb)))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_const c)) sfb))
    (asl_prog_block (holfoot_prog_new n v tL::progL)) Q
```

(32) HOLFOOT_COND_INFERENCE___prog_new_1 122

```
⊢ v ∈: wpb ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT (holfoot_ap_points_to (var_res_exp_var v) FEMPTY)
          (BAG_IMAGE (var_res_prop_varlist_update [(v,c)]) sfb)))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE DISJOINT_FMAP_UNION
    (var_res_prop DISJOINT_FMAP_UNION (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION (var_res_exp_var v)
             (var_res_exp_const c)) sfb))
    (asl_prog_block
       (holfoot_prog_new (var_res_exp_const 1) v tL::progL)) Q
```

(33) holfoot_dispose_action_def 122

```
⊢ holfoot_dispose_action me e s =
   (let loc_opt = e (FST s) in
    let m_opt = me (FST s)
    in
      if IS_NONE m_opt then
        NONE
      else
        (let m = THE m_opt
         in
           if m = 0 then
             SOME {s}
           else if IS_NONE loc_opt then
             NONE
           else
             (let loc = THE loc_opt
              in
                if
                  ¬(IMAGE (λn'. loc + n') (count m) ⊆ FDOM (SND s)) ∨
                  (loc = 0)
                then
                  NONE
                else
                  SOME
                    {(FST s,
                      DRESTRICT (SND s)
                        (COMPL (IMAGE (λn'. loc + n') (count m))))}))
```

(34) holfoot_field_assign_action_def 124

```
⊢ holfoot_field_assign_action e₁ t e₂ s =
   (let e1_opt = e₁ (FST s) in
    let e2_opt = e₂ (FST s)
    in
      if IS_NONE e1_opt ∨ IS_NONE e2_opt then
        NONE
      else
        (let e1_v = THE e1_opt in
         let e2_v = THE e2_opt
         in
           if e1_v ∉ FDOM (SND s) ∨ (e1_v = 0) then
             NONE
           else
             SOME
               {(FST s,SND s |+ (e1_v,(t =+ e2_v) (SND s ' e1_v)))}))
```

(35) holfoot_field_lookup_action_def 123

```
⊢ holfoot_field_lookup_action v e t s =
  (let loc_opt = e (FST s)
   in
    if
      ¬var_res_sl___has_write_permission v (FST s) ∨ IS_NONE loc_opt
    then
      NONE
    else
      (let loc = THE loc_opt
       in
        if loc ∉ FDOM (SND s) ∨ (loc = 0) then
          NONE
        else
          SOME {var_res_ext_state_var_update (v,SND s ' loc t) s}))
```

(36) holfoot_implies_in_heap_def 124

```
⊢ holfoot_implies_in_heap =
  holfoot_implies_in_heap_pred (λX x. x ≠ 0 ∧ x ∈ X)
```

(37) holfoot_implies_in_heap_or_null_def 124

```
⊢ holfoot_implies_in_heap_or_null =
  holfoot_implies_in_heap_pred (λX x. (x = 0) ∨ x ∈ X)
```

(38) holfoot_implies_in_heap_or_null___const_null 125

```
⊢ holfoot_implies_in_heap_or_null B b (var_res_exp_const 0)
```

(39) holfoot_implies_in_heap_or_null___equal_null 125

```
⊢ (∀ B e sfb.
     IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
     holfoot_implies_in_heap_or_null B
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION
             (var_res_exp_const 0) e) sfb) e) ∧
  (∀ B e sfb.
     IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
     holfoot_implies_in_heap_or_null B
       (BAG_INSERT
          (var_res_prop_equal DISJOINT_FMAP_UNION e
             (var_res_exp_const 0)) sfb) e) ∧
  (∀ B e sfb.
     B ≠ {||} ∧
     IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
     holfoot_implies_in_heap_or_null B
       (BAG_INSERT (var_res_prop_weak_equal (var_res_exp_const 0) e)
          sfb) e) ∧
  ∀ B e sfb.
    IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
    holfoot_implies_in_heap_or_null B
      (BAG_INSERT (var_res_prop_weak_equal e (var_res_exp_const 0))
         sfb) e
```

(40) holfoot_implies_in_heap_or_null___implies_equal 125

```
⊢ b1 ⊎ b2 ≼ sfb ∧ holfoot_implies_in_heap_or_null sfb b1 e ∧
  holfoot_implies_in_heap_or_null sfb b2 e ⇒
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ⇒
  var_res_prop_implies DISJOINT_FMAP_UNION (wpb,rpb) sfb
    {|var_res_prop_equal DISJOINT_FMAP_UNION e (var_res_exp_const 0)|}
```

(41) holfoot_implies_in_heap_or_null___SUB_BAG 125

```
⊢ sfb1 ≼ sfb2 ⇒
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  holfoot_implies_in_heap_or_null B sfb1 e ⇒
  holfoot_implies_in_heap_or_null B sfb2 e
```

(42) holfoot_implies_in_heap_pred_def 124

```
⊢ holfoot_implies_in_heap_pred p B b e ⇔
  ∀ st st2 h1 h2.
    VAR_RES_STACK_IS_SUBSTATE st2 st ∧
    (st,h1) ∈ var_res_bigstar DISJOINT_FMAP_UNION B ∧
    (st2,h2) ∈ var_res_bigstar DISJOINT_FMAP_UNION b ⇒
    IS_SOME (e st) ∧ p (FDOM h2) (THE (e st))
```

(43) holfoot_implies_in_heap___implies_unequal 125

```
⊢ b1 ⊎ b2 ≤ sfb ∧ holfoot_implies_in_heap sfb b1 e1 ∧
  holfoot_implies_in_heap sfb b2 e2 ⇒
  var_res_implies_unequal DISJOINT_FMAP_UNION sfb e1 e2
```

(44) holfoot_implies_in_heap___implies_unequal___null 125

```
⊢ b ≤ sfb ∧ holfoot_implies_in_heap sfb b e ⇒
  var_res_implies_unequal DISJOINT_FMAP_UNION sfb e
    (var_res_exp_const 0)
```

(45) holfoot_implies_in_heap___implies___or_null 125

```
⊢ holfoot_implies_in_heap B b e ⇒
  holfoot_implies_in_heap_or_null B b e
```

(46) holfoot_implies_in_heap___or_null___implies_unequal 125

```
⊢ b1 ⊎ b2 ≤ sfb ∧ holfoot_implies_in_heap sfb b1 e1 ∧
  holfoot_implies_in_heap_or_null sfb b2 e2 ⇒
  var_res_implies_unequal DISJOINT_FMAP_UNION sfb e1 e2
```

(47) holfoot_implies_in_heap___SUB_BAG 125

```
⊢ sfb1 ≤ sfb2 ⇒
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  holfoot_implies_in_heap B sfb1 e ⇒
  holfoot_implies_in_heap B sfb2 e
```

(48) holfoot_new_action_def 122

```
⊢ holfoot_new_action me v tagL s =
  if
    ¬var_res_sl___has_write_permission v (FST s) ∨
    ¬IS_SOME (me (FST s))
  then
    NONE
  else
    (let m = THE (me (FST s))
     in
       SOME
         (λs′.
            ∃n XL.
              n ≠ 0 ∧
              (∀m′. n ≤ m′ ∧ m′ < n + m ⇒ m′ ∉ FDOM (SND s)) ∧
              (LENGTH XL = m) ∧
              (s′ =
               (FST s |+ (v,n,var_res_write_permission),
                SND s |++
                MAP (λm′. (n + m′,EL m′ XL)) (COUNT_LIST m)))))
```

(49) holfoot_separation_combinator_def 117

```
⊢ holfoot_separation_combinator =
  VAR_RES_COMBINATOR DISJOINT_FMAP_UNION
```

(50) IS_SEPARATION_ALGEBRA___holfoot_separation_combinator 117

```
⊢ IS_SEPARATION_ALGEBRA holfoot_separation_combinator (FEMPTY,FEMPTY)
```

(51) IS_SEPARATION_COMBINATOR___holfoot_separation_combinator 117

```
⊢ IS_SEPARATION_COMBINATOR holfoot_separation_combinator
```

(52) VAR_RES_FRAME_SPLIT___data_array___data_array___SAME_EXP_LENGTH 129

```
⊢ set (MAP FST data2) ⊆ set (MAP FST data1) ∧
  ALL_DISTINCT (MAP FST data2) ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) n ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context
     (BAG_INSERT (holfoot_ap_data_array e n data1) sfb_split)
     (BAG_INSERT (holfoot_ap_data_array e n data2) sfb_imp)
     sfb_restP ⇐⇒
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_data_array e n data1) sfb_context)
     sfb_split
     (BAG_INSERT
        (var_res_bool_proposition DISJOINT_FMAP_UNION
           (EVERY (λx. MEM x data1) data2)) sfb_imp) sfb_restP)
```

(53) VAR_RES_FRAME_SPLIT___data_list_seg___REMOVE_START 128

```
⊢ holfoot_implies_in_heap_or_null (sfb_split ⊎ sfb_context)
    (sfb_split ⊎ sfb_context) e3 ⇒
  set (MAP FST data2) ⊆ set (MAP FST data1) ∧
  (ALL_DISTINCT (tl::MAP FST data1) ⇒ ALL_DISTINCT (MAP FST data2)) ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e1 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e2 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e3 ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data1 e2) sfb_split)
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data2 e3) sfb_imp)
     sfb_restP ⇐⇒
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data1 e2)
        sfb_context) sfb_split
     (BAG_INSERT
        (var_res_bool_proposition DISJOINT_FMAP_UNION
           (EVERY
              (λx.
                 MEM (FST x,TAKE (LENGTH (SND (HD data1))) (SND x))
                   data1) data2))
        (BAG_INSERT
           (holfoot_ap_data_list_seg tl e2
              (MAP
                 (λx. (FST x,DROP (LENGTH (SND (HD data1))) (SND x)))
                 data2) e3) sfb_imp)) sfb_restP)
```

(54) VAR_RES_FRAME_SPLIT___data_list_seg___SAME_START_END___REMOVE 127

```
⊢ (set (MAP FST data2) ⊆ set (MAP FST data1) ∧
   ALL_DISTINCT (MAP FST data2)) ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e1 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e2 ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data1 e2) sfb_split)
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data2 e2) sfb_imp)
     sfb_restP ⇐⇒
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data1 e2)
        sfb_context) sfb_split
     (BAG_INSERT
        (var_res_bool_proposition DISJOINT_FMAP_UNION
           (EVERY (λx. MEM x data1) data2)) sfb_imp) sfb_restP)
```

(55) VAR_RES_FRAME_SPLIT___data_tree___SAME_EXP___REMOVE 129

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context
     (BAG_INSERT (holfoot_ap_data_tree tagL e (dtagL,data1))
        sfb_split)
     (BAG_INSERT (holfoot_ap_data_tree tagL e (dtagL,data2)) sfb_imp)
     sfb_restP ⟺
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_data_tree tagL e (dtagL,data1))
        sfb_context) sfb_split
     (BAG_INSERT
        (var_res_bool_proposition DISJOINT_FMAP_UNION
           (data1 = data2)) sfb_imp) sfb_restP)
```

(56) VAR_RES_FRAME_SPLIT___points_to___data_list_seg 127

```
⊢ var_res_implies_unequal DISJOINT_FMAP_UNION
    (sfb_context ⊎ BAG_INSERT (holfoot_ap_points_to e1 L) sfb_split)
    e1 e2 ⇒
  tl ∈ FDOM L ∧ set (MAP FST data) ⊆ FDOM L ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e1 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e2 ∧
  FEVERY
    (λ x.
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
         (SET_OF_BAG (wpb ⊎ rpb)) (SND x)) L ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context (BAG_INSERT (holfoot_ap_points_to e1 L) sfb_split)
     (BAG_INSERT (holfoot_ap_data_list_seg tl e1 data e2) sfb_imp)
     sfb_restP ⟺
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_points_to e1 L) sfb_context) sfb_split
     (LIST_TO_BAG
        (MAP
           (λ x.
              var_res_prop_equal DISJOINT_FMAP_UNION (L ' (FST x))
                (var_res_exp_const (HD (SND x)))) data) ⊎
      BAG_INSERT
        (var_res_bool_proposition DISJOINT_FMAP_UNION
           (EVERY (λ x. ¬NULL (SND x)) data ∧
            ALL_DISTINCT (tl::MAP FST data)))
        (BAG_INSERT
           (holfoot_ap_data_list_seg tl (L ' tl)
              (MAP (λ x. (FST x,TL (SND x))) data) e2) sfb_imp))
     sfb_restP)
```

(57) VAR_RES_FRAME_SPLIT___points_to___data_tree 128

```
⊢ set (tagL ++ dtagL) ⊆ FDOM L ∧ ¬NULL tagL ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ∧
  FEVERY
    (λ x.
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
         (SET_OF_BAG (wpb ⊎ rpb)) (SND x)) L ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context (BAG_INSERT (holfoot_ap_points_to e L) sfb_split)
     (BAG_INSERT (holfoot_ap_data_tree tagL e (dtagL,data)) sfb_imp)
     sfb_restP ⟺
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_points_to e L) sfb_context) sfb_split
     (BAG_INSERT
        (asl_exists_list dtagL
           (λ v.
              asl_exists_list tagL
                (λ lL.
                   asl_exists_list tagL
                     (λ tL.
                        asl_trivial_cond (data = node v tL)
                          (asl_bigstar_list
                             holfoot_separation_combinator
                             (MAP
                                (λ x.
                                   var_res_prop_equal
                                     DISJOINT_FMAP_UNION
                                     (L ' (FST x))
                                     (var_res_exp_const (SND x)))
                                (ZIP (tagL ++ dtagL,lL ++ v)) ++
                              MAP
                                (λ lt.
                                   holfoot_ap_data_tree tagL
                                     (var_res_exp_const (FST lt))
                                     (dtagL,SND lt))
                                (ZIP (lL,tL))))))))) sfb_imp)
     sfb_restP)
```

(58) VAR_RES_FRAME_SPLIT___points_to___points_to___SUBMAP 126

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ∧
  FEVERY
    (λ x.
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
         (SET_OF_BAG (wpb ⊎ rpb)) (SND x)) L ∧
  FEVERY
    (λ x.
       ¬MEM (FST x) l' ∨
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
         (SET_OF_BAG (wpb ⊎ rpb)) (SND x)) L' ∧
  FEVERY (λ (t,a). t ∈ FDOM L ∧ (MEM t l' ∨ (a = L ' t))) L' ∧
  EVERY (λ t. t ∈ FDOM L') l' ⇒
  (VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     sfb_context (BAG_INSERT (holfoot_ap_points_to e L) sfb_split)
     (BAG_INSERT (holfoot_ap_points_to e L') sfb_imp) sfb_restP ⟺
   VAR_RES_FRAME_SPLIT DISJOINT_FMAP_UNION sr (wpb,rpb) wpb'
     (BAG_INSERT (holfoot_ap_points_to e L) sfb_context) sfb_split
     (BAG_INSERT
        (asl_bigstar_list holfoot_separation_combinator
           (MAP
              (λ t.
                 var_res_prop_equal DISJOINT_FMAP_UNION (L ' t)
                   (L' ' t)) l' ++
            [var_res_prop_stack_true DISJOINT_FMAP_UNION])) sfb_imp)
     sfb_restP)
```

(59) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___data_list_seg 119, 130

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
    startExp ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
    endExp ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
    (holfoot_ap_data_list_seg tl startExp data endExp)
```

(60) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___holfoot_ap_data_array 121

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs e ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs n ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
    (holfoot_ap_data_array e n data)
```

(61) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___holfoot_ap_data_tree 120

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
    startExp ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
    (holfoot_ap_data_tree tagL startExp data)
```

(62) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___points_to 118

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs e1 ∧
  FEVERY
    (λ x.
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
         (SND x)) L ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
    (holfoot_ap_points_to e1 L)
```

(63) var_res_prop_varlist_update___holfoot_ap_data_array 121

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ∧
  IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS n) ⇒
  (var_res_prop_varlist_update vcL (holfoot_ap_data_array e n data) =
   holfoot_ap_data_array (var_res_exp_varlist_update vcL e)
     (var_res_exp_varlist_update vcL n) data)
```

(64) var_res_prop_varlist_update___holfoot_ap_data_list_seg 119

```
⊢ IS_SOME
    (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS startExp) ∧
  IS_SOME
    (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS endExp) ⇒
  (var_res_prop_varlist_update vcL
     (holfoot_ap_data_list_seg tl startExp data endExp) =
   holfoot_ap_data_list_seg tl
     (var_res_exp_varlist_update vcL startExp) data
     (var_res_exp_varlist_update vcL endExp))
```

(65) var_res_prop_varlist_update___holfoot_ap_data_tree 120

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  (var_res_prop_varlist_update vcL
     (holfoot_ap_data_tree tagL e data) =
   holfoot_ap_data_tree tagL (var_res_exp_varlist_update vcL e) data)
```

(66) var_res_prop_varlist_update___holfoot_ap_points_to 118

```
⊢ var_res_prop_varlist_update vcL (holfoot_ap_points_to e L) =
  holfoot_ap_points_to (var_res_exp_varlist_update vcL e)
    (var_res_exp_varlist_update vcL o_f L)
```

C.2 separationLogicTheory

(67) asla_annihilation_def 70

```
⊢ asla_annihilation f p = best_local_action f p (asl_emp f)
```

(68) asla_annihilation_PRECISE_IN_STATE_THM 70

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  (asla_annihilation f P q =
   (let v s0 = ∃ s1. s1 ∈ P ∧ (SOME q = f (SOME s0) (SOME s1))
    in
      if v = ∅ then NONE else if SING v then SOME v else SOME ∅))
```

(69) asla_assume_def 72

```
⊢ asla_assume f P =
  (λ s.
     if s ∈ P then
       SOME {s}
     else if s ∈ ASL_INTUITIONISTIC_NEGATION f P then
       SOME ∅
     else
       NONE)
```

(70) asla_check_def 75

```
⊢ asla_check f a1 a2 =
  (λ s.
     if
       ∃ s1 s2.
         (SOME s = f (SOME s1) (SOME s2)) ∧ IS_SOME (a1 s1) ∧
         IS_SOME (a2 s2)
     then
       SOME {s}
     else
       NONE)
```

(71) asla_choice_REWRITE 66

```
⊢ asla_choice actions =
  (λ x. SUP_fasl_order (IMAGE (λ f. f x) actions))
```

(72) asla_diverge_def 66

```
⊢ asla_diverge = (λ s. SOME ∅)
```

(73) asla_fail_def 66

```
⊢ asla_fail = (λ s. NONE)
```

(74) asla_materialisation_def 70

```
⊢ asla_materialisation f p = best_local_action f (asl_emp f) p
```

(75) asla_seq_def 66

```
⊢ asla_seq a1 a2 =
  (λ s.
     if a1 s = NONE then
       NONE
     else
       SUP_fasl_order (IMAGE a2 (THE (a1 s))))
```

(76) asla_skip_def 66

```
⊢ asla_skip = (λ s. SOME {s})
```

(77) asl_and_def 63

```
⊢ asl_and = (λ P Q s. s ∈ P ∧ s ∈ Q)
```

(78) ASL_ATOMIC_ACTION_SEM_def 75

```
⊢ (ASL_ATOMIC_ACTION_SEM (f,lock_env) (asl_aa_pc pc) =
   EVAL_asl_prim_command f pc) ∧
  (ASL_ATOMIC_ACTION_SEM (f,lock_env) (asl_aa_check pc1 pc2) =
   asla_check f (EVAL_asl_prim_command f pc1)
     (EVAL_asl_prim_command f pc2)) ∧
  (ASL_ATOMIC_ACTION_SEM (f,lock_env) (asl_aa_prolaag l) =
   asla_materialisation f (lock_env l)) ∧
  (ASL_ATOMIC_ACTION_SEM (f,lock_env) (asl_aa_verhoog l) =
   asla_annihilation f (lock_env l))
```

(79) asl_bigstar_list_REWRITE 97

```
⊢ (∀ f. asl_bigstar_list f [] = asl_emp f) ∧
  ∀ f h l.
    asl_bigstar_list f (h::l) = asl_star f h (asl_bigstar_list f l)
```

(80) asl_emp_ALGEBRA 64

```
⊢ IS_SEPARATION_ALGEBRA f u ⇒ (asl_emp f = {u})
```

(81) asl_emp_def 62

```
⊢ asl_emp f = (λ u. ∃ x. f (SOME u) (SOME x) = SOME x)
```

(82) asl_emp_DISJOINT_FMAP_UNION 63, 117

```
⊢ asl_emp DISJOINT_FMAP_UNION = {FEMPTY}
```

(83) asl_exists_def 63

```
⊢ (asl_exists) = (λP s. ∃x. s ∈ P x)
```

(84) asl_false_def 63

```
⊢ asl_false = ∅
```

(85) asl_forall_def 63

```
⊢ (asl_forall) = (λP s. ∀x. s ∈ P x)
```

(86) ASL_INFERENCE_asl_quant 81

```
⊢ (ASL_PROGRAM_HOARE_TRIPLE xenv penv (asl_exists x. P x) p Q′ ⟺
   ∀x. ASL_PROGRAM_HOARE_TRIPLE xenv penv (P x) p Q′) ∧
  (ASL_PROGRAM_HOARE_TRIPLE xenv penv P′ p (asl_forall x. Q x) ⟺
   ∀x. ASL_PROGRAM_HOARE_TRIPLE xenv penv P′ p (Q x)) ∧
  ((∃x. ASL_PROGRAM_HOARE_TRIPLE xenv penv (P x) p Q′) ⇒
   ASL_PROGRAM_HOARE_TRIPLE xenv penv (asl_forall x. P x) p Q′) ∧
  ((∃x. ASL_PROGRAM_HOARE_TRIPLE xenv penv P′ p (Q x)) ⇒
   ASL_PROGRAM_HOARE_TRIPLE xenv penv P′ p (asl_exists x. Q x))
```

(87) ASL_INFERENCE_assume 81

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  asl_predicate_IS_DECIDED (FST xenv) P c ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_prim_command (asl_pc_assume c))
    (asl_and P (EVAL_asl_predicate (FST xenv) c))
```

(88) ASL_INFERENCE_assume_seq_STRONG 84, 103

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (ASL_PROGRAM_HOARE_TRIPLE xenv penv P
     (asl_prog_seq (asl_prog_prim_command (asl_pc_assume c)) prog)
     Q ⟺
   asl_predicate_IS_DECIDED (FST xenv) P c ∧
   ASL_PROGRAM_HOARE_TRIPLE xenv penv
     (asl_and P (EVAL_asl_predicate (FST xenv) c)) prog Q)
```

(89) ASL_INFERENCE_COMBINE_INTER 81

```
⊢ (∀P Q. (P,Q) ∈ PQ ⇒ ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q) ∧
  PQ ≠ ∅ ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv (BIGINTER (IMAGE FST PQ)) prog
    (BIGINTER (IMAGE SND PQ))
```

(90) ASL_INFERENCE_COMBINE_UNION 81

```
⊢ (∀P Q. (P,Q) ∈ PQ ⇒ ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q) ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv (BIGUNION (IMAGE FST PQ)) prog
    (BIGUNION (IMAGE SND PQ))
```

(91) ASL_INFERENCE_FRAME 80

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q ⇒
  ∀x.
    ASL_PROGRAM_HOARE_TRIPLE xenv penv (asl_star (FST xenv) P x) prog
      (asl_star (FST xenv) Q x)
```

(92) ASL_INFERENCE_prog_choice_STRONG 82

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_choice p1 p2) Q ⟺
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P p1 Q ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P p2 Q
```

(93) ASL_INFERENCE_prog_cond 82

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_seq (asl_prog_assume c) pTrue) Q ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_seq (asl_prog_assume (asl_pred_neg c)) pFalse) Q ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_cond c pTrue pFalse)
    Q
```

(94) ASL_INFERENCE_prog_cond_critical_section 83

```
⊢ IS_SEPARATION_COMBINATOR f ∧ (lock_env l = R) ∧
  asl_predicate_IS_DECIDED f (asl_star f P R) c ∧
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv
    (asl_and (asl_star f P R) (EVAL_asl_predicate f c)) p
    (asl_star f Q R) ∧ ASL_IS_PRECISE f R ⇒
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv P
    (asl_prog_cond_critical_section l c p) Q
```

(95) ASL_INFERENCE_prog_critical_section 82

```
⊢ IS_SEPARATION_COMBINATOR f ∧ (lock_env l = R) ∧
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv (asl_star f P R) p
    (asl_star f Q R) ∧ ASL_IS_PRECISE f R ⇒
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv P
    (asl_prog_critical_section l p) Q
```

(96) ASL_INFERENCE_prog_diverge 81

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P asl_prog_diverge Q
```

(97) ASL_INFERENCE_prog_kleene_star_STRONG 82

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_kleene_star p) P ⟺
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P p P
```

(98) ASL_INFERENCE_prog_lock_declaration 83

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv P p Q ∧
  (lock_env l = R) ∧ ASL_IS_PRECISE f R ⇒
  ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv (asl_star f P R)
    (asl_prog_lock_declaration l p) (asl_star f Q R)
```

(99) ASL_INFERENCE_prog_parallel 82, 87

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P1 p1 Q1 ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P2 p2 Q2 ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv (asl_star (FST xenv) P1 P2)
    (asl_prog_parallel p1 p2) (asl_star (FST xenv) Q1 Q2)
```

(100) ASL_INFERENCE_prog_procedure_call 83, 88

```
⊢ name ∈ FDOM penv ⇒
  (ASL_PROGRAM_HOARE_TRIPLE xenv penv P (penv ' name arg) Q ⟺
   ASL_PROGRAM_HOARE_TRIPLE xenv penv P
     (asl_prog_procedure_call name arg) Q)
```

(101) ASL_INFERENCE_prog_quant_best_local_action 81

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv (qP arg)
    (asl_prog_quant_best_local_action qP qQ) (qQ arg)
```

(102) ASL_INFERENCE_prog_quant_best_local_action2 81

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (∃ arg.
     ASL_PROGRAM_HOARE_TRIPLE xenv penv P
       (asl_prog_best_local_action (qP arg) (qQ arg)) Q) ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_quant_best_local_action qP qQ) Q
```

(103) ASL_INFERENCE_prog_seq_STRONG 82, 83

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_seq p1 p2) R ⟺
  ∃ Q.
    ASL_PROGRAM_HOARE_TRIPLE xenv penv P p1 Q ∧
    ASL_PROGRAM_HOARE_TRIPLE xenv penv Q p2 R
```

(104) ASL_INFERENCE_prog_skip 81

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P asl_prog_skip P
```

(105) ASL_INFERENCE_prog_while 82, 87

```
⊢ asl_predicate_IS_DECIDED (FST xenv) P c ∧
  IS_SEPARATION_COMBINATOR (FST xenv) ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv
    (asl_and P (EVAL_asl_predicate (FST xenv) c)) p P ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_while c p)
    (asl_and P (EVAL_asl_predicate (FST xenv) (asl_pred_neg c)))
```

(106) ASL_INFERENCE_prog_while_frame 87

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  (∀ x.
     ASL_PROGRAM_HOARE_TRIPLE xenv penv (I x)
       (asl_prog_seq (asl_prog_assume c) p) (I x)) ∧
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_block
       (asl_prog_quant_best_local_action I I::
            asl_prog_assume (asl_pred_neg c)::pL)) Q ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P
    (asl_prog_block (asl_prog_while c p::pL)) Q
```

(107) ASL_INFERENCE_prog_while_frame___loop_spec 82, 87

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  (∀ x.
     ASL_PROGRAM_HOARE_TRIPLE xenv penv (P x)
       (asl_prog_block (asl_prog_assume (asl_pred_neg c)::pL))
       (Q x)) ∧
  (∀ x.
     ASL_PROGRAM_HOARE_TRIPLE xenv penv (P x)
       (asl_prog_block
          [asl_prog_assume c; p;
           asl_prog_quant_best_local_action P Q]) (Q x)) ⇒
  ∀ x.
    ASL_PROGRAM_HOARE_TRIPLE xenv penv (P x)
      (asl_prog_block (asl_prog_while c p::pL)) (Q x)
```

(108) ASL_INFERENCE_sp 84

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_seq p1 p2) Q ⟺
  ∃ sp.
    (asl_sp_opt xenv penv P p1 = SOME sp) ∧
    ASL_PROGRAM_HOARE_TRIPLE xenv penv sp p2 Q
```

(109) ASL_INFERENCE_STRENGTHEN 81

```
⊢ P2 ⊆ P1 ∧ Q1 ⊆ Q2 ∧ ASL_PROGRAM_HOARE_TRIPLE xenv penv P1 prog Q1 ⇒
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P2 prog Q2
```

(110) ASL_INFERENCE_wlp 84

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_seq p1 p2) Q ⟺
  ASL_PROGRAM_HOARE_TRIPLE xenv penv P p1 (asl_wlp xenv penv p2 Q)
```

(111) ASL_INFERENCE___choose_constants 83

```
⊢ IS_SEPARATION_COMBINATOR f ∧ (LENGTH cL = LENGTH L) ∧
  EVERY
    (λ (e,c).
       (∀ s. s ∈ P ⇒ (e s = SOME c)) ∧
       ∀ s1 s2.
         (e s1 = SOME c) ∧ ASL_IS_SUBSTATE f s1 s2 ⇒ (e s2 = SOME c))
    (ZIP (L,cL)) ⇒
  (ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv P
     (asl_prog_seq (asl_prog_choose_constants prog L) prog2) Q ⟺
   ASL_PROGRAM_HOARE_TRIPLE (f,lock_env) penv P
     (asl_prog_seq (prog cL) prog2) Q)
```

(112) ASL_INFERENCE___PROCEDURE_SPEC___DIRECT 88

```
⊢ ASL_PROCEDURE_SPEC___wellformed_spec penv specs ∧
  (∀ penv'.
     ASL_PROCEDURE_SPEC xenv penv' specs ⇒
     ∀ name abst.
       MEM (name,abst) specs ⇒
       ∀ arg.
         ASL_PROGRAM_IS_ABSTRACTION xenv penv' (penv ' name arg)
           (abst arg)) ⇒
  ASL_PROCEDURE_SPEC xenv penv specs
```

(113) ASL_INTUITIONISTIC_NEGATION_def 72

```
⊢ ASL_INTUITIONISTIC_NEGATION f P =
  (λ s. ∀ s'. ASL_IS_SEPARATE f s s' ⇒ THE (f (SOME s) (SOME s')) ∉ P)
```

(114) ASL_IS_INTUITIONISTIC_def 71

```
⊢ ASL_IS_INTUITIONISTIC f P ⟺ (asl_star f P 𝒰(:α) = P)
```

(115) ASL_IS_INTUITIONISTIC___EVAL_asl_predicate 73

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_IS_INTUITIONISTIC f (EVAL_asl_predicate f p)
```

(116) ASL_IS_INTUITIONISTIC___REWRITE 71

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ∀ P.
    ASL_IS_INTUITIONISTIC f P ⟺
    ∀ s1 s2. s1 ∈ P ∧ ASL_IS_SUBSTATE f s1 s2 ⇒ s2 ∈ P
```

(117) ASL_IS_LOCAL_ACTION_def 67

```
⊢ ASL_IS_LOCAL_ACTION f op ⟺
  ∀ s1 s2.
    ASL_IS_SEPARATE f s1 s2 ⇒
    fasl_order (op (THE (f (SOME s1) (SOME s2))))
      (fasl_star f (op s1) (SOME {s2}))
```

(118) ASL_IS_LOCAL_ACTION___asla_assume 72

```
⊢ IS_SEPARATION_COMBINATOR f ∧ ASL_IS_INTUITIONISTIC f P ⇒
  ASL_IS_LOCAL_ACTION f (asla_assume f P)
```

(119) ASL_IS_LOCAL_ACTION___asla_choice 68, 76

```
⊢ (∀ op. op ∈ OP ⇒ ASL_IS_LOCAL_ACTION f op) ⇒
  ASL_IS_LOCAL_ACTION f (asla_choice OP)
```

(120) ASL_IS_LOCAL_ACTION___asla_diverge 68

```
⊢ ASL_IS_LOCAL_ACTION f asla_diverge
```

(121) ASL_IS_LOCAL_ACTION___asla_fail 68, 73

```
⊢ ASL_IS_LOCAL_ACTION f asla_fail
```

(122) ASL_IS_LOCAL_ACTION___asla_seq 68, 76

```
⊢ ASL_IS_LOCAL_ACTION f a1 ∧ ASL_IS_LOCAL_ACTION f a2 ⇒
  ASL_IS_LOCAL_ACTION f (asla_seq a1 a2)
```

(123) ASL_IS_LOCAL_ACTION___asla_skip 68

```
⊢ ASL_IS_LOCAL_ACTION f asla_skip
```

(124) ASL_IS_LOCAL_ACTION___ASL_ATOMIC_ACTION_SEM 76

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_IS_LOCAL_ACTION f (ASL_ATOMIC_ACTION_SEM (f,lock_env) aa)
```

(125) ASL_IS_LOCAL_ACTION___ASL_PROGRAM_SEM 76, 80

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_IS_LOCAL_ACTION (FST xenv) (ASL_PROGRAM_SEM xenv penv prog)
```

(126) ASL_IS_LOCAL_ACTION___ASL_TRACE_SEM 76

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_IS_LOCAL_ACTION f (ASL_TRACE_SEM (f,lock_env) t)
```

(127) ASL_IS_LOCAL_ACTION___EVAL_asl_prim_command 73

$\vdash$ ASL_IS_LOCAL_ACTION f (EVAL_asl_prim_command f c)

(128) ASL_IS_LOCAL_ACTION___HOARE_TRIPLE 67

$\vdash$ ASL_IS_LOCAL_ACTION f a $\iff$
  $\forall P\ Q$.
    HOARE_TRIPLE P a Q $\Rightarrow$
    $\forall x$. HOARE_TRIPLE (asl_star f P x) a (asl_star f Q x)

(129) ASL_IS_LOCAL_ACTION___materialisation_annihilation 70

$\vdash$ IS_SEPARATION_COMBINATOR f $\Rightarrow$
  ASL_IS_LOCAL_ACTION f (asla_materialisation f p) $\wedge$
  ASL_IS_LOCAL_ACTION f (asla_annihilation f p)

(130) ASL_IS_LOCAL_ACTION___simple_heap_examples 68

$\vdash$ $\neg$ASL_IS_LOCAL_ACTION DISJOINT_FMAP_UNION
    (λh. **if** h = FEMPTY **then** SOME $\{h\}$ **else** NONE) $\wedge$
  ASL_IS_LOCAL_ACTION DISJOINT_FMAP_UNION
    (λh. **if** h = FEMPTY **then** SOME $\{h\}$ **else** SOME $\emptyset$)

(131) ASL_IS_PRECISE_def 70

$\vdash$ ASL_IS_PRECISE f p $\iff$
  $\forall x\ y_1\ y_2$.
    $y_1 \in p \wedge y_2 \in p \wedge$ ASL_IS_SUBSTATE f y_1 x $\wedge$
    ASL_IS_SUBSTATE f y_2 x $\Rightarrow$
    $(y_1 = y_2)$

(132) ASL_IS_SEPARATE_def 62

$\vdash$ ASL_IS_SEPARATE f x_1 x_2 $\iff$ IS_SOME (f (SOME x_1) (SOME x_2))

(133) ASL_IS_SUBSTATE_def 62

$\vdash$ ASL_IS_SUBSTATE f s_0 s_2 $\iff$ $\exists s_1$. f (SOME s_0) (SOME s_1) = SOME s_2

(134) asl_magic_wand_def 63

$\vdash$ asl_magic_wand f P Q =
  (λs. $\forall s_1\ s_2$. (SOME s_2 = f (SOME s_1) (SOME s)) $\wedge$ $s_1 \in P \Rightarrow s_2 \in Q$)

(135) asl_neg_def 63

$\vdash$ asl_neg = ($\lambda P\ s.\ s \notin P$)

(136) asl_or_def 63

```
⊢ asl_or = (λP Q s. s ∈ P ∨ s ∈ Q)
```

(137) asl_predicate_IS_DECIDED_def 72

```
⊢ asl_predicate_IS_DECIDED f P c ⟺
  ∀s.
    s ∈ P ⇒
    s ∈ EVAL_asl_predicate f c ∨
    s ∈ EVAL_asl_predicate f (asl_pred_neg c)
```

(138) ASL_PROGRAM_HOARE_TRIPLE_def 76

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q ⟺
  HOARE_TRIPLE P (ASL_PROGRAM_SEM xenv penv prog) Q
```

(139) ASL_PROGRAM_IS_ABSTRACTION_def 76

```
⊢ ASL_PROGRAM_IS_ABSTRACTION xenv penv prog1 prog2 ⟺
  fasl_action_order (ASL_PROGRAM_SEM xenv penv prog1)
    (ASL_PROGRAM_SEM xenv penv prog2)
```

(140) ASL_PROGRAM_IS_ABSTRACTION___ALTERNATIVE_DEF 76, 81, 85

```
⊢ ASL_PROGRAM_IS_ABSTRACTION xenv penv prog1 prog2 ⟺
  ∀P Q.
    ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog2 Q ⇒
    ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog1 Q
```

(141) ASL_PROGRAM_IS_ABSTRACTION___asl_prog_lock_declaration 86, 105

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  ASL_IS_PRECISE (FST xenv) (SND xenv l) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_lock_declaration l p)
    (asl_prog_block
       [asl_prog_prim_command
          (asl_pc_shallow_command
             (λf. asla_annihilation f (SND xenv l))); p;
        asl_prog_prim_command
          (asl_pc_shallow_command
             (λf. asla_materialisation f (SND xenv l)))])
```

(142) ASL_PROGRAM_IS_ABSTRACTION___assume_and 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_assume (asl_pred_and P1 P2))
    (asl_prog_seq (asl_prog_assume P1) (asl_prog_assume P2))
```

(143) ASL_PROGRAM_IS_ABSTRACTION__assume_and__LOST_INFORMATION 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  asl_predicate_IS_DECIDED_IN_STATE (FST xenv) s P1 ∧
  (s ∈ EVAL_asl_predicate (FST xenv) P1 ⇒
   asl_predicate_IS_DECIDED_IN_STATE (FST xenv) s P2) ⇒
  (ASL_PROGRAM_SEM xenv penv
     (asl_prog_seq (asl_prog_assume P1) (asl_prog_assume P2)) s =
   ASL_PROGRAM_SEM xenv penv (asl_prog_assume (asl_pred_and P1 P2))
     s)
```

(144) ASL_PROGRAM_IS_ABSTRACTION__assume_neg_and 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_assume (asl_pred_neg (asl_pred_and P1 P2)))
    (asl_prog_assume
       (asl_pred_or (asl_pred_neg P1) (asl_pred_neg P2)))
```

(145) ASL_PROGRAM_IS_ABSTRACTION__assume_neg_neg 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_assume (asl_pred_neg (asl_pred_neg P)))
    (asl_prog_assume P)
```

(146) ASL_PROGRAM_IS_ABSTRACTION__assume_neg_or 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_assume (asl_pred_neg (asl_pred_or P1 P2)))
    (asl_prog_assume
       (asl_pred_and (asl_pred_neg P1) (asl_pred_neg P2)))
```

(147) ASL_PROGRAM_IS_ABSTRACTION__assume_or 85

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_assume (asl_pred_or P1 P2))
    (asl_prog_choice (asl_prog_assume P1) (asl_prog_assume P2))
```

(148) ASL_PROGRAM_IS_ABSTRACTION__best_local_action 86, 100

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (ASL_PROGRAM_IS_ABSTRACTION xenv penv prog
     (asl_prog_best_local_action P Q) ⇔
   ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q)
```

(149) ASL_PROGRAM_IS_ABSTRACTION___choice 85

```
⊢ ASL_PROGRAM_IS_ABSTRACTION xenv penv prog1 prog1' ∧
  ASL_PROGRAM_IS_ABSTRACTION xenv penv prog2 prog2' ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv (asl_prog_choice prog1 prog2)
    (asl_prog_choice prog1' prog2')
```

(150) ASL_PROGRAM_IS_ABSTRACTION___cond 86

```
⊢ ASL_PROGRAM_IS_ABSTRACTION xenv penv prog1 prog1' ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv prog2 prog2' ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv (asl_prog_cond c prog1 prog2)
    (asl_prog_cond c prog1' prog2')
```

(151) ASL_PROGRAM_IS_ABSTRACTION___kleene_star 86

```
⊢ ASL_PROGRAM_IS_ABSTRACTION xenv penv prog prog' ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv (asl_prog_kleene_star prog)
    (asl_prog_kleene_star prog')
```

(152) ASL_PROGRAM_IS_ABSTRACTION___parallel 87

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ∧
  ASL_PROGRAM_IS_ABSTRACTION xenv penv p1
    (asl_prog_quant_best_local_action qP1 qQ1) ∧
  ASL_PROGRAM_IS_ABSTRACTION xenv penv p2
    (asl_prog_quant_best_local_action qP2 qQ2) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv (asl_prog_parallel p1 p2)
    (asl_prog_quant_best_local_action
       (λ(a1,a2). asl_star (FST xenv) (qP1 a1) (qP2 a2))
       (λ(a1,a2). asl_star (FST xenv) (qQ1 a1) (qQ2 a2)))
```

(153) ASL_PROGRAM_IS_ABSTRACTION___prog_critical_section_2 .... 86, 105

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  ASL_PROGRAM_IS_ABSTRACTION xenv penv
    (asl_prog_critical_section l p)
    (asl_prog_block
       [asl_prog_prim_command
          (asl_pc_shallow_command
             (λf. asla_materialisation f (SND xenv l))); p;
        asl_prog_prim_command
          (asl_pc_shallow_command
             (λf. asla_annihilation f (SND xenv l)))])
```

(154) ASL_PROGRAM_IS_ABSTRACTION___quant_best_local_action 86, 100

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (ASL_PROGRAM_IS_ABSTRACTION xenv penv prog
     (asl_prog_quant_best_local_action P Q) ⇔
   ∀arg. ASL_PROGRAM_HOARE_TRIPLE xenv penv (P arg) prog (Q arg))
```

(155) ASL_PROGRAM_IS_ABSTRACTION___REFL 85

⊢ ASL_PROGRAM_IS_ABSTRACTION *xenv penv p p*

(156) ASL_PROGRAM_IS_ABSTRACTION___seq 85

⊢ ASL_PROGRAM_IS_ABSTRACTION *xenv penv* $prog_1$ $prog'_1$ ∧
ASL_PROGRAM_IS_ABSTRACTION *xenv penv* $prog_2$ $prog'_2$ ⇒
ASL_PROGRAM_IS_ABSTRACTION *xenv penv* (asl_prog_seq $prog_1$ $prog_2$)
(asl_prog_seq $prog'_1$ $prog'_2$)

(157) ASL_PROGRAM_IS_ABSTRACTION___TRANSITIVE 85

⊢ ASL_PROGRAM_IS_ABSTRACTION *xenv penv* p_1 p_2 ⇒
ASL_PROGRAM_IS_ABSTRACTION *xenv penv* p_2 p_3 ⇒
ASL_PROGRAM_IS_ABSTRACTION *xenv penv* p_1 p_3

(158) ASL_PROGRAM_IS_ABSTRACTION___while 86

⊢ ASL_PROGRAM_IS_ABSTRACTION *xenv penv prog* $prog'$ ⇒
ASL_PROGRAM_IS_ABSTRACTION *xenv penv* (asl_prog_while *c prog*)
(asl_prog_while *c* $prog'$)

(159) ASL_PROGRAM_SEM_def 75

⊢ ASL_PROGRAM_SEM *xenv penv prog* =
ASL_TRACE_SET_SEM *xenv* (ASL_PROGRAM_TRACES *penv prog*)

(160) ASL_PROGRAM_SEM___prog_seq 77

⊢ ASL_PROGRAM_SEM *xenv penv* (asl_prog_seq $prog_1$ $prog_2$) =
asla_seq (ASL_PROGRAM_SEM *xenv penv* $prog_1$)
(ASL_PROGRAM_SEM *xenv penv* $prog_2$)

(161) ASL_PROGRAM_TRACES_def 74

⊢ ASL_PROGRAM_TRACES *penv prog* =
BIGUNION (IMAGE (ASL_PROTO_TRACES_EVAL *penv*) *prog*)

(162) asl_prog_choice_def 78

⊢ asl_prog_choice = (∪)

(163) asl_prog_choose_constants_def 79

```
⊢ asl_prog_choose_constants prog expL =
  asl_prog_ndet
    (IMAGE
       (λ constL.
          asl_prog_seq
            (asl_prog_prim_command
               (asl_pc_assume
                  (asl_pred_bigand
                     (MAP
                        (λ x.
                           asl_pred_prim
                             (λ f s. FST x s = SOME (SND x)))
                        (ZIP (expL,constL))))) (prog constL))
       (λ l. LENGTH l = LENGTH expL))
```

(164) asl_prog_cond_critical_section_def 78

```
⊢ asl_prog_cond_critical_section l c p =
  asl_prog_critical_section l
    (asl_prog_seq (asl_prog_prim_command (asl_pc_assume c)) p)
```

(165) asl_prog_cond_def 78

```
⊢ asl_prog_cond c pTrue pFalse =
  asl_prog_choice
    (asl_prog_seq (asl_prog_prim_command (asl_pc_assume c)) pTrue)
    (asl_prog_seq
       (asl_prog_prim_command (asl_pc_assume (asl_pred_neg c)))
       pFalse)
```

(166) asl_prog_critical_section_def 77

```
⊢ asl_prog_critical_section l p = IMAGE (asl_pt_critical_section l) p
```

(167) asl_prog_ext_procedure_call_def 106

```
⊢ asl_prog_ext_procedure_call name (ref_argL,val_argL) =
  asl_prog_choose_constants
    (λ constL. asl_prog_procedure_call name (ref_argL,constL))
    val_argL
```

(168) asl_prog_kleene_star_def 78

```
⊢ asl_prog_kleene_star p = (λ pt. ∃ n. pt ∈ asl_prog_repeat_num n p)
```

(169) asl_prog_lock_declaration_def 77

```
⊢ asl_prog_lock_declaration l p = IMAGE (asl_pt_lock_declaration l) p
```

(170) asl_prog_ndet__HOARE_TRIPLE 82

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P (asl_prog_ndet pset) Q ⟺
  ∀ prog. prog ∈ pset ⇒ ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q
```

(171) asl_prog_parallel_def 77

```
⊢ asl_prog_parallel p₁ p₂ =
  (λ pt.
     ∃ pt₁ pt₂. (pt = asl_pt_parallel pt₁ pt₂) ∧ pt₁ ∈ p₁ ∧ pt₂ ∈ p₂)
```

(172) asl_prog_repeat_num_def 78

```
⊢ (∀ p. asl_prog_repeat_num 0 p = asl_prog_skip) ∧
  ∀ n p.
    asl_prog_repeat_num (SUC n) p =
    (λ pt.
       ∃ pt₁ pt₂.
         (pt = asl_pt_seq pt₁ pt₂) ∧ pt₁ ∈ p ∧
         pt₂ ∈ asl_prog_repeat_num n p)
```

(173) asl_prog_seq_def 77

```
⊢ asl_prog_seq p₁ p₂ =
  (λ pt.
     ∃ pt₁ pt₂.
       (pt = asl_pt_seq pt₁ pt₂) ∧ pt₁ ∈ p₁ ∧
       pt₂ ∈ asl_pt_diverge INSERT p₂)
```

(174) asl_prog_while_def 78

```
⊢ asl_prog_while c p =
  asl_prog_seq
    (asl_prog_kleene_star
       (asl_prog_seq (asl_prog_prim_command (asl_pc_assume c)) p))
    (asl_prog_prim_command (asl_pc_assume (asl_pred_neg c)))
```

(175) ASL_PROTO_TRACES_EVAL_def 74

```
⊢ ASL_PROTO_TRACES_EVAL penv prog =
  (λ t. ∃ n. t ∈ ASL_PROTO_TRACES_EVAL_PROC n penv prog)
```

(176) ASL_PROTO_TRACES_EVAL_PROC_THM 74

```
⊢ (ASL_PROTO_TRACES_EVAL_PROC n penv (asl_pt_prim_command pc) =
    { [asl_aa_pc pc] }) ∧
   (ASL_PROTO_TRACES_EVAL_PROC n penv (asl_pt_seq p1 p2) =
    { t1 ++ t2 |
      t1 ∈ ASL_PROTO_TRACES_EVAL_PROC n penv p1 ∧
      t2 ∈ ASL_PROTO_TRACES_EVAL_PROC n penv p2 }) ∧
   (ASL_PROTO_TRACES_EVAL_PROC n penv (asl_pt_parallel p1 p2) =
    BIGUNION
      { ASL_TRACE_ZIP t1 t2 |
        t1 ∈ ASL_PROTO_TRACES_EVAL_PROC n penv p1 ∧
        t2 ∈ ASL_PROTO_TRACES_EVAL_PROC n penv p2 }) ∧
   (ASL_PROTO_TRACES_EVAL_PROC n penv (asl_pt_lock_declaration l p) =
    IMAGE
      (λ t.
         ASL_TRACE_REMOVE_LOCKS { l }
           ([asl_aa_verhoog l] ++ t ++ [asl_aa_prolaag l]))
      (ASL_PROTO_TRACES_EVAL_PROC n penv p ∩
       ASL_TRACE_IS_LOCK_SYNCHRONISED l)) ∧
   (ASL_PROTO_TRACES_EVAL_PROC n penv (asl_pt_critical_section l p) =
    IMAGE (λ t. [asl_aa_prolaag l] ++ t ++ [asl_aa_verhoog l])
      (ASL_PROTO_TRACES_EVAL_PROC n penv p)) ∧
   (ASL_PROTO_TRACES_EVAL_PROC 0 penv
      (asl_pt_procedure_call name arg) =
    if name ∉ FDOM penv then { [asl_aa_fail] } else ∅) ∧
   (ASL_PROTO_TRACES_EVAL_PROC (SUC n) penv
      (asl_pt_procedure_call name arg) =
    if name ∉ FDOM penv then
      { [asl_aa_fail] }
    else
      ASL_PROGRAM_TRACES_PROC n penv (penv ' name arg))
```

(177) asl_septraction_def 63

```
⊢ asl_septraction f P Q =
   (λ s. ∃ s1 s2. (SOME s2 = f (SOME s1) (SOME s)) ∧ s1 ∈ P ∧ s2 ∈ Q)
```

(178) asl_sp_opt_def 83

```
⊢ asl_sp_opt xenv penv P prog =
   (let Qset Q = ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q
    in
      if Qset = ∅ then NONE else SOME (BIGINTER Qset))
```

(179) asl_sp_opt_THM 83

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog sp ∧
   (∀ Q. ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q ⇒ sp ⊆ Q) ⟺
   (SOME sp = asl_sp_opt xenv penv P prog)
```

(180) asl_sp_opt___prog_assume 84

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (asl_sp_opt xenv penv P (asl_prog_assume c) =
   if asl_predicate_IS_DECIDED (FST xenv) P c then
     SOME (asl_and P (EVAL_asl_predicate (FST xenv) c))
   else
     NONE)
```

(181) asl_sp_opt___prog_ndet 84

```
⊢ asl_sp_opt xenv penv P (asl_prog_ndet pset) =
  if ∀prog. prog ∈ pset ⇒ IS_SOME (asl_sp_opt xenv penv P prog) then
    SOME
      (BIGUNION
         (IMAGE (λprog. THE (asl_sp_opt xenv penv P prog)) pset))
  else
    NONE
```

(182) asl_sp_opt___prog_seq 84

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (asl_sp_opt xenv penv P (asl_prog_seq p1 p2) =
   (let P1_opt = asl_sp_opt xenv penv P p1
    in
      if IS_SOME P1_opt then
        asl_sp_opt xenv penv (THE P1_opt) p2
      else
        NONE))
```

(183) asl_star_def 62

```
⊢ asl_star =
  (λf P Q x. ∃p q. (SOME x = f (SOME p) (SOME q)) ∧ p ∈ P ∧ q ∈ Q)
```

(184) ASL_TRACE_IS_LOCK_SYNCHRONISED_def 74

```
⊢ ASL_TRACE_IS_LOCK_SYNCHRONISED l t ⇔
  LIST_STAR [asl_aa_prolaag l; asl_aa_verhoog l]
    (ASL_TRACE_GET_LOCKS {l} t)
```

(185) ASL_TRACE_REMOVE_LOCKS_def 74

```
⊢ ASL_TRACE_REMOVE_LOCKS L =
  FILTER (λx. ¬ASL_IS_LOCK_ATOMIC_ACTION L x)
```

(186) ASL_TRACE_SEM_def 75

```
⊢ ASL_TRACE_SEM xenv t =
  asla_big_seq (MAP (ASL_ATOMIC_ACTION_SEM xenv) t)
```

(187) ASL_TRACE_ZIP___REWRITE 74

```
⊢ (ASL_TRACE_ZIP [] t = {t}) ∧ (ASL_TRACE_ZIP t [] = {t}) ∧
  (ASL_TRACE_ZIP (aa1::t1) (aa2::t2) =
   (let z1 = IMAGE (λx. aa1::x) (ASL_TRACE_ZIP t1 (aa2::t2)) in
    let z2 = IMAGE (λx. aa2::x) (ASL_TRACE_ZIP (aa1::t1) t2) in
    let z3 = z1 ∪ z2
    in
      if
        ASL_IS_PRIM_COMMAND_ATOMIC_ACTION aa1 ∧
        ASL_IS_PRIM_COMMAND_ATOMIC_ACTION aa2
      then
        IMAGE
          (λx.
             asl_aa_check (ASL_GET_PRIM_COMMAND_ATOMIC_ACTION aa1)
               (ASL_GET_PRIM_COMMAND_ATOMIC_ACTION aa2)::x) z3
      else
        z3))
```

(188) asl_trivial_cond_def 63

```
⊢ asl_trivial_cond = (λc P. if c then P else asl_false)
```

(189) asl_true_def 63

```
⊢ asl_true = 𝒰(:α)
```

(190) asl_wlp_def 83

```
⊢ asl_wlp xenv penv prog Q =
  BIGUNION (λP. ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q)
```

(191) asl_wlp_THM 83

```
⊢ ASL_PROGRAM_HOARE_TRIPLE xenv penv wlp prog Q ∧
  (∀P. ASL_PROGRAM_HOARE_TRIPLE xenv penv P prog Q ⇒ P ⊆ wlp) ⟺
  (wlp = asl_wlp xenv penv prog Q)
```

(192) asl_wlp___prog_seq 84

```
⊢ IS_SEPARATION_COMBINATOR (FST xenv) ⇒
  (asl_wlp xenv penv (asl_prog_seq p1 p2) Q =
   asl_wlp xenv penv p1 (asl_wlp xenv penv p2 Q))
```

(193) best_local_action_def 69

```
⊢ best_local_action f P1 P2 s =
  (let set p =
         ∃s0 s1.
           (SOME s = f (SOME s0) (SOME s1)) ∧ s1 ∈ P1 ∧
           (p = fasl_star f (SOME P2) (SOME {s0}))
   in
     INF_fasl_order set)
```

(194) best_local_action_THM 69, 81

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_IS_LOCAL_ACTION f (best_local_action f P1 P2) ∧
  HOARE_TRIPLE P1 (best_local_action f P1 P2) P2 ∧
  ∀ g.
    ASL_IS_LOCAL_ACTION f g ∧ HOARE_TRIPLE P1 g P2 ⇒
    fasl_action_order g (best_local_action f P1 P2)
```

(195) best_local_action___ALTERNATIVE_DEF 69

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  (BIGSUP fasl_action_order 𝒰(:α asl_action)
     (λ g. ASL_IS_LOCAL_ACTION f g ∧ HOARE_TRIPLE P1 g P2) =
   SOME (best_local_action f P1 P2))
```

(196) DISJOINT_FMAP_UNION_def 63, 117

```
⊢ DISJOINT_FMAP_UNION =
  BIN_OPTION_MAP () (λ m1 m2. DISJOINT (FDOM m1) (FDOM m2))
```

(197) EVAL_asl_predicate_def 73

```
⊢ (∀ f pp.
     EVAL_asl_predicate f (asl_pred_prim pp) =
     if ASL_IS_INTUITIONISTIC f (pp f) then pp f else asl_false) ∧
  (∀ f. EVAL_asl_predicate f asl_pred_true = asl_true) ∧
  (∀ f. EVAL_asl_predicate f asl_pred_false = asl_false) ∧
  (∀ f p.
     EVAL_asl_predicate f (asl_pred_neg p) =
     ASL_INTUITIONISTIC_NEGATION f (EVAL_asl_predicate f p)) ∧
  (∀ f p1 p2.
     EVAL_asl_predicate f (asl_pred_and p1 p2) =
     asl_and (EVAL_asl_predicate f p1) (EVAL_asl_predicate f p2)) ∧
  ∀ f p1 p2.
    EVAL_asl_predicate f (asl_pred_or p1 p2) =
    asl_or (EVAL_asl_predicate f p1) (EVAL_asl_predicate f p2)
```

(198) EVAL_asl_prim_command_def 73

```
⊢ EVAL_asl_prim_command f (asl_pc_shallow_command sc) =
  if ASL_IS_LOCAL_ACTION f (sc f) then sc f else asla_fail
```

(199) fasl_action_order_def 68

```
⊢ fasl_action_order f g ⇔
  ∀ P Q. HOARE_TRIPLE P g Q ⇒ HOARE_TRIPLE P f Q
```

(200) fasl_action_order_IS_WEAK_ORDER 68

```
⊢ WeakOrder fasl_action_order
```

(201) fasl_action_order_POINTWISE_DEF 68

⊢ fasl_action_order a_1 a_2 ⟺ ∀ s. fasl_order (a_1 s) (a_2 s)

(202) fasl_action_order___IS_COMPLETE_LATTICE 68

⊢ IS_SEPARATION_COMBINATOR f ⇒
IS_COMPLETE_LATTICE fasl_action_order (ASL_IS_LOCAL_ACTION f)

(203) fasl_order_def 67

⊢ (fasl_order NONE NONE ⟺ T) ∧
(fasl_order (SOME v_2) NONE ⟺ T) ∧
(fasl_order NONE (SOME x) ⟺ F) ∧
(fasl_order (SOME x) (SOME y) ⟺ x ⊆ y)

(204) fasl_order_IS_WEAK_ORDER 68

⊢ WeakOrder fasl_order

(205) fasl_star_DIRECT_DEF 67

⊢ (fasl_star f NONE $Qopt$ = NONE) ∧ (fasl_star f $Popt$ NONE = NONE) ∧
(fasl_star f (SOME P) (SOME Q) = SOME (asl_star f P Q))

(206) HOARE_TRIPLE_REWRITE 65

⊢ HOARE_TRIPLE P f Q ⟺ ∀ s. s ∈ P ⇒ ∃ S. (f s = SOME S) ∧ S ⊆ Q

(207) ID_SEPARATION_COMBINATOR_def 65

⊢ ID_SEPARATION_COMBINATOR = BIN_OPTION_MAP (λ x' y'. x') (=)

(208) ID_SEPARATION_COMBINATOR___THMS 65

⊢ IS_SEPARATION_COMBINATOR ID_SEPARATION_COMBINATOR ∧
(asl_emp ID_SEPARATION_COMBINATOR = $\mathcal{U}$(:β)) ∧
(asl_star ID_SEPARATION_COMBINATOR = (∩)) ∧
(ASL_IS_SEPARATE ID_SEPARATION_COMBINATOR = (=)) ∧
(ASL_IS_SUBSTATE ID_SEPARATION_COMBINATOR = (=)) ∧
(ASL_IS_PRECISE_IN_STATE ID_SEPARATION_COMBINATOR = K (K T)) ∧
(ASL_IS_PRECISE ID_SEPARATION_COMBINATOR = K T)

(209) INF_fasl_action_order_LOCAL 68

⊢ IS_SEPARATION_COMBINATOR f ∧
(∀ op. op ∈ OP ⇒ ASL_IS_LOCAL_ACTION f op) ⇒
ASL_IS_LOCAL_ACTION f (INF_fasl_action_order OP)

(210) INF_fasl_action_order_REWRITE 68

```
⊢ INF_fasl_action_order actions =
  (λ s.
     if ∃ a. a ∈ actions ∧ IS_SOME (a s) then
       SOME
         (BIGINTER (IMAGE THE (IS_SOME ∩ IMAGE (λ a. a s) actions)))
     else
       NONE)
```

(211) INF_fasl_action_order_THM 68

```
⊢ IS_INFIMUM fasl_action_order 𝒰(:α asl_action) M
    (INF_fasl_action_order M)
```

(212) INF_fasl_order_def 66

```
⊢ INF_fasl_order M =
  if ∀ x. x ∈ M ⇒ (x = NONE) then
    NONE
  else
    SOME (BIGINTER (IMAGE THE ((λ x. IS_SOME x) ∩ M)))
```

(213) IS_COMM_MONOID___asl_star_emp 63

```
⊢ IS_SEPARATION_COMBINATOR f ⇒ COMM_MONOID (asl_star f) (asl_emp f)
```

(214) IS_SEPARATION_ALGEBRA_def 64

```
⊢ IS_SEPARATION_ALGEBRA f u ⟺
  (∀ x. f NONE x = NONE) ∧ (∀ x. f (SOME u) (SOME x) = SOME x) ∧
  COMM f ∧ ASSOC f ∧ OPTION_IS_LEFT_CANCELLATIVE f
```

(215) IS_SEPARATION_ALGEBRA___COMBINATOR_DEF 64

```
⊢ IS_SEPARATION_ALGEBRA f u ⟺
  IS_SEPARATION_COMBINATOR f ∧ ∀ x. f (SOME u) (SOME x) = SOME x
```

(216) IS_SEPARATION_ALGEBRA___FINITE_MAP 117

```
⊢ IS_SEPARATION_ALGEBRA DISJOINT_FMAP_UNION FEMPTY
```

(217) IS_SEPARATION_COMBINATOR_def 62

```
⊢ IS_SEPARATION_COMBINATOR f ⟺
  (∀ x. f NONE x = NONE) ∧ (∀ x. ∃ u. f (SOME u) (SOME x) = SOME x) ∧
  COMM f ∧ ASSOC f ∧ OPTION_IS_LEFT_CANCELLATIVE f
```

(218) IS_SEPARATION_COMBINATOR_NEUTRAL_ELEMENT_FUNCTION_11 64

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  IS_SEPARATION_COMBINATOR_NEUTRAL_ELEMENT_FUNCTION f uf1 ∧
  IS_SEPARATION_COMBINATOR_NEUTRAL_ELEMENT_FUNCTION f uf2 ⇒
  (uf1 = uf2)
```

(219) IS_SEPARATION_COMBINATOR_NEUTRAL_ELEMENT_FUNCTION_EQS 64

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  IS_SEPARATION_COMBINATOR_NEUTRAL_ELEMENT_FUNCTION f uf ∧
  (f (SOME s1) (SOME s2) = SOME s3) ⇒
  (uf s1 = uf s2) ∧ (uf s1 = uf s3) ∧ (uf s2 = uf s3)
```

(220) IS_SEPARATION_COMBINATOR___FINITE_MAP 63, 117

```
⊢ IS_SEPARATION_COMBINATOR DISJOINT_FMAP_UNION
```

(221) IS_SEPARATION_COMBINATOR___NEURAL_ELEMENT_IDEMPOTENT 64

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ∀x u.
    (f (SOME u) (SOME x) = SOME x) ⇒ (f (SOME u) (SOME u) = SOME u)
```

(222) IS_SEPARATION_COMBINATOR___NEURAL_ELEMENT_IS_NEUTRAL 64

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ∀x1 x2 x3 u.
    (f (SOME u) (SOME x1) = SOME x1) ∧
    (f (SOME u) (SOME x2) = SOME x3) ⇒
    (x3 = x2)
```

(223) LOCALITY_CHARACTERISATION 67

```
⊢ ASL_IS_LOCAL_ACTION f op ⟺
  TRANS_FUNC_SAFETY_MONOTONICITY f op ∧
  TRANS_FUNC_FRAME_PROPERTY f op
```

(224) PRODUCT_SEPARATION_COMBINATOR_def 64

```
⊢ (PRODUCT_SEPARATION_COMBINATOR f1 f2 NONE NONE = NONE) ∧
  (PRODUCT_SEPARATION_COMBINATOR f1 f2 NONE (SOME v9) = NONE) ∧
  (PRODUCT_SEPARATION_COMBINATOR f1 f2 (SOME (v11,v12)) NONE =
   NONE) ∧
  (PRODUCT_SEPARATION_COMBINATOR f1 f2 (SOME (x1,x2))
     (SOME (y1,y2)) =
   (let z1 = f1 (SOME x1) (SOME y1) in
    let z2 = f2 (SOME x2) (SOME y2)
    in
      if IS_SOME z1 ∧ IS_SOME z2 then
        SOME (THE z1,THE z2)
      else
        NONE))
```

(225) PRODUCT_SEPARATION_COMBINATOR_THM 64

```
⊢ IS_SEPARATION_COMBINATOR f1 ∧ IS_SEPARATION_COMBINATOR f2 ⇒
  IS_SEPARATION_COMBINATOR (PRODUCT_SEPARATION_COMBINATOR f1 f2)
```

(226) PRODUCT_SEPARATION_COMBINATOR___ALGEBRA_THM 64

```
⊢ IS_SEPARATION_ALGEBRA f1 u1 ∧ IS_SEPARATION_ALGEBRA f2 u2 ⇒
  IS_SEPARATION_ALGEBRA (PRODUCT_SEPARATION_COMBINATOR f1 f2) (u1,u2)
```

(227) PRODUCT_SEPARATION_COMBINATOR___asl_star 94

```
⊢ asl_star (PRODUCT_SEPARATION_COMBINATOR f1 f2) P1 P2 =
  (λ (x,y).
     ∃ x1 x2 y1 y2.
       (f1 (SOME x1) (SOME x2) = SOME x) ∧
       (f2 (SOME y1) (SOME y2) = SOME y) ∧ P1 (x1,y1) ∧ P2 (x2,y2))
```

(228) quant_best_local_action_def 71

```
⊢ quant_best_local_action f qP1 qP2 =
  INF_fasl_action_order
    (λ g. ∃ x. g = best_local_action f (qP1 x) (qP2 x))
```

(229) quant_best_local_action_REWRITE 71

```
⊢ quant_best_local_action f qP1 qP2 s =
  (let set p =
         ∃ x s0 s1.
           (SOME s = f (SOME s0) (SOME s1)) ∧ s1 ∈ qP1 x ∧
           (p = fasl_star f (SOME (qP2 x)) (SOME {s0}))
   in
     INF_fasl_order set)
```

(230) quant_best_local_action_THM 71

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_IS_LOCAL_ACTION f (quant_best_local_action f qP1 qP2) ∧
  (∀ x.
     HOARE_TRIPLE (qP1 x) (quant_best_local_action f qP1 qP2)
       (qP2 x)) ∧
  ∀ g.
    ASL_IS_LOCAL_ACTION f g ∧ (∀ x. HOARE_TRIPLE (qP1 x) g (qP2 x)) ⇒
    fasl_action_order g (quant_best_local_action f qP1 qP2)
```

(231) quant_best_local_action___ALTERNATIVE_DEF 71

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  (BIGSUP fasl_action_order 𝒰(:α asl_action)
     (λ g.
        ASL_IS_LOCAL_ACTION f g ∧
        ∀ x. HOARE_TRIPLE (qP1 x) g (qP2 x)) =
   SOME (quant_best_local_action f qP1 qP2))
```

(232) SUP_fasl_action_order_def 68

```
⊢ SUP_fasl_action_order M = (λ x. SUP_fasl_order (IMAGE (λ f. f x) M))
```

(233) SUP_fasl_action_order_LOCAL 68

```
⊢ (∀ op. op ∈ OP ⇒ ASL_IS_LOCAL_ACTION f op) ⇒
  ASL_IS_LOCAL_ACTION f (SUP_fasl_action_order OP)
```

(234) SUP_fasl_action_order_THM 68

```
⊢ IS_SUPREMUM fasl_action_order 𝒰(:α asl_action) M
    (SUP_fasl_action_order M)
```

(235) SUP_fasl_order_def 66

```
⊢ SUP_fasl_order M =
  if NONE ∈ M then NONE else SOME (BIGUNION (IMAGE THE M))
```

(236) TRANS_FUNC_FRAME_PROPERTY_def 67

```
⊢ TRANS_FUNC_FRAME_PROPERTY f op ⟺
  ∀ s1 s2 s3 v1 v3 t.
    (f (SOME s1) (SOME s2) = SOME s3) ∧ (op s1 = SOME v1) ∧
    (op s3 = SOME v3) ∧ t ∈ v3 ⇒
    ∃ t'. (SOME t = f (SOME t') (SOME s2)) ∧ t' ∈ v1
```

(237) TRANS_FUNC_SAFETY_MONOTONICITY_def 67

```
⊢ TRANS_FUNC_SAFETY_MONOTONICITY f op ⟺
  ∀ s1 s2. ASL_IS_SUBSTATE f s1 s2 ∧ IS_SOME (op s1) ⇒ IS_SOME (op s2)
```

C.3 vars_as_resourceTheory

(238) asl_emp___VAR_RES_COMBINATOR 95

```
⊢ asl_emp (VAR_RES_COMBINATOR f) =
  (λ s. (FST s = FEMPTY) ∧ SND s ∈ asl_emp f)
```

(239) ASL_INTUITIONISTIC_NEGATION___weak_prop_expression 103

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  EVERY
    (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
       (FDOM (FST s))) el ⇒
  (ASL_INTUITIONISTIC_NEGATION (VAR_RES_COMBINATOR f)
     (var_res_prop_weak_expression p el) s ⟺
   var_res_prop_weak_expression (λ l. ¬p l) el s)
```

(240) ASL_IS_INTUITIONISTIC___weak_expression 96, 103

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  EVERY
    (λ e.
       IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e))
    el ⇒
  ASL_IS_INTUITIONISTIC (VAR_RES_COMBINATOR f)
    (var_res_prop_weak_expression p el)
```

(241) ASL_IS_LOCAL_ACTION___var_res_assign_action 107

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  ASL_IS_LOCAL_ACTION (VAR_RES_COMBINATOR f)
    (var_res_assign_action v e)
```

(242) ASL_IS_LOCAL_ACTION___var_res_dispose_var_action 109

```
⊢ ASL_IS_LOCAL_ACTION (VAR_RES_COMBINATOR f)
    (var_res_dispose_var_action v)
```

(243) ASL_IS_LOCAL_ACTION___var_res_new_var_init_action 109

```
⊢ IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e) ⇒
  ASL_IS_LOCAL_ACTION (VAR_RES_COMBINATOR f)
    (var_res_new_var_init_action v e)
```

(244) asl_predicate_IS_DECIDED___var_res_pred 103

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  EVERY (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs)
    el ∧ (∀ s. s ∈ P ⇒ vs ⊆ FDOM (FST s)) ⇒
  asl_predicate_IS_DECIDED (VAR_RES_COMBINATOR f) P
    (var_res_pred p el)
```

(245) ASL_PROGRAM_IS_ABSTRACTION___var_res_best_local_action 100

```
⊢ IS_VAR_RES_COMBINATOR (FST xenv) ⇒
  (ASL_PROGRAM_IS_ABSTRACTION xenv penv prog
     (var_res_prog_best_local_action P Q) ⇐⇒
   VAR_RES_HOARE_TRIPLE xenv penv P prog Q)
```

(246) ASL_PROGRAM_IS_ABSTRACTION___var_res_prog_aquire_lock 105

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  ASL_PROGRAM_IS_ABSTRACTION (VAR_RES_COMBINATOR f,lenv) penv
    (var_res_prog_aquire_lock f c wpb sfb)
    (asl_prog_seq
       (var_res_prog_cond_best_local_action
          (var_res_prop f ({||},{||}) {||}) (var_res_prop f (wpb,{||}) sfb))
       (asl_prog_assume c))
```

(247) ASL_PROGRAM_IS_ABSTRACTION___var_res_prog_eval_expressions 106

```
⊢ (∀ constL.
      ASL_PROGRAM_IS_ABSTRACTION xenv penv (prog constL)
        (prog′ constL)) ⇒
   ASL_PROGRAM_IS_ABSTRACTION xenv penv
     (var_res_prog_eval_expressions prog expL)
     (var_res_prog_eval_expressions prog′ expL)
```

(248) ASL_PROGRAM_IS_ABSTRACTION___var_res_prog_parallel_procedure_call 106

```
⊢ IS_VAR_RES_COMBINATOR (FST xenv) ⇒
   ∀ qP1 qP2 qQ1 qQ2.
     (∀ arg.
        ASL_PROGRAM_IS_ABSTRACTION xenv penv
          (asl_prog_procedure_call name1 arg)
          (var_res_prog_quant_best_local_action (qP1 arg)
             (qQ1 arg))) ∧
     (∀ arg.
        ASL_PROGRAM_IS_ABSTRACTION xenv penv
          (asl_prog_procedure_call name2 arg)
          (var_res_prog_quant_best_local_action (qP2 arg)
             (qQ2 arg))) ⇒
     ASL_PROGRAM_IS_ABSTRACTION xenv penv
       (var_res_prog_parallel_procedure_call name1 arg1 name2 arg2)
       (var_res_prog_eval_expressions
          (λ constL1.
             var_res_prog_eval_expressions
               (λ constL2.
                  var_res_prog_quant_best_local_action
                    (λ (arg′1,arg′2).
                       asl_star (FST xenv)
                         (qP1 (FST arg1,constL1) arg′1)
                         (qP2 (FST arg2,constL2) arg′2))
                    (λ (arg′1,arg′2).
                       asl_star (FST xenv)
                         (qQ1 (FST arg1,constL1) arg′1)
                         (qQ2 (FST arg2,constL2) arg′2))) (SND arg2))
          (SND arg1))
```

(249) ASL_PROGRAM_IS_ABSTRACTION___var_res_prog_release_lock 105

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
   ASL_PROGRAM_IS_ABSTRACTION (VAR_RES_COMBINATOR f,lenv) penv
     (var_res_prog_release_lock f wpb sfb)
     (var_res_prog_cond_best_local_action
        (var_res_prop f (wpb,{||}) sfb) (var_res_prop f ({||},{||}) {||}))
```

(250) ASL_PROGRAM_IS_ABSTRACTION___var_res_quant_best_local_action 100

```
⊢ IS_VAR_RES_COMBINATOR (FST xenv) ⇒
   (ASL_PROGRAM_IS_ABSTRACTION xenv penv prog
      (var_res_prog_quant_best_local_action qP qQ) ⟺
    ∀ arg. VAR_RES_HOARE_TRIPLE xenv penv (qP arg) prog (qQ arg))
```

(251) asl_star___VAR_RES_IS_STACK_IMPRECISE 94

$\vdash$ VAR_RES_IS_STACK_IMPRECISE P_1 $\wedge$ VAR_RES_IS_STACK_IMPRECISE P_2 $\Rightarrow$
(asl_star (VAR_RES_COMBINATOR f) P_1 P_2 =
($\lambda\, s$.
$\exists\, es_1\ es_2$.
(f (SOME es_1) (SOME es_2) = SOME (SND s)) $\wedge$ (FST s,es_1) $\in$ P_1 $\wedge$
(FST s,es_2) $\in$ P_2))

(252) asl_star___var_res_prop_stack_true___STACK_IMPRECISE 95

$\vdash$ IS_SEPARATION_COMBINATOR f $\Rightarrow$
$\forall\, P$.
VAR_RES_IS_STACK_IMPRECISE P $\Rightarrow$
(asl_star (VAR_RES_COMBINATOR f) (var_res_prop_stack_true f) P =
P)

(253) asl_star___var_res_prop___PROP 101

$\vdash$ BAG_DISJOINT wpb_1 wpb_2 $\wedge$ BAG_DISJOINT wpb_1 rpb_2 $\wedge$
BAG_DISJOINT wpb_2 rpb_1 $\wedge$ var_res_prop___COND f (wpb_1,rpb_1) sfb_1 $\wedge$
var_res_prop___COND f (wpb_2,rpb_2) sfb_2 $\Rightarrow$
(asl_star (VAR_RES_COMBINATOR f)
(var_res_prop___PROP f (wpb_1,rpb_1) sfb_1)
(var_res_prop___PROP f (wpb_2,rpb_2) sfb_2) =
var_res_prop___PROP f (wpb_1 $\uplus$ wpb_2,BAG_MERGE rpb_1 rpb_2)
(sfb_1 $\uplus$ sfb_2))

(254) IS_PERMISSION_STRUCTURE_def 90

$\vdash$ IS_PERMISSION_STRUCTURE (f,$total_perm$) $\iff$
ASSOC f $\wedge$ COMM f $\wedge$ OPTION_IS_LEFT_CANCELLATIVE f $\wedge$
($\forall\, C$. f NONE C = NONE) $\wedge$
($\forall\, c$. $\exists\, c_1\ c_2$. f (SOME c_1) (SOME c_2) = SOME c) $\wedge$
($\forall\, c$. f (SOME $total_perm$) (SOME c) = NONE) $\wedge$
$\forall\, c_1\ c_2$. f (SOME c_1) (SOME c_2) $\neq$ SOME c_1

(255) IS_SEPARATION_ALGEBRA___VAR_RES_COMBINATOR 92

$\vdash$ IS_SEPARATION_ALGEBRA f u $\Rightarrow$
IS_SEPARATION_ALGEBRA (VAR_RES_COMBINATOR f) (FEMPTY,u)

(256) IS_SEPARATION_COMBINATOR___VAR_RES_COMBINATOR 92

$\vdash$ IS_SEPARATION_COMBINATOR f $\Rightarrow$
IS_SEPARATION_COMBINATOR (VAR_RES_COMBINATOR f)

(257) IS_VAR_RES_SUBPERMISSION_def 90

$\vdash$ IS_VAR_RES_SUBPERMISSION p_1 p_2 $\iff$
(p_1 = p_2) $\vee$
$\exists\, p$. var_res_permission_combine (SOME p_1) (SOME p) = SOME p_2

(258) var_res_assign_action_def 107

```
⊢ var_res_assign_action v e s =
  (let ev_opt = e (FST s)
   in
     if
       var_res_sl___has_write_permission v (FST s) ∧ IS_SOME ev_opt
     then
       SOME {var_res_ext_state_var_update (v,THE ev_opt) s}
     else
       NONE)
```

(259) var_res_best_local_action_def 100

```
⊢ var_res_best_local_action f P Q =
  quant_best_local_action f (λx s. s ∈ P ∧ (s = x))
    (λx s.
       s ∈ Q ∧ VAR_RES_STACK___IS_EQUAL_UPTO_VALUES (FST x) (FST s))
```

(260) var_res_bigstar_list_REWRITE 97

```
⊢ (∀f.
     IS_SEPARATION_COMBINATOR f ⇒
     (var_res_bigstar_list f [] = var_res_prop_stack_true f)) ∧
  ∀f p pL.
    IS_SEPARATION_COMBINATOR f ⇒
    (var_res_bigstar_list f (p::pL) =
     asl_star (VAR_RES_COMBINATOR f) p (var_res_bigstar_list f pL))
```

(261) var_res_bigstar_REWRITE 97

```
⊢ (∀f.
     IS_SEPARATION_COMBINATOR f ⇒
     (var_res_bigstar f {||} = var_res_prop_stack_true f)) ∧
  ∀f p pL.
    IS_SEPARATION_COMBINATOR f ⇒
    (var_res_bigstar f (BAG_INSERT p pL) =
     asl_star (VAR_RES_COMBINATOR f) p (var_res_bigstar f pL))
```

(262) var_res_bool_proposition_def 96

```
⊢ var_res_bool_proposition f c =
  var_res_stack_proposition f T (λs. c)
```

(263) VAR_RES_COMBINATOR_def 92

```
⊢ VAR_RES_COMBINATOR f =
  PRODUCT_SEPARATION_COMBINATOR VAR_RES_STACK_COMBINE f
```

(264) var_res_cond_best_local_action_def 100

```
⊢ var_res_cond_best_local_action f P Q =
  if ¬FST P ∨ ¬FST Q then
    asla_diverge
  else
    var_res_best_local_action f (SND P) (SND Q)
```

(265) VAR_RES_COND_HOARE_TRIPLE_def 99

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P prog Q ⟺
  IS_SEPARATION_COMBINATOR f ∧ FST P ∧ FST Q ⇒
  VAR_RES_HOARE_TRIPLE (VAR_RES_COMBINATOR f,K asl_false) FEMPTY
    (SND P) prog (SND Q)
```

(266) VAR_RES_COND_HOARE_TRIPLE___COND_PROP_STRONG_IMP 108

```
⊢ COND_PROP___STRONG_IMP P1 P2 ⇒
  VAR_RES_COND_HOARE_TRIPLE f P2 prog Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P1 prog Q
```

(267) VAR_RES_COND_HOARE_TRIPLE___PROGRAM_ABSTRACTION 101

```
⊢ (IS_SEPARATION_COMBINATOR f ⇒
   VAR_RES_PROGRAM_IS_ABSTRACTION f prog1 prog2 ∧
   VAR_RES_COND_HOARE_TRIPLE f P prog2 Q) ⇒
  VAR_RES_COND_HOARE_TRIPLE f P prog1 Q
```

(268) VAR_RES_COND_HOARE_TRIPLE___SOLVE 112

```
⊢ SET_OF_BAG wpb' ⊆ SET_OF_BAG wpb ∧
  SET_OF_BAG rpb' ⊆ SET_OF_BAG (wpb ⊎ rpb) ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) {||} {||} sfb sfb'
    (BAG_EVERY (VAR_RES_IS_PURE_PROPOSITION f)) ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (asl_prog_block []) (var_res_prop f (wpb',rpb') sfb')
```

(269) VAR_RES_COND_INFERENCE___asl_exists_pre 102

```
⊢ (∀ y.
     VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
       (P y)) ⇒
  (VAR_RES_COND_HOARE_TRIPLE f
     (var_res_prop f (wpb,rpb) (BAG_INSERT (asl_exists y. P y) sfb))
     prog Q ⟺
   ∀ y.
     VAR_RES_COND_HOARE_TRIPLE f
       (var_res_prop f (wpb,rpb) (BAG_INSERT (P y) sfb)) prog Q)
```

(270) VAR_RES_COND_INFERENCE___asl_star_pre 102

```
⊢ IS_VAR_RES_COMBINATOR f′ ∧ (GET_VAR_RES_COMBINATOR f′ = f) ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
    P1 ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
    P2 ⇒
  (VAR_RES_COND_HOARE_TRIPLE f
     (var_res_prop f (wpb,rpb) (BAG_INSERT (asl_star f′ P1 P2) sfb))
     prog Q ⟺
   VAR_RES_COND_HOARE_TRIPLE f
     (var_res_prop f (wpb,rpb) (BAG_INSERT P1 (BAG_INSERT P2 sfb)))
     prog Q)
```

(271) VAR_RES_COND_INFERENCE___asl_trivial_cond 102

```
⊢ VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT (asl_trivial_cond c P) sfb)) prog post ⟺
  c ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb) (BAG_INSERT P sfb)) prog post
```

(272) VAR_RES_COND_INFERENCE___CONST_INTRO 102

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ⇒
  (VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
     post ⟺
   ∀ c.
     VAR_RES_COND_HOARE_TRIPLE f
       (var_res_prop f (wpb,rpb)
          (BAG_INSERT (var_res_prop_equal f e (var_res_exp_const c))
             sfb)) prog post)
```

(273) VAR_RES_COND_INFERENCE___CONST_INTRO_READ 102

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG rpb) e ⇒
  (VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
     (var_res_prop f (wpb′,rpb) sfb′) ⟺
   ∀ c.
     VAR_RES_COND_HOARE_TRIPLE f
       (var_res_prop f (wpb,rpb)
          (BAG_INSERT (var_res_prop_equal f e (var_res_exp_const c))
             sfb)) prog
       (var_res_prop f (wpb′,rpb)
          (BAG_INSERT (var_res_prop_equal f e (var_res_exp_const c))
             sfb′)))
```

(274) VAR_RES_COND_INFERENCE___eval_expressions___NIL 106

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (var_res_prog_eval_expressions prog []::progL))
    Q ⟺
  VAR_RES_COND_HOARE_TRIPLE f P (asl_prog_block (prog []::progL)) Q
```

(275) VAR_RES_COND_INFERENCE___eval_expressions___ONE 106

```
⊢ (e = var_res_exp_const c) ∨
  var_res_prop_equal f e (var_res_exp_const c) ∈: sfb ⇒
  (VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
     (asl_prog_block
        (var_res_prog_eval_expressions prog (e::L)::progL)) Q ⟺
   VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
     (asl_prog_block
        (var_res_prog_eval_expressions (λL. prog (c::L)) L::progL))
     Q)
```

(276) VAR_RES_COND_INFERENCE___FRAME 102

```
⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG rpb) P ∧
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
    (var_res_prop f (wpb',rpb) sfb') ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb) (BAG_INSERT P sfb)) prog
    (var_res_prop f (wpb',rpb) (BAG_INSERT P sfb'))
```

(277) VAR_RES_COND_INFERENCE___loop_spec_GENERAL 104

```
⊢ (FST P ⇒ ∀x. FST (pre x) ∧ FST (post x)) ⇒
  (∀x.
     VAR_RES_COND_HOARE_TRIPLE f (pre x)
       (asl_prog_block (asl_prog_assume (asl_pred_neg c)::prog_1))
       (post x)) ∧
  (∀x.
     VAR_RES_COND_HOARE_TRIPLE f (pre x)
       (asl_prog_block
          [asl_prog_assume c; p;
           var_res_prog_cond_quant_best_local_action pre post])
       (post x)) ∧
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (var_res_prog_cond_quant_best_local_action pre post::prog_2))
    Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_block (asl_prog_while c p::prog_1)::prog_2)) Q
```

(278) VAR_RES_COND_INFERENCE___prog_assume_and 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume P1::asl_prog_assume P2::progL))
    Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume (asl_pred_and P1 P2)::progL)) Q
```

(279) VAR_RES_COND_INFERENCE___prog_assume_neg_and 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_assume
          (asl_pred_or (asl_pred_neg P1) (asl_pred_neg P2))::progL))
    Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_assume (asl_pred_neg (asl_pred_and P1 P2))::progL))
    Q
```

(280) VAR_RES_COND_INFERENCE___prog_assume_neg_neg 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume p::progL)) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_assume (asl_pred_neg (asl_pred_neg p))::progL)) Q
```

(281) VAR_RES_COND_INFERENCE___prog_assume_neg_or 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_assume
          (asl_pred_and (asl_pred_neg P1) (asl_pred_neg P2))::progL))
    Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (asl_prog_assume (asl_pred_neg (asl_pred_or P1 P2))::progL)) Q
```

(282) VAR_RES_COND_INFERENCE___prog_assume_neg_pred 103

```
⊢ EVERY
    (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
       (SET_OF_BAG (wpb ⊎ rpb))) el ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT (var_res_prop_expression f T (λl. ¬p l) el) sfb))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (asl_prog_block
       (asl_prog_assume (asl_pred_neg (var_res_pred p el))::progL)) Q
```

(283) VAR_RES_COND_INFERENCE___prog_assume_or 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume P1::progL)) Q ∧
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume P2::progL)) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume (asl_pred_or P1 P2)::progL)) Q
```

(284) VAR_RES_COND_INFERENCE___prog_assume_pred 103

```
⊢ EVERY
    (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
       (SET_OF_BAG (wpb ⊎ rpb))) el ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT (var_res_prop_expression f T p el) sfb))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (asl_prog_block (asl_prog_assume (var_res_pred p el)::progL)) Q
```

(285) VAR_RES_COND_INFERENCE___prog_call_by_value_arg 110

```
⊢ (∀ v.
     VAR_RES_COND_HOARE_TRIPLE f
       (var_res_prop f (BAG_INSERT v wpb,rpb)
          (BAG_INSERT
             (var_res_prop_equal f (var_res_exp_var v)
                (var_res_exp_const c)) sfb)) (body v)
       (COND_PROP___STRONG_EXISTS
          (λ x'. var_res_prop f (BAG_INSERT v wpb,rpb) (sfb' x')))) ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (var_res_prog_call_by_value_arg body c)
    (COND_PROP___STRONG_EXISTS
       (λ x'. var_res_prop f (wpb,rpb) (sfb' x')))
```

(286) VAR_RES_COND_INFERENCE___prog_choice 101

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P (asl_prog_block (p1::prog)) Q ∧
  VAR_RES_COND_HOARE_TRIPLE f P (asl_prog_block (p2::prog)) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_choice p1 p2::prog)) Q
```

(287) VAR_RES_COND_INFERENCE___prog_cond 104

```
⊢ VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume c::pTrue::prog)) Q ∧
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_assume (asl_pred_neg c)::pFalse::prog))
    Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_cond c pTrue pFalse::prog)) Q
```

(288) VAR_RES_COND_INFERENCE___prog_local_var 110

```
⊢ (∀ v.
     VAR_RES_COND_HOARE_TRIPLE f
       (var_res_prop f (BAG_INSERT v wpb,rpb) sfb) (body v)
       (COND_PROP___STRONG_EXISTS
          (λ x'. var_res_prop f (BAG_INSERT v wpb,rpb) (sfb' x')))) ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (var_res_prog_local_var body)
    (COND_PROP___STRONG_EXISTS
       (λ x'. var_res_prop f (wpb,rpb) (sfb' x')))
```

(289) VAR_RES_COND_INFERENCE___prog_while_GENERAL 104

```
⊢ (FST P ⇒ ∀ x. FST (Inv x)) ⇒
  (∀ x.
     VAR_RES_COND_HOARE_TRIPLE f (Inv x)
       (asl_prog_block (asl_prog_assume c::pL)) (Inv x)) ∧
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (var_res_prog_cond_quant_best_local_action Inv Inv::
          asl_prog_assume (asl_pred_neg c)::prog)) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block (asl_prog_while c (asl_prog_block pL)::prog)) Q
```

(290) VAR_RES_COND_INFERENCE___prop_implies 116

```
⊢ var_res_prop_implies f (wpb,rpb) sfb sfb' ⇒
  (VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
     post ⟺
   VAR_RES_COND_HOARE_TRIPLE f
     (var_res_prop f (wpb,rpb) (sfb ⊎ sfb')) prog post)
```

(291) VAR_RES_COND_INFERENCE___prop_implies_eq 115

```
⊢ var_res_prop_implies_eq f (wpb,rpb) {||} sfb sfb' ⇒
  (VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
     post ⟺
   VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb') prog
     post)
```

(292) VAR_RES_COND_INFERENCE___STRENGTHEN_PERMS 101

```
⊢ (∀ v. v ∈: wpb1 ⇒ v ∈: wpb2) ∧ (wpb1 ⊎ rpb1 = wpb2 ⊎ rpb2) ∧
  (wpb1' ⊎ rpb1 = wpb2' ⊎ rpb2) ∧
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb1,rpb1) sfb1) prog
    (var_res_prop f (wpb1',rpb1) sfb2) ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb2,rpb2) sfb1) prog
    (var_res_prop f (wpb2',rpb2) sfb2)
```

(293) VAR_RES_COND_INFERENCE___var_res_bool_proposition 102

```
⊢ VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT (var_res_bool_proposition f c) sfb)) prog post ⟺
  c ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
    post
```

(294) VAR_RES_COND_INFERENCE___var_res_prog_assign 108

```
⊢ v ∈: wpb ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal f (var_res_exp_var v)
             (var_res_exp_varlist_update [(v,c)] e))
          (BAG_IMAGE (var_res_prop_varlist_update [(v,c)]) sfb)))
    (asl_prog_block progL) Q ⇒
  VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT
          (var_res_prop_equal f (var_res_exp_var v)
             (var_res_exp_const c)) sfb))
    (asl_prog_block (var_res_prog_assign v e::progL)) Q
```

(295) VAR_RES_COND_INFERENCE___var_res_prog_cond_best_local_action___VAR_CHANGE 112

```
⊢ SET_OF_BAG wpb′ ⊆ SET_OF_BAG wpb ∧
  SET_OF_BAG rpb′ ⊆ SET_OF_BAG (wpb ⊎ rpb) ⇒
  VAR_RES_FRAME_SPLIT f rfc (wpb,rpb) wpb′ {||} sfb sfb′
    (λ sfb‴.
       VAR_RES_COND_HOARE_TRIPLE f
         (var_res_prop f (wpb − wpb′ ⊎ wpb″,rpb) (sfb″ ⊎ sfb‴))
         (asl_prog_block progL) Q) ⇒
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb)
    (asl_prog_block
       (var_res_prog_cond_best_local_action
          (var_res_prop f (wpb′,rpb′) sfb′)
          (var_res_prop f (wpb″,rpb′) sfb″)::progL)) Q
```

(296) VAR_RES_COND_INFERENCE___var_res_prog_cond_quant_best_local_action 110

```
⊢ (∃ x.
     VAR_RES_COND_HOARE_TRIPLE f P
       (asl_prog_block
          (var_res_prog_cond_best_local_action (qP x) (qQ x)::progL))
       Q) ⇒
  VAR_RES_COND_HOARE_TRIPLE f P
    (asl_prog_block
       (var_res_prog_cond_quant_best_local_action qP qQ::progL)) Q
```

(297) VAR_RES_COND_INFERENCE___var_res_prop_stack_true 102

```
⊢ VAR_RES_COND_HOARE_TRIPLE f
    (var_res_prop f (wpb,rpb)
       (BAG_INSERT (var_res_prop_stack_true f) sfb)) prog post ⟺
  VAR_RES_COND_HOARE_TRIPLE f (var_res_prop f (wpb,rpb) sfb) prog
    post
```

(298) var_res_cond_quant_best_local_action_def 100

```
⊢ var_res_cond_quant_best_local_action f qP qQ =
  if ¬(∀x. FST (qP x)) ∨ ¬∀x. FST (qQ x) then
    asla_diverge
  else
    var_res_quant_best_local_action f (λx. SND (qP x))
      (λx. SND (qQ x))
```

(299) var_res_dispose_var_action_def 109

```
⊢ var_res_dispose_var_action v s =
  if ¬var_res_sl___has_write_permission v (FST s) then
    NONE
  else
    SOME {(FST s \\ v,SND s)}
```

(300) var_res_exp_binop_def 92

```
⊢ var_res_exp_binop bop e1 e2 =
  var_res_exp_op (λl. bop (EL 0 l) (EL 1 l)) [e1; e2]
```

(301) var_res_exp_const_def 92

```
⊢ var_res_exp_const c = K (SOME c)
```

(302) var_res_exp_full_prop_def 93

```
⊢ var_res_exp_full_prop P eL =
  (λstate.
     (let e_optL = MAP (λe. e (FST state)) eL
      in
        EVERY IS_SOME e_optL ∧ P (MAP THE e_optL) (SND state)))
```

(303) var_res_exp_op_def 92

```
⊢ var_res_exp_op f el =
  (λs.
     (let el' = MAP (λe. e s) el
      in
        if EVERY IS_SOME el' then SOME (f (MAP THE el')) else NONE))
```

(304) var_res_exp_varlist_update___const_EVAL 107

```
⊢ var_res_exp_varlist_update vL (var_res_exp_const c) =
  var_res_exp_const c
```

(305) var_res_exp_varlist_update___var_EVAL 107

```
⊢ var_res_exp_varlist_update ((v1,c)::vL) (var_res_exp_var v2) =
  if v1 = v2 then
    var_res_exp_const c
  else
    var_res_exp_varlist_update vL (var_res_exp_var v2)
```

(306) var_res_exp_varlist_update___var_res_exp_op_EVAL 107

```
⊢ var_res_exp_varlist_update vL (var_res_exp_op f eL) =
  var_res_exp_op f (MAP (var_res_exp_varlist_update vL) eL)
```

(307) var_res_exp_var_def 92

```
⊢ var_res_exp_var var =
  (λ stack.
     if var ∈ FDOM stack then SOME (FST (stack ' var)) else NONE)
```

(308) var_res_exp_var_update_def 107

```
⊢ var_res_exp_var_update vc e =
  (λ s. e (var_res_state_var_update (FST vc) (SND vc) s))
```

(309) var_res_ext_state_var_update_def 107

```
⊢ var_res_ext_state_var_update vc s =
  (var_res_state_var_update (FST vc) (SND vc) (FST s),SND s)
```

(310) VAR_RES_FRAME_SPLIT_def 111

```
⊢ VAR_RES_FRAME_SPLIT f rfc (wpb,rpb) wpb' sfb_context sfb_split
    sfb_imp sfb_restP ⟺
  VAR_RES_FRAME_SPLIT___sfb_restP_OK f (wpb − wpb',rpb − wpb')
    sfb_restP ⇒
  ∃ sfb_rest.
    sfb_restP sfb_rest ∧
    var_res_prop___COND f (wpb − wpb',rpb − wpb') sfb_rest ∧
    (var_res_prop___COND f (wpb,rpb)
       (sfb_context ⊎ (sfb_split ⊎ sfb_imp)) ⇒
     ∀ s.
       var_res_prop___PROP f (wpb,rpb) (sfb_split ⊎ sfb_context) s ⇒
       var_res_prop___PROP f (wpb,rpb)
         (sfb_imp ⊎ (sfb_rest ⊎ sfb_context)) s)
```

(311) VAR_RES_FRAME_SPLIT___asl_exists___context 112

```
⊢ (∀ y.
     VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
       (P y)) ⇒
  (∀ y.
     VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
       (BAG_INSERT (P y) sfb_context) sfb_split sfb_imp sfb_restP) ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
    (BAG_INSERT (asl_exists y. P y) sfb_context) sfb_split sfb_imp
    sfb_restP
```

(312) VAR_RES_FRAME_SPLIT___asl_exists___imp 113

```
⊢ (∀ y.
     VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
       (P y)) ⇒
  (∃ y.
     VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
       (BAG_INSERT (P y) sfb_imp) sfb_restP) ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
    (BAG_INSERT (asl_exists y. P y) sfb_imp) sfb_restP
```

(313) VAR_RES_FRAME_SPLIT___asl_exists___split 112

```
⊢ (∀ y.
     VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
       (P y)) ⇒
  (VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context
     (BAG_INSERT (asl_exists y. P y) sfb_split) sfb_imp sfb_restP ⇔
   ∀ y.
     VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context
       (BAG_INSERT (P y) sfb_split) sfb_imp sfb_restP)
```

(314) VAR_RES_FRAME_SPLIT___asl_star___imp 113

```
⊢ IS_VAR_RES_COMBINATOR f ∧ (GET_VAR_RES_COMBINATOR f = f′) ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
    P1 ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
    P2 ⇒
  (VAR_RES_FRAME_SPLIT f′ sr (wpb,rpb) wpb′ sfb_context sfb_split
     (BAG_INSERT (asl_star f P1 P2) sfb_imp) sfb_restP ⇔
   VAR_RES_FRAME_SPLIT f′ sr (wpb,rpb) wpb′ sfb_context sfb_split
     (BAG_INSERT P1 (BAG_INSERT P2 sfb_imp)) sfb_restP)
```

(315) VAR_RES_FRAME_SPLIT___bool_proposition___context 112

```
⊢ VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
    (BAG_INSERT (var_res_bool_proposition f c) sfb_context) sfb_split
    sfb_imp sfb_restP ⇔
  c ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
    sfb_imp sfb_restP
```

(316) VAR_RES_FRAME_SPLIT___bool_proposition___imp 113

```
⊢ VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
    (BAG_INSERT (var_res_bool_proposition f c1)
       (BAG_INSERT (var_res_bool_proposition f c2) sfb_imp))
    sfb_restP ⟺
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
    (BAG_INSERT (var_res_bool_proposition f (c1 ∧ c2)) sfb_imp)
    sfb_restP
```

(317) VAR_RES_FRAME_SPLIT___bool_proposition___split 112

```
⊢ VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context
    (BAG_INSERT (var_res_bool_proposition f c) sfb_split) sfb_imp
    sfb_restP ⟺
  c ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context sfb_split
    sfb_imp sfb_restP
```

(318) VAR_RES_FRAME_SPLIT___equal_const_SING 113

```
⊢ VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
    (BAG_INSERT
       (var_res_prop_equal f (var_res_exp_var v)
          (var_res_exp_const c))
       (BAG_IMAGE (var_res_prop_var_update (v,c)) sfb_context))
    (BAG_IMAGE (var_res_prop_var_update (v,c)) sfb_split)
    (BAG_IMAGE (var_res_prop_var_update (v,c)) sfb_imp) sfb_restP ⇒
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context
    (BAG_INSERT
       (var_res_prop_equal f (var_res_exp_var v)
          (var_res_exp_const c)) sfb_split) sfb_imp sfb_restP
```

(319) VAR_RES_FRAME_SPLIT___FRAME 113

```
⊢ VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ sfb_context
    (BAG_INSERT sf sfb_split) (BAG_INSERT sf sfb_imp) sfb_restP ⟺
  VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′ (BAG_INSERT sf sfb_context)
    sfb_split sfb_imp sfb_restP
```

(320) VAR_RES_FRAME_SPLIT___PURE_PROPOSITION___CONTEXT_FRAME 113

```
⊢ VAR_RES_IS_PURE_PROPOSITION f sf ⇒
  (VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
     (BAG_INSERT sf sfb_context) sfb_split (BAG_INSERT sf sfb_imp)
     sfb_restP ⟺
   VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb′
     (BAG_INSERT sf sfb_context) sfb_split sfb_imp sfb_restP)
```

(321) VAR_RES_FRAME_SPLIT__PURE_PROPOSITION__TO_CONTEXT 113, 114

```
⊢ VAR_RES_IS_PURE_PROPOSITION f sf ⇒
  (VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb' sfb_context
     (BAG_INSERT sf sfb_split) sfb_imp sfb_restP ⟺
   VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb'
     (BAG_INSERT sf sfb_context) sfb_split sfb_imp sfb_restP)
```

(322) VAR_RES_FRAME_SPLIT___sfb_restP_OK_def 111

```
⊢ VAR_RES_FRAME_SPLIT___sfb_restP_OK f (wpb,rpb) sfb_restP ⟺
  (∃ sfb. sfb_restP sfb ∧ var_res_prop___COND f (wpb,rpb) sfb) ∧
  ∀ sfbS.
    (∀ sfb.
       sfb ∈ sfbS ⇒
       var_res_prop___COND f (wpb,rpb) sfb ∧ sfb_restP sfb) ⇒
    sfb_restP {|(λ s. ∃ sfb. sfb ∈ sfbS ∧ s ∈ var_res_bigstar f sfb)|}
```

(323) VAR_RES_FRAME_SPLIT___SOLVE_WEAK 114

```
⊢ BAG_EVERY
    (VAR_RES_IS_STACK_IMPRECISE___USED_VARS
       (SET_OF_BAG (wpb ⊎ rpb − wpb'))) sfb_split ⇒
  sfb_restP sfb_split ⇒
  VAR_RES_FRAME_SPLIT f rfc (wpb,rpb) wpb' sfb_context sfb_split {||}
    sfb_restP
```

(324) VAR_RES_FRAME_SPLIT___SOLVE_WEAK___bool_prop 114

```
⊢ BAG_EVERY
    (VAR_RES_IS_STACK_IMPRECISE___USED_VARS
       (SET_OF_BAG (wpb ⊎ rpb − wpb'))) sfb_split ⇒
  b ∧ (b ⇒ sfb_restP sfb_split) ⇒
  VAR_RES_FRAME_SPLIT f rfc (wpb,rpb) wpb' sfb_context sfb_split
    {|var_res_bool_proposition f b|} sfb_restP
```

(325) VAR_RES_FRAME_SPLIT___var_res_prop_implies_eq___split 115

```
⊢ var_res_prop_implies_eq f (wpb,rpb) sfb_context sfb_split
    sfb_split' ⇒
  (VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb' sfb_context sfb_split
     sfb_imp sfb_restP ⟺
   VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb' sfb_context sfb_split'
     sfb_imp sfb_restP)
```

(326) VAR_RES_FRAME_SPLIT___var_res_prop_implies___split 116

```
⊢ var_res_prop_implies f (wpb,rpb) (sfb_context ⊎ sfb_split) sfb ⇒
  (VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb' sfb_context sfb_split
     sfb_imp sfb_restP ⟺
   VAR_RES_FRAME_SPLIT f sr (wpb,rpb) wpb' sfb_context
     (sfb ⊎ sfb_split) sfb_imp sfb_restP)
```

(327) VAR_RES_HOARE_TRIPLE_def 99

```
⊢ VAR_RES_HOARE_TRIPLE xenv penv P prog Q ⟺
  ∀x.
    ASL_PROGRAM_HOARE_TRIPLE xenv penv (λs. s ∈ P ∧ (s = x)) prog
      (λs.
         s ∈ Q ∧
         VAR_RES_STACK___IS_EQUAL_UPTO_VALUES (FST x) (FST s))
```

(328) var_res_implies_unequal_def 115

```
⊢ var_res_implies_unequal f b e1 e2 ⟺
  ∀s.
    IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e1) ∧
    IS_SOME (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e2) ∧
    IS_SEPARATION_COMBINATOR f ∧ s ∈ var_res_bigstar f b ⇒
    s ∈ var_res_prop_weak_unequal e1 e2
```

(329) var_res_implies_unequal___prop_implies 116, 125

```
⊢ var_res_implies_unequal f sfb e1 e2 ⇒
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e1 ∧
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET
    (SET_OF_BAG (wpb ⊎ rpb)) e2 ⇒
  var_res_prop_implies f (wpb,rpb) sfb {|var_res_prop_unequal f e1 e2|}
```

(330) var_res_implies_unequal___trivial_context 116

```
⊢ c1 ≠ c2 ⇒
  var_res_implies_unequal f b (var_res_exp_const c1)
    (var_res_exp_const c2)
```

(331) var_res_implies_unequal___trivial_unequal 116

```
⊢ var_res_prop_unequal f e1 e2 ∈: b ⇒
  var_res_implies_unequal f b e1 e2
```

(332) VAR_RES_INFERENCE___FRAME 101

```
⊢ IS_SEPARATION_COMBINATOR f ∧ (FST xenv = VAR_RES_COMBINATOR f) ∧
  VAR_RES_HOARE_TRIPLE xenv penv P prog Q ⇒
  VAR_RES_HOARE_TRIPLE xenv penv
    (asl_star (VAR_RES_COMBINATOR f) P R) prog
    (asl_star (VAR_RES_COMBINATOR f) Q R)
```

(333) VAR_RES_INFERENCE___prog_kleene_star 101

```
⊢ VAR_RES_HOARE_TRIPLE xenv penv P prog P ⇒
  VAR_RES_HOARE_TRIPLE xenv penv P (asl_prog_kleene_star prog) P
```

(334) VAR_RES_INFERENCE___prog_parallel 101

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  VAR_RES_HOARE_TRIPLE (VAR_RES_COMBINATOR f,lock_env) penv P1 p1
    Q1 ∧
  VAR_RES_HOARE_TRIPLE (VAR_RES_COMBINATOR f,lock_env) penv P2 p2
    Q2 ⇒
  VAR_RES_HOARE_TRIPLE (VAR_RES_COMBINATOR f,lock_env) penv
    (asl_star (VAR_RES_COMBINATOR f) P1 P2) (asl_prog_parallel p1 p2)
    (asl_star (VAR_RES_COMBINATOR f) Q1 Q2)
```

(335) VAR_RES_INFERENCE___prog_seq 101

```
⊢ VAR_RES_HOARE_TRIPLE xenv penv P p1 Q ∧
  VAR_RES_HOARE_TRIPLE xenv penv Q p2 R ⇒
  VAR_RES_HOARE_TRIPLE xenv penv P (asl_prog_seq p1 p2) R
```

(336) VAR_RES_IS_PURE_PROPOSITION_def 112

```
⊢ VAR_RES_IS_PURE_PROPOSITION f P ⇔ ∀s. s ∈ P ⇒ SND s ∈ asl_emp f
```

(337) VAR_RES_IS_PURE_PROPOSITION___pure_proposition 112

```
⊢ VAR_RES_IS_PURE_PROPOSITION f (var_res_stack_proposition f T p)
```

(338) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_REL___REWRITE 92

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_REL e vs ⇔
  FINITE vs ∧ (∀st. IS_SOME (e st) ⇔ vs ⊆ FDOM st) ∧
  ∀st1 st2.
    vs ⊆ FDOM st1 ∧ vs ⊆ FDOM st2 ∧
    (∀v. v ∈ vs ⇒ (FST (st1 ' v) = FST (st2 ' v))) ⇒
    (e st1 = e st2)
```

(339) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET___REWRITE 92

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs e ⇔
  ∃vs'.
    vs' ⊆ vs ∧
    VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_REL e vs'
```

(340) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET___VAR_CONST_EVAL 93, 130

```
⊢ (∀vs c.
     VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
       (var_res_exp_const c)) ∧
  ∀vs v.
    VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
      (var_res_exp_var v) ⇔ v ∈ vs
```

(341) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET___var_res_exp_op 93

```
⊢ EVERY
    (λ e.
       VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs e)
    el ⇒
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
    (var_res_exp_op f el)
```

(342) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET___var_res_exp_var_update 107

```
⊢ VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs e ⇒
  VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs
    (var_res_exp_var_update vc e)
```

(343) VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS___var_res_exp_var_update 107

```
⊢ (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e = SOME vs) ⇒
  (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS
     (var_res_exp_var_update vc e) =
   SOME (vs DELETE FST vc))
```

(344) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___ALTERNATIVE_DEF 94

```
⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs P ⟺
  ∀ s s2.
    s2 ∈ P ∧ (SND s2 = SND s) ∧ FDOM (FST s2) ∩ vs ⊆ FDOM (FST s) ∧
    (∀ v.
       v ∈ FDOM (FST s2) ∧ v ∈ vs ⇒
       (FST (FST s ' v) = FST (FST s2 ' v))) ⇒
    s ∈ P
```

(345) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_and 95

```
⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P1 ∧
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P2 ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (asl_and P1 P2)
```

(346) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_exists_direct 95

```
⊢ (∀ x. VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (P x)) ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (asl_exists x. P x)
```

(347) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_false 95

```
⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs asl_false
```

(348) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_forall 95

```
⊢ (∀ x. VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (P x)) ⇒
  VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (asl_forall x. P x)
```

(349) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_or 95

⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P_1 ∧
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P_2 ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (asl_or P_1 P_2)

(350) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_star 95

⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P_1 ∧
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P_2 ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS
(asl_star (VAR_RES_COMBINATOR f) P_1 P_2)

(351) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_trivial_cond 95

⊢ (c ⇒ VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS P) ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (asl_trivial_cond c P)

(352) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___asl_true 95

⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs asl_true

(353) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___var_res_bigstar 97

⊢ IS_SEPARATION_COMBINATOR f ∧
BAG_EVERY (VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS) sfb ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS (var_res_bigstar f sfb)

(354) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___var_res_bigstar_list 97

⊢ IS_SEPARATION_COMBINATOR f ∧
EVERY (VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS) L ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS exS
(var_res_bigstar_list f L)

(355) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___var_res_exp_full_prop 94

⊢ EVERY (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS_SUBSET vs)
eL ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
(var_res_exp_full_prop P eL)

(356) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___var_res_prop_stack_true 95

⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
(var_res_prop_stack_true f)

(357) VAR_RES_IS_STACK_IMPRECISE___USED_VARS___var_res_prop_var_update___INSERT 108

⊢ VAR_RES_IS_STACK_IMPRECISE___USED_VARS (FST vc INSERT vs) P ⇒
VAR_RES_IS_STACK_IMPRECISE___USED_VARS vs
(var_res_prop_var_update vc P)

(358) var_res_lock_invariant_def 105

```
⊢ var_res_lock_invariant f wp P =
  (λ s.
     (FDOM (FST s) = wp) ∧
     (∀ v. v ∈ wp ⇒ (SND (FST s ’ v) = var_res_write_permission)) ∧
     s ∈
     asl_star (VAR_RES_COMBINATOR f) (var_res_prop_stack_true f) P)
```

(359) var_res_map_def 97

```
⊢ var_res_map f P l = var_res_bigstar_list f (MAP P l)
```

(360) var_res_new_var_init_action_def 109

```
⊢ var_res_new_var_init_action v e s =
  (let e_opt = e (FST s)
   in
     if IS_NONE e_opt then
       NONE
     else if v ∈ FDOM (FST s) then
       SOME ∅
     else
       SOME {var_res_ext_state_var_update (v,THE e_opt) s})
```

(361) VAR_RES_PROGRAM_IS_ABSTRACTION___var_res_prog_assign 108

```
⊢ IS_SEPARATION_COMBINATOR f ∧
  (VAR_RES_IS_STACK_IMPRECISE_EXPRESSION___USED_VARS e = SOME vs) ⇒
  VAR_RES_PROGRAM_IS_ABSTRACTION f (var_res_prog_assign v e)
    (var_res_prog_cond_best_local_action
       (var_res_prop f ({|v|},BAG_OF_SET (vs DELETE v))
          {|var_res_prop_equal f (var_res_exp_var v)
              (var_res_exp_const c)|})
       (var_res_prop f ({|v|},BAG_OF_SET (vs DELETE v))
          {|var_res_prop_equal f (var_res_exp_var v)
              (var_res_exp_var_update (v,c) e)|}))
```

(362) var_res_prog_call_by_value_arg_def 109

```
⊢ var_res_prog_call_by_value_arg prog_body c =
  asl_prog_forall
    (λ x.
       asl_prog_seq
         (var_res_prog_new_var_init x (var_res_exp_const c))
         (asl_prog_seq (prog_body x) (var_res_prog_dispose_var x)))
```

(363) var_res_prog_eval_expressions_def 105

```
⊢ var_res_prog_eval_expressions prog expL =
  asl_prog_choose_constants prog (MAP (λ e s. e (FST s)) expL)
```

(364) var_res_prog_local_var_def 109

```
⊢ var_res_prog_local_var prog_body =
  asl_prog_ndet
    (λ p. ∃ c. p = var_res_prog_call_by_value_arg prog_body c)
```

(365) var_res_prog_parallel_procedure_call_THM 106

```
⊢ var_res_prog_parallel_procedure_call name1 (ref1,expL1) name2
    (ref2,expL2) =
  var_res_prog_eval_expressions
    (λ constL1.
       var_res_prog_eval_expressions
         (λ constL2.
            asl_prog_parallel
              (asl_prog_procedure_call name1 (ref1,constL1))
              (asl_prog_procedure_call name2 (ref2,constL2))) expL2)
    expL1
```

(366) var_res_prog_procedure_call_def 106

```
⊢ var_res_prog_procedure_call name (ref,expL) =
  asl_prog_ext_procedure_call name (ref,MAP (λ e s. e (FST s)) expL)
```

(367) var_res_prop_binexpression___ALTERNATIVE_DEF 96

```
⊢ var_res_prop_binexpression f emp p e1 e2 =
  var_res_prop_expression f emp (λ l. p (HD l) (HD (TL l))) [e1; e2]
```

(368) var_res_prop_equal_def 96

```
⊢ var_res_prop_equal f p1 p2 =
  var_res_prop_binexpression f T (=) p1 p2
```

(369) var_res_prop_expression___ALTERNATIVE_DEF 96

```
⊢ var_res_prop_expression f emp p eL =
  var_res_exp_full_prop (λ vl s. p vl ∧ (s ∈ asl_emp f ∨ ¬emp)) eL
```

(370) var_res_prop_expression___CONS_CONST 96

```
⊢ var_res_prop_expression f emp p (var_res_exp_const c::eL) =
  var_res_prop_expression f emp (λ l. p (c::l)) eL
```

(371) var_res_prop_expression___NIL 96

```
⊢ var_res_prop_expression f emp p [] =
  if emp then var_res_bool_proposition f (p []) else K (p [])
```

(372) var_res_prop_implies_def 115

⊢ var_res_prop_implies f (wpb,rpb) sfb sfb' $\iff$
  var_res_prop_implies_eq f (wpb,rpb) sfb {||} sfb'

(373) var_res_prop_implies_eq_def 115

⊢ var_res_prop_implies_eq f (wpb,rpb) sfb sfb_1 sfb_1' $\iff$
  (var_res_prop f (wpb,rpb) (sfb $\uplus$ sfb_1) =
   var_res_prop f (wpb,rpb) (sfb $\uplus$ sfb_1'))

(374) var_res_prop_implies___UNION 116, 125

⊢ var_res_prop_implies f (wpb,rpb) sfb sfb' $\wedge$
  var_res_prop_implies f (wpb,rpb) sfb sfb'' $\Rightarrow$
  var_res_prop_implies f (wpb,rpb) sfb (sfb' $\uplus$ sfb'')

(375) var_res_prop_input_ap_distinct_def 97

⊢ var_res_prop_input_ap_distinct f (wp,rp) d P =
  asl_and (K (ALL_DISTINCT d))
    (var_res_prop_internal___PROP f ({||},{||}) (wp,rp) {||} P)

(376) var_res_prop_stack_true_REWRITE 95

⊢ var_res_prop_stack_true f = (λ $state$. SND $state$ $\in$ asl_emp f)

(377) var_res_prop_unequal_def 96

⊢ var_res_prop_unequal f p_1 p_2 =
  var_res_prop_binexpression f T (λ n_1 n_2. n_1 $\neq$ n_2) p_1 p_2

(378) var_res_prop_varlist_update___asl_star 108

⊢ VAR_RES_IS_STACK_IMPRECISE p_1 $\wedge$ VAR_RES_IS_STACK_IMPRECISE p_2 $\Rightarrow$
  (var_res_prop_varlist_update vL
     (asl_star (VAR_RES_COMBINATOR f) p_1 p_2) =
   asl_star (VAR_RES_COMBINATOR f)
     (var_res_prop_varlist_update vL p_1)
     (var_res_prop_varlist_update vL p_2))

(379) var_res_prop_varlist_update___BOOL 108

```
⊢ (var_res_prop_varlist_update vcL (asl_and p1 p2) =
    asl_and (var_res_prop_varlist_update vcL p1)
      (var_res_prop_varlist_update vcL p2)) ∧
   (var_res_prop_varlist_update vcL (asl_or p1 p2) =
    asl_or (var_res_prop_varlist_update vcL p1)
      (var_res_prop_varlist_update vcL p2)) ∧
   (var_res_prop_varlist_update vcL (asl_cond p1 p2 p3) =
    asl_cond (var_res_prop_varlist_update vcL p1)
      (var_res_prop_varlist_update vcL p2)
      (var_res_prop_varlist_update vcL p3)) ∧
   (var_res_prop_varlist_update vcL (K cp) = K cp) ∧
   (var_res_prop_varlist_update vcL asl_false = asl_false) ∧
   (var_res_prop_varlist_update vcL (var_res_prop_stack_true f) =
    var_res_prop_stack_true f) ∧
   (var_res_prop_varlist_update vcL (var_res_bool_proposition f cp) =
    var_res_bool_proposition f cp) ∧
   (var_res_prop_varlist_update vcL (asl_exists x. p x) =
    asl_exists x. var_res_prop_varlist_update vcL (p x))
```

(380) var_res_prop_varlist_update___var_res_exp_full_prop 108

```
⊢ var_res_prop_varlist_update vcL (var_res_exp_full_prop P eL) =
  var_res_exp_full_prop P (MAP (var_res_exp_varlist_update vcL) eL)
```

(381) var_res_prop_varlist_update___var_res_prop_expression 108

```
⊢ var_res_prop_varlist_update vcL
    (var_res_prop_expression f emp p el) =
  var_res_prop_expression f emp p
    (MAP (var_res_exp_varlist_update vcL) el)
```

(382) var_res_prop_var_update_def 107

```
⊢ var_res_prop_var_update vc P =
  (λs. var_res_ext_state_var_update vc s ∈ P)
```

(383) var_res_prop_weak_equal_def 96

```
⊢ var_res_prop_weak_equal = var_res_prop_weak_binexpression (=)
```

(384) var_res_prop_weak_expression_def 96

```
⊢ var_res_prop_weak_expression p el =
  var_res_prop_expression ARB F p el
```

(385) var_res_prop_weak_expression_TF 104

```
⊢ (var_res_prop_weak_expression (K T) [] = asl_true) ∧
  (var_res_prop_weak_expression (K F) [] = asl_false)
```

(386) var_res_prop_weak_unequal_def 96

```
⊢ var_res_prop_weak_unequal =
  var_res_prop_weak_binexpression (λ n1 n2. n1 ≠ n2)
```

(387) var_res_prop___COND___REWRITE 98

```
⊢ var_res_prop___COND f (wpb,rpb) sfb ⟺
  FINITE_BAG sfb ∧ IS_SEPARATION_COMBINATOR f ∧
  BAG_ALL_DISTINCT (wpb ⊎ rpb) ∧
  ∀ sf.
    sf ∈: sfb ⇒
    VAR_RES_IS_STACK_IMPRECISE___USED_VARS (SET_OF_BAG (wpb ⊎ rpb))
      sf
```

(388) var_res_prop___equal_const 108

```
⊢ COND_PROP___STRONG_IMP
    (var_res_prop f (wpb,rpb)
       (var_res_prop___var_eq_const_BAG f vcL ⊎ sfb))
    (var_res_prop f (wpb,rpb)
       (var_res_prop___var_eq_const_BAG f vcL ⊎
        BAG_IMAGE (var_res_prop_varlist_update vcL) sfb))
```

(389) var_res_prop___PROP___REWRITE 98

```
⊢ IS_SEPARATION_COMBINATOR f ⇒
  (var_res_prop___PROP f (wpb,rpb) sfb =
   (λ s.
      (∀ v. v ∈: wpb ⇒ var_res_sl___has_write_permission v (FST s)) ∧
      (∀ v. v ∈: rpb ⇒ var_res_sl___has_read_permission v (FST s)) ∧
      s ∈ var_res_bigstar f sfb))
```

(390) var_res_prop___REWRITE 98

```
⊢ var_res_prop f (wpb,rpb) sfb =
  (var_res_prop___COND f (wpb,rpb) sfb,
   if var_res_prop___COND f (wpb,rpb) sfb then
     var_res_prop___PROP f (wpb,rpb) sfb
   else
     asl_false)
```

(391) var_res_quant_best_local_action_def 100

```
⊢ var_res_quant_best_local_action f qP qQ =
  quant_best_local_action f (λ x s. s ∈ qP (FST x) ∧ (s = SND x))
    (λ x s.
       s ∈ qQ (FST x) ∧
       VAR_RES_STACK___IS_EQUAL_UPTO_VALUES (FST (SND x)) (FST s))
```

(392) VAR_RES_STACK_COMBINE_def 91

```
⊢ VAR_RES_STACK_COMBINE =
  BIN_OPTION_MAP (FMERGE VAR_RES_STACK_COMBINE___MERGE_FUNC)
    VAR_RES_STACK_IS_SEPARATE
```

(393) VAR_RES_STACK_COMBINE___IS_SEPARATION_ALGEBRA 91

```
⊢ IS_SEPARATION_ALGEBRA VAR_RES_STACK_COMBINE FEMPTY
```

(394) VAR_RES_STACK_COMBINE___IS_SEPARATION_COMBINATOR 91

```
⊢ IS_SEPARATION_COMBINATOR VAR_RES_STACK_COMBINE
```

(395) VAR_RES_STACK_IS_SEPARATE_def 91

```
⊢ VAR_RES_STACK_IS_SEPARATE s1 s2 ⟺
  ∀x.
    x ∈ FDOM s1 ∧ x ∈ FDOM s2 ⇒
    (FST (s1 ' x) = FST (s2 ' x)) ∧
    IS_SOME
      (var_res_permission_combine (SOME (SND (s1 ' x)))
         (SOME (SND (s2 ' x))))
```

(396) VAR_RES_STACK_IS_SUBSTATE_REWRITE 91

```
⊢ VAR_RES_STACK_IS_SUBSTATE st1 st2 ⟺
  FDOM st1 ⊆ FDOM st2 ∧
  ∀v.
    v ∈ FDOM st1 ⇒
    (FST (st1 ' v) = FST (st2 ' v)) ∧
    IS_VAR_RES_SUBPERMISSION (SND (st1 ' v)) (SND (st2 ' v))
```

(397) VAR_RES_STACK___IS_EQUAL_UPTO_VALUES_def 98

```
⊢ VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st1 st2 ⟺
  (∀x.
     x ∈ FDOM st1 ∧ x ∈ FDOM st2 ⇒ (SND (st1 ' x) = SND (st2 ' x))) ∧
  (∀x.
     x ∈ FDOM st1 ∧ x ∉ FDOM st2 ⇒
     (SND (st1 ' x) = var_res_write_permission)) ∧
  ∀x.
    x ∉ FDOM st1 ∧ x ∈ FDOM st2 ⇒
    (SND (st2 ' x) = var_res_write_permission)
```

(398) VAR_RES_STACK___IS_EQUAL_UPTO_VALUES___REFL 101

```
⊢ VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st st
```

(399) VAR_RES_STACK___IS_EQUAL_UPTO_VALUES___SYM 101

⊢ VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_1 st_2 $\iff$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_2 st_1

(400) VAR_RES_STACK___IS_EQUAL_UPTO_VALUES___TRANS 101

⊢ VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_1 st_2 $\wedge$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_2 st_3 $\Rightarrow$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_1 st_3

(401) VAR_RES_STACK___IS_EQUAL_UPTO_VALUES___VAR_RES_STACK_COMBINE 101

⊢ (VAR_RES_STACK_COMBINE (SOME st_{11}) (SOME st_{12}) = SOME st_1) $\wedge$
(VAR_RES_STACK_COMBINE (SOME st_{21}) (SOME st_{22}) = SOME st_2) $\wedge$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_{11} st_{21} $\wedge$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_{12} st_{22} $\Rightarrow$
VAR_RES_STACK___IS_EQUAL_UPTO_VALUES st_1 st_2

(402) var_res_state_var_update_def 107

⊢ var_res_state_var_update v c s =
s |+ (v,c,var_res_write_permission)