Technical Report

UCAM-CL-TR-948
ISSN 1476-2986

Number 948

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Reasoning about effectful programs and evaluation order

Dylan McDermott

June 2020

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

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

This technical report is based on a dissertation submitted October 2019 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Homerton College.

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

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

ISSN 1476-2986

Abstract

Program transformations have various applications, such as in compiler optimizations. These transformations are often effect-dependent: replacing one program with another relies on some restriction on the side-effects of subprograms. For example, we cannot eliminate a dead computation that raises an exception, or a duplicated computation that prints to the screen. Effect-dependent program transformations can be described formally using effect systems, which annotate types with information about the side-effects of expressions.

In this thesis, we extend previous work on effect systems and correctness of effect-dependent transformations in two related directions.

First, we consider evaluation order. Effect systems for call-by-value languages are well-known, but are not sound for other evaluation orders. We describe sound and precise effect systems for various evaluation orders, including call-by-name. We also describe an effect system for Levy's call-by-push-value, and show that this subsumes those for call-by-value and call-by-name. This naturally leads us to consider effect-dependent transformations that replace one evaluation order with another. We show how to use the call-by-push-value effect system to prove the correctness of transformations that replace call-by-value with call-by-name, using an argument based on logical relations. Finally, we extend call-by-push-value to additionally capture call-by-need. We use our extension to show a classic example of a relationship between evaluation orders: if the side-effects are restricted to (at most) nontermination, then call-by-name is equivalent to call-by-need.

The second direction we consider is non-invertible transformations. A program transformation is non-invertible if only one direction is correct. Such transformations arise, for example, when considering undefined behaviour, nondeterminism, or concurrency. We present a general framework for verifying noninvertible effect-dependent transformations, based on our effect system for call-by-push-value. The framework includes a non-symmetric notion of correctness for effect-dependent transformations, and a denotational semantics based on order-enriched category theory that can be used to prove correctness.

Acknowledgements

Most importantly, I thank Alan Mycroft for his enthusiastic supervision during my PhD. We had many useful discussions about a wide range of topics, and he provided a lot encouragement, and advice on how to write papers.

I also thank all of the people who I had helpful technical discussions with, including Paul Downen, Paul Blain Levy, Martin Hyland, Dominic Orchard, Tomas Petricek, Sam Staton, and Tarmo Uustalu. I'm particularly grateful to Ohad Kammar for collaboration and many conversations about category theory and denotational semantics.

My friends and colleagues at the Computer Lab deserve thanks for their help and support. My office-mate Ian Orton provided many entertaining (and occasionally useful) distractions after 12:30pm, along with Sam Ainsworth, Aurore Alcolei, Matthew Daggitt, Hugo Paquet and Philip Saville, Nathanael Arkor, Adam Ó Conghaile, Andrej Ivašković, Ben Simner, Shaun Steenkamp and Dima Szamozvancev. And probably some others I forgot to mention. I also thank the admin staff at the CL, especially Megan Sammons for her tireless work organising supervisions.

I'm grateful to Matthew Parkinson, Dimitrios Vytiniotis and others at MSR Cambridge for an interesting internship project, discussions about PhDs, and a surprisingly large amount of fun debugging garbage collectors.

Finally, I thank the EPSRC for funding this work.

Contents

Chapter 1

Introduction

At a high level, the question considered in this thesis is:

When is it correct to replace one program phrase with another?

Suppose we are writing an optimizing compiler. The optimizations are applied locally, replacing expressions e_1 (which form part of the program) with other expressions e_2. Setting aside the question of whether replacing e_1 with e_2 actually improves the program (we do not consider this), we want to know whether it is *valid* to do so. The output of the compiler should not do anything the original program could not, so it is valid to replace e_1 with e_2 when *every observable behaviour of* e_2 *is an observable behaviour of* e_1. We develop techniques for formal reasoning about programs that can be used to prove the validity of program transformations.

Of course, formal reasoning about programs is not a new area. To motivate the specific problems we consider, we give three examples of common program transformations:

	Old expression	Replacement
Dead code elimination (where x not free in e')	$\textbf{let}\ x = e\ \textbf{in}\ e'$	e'
Inlining	$(\lambda x.\, e)\ e'$	$e'[x \mapsto e]$
Common subexpression elimination	$\textbf{let}\ x_1 = e\ \textbf{in}\ \textbf{let}\ x_2 = e\ \textbf{in}\ e'\ (x_1, x_2)$	$\textbf{let}\ x = e\ \textbf{in}\ e'\ (x, x)$

These examples highlight three important aspects of program transformations:

- *They are often effect-dependent.* The first example is valid if e has no side-effects, but not e.g. if e changes some state that e' then reads from, or if e raises an exception. (For these examples, **let** represents eager evaluation, so first e is evaluated, and then e' is.) The first example is therefore *effect-dependent*, which means applying it requires static knowledge of the side-effects of the expressions involved. The wide array of side-effects available in practical languages means that a majority of the transformations we would like to perform are effect-dependent. We therefore focus on verifying effect-dependent transformations throughout this thesis, continuing a long line of work started by Tolmach [95] and Benton et al. [12].
- *They often depend on evaluation order.* Consider the second example. This is clearly valid in call-by-name languages (it is just β-reduction), even with arbitrary side-effects. It is not in general valid in call-by-value languages. We cannot consider program transformations

without thinking about evaluation order. Even worse, the side-effects of expressions depend on evaluation order, so we cannot validate *any* effect-dependent transformations without considering evaluation order. Previous work on effect-dependent transformations has mostly considered only a single evaluation order (usually call-by-value). We focus on capturing a range of evaluation orders. This aspect also suggests another useful class of program transformations: those that replace one evaluation order with another, such as call-by-value with call-by-name.

- *They are often noninvertible.* One way to justify the validity of the third example (or in general, any program transformation) is to show that both sides have the *same* semantics. This is the method normally used, but it is unsuitable for many examples. Consider the third example, and suppose that the only side-effect of e is to make nondeterministic choices internally. We could allow the compiler to make nondeterministic choices statically, by replacing the expression $(e_1 \textbf{ or } e_2)$ with e_1 or with e_2. If this is the case, then we can replace the expression on the left of the example, which makes the choices in e twice, with the one on the right, even though it does not have the same semantics. Program transformations are in general *noninvertible*: only one direction is valid.

This thesis constructs a general framework for proving the validity of program transformations with all three of these aspects.

1.1 Approach

We have stated our aim, now we explain the approach that we take. Throughout, we express program transformations inside some *intermediate language* that we construct for this purpose. Given an expression that we want to transform, the first step is to *translate* it into our intermediate language of choice. We then reason about the translation of the expression. The intermediate languages we use in this thesis (GCBPV in Section 2.7, ECBPV in Section 5.1) are based on Levy's [56] call-by-push-value calculus (CBPV). (We use e for expressions in source languages, and M for terms in intermediate languages, so each expression e can be translated into a term M.)

Using intermediate languages in this way has several advantages. They can be designed specifically for validating program transformations, ignoring considerations that apply to source languages (we do not need to write programs directly in intermediate languages, because intermediate-language programs are constructed by the translations). In particular, the intermediate language can be close to the denotational semantics. (For CBPV, several side-effects have simpler models than e.g. for monadic intermediate languages [78]. This is one of the reasons why CBPV is useful for verifying program transformations.)

However, the main advantage of intermediate languages is that we can design them so that evaluation order is irrelevant (for a given intermediate-language program, every evaluation order gives the same behaviour). The evaluation order of the source language is encoded in the intermediate language *programs* (in the syntax) by the translation. We use intermediate languages that capture several evaluation orders. CBPV and the variants we use capture both call-by-value and call-by-name, so for every source-language expression e we have two intermediate-language terms: a call-by-value translation $(\!|e|\!)^{\mathrm{v}}$ and a call-by-name translation $(\!|e|\!)^{\mathrm{n}}$. (We also have other translations in Chapters 2 and 5.) Encoding evaluation orders inside programs in this way allows us to consider transformations that replace one evaluation order with another inside a single expression, since they really can just transform the syntax of the program. There is no change in the semantics of the intermediate language itself. For call-by-value and call-by-name, we relate the behaviours of $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$, which are terms in the same language.

We use *effect systems* [63] to enable *effect-dependent* transformations, following previous work. Effect systems refine type systems so that they statically (over-)estimate the side-effects of expressions. (We do not attempt to define[1] *side-effect* precisely, but we use a broad interpretation, which includes at least nontermination, nondeterministic and probabilistic choice, and exceptions.) Effect systems are the traditional way of working with effect-dependent transformations. They are a lightweight technique (requiring only minor modifications to type systems), and are expressive (we can use them to restrict the order and the number of times that side-effects are used, not just *whether* particular side-effects are used). They are also amenable to mechanization (for example, type inference algorithms can be extended to *effect* inference algorithms). We design several effect systems to account for the fact that the side-effects of expressions depend on evaluation order.

We formally define *validity* of program transformations in terms of *contextual preorders* $\preccurlyeq_{\mathrm{ctx}}$. It is correct to replace M with N when $M \preccurlyeq_{\mathrm{ctx}} N$, which means that every observable behaviour of N is an observable behaviour of M. For example, if the only side-effect is nondeterministic choice, this might mean that every possible result of a program containing N is a possible result of the same program but containing M instead: the transformation does not make the set of possible results of a program larger. If we want to validate a transformation that replaces e with e' inside a call-by-value source language, we show that $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} (\!|e'|\!)^{\mathrm{v}}$. Contextual preorders might not be symmetric in examples; this allows us to consider noninvertible transformations.

Comparing evaluation orders requires more work than just asking whether the contextual preorder relates them. Everything we do is typed, and it only makes sense to ask if $M \preccurlyeq_{\mathrm{ctx}} N$ if M and N have the same type, but in general this is not the case if M and N are two different translations of the same expression. Call-by-name functions take (thunked) computations as arguments, call-by-value functions take values as arguments (the distinction is important in CBPV), so if e is a function, it does not make sense to replace the call-by-value translation $(\!|e|\!)^{\mathrm{v}}$ with the call-by-name translation $(\!|e|\!)^{\mathrm{n}}$. To fix this, we need to add some extra glue to the transformations. To pass a call-by-name argument to a call-by-value function it needs to be evaluated first; to pass a call-by-value argument to a call-by-name function we need to convert it into a trivial computation. To compare call-by-value and call-by-name, we therefore construct maps between the two evaluation orders, and compose these with the two translations (this is similar to the technique used by Reynolds [89] to relate direct and continuation semantics). Other comparisons between evaluation orders require similar considerations.

We define contextual preorders in terms of *inequational theories*, which are specified by axioms characterizing the constructs in the intermediate language. These axioms include β- and η-laws, and also laws capturing the behaviour of the side-effects, for example, associativity of nondeterministic choice, or that two reads from the same location can be merged. Different axioms are used for different side-effects. We prefer inequational theories to e.g. operational semantics for two reasons: theories allow us to apply inequations under lambdas (which is important because we mainly consider open terms), and they allow uniformly specifying various side-effects (e.g. the usual way of specifying operational semantics for global state is to carry around an extra state parameter; this is not required for inequational theories).

To prove instances of contextual preorders, and hence to prove the validity of program transformations, we employ two different techniques. The first is to use logical relations on

[1] The best we can do is cheat by saying that a side-effect is roughly anything that we can capture in the languages we use. (This includes the usual examples.) A proper definition could be along the lines of Sabry [91]: a side-effect is anything that breaks equivalences between evaluation orders. (Though unlike Sabry we do not treat divergence and errors as special cases.)

the syntax (type-indexed families of binary relations on terms). These lie somewhere between inequational theories and contextual preorders. We construct them so that they relate more pairs of terms than the inequational theory (in most cases, the program transformations we consider are not instances of the inequational theory), and also so that they only relate M to N when $M \preccurlyeq_{\mathrm{ctx}} N$ (which means we can actually use them to validate program transformations). We use logical relations to relate call-by-value and call-by-name executions of expressions in Chapter 3. However, they turn out to be quite cumbersome to use. We therefore prefer the second technique, which is to use denotational semantics.

We define a *categorical* semantics of GCBPV in Chapter 4, and use it to verify noninvertible program transformations. The semantics allows for models in various categories, which allows us to model as wide a range of side-effects as possible. The categories we use are *order-enriched*, which enables us to reason about noninvertible transformations: to show $M \preccurlyeq_{\mathrm{ctx}} N$ holds, we show $[\![M]\!] \sqsubseteq [\![N]\!]$, where $\sqsubseteq$ is part of the data of the model.

When we relate call-by-name and call-by-need evaluation in Chapter 5 we are forced to go back to logical relations on the syntax because we do not have a general call-by-need denotational semantics. We actually need to use a more powerful notion of logical relation than previously (*Kripke logical relations of varying arity* [39]), because of the extra difficulties that call-by-need introduces.

Our framework for validating program transformations therefore has several components. We have various source-language effect systems, one for each evaluation order we wish to consider. We also have intermediate languages that capture all of these evaluation orders via translations from source languages ($(\!|-|\!)^{\mathrm{v}}$, $(\!|-|\!)^{\mathrm{n}}$, etc.). We define a notion of inequational theory for each intermediate language, and use these to define contextual preorders. Finally, we also include machinery (logical relations and denotational semantics) for proving instances of contextual preorders. Together, these components enable us to validate of all of the kinds mentioned above.

1.2 Contributions

This thesis develops a framework for proving the correctness of effect-dependent program transformations:

- We develop tools for formulating effect-dependent transformations for various evaluation orders, including two forms of call-by-name (Chapter 2). This includes source-language effect systems for each of the evaluation orders we consider, and an intermediate language (GCBPV) we use for reasoning throughout most of this thesis.

- We establish a novel reasoning principle for relating call-by-value and call-by-name (Chapter 3), which enables us to prove the validity of program transformations that replace call-by-value with call-by-name, and vice versa. We technique we use to do this is new, and we expect it to work more generally (to relate other evaluation orders).

- We develop machinery for proving the validity of noninvertible program transformations (Chapter 4), including an order-enriched denotational semantics for GCBPV. We apply this to examples including undefined behaviour, nondeterminism, and shared global state.

- We extend GCBPV to additionally capture call-by-need evaluation (Chapter 5). This extension enables us to give a new proof technique for relating evaluation orders. We exemplify the technique by proving the classic result that call-by-name and call-by-need are equivalent if the only side-effect is nontermination.

Chapter 2

Effect systems and evaluation orders

We formalize effect-dependent program transformations using *effect systems*. The goal of an effect system is to statically associate computations with *effects* ε, which abstractly represent the side-effects the computation has. Hence by requiring computations to be associated with a chosen effect ε, we can view effect systems as a way to restrict the side-effects of computations.

As noted in the introduction, different expressions may have different side-effects depending on the evaluation order used. The effect system used must therefore reflect the evaluation order. Using a call-by-value effect system to analyse a call-by-name language could lead to imprecise results (leading us to miss valid effect-dependent transformations), or worse, might be unsound (leading us to incorrectly believe certain transformations are valid).

To reason about transformations that change the evaluation order it is useful to have a single *intermediate* language that allows programs to express their evaluation order, rather than one language for each evaluation order. Two such languages are Moggi's *monadic metalanguage* [78] and Levy's *call-by-push-value* [56]. We can use these to reason about *source* languages with a single evaluation order by translating from source to intermediate. Since valid changes in evaluation order are also effect-dependent, we therefore also wish to define effect systems for intermediate languages.

This chapter serves two purposes. First, it provides a general introduction to effect systems. In particular we explain *effect algebras* (Section 2.1), which provide the abstract notion of *effect* we track, the well-known call-by-value effect system (Section 2.3), and the less well-known effect system for the monadic metalanguage (Section 2.5). Second, it also makes two key contributions:

- We describe effect systems for two different forms of call-by-name (Sections 2.4 and 2.6). These allow us to do effect-dependent reasoning in call-by-name languages. We contrast the two forms of call-by-name in Section 2.6.
- We describe an effect system for call-by-push-value (Section 2.7), and show that translations from source languages into call-by-push-value respect the effect systems. We use this call-by-push-value effect system as the basis for work in later chapters of this thesis.

Source and intermediate languages Before continuing, we clarify what we mean by *source* and *intermediate* languages. If a language is intended primarily to be used for reasoning (e.g. proving correctness of program transformations), then we call it an intermediate language. In this chapter, the intermediate languages are the monadic metalanguage (Section 2.5) and call-by-push-value (Section 2.7). We focus on making them easy to reason about (for example, by ensuring that their effect systems are syntax-directed, and by making sequencing of side-effecting computations explicit). Source languages are intended to more closely resemble programming languages (in particular, sequencing of computations is implicit as it is in ML), and formal reasoning about source languages is intended to be done by translating source programs into an intermediate language, and then reasoning in the intermediate language. In this thesis, we concentrate mainly on the intermediate languages, and hence we keep the source languages as small as possible (we only include booleans, unit type and function types).

2.1 Effect algebras

An *effect* ε abstractly represents some collection of side-effects. Effect systems statically assign effects to program fragments; when the effect ε is assigned, this means the possible runtime side-effects of the program fragment are restricted to ε. To do this, we require the effects to have some extra structure, called an *effect algebra*[1]. There are various forms of effect algebra. Commonly effects are assumed to form semilattices, but other structures, such as *effectors* [94] or *effect quantales* [31] are also used. Here we follow Katsumata [45] and use *preordered monoids*.

Definition 2.1.1 (Preordered monoid) A *preordered monoid* $(\mathcal{E}, \leq, \cdot, 1)$ consists of a monoid $(\mathcal{E}, \cdot, 1)$ and a preorder $\leq$ on $\mathcal{E}$, such that the binary operation $\cdot$ is monotone in each argument separately. ◀

Preordered monoids have several advantages: they are simple, and they generalize the other forms of effect algebra we have just mentioned (Katsumata postulates that effects *always* form a preordered monoid). They can also be easily extended with additional data (such as iteration operators for recursion).

The effects ε are elements of the set $\mathcal{E}$. The *subeffecting relation* $\leq$ provides a notion of approximation of effects: $\varepsilon \leq \varepsilon'$ means ε is more restrictive than ε'. The *multiplication* $\cdot$ represents sequencing of effects: $\varepsilon \cdot \varepsilon'$ is the effect of running a computation with effect ε followed by a computation with effect ε'. Finally, the *unit* 1 is used for computations with no side-effects. Each effect system is parameterized by the effect algebra, so we can instantiate an effect system with different effect algebras for different use cases.

The simplest effect algebra is the trivial preordered monoid, in which $\mathcal{E}$ is a singleton $\{\star\}$. When instantiated with the trivial preordered monoid, effect systems do not track effect information, and are the same as ordinary type systems. More interesting examples are *Gifford-style effect algebras*, which are used for example by Lucassen and Gifford [63].

Example 2.1.2 *Gifford-style* effect algebras have the form $(\mathcal{P}\Sigma, \subseteq, \cup, \emptyset)$, where Σ is some set of operations (e.g. $\Sigma := \{\mathsf{raise}, \mathsf{get}, \mathsf{put}\}$), and $\mathcal{P}\Sigma$ is the powerset. In this case, effects $\varepsilon \subseteq \Sigma$ give the set of operations that a computation *may* use during its execution. For example, a computation with effect $\{\mathsf{raise}, \mathsf{get}\}$ may raise an exception or get the value of the state, but does not change the value of the state. The subeffecting relation $\subseteq$ allows additional operations (that are not used) to be included in the effect (this is useful to balance the two branches of **if**-expressions). We can use a Gifford-style effect system to express effect-dependent program transformations. For example, consider

$$\begin{array}{lll} \mathbf{let}\; x = e\; \mathbf{in} & \qquad\qquad & \mathbf{let}\; x = e\; \mathbf{in} \\ \mathbf{let}\; y = e\; \mathbf{in} & & \\ (x, y) & & (x, x) \end{array}$$

[1] For the avoidance of doubt, these have nothing to do with effect algebras in quantum theory [22].

If the effect of e is $\{\mathsf{raise}, \mathsf{get}\}$, then it is correct to replace the expression on the left with the one on the right. However, if e uses put, this might not be the case. ◀

This example gives a *may* analysis: it allows overapproximation of effects. In general, if a preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ describes a may analysis, then the preordered monoid $(\mathcal{E}, \geq, \cdot, 1)$ (which is the same but with the opposite order) describes the corresponding must analysis that allows under- instead of over-approximation.

Example 2.1.3 Consider a language with an operation flip that nondeterministically chooses either **true** or **false**. We can use the preordered monoid $(\mathbb{N}_+, \leq, \cdot, 1)$, where $\mathbb{N}_+$ is the set of positive integers, $\cdot$ is the usual integer multiplication, and the unit is the integer 1, to statically (over-)estimate the number of potential results of computations. The effect of flip would be the integer 2. ◀

Example 2.1.4 Consider a language with global state, including operations get and put that read from and write to the state. In general, to evaluate an expression we might have to supply an initial state, but if put must be used before get, then we do not. We can determine whether this is the case statically using a preordered monoid with underlying preorder $\{\mathsf{pf} \leq 1 \leq \mathsf{gf}\}$. The effect pf means must set the value of the state, and does not get the value before setting it; the unit 1 means does not get before setting the value, but may do neither; and gf means may get the value first. The multiplication is defined by:

$$\mathsf{pf} \cdot \varepsilon \; = \; \mathsf{pf} \qquad 1 \cdot \varepsilon \; = \; \varepsilon \qquad \mathsf{gf} \cdot \varepsilon \; = \; \mathsf{gf}$$

The effect of applying get is gf, and the effect of applying put is pf. Programs that have the effect pf or the effect 1 do not require an initial state. For example, if e' has effect pf, then it is correct to replace $(e;\, e')$ with e'. Even if e changes the state, it will be overwritten by e', and hence will not change the behaviour of e'. ◀

In some cases, it is useful to assume effect algebras with more structure than just a preordered monoid. For example, for languages that include fixed points, we might assume an operator $(-)^*$ that assigns to each effect $\varepsilon \in \mathcal{E}$ an effect $\varepsilon^* \in \mathcal{E}$ of a recursive computation. Mycroft et al. [80] discuss several cases in which we want additional structure. In this thesis, to keep the discussion general, we mostly assume only a preordered monoid. We will occasionally require $\mathcal{E}$ to have *bounded binary joins*:

Definition 2.1.5 A *partially ordered monoid* is a preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ in which $\leq$ is antisymmetric. An element $\varepsilon' \in \mathcal{E}$ is an *upper bound* of $\varepsilon_1, \varepsilon_2 \in \mathcal{E}$ if $\varepsilon_1 \leq \varepsilon'$ and $\varepsilon_2 \leq \varepsilon'$. It is the *join* of ε_1 and ε_2 if additionally $\varepsilon' \leq \varepsilon''$ for all upper bounds ε''. A partially ordered monoid has *bounded binary joins* if for each pair of elements $\varepsilon_1, \varepsilon_2 \in \mathcal{E}$, existence of an upper bound implies the existence of a join $\varepsilon_1 \vee \varepsilon_2$. ◀

In this definition, we restrict to partial orders so that joins are unique. All of the above examples are partially ordered monoids with bounded binary joins. Joins are also useful e.g. for effect inference, but we do not discuss inference in this thesis.

2.2 Simply-typed lambda calculus

The source language we consider is based on the simply-typed lambda calculus. In later sections of this chapter we consider various evaluation orders, and give effect systems that correspond to

$$\frac{}{\Gamma \vdash x : \tau}\ \text{if } (x : \tau) \in \Gamma \qquad \frac{\Gamma \vdash e : \mathrm{car}_{\mathrm{op}}}{\Gamma \vdash \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}}} \qquad \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}$$

Figure 2.1: Source language type system

them. Here we give the syntax and type system of the source language without yet considering how to track effects.

The syntax of source-language types τ and expressions e is as follows:

$$\begin{aligned} \tau &::= \mathbf{unit} \mid \mathbf{bool} \mid \tau \to \tau' \\ e &::= x \mid \mathrm{op}\, e \mid () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \end{aligned}$$

Most of the syntax is standard. The particular choice of types is unimportant; we choose the unit type, booleans and function types because we use them in examples.

We assume some set Σ of *operations*, ranged over by op. The purpose of these is to provide side-effects. We could for example have operations get and put for interacting with global state, or raise for raising exceptions. Each operation takes exactly one argument; hence the syntax of expressions includes op e, which means apply the operation op to the argument e. We can include nullary operations by using **unit** for the argument. If we had included products in the syntax we could also have operations with more than one argument (we cannot get multiple arguments by currying because of the restriction to ground types below). We do not give any examples involving operations of more than one argument, so do not need to consider product types here.

The operations op are *generic effects* [85]. Including operations in this way, rather than for example just assuming an arbitrary collection of constants c, is useful when we consider equivalences between evaluation orders (see Chapter 3).

To specify the type system, we assume an assignment of types for the operations op. The *ground types* are **unit** and **bool** (i.e. types that do not include functions). We restrict the operations to ground types (again this is useful for equivalences between evaluation orders). We collect the data required to specify the type system into a notion of *signature*. Signatures also include an effect algebra. This effect algebra is not used in this section, but is used in each of our effect systems.

Definition 2.2.1 (Signature) A *(source-language) signature* consists of the following data:

- A preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ of *effects*.
- A set Σ of *operations*.
- For each operation $\mathrm{op} \in \Sigma$, ground types $\mathrm{car}_{\mathrm{op}}$ and $\mathrm{ar}_{\mathrm{op}}$, respectively called the *coarity* and *arity* of op, and an effect $\mathrm{eff}_{\mathrm{op}} \in \mathcal{E}$. ◀

$$\frac{}{\Gamma \vdash_{\mathrm{v}} x : \tau \mathbin{\&} 1}\ \text{if } (x : \tau) \in \Gamma \qquad \frac{\Gamma \vdash_{\mathrm{v}} e : \mathrm{car}_{\mathrm{op}} \mathbin{\&} \varepsilon}{\Gamma \vdash_{\mathrm{v}} \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \mathbin{\&} \varepsilon \cdot \mathrm{eff}_{\mathrm{op}}} \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} () : \mathbf{unit} \mathbin{\&} 1}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{true} : \mathbf{bool} \mathbin{\&} 1} \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{false} : \mathbf{bool} \mathbin{\&} 1}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} e_1 : \mathbf{bool} \mathbin{\&} \varepsilon \qquad \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon' \qquad \Gamma \vdash_{\mathrm{v}} e_3 : \tau \mathbin{\&} \varepsilon'}{\Gamma \vdash_{\mathrm{v}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \mathbin{\&} \varepsilon \cdot \varepsilon'}$$

$$\frac{\Gamma, x : \tau \vdash_{\mathrm{v}} e : \tau' \mathbin{\&} \varepsilon}{\Gamma \vdash_{\mathrm{v}} \lambda x{:}\tau.\, e : \tau \xrightarrow{\varepsilon} \tau' \mathbin{\&} 1} \qquad \frac{\Gamma \vdash_{\mathrm{v}} e_1 : \tau \xrightarrow{\varepsilon_3} \tau' \mathbin{\&} \varepsilon_1 \qquad \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon_2}{\Gamma \vdash_{\mathrm{v}} e_1\, e_2 : \tau' \mathbin{\&} \varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon}{\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon'}\ \text{if } \varepsilon \leq \varepsilon'$$

Figure 2.2: Call-by-value effect system

The coarity of an operation is the type of its argument; the arity is the type of its result.[2] For example, for global state storing a value of type **bool**, we would have an operation get with coarity $\mathrm{car}_{\mathrm{get}} \coloneqq \mathbf{unit}$ and arity $\mathrm{ar}_{\mathrm{get}} \coloneqq \mathbf{bool}$, and an operation put with coarity **bool** and arity **unit**. For binary nondeterministic choice we would have an operation flip with coarity **unit** and arity **bool**, which nondeterministically chooses between **true** and **false**. (We can use this to make one form of nondeterministic choice between two expressions e_1, e_2 of an arbitrary type by writing **if** flip () **then** e_1 **else** e_2.)

Each operation op is also associated with the effect $\mathrm{eff}_{\mathrm{op}}$ of using it. One option would be to use a Gifford-style effect algebra (Section 2.1), so that $\mathrm{eff}_{\mathrm{op}} \coloneqq \{\mathrm{op}\}$ for each $\mathrm{op} \in \Sigma$, and the preordered monoid is $(\mathcal{P}\Sigma, \subseteq, \cup, \emptyset)$. However, we allow other effect algebras to be used.

We assume a fixed signature to specify the type system. A *typing context* Γ is an ordered list of (variable, type) pairs such that no variable appears more than once. We write $\diamond$ for the empty typing context. The typing judgment $\Gamma \vdash e : \tau$ is defined inductively by the rules in Figure 2.1. Rules that add variables to typing contexts implicitly assume that those variables are fresh. (This is the case for all of the typing rules in this thesis.)

2.3 Call-by-value

We augment the source language type system so that it tracks the effects of expressions. The first effect system we present is well-known: it is for the traditional example of a call-by-value source language. The call-by-value effect system consists of a typing judgment of the form $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$, which assigns to each expression e an effect ε in addition to the type τ.

[2] The reader may be concerned that these are the wrong way around, since the arity should specify the type of the argument and the coarity the type of the result. However, *generic effects* with argument type τ and result type τ' are in bijection with *algebraic operations* [85], which are τ''-indexed families of maps from $(\tau' \to \tau'')$ to $(\tau \to \tau'')$ satisfying certain conditions. When taking the algebraic operations view, the arity and coarity are the correct way around. Our usage of the terms arity and coarity matches e.g. Katsumata [44].

$$\frac{e \overset{\mathrm{v}}{\rightsquigarrow} e'}{\mathrm{op}\, e \overset{\mathrm{v}}{\rightsquigarrow} \mathrm{op}\, e'} \qquad \begin{array}{l} \textbf{if true then } e_2 \textbf{ else } e_3 \overset{\mathrm{v}}{\rightsquigarrow} e_2 \\ \textbf{if false then } e_2 \textbf{ else } e_3 \overset{\mathrm{v}}{\rightsquigarrow} e_3 \end{array} \qquad (\lambda x\!:\!\tau.\, e)\, v \overset{\mathrm{v}}{\rightsquigarrow} e[x \mapsto v]$$

$$\frac{e_1 \overset{\mathrm{v}}{\rightsquigarrow} e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \overset{\mathrm{v}}{\rightsquigarrow} \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3} \quad \frac{e_1 \overset{\mathrm{v}}{\rightsquigarrow} e_1'}{e_1\, e_2 \overset{\mathrm{v}}{\rightsquigarrow} e_1'\, e_2} \quad \frac{e_2 \overset{\mathrm{v}}{\rightsquigarrow} e_2'}{(\lambda x\!:\!\tau.\, e)\, e_2 \overset{\mathrm{v}}{\rightsquigarrow} (\lambda x\!:\!\tau.\, e)\, e_2'}$$

Figure 2.3: Call-by-value reduction relation

To add effects, we replace the syntax of types and expressions with the following:

$$\begin{aligned} \tau, \tau' &::= \textbf{unit} \mid \textbf{bool} \mid \tau \xrightarrow{\varepsilon} \tau' \\ e &::= x \mid \mathrm{op}\, e \mid () \mid \textbf{true} \mid \textbf{false} \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \end{aligned}$$

The only difference with the previous syntax is that each function type is annotated with a *latent* effect ε. The latent effect is the effect of applying the function.

We again assume a fixed source-language signature (Definition 2.2.1) to specify the type system. Since arities and coarities are required to be ground types, it does not matter that the syntax of types has changed (the ground types are still **unit** and **bool**). Typing contexts Γ are ordered lists of (variable, type) pairs without repetitions, as before. The typing judgment $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ is defined inductively by the rules in Figure 2.2. For operations, we multiply the effect of the argument by the effect of the operation itself (the argument is evaluated before the operation is performed). Function application evaluates the function, then the argument, and then the body of the function; hence the effect in the conclusion of the typing rule for application is $\varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3$. A different evaluation order would require a different effect here. The final rule is subeffecting, which allows effects to be overapproximated. The overapproximation is necessary for programs that use **if**: we do not attempt to determine which branch will be taken, so we require both branches to have the same effect.

We give a call-by-value operational semantics for the source language. It consists of a small-step reduction relation $\overset{\mathrm{v}}{\rightsquigarrow}$. The definition is straightforward: it is the smallest relation closed under the rules in Figure 2.3. To specify β-reduction of functions, we use the following syntax of *values* v, which form a subset of expressions:

$$v ::= () \mid \textbf{true} \mid \textbf{false} \mid \lambda x\!:\!\tau.\, e$$

Values are pure: if $\Gamma \vdash_{\mathrm{v}} v : \tau \mathbin{\&} \varepsilon$ for any effect ε, then $\Gamma \vdash_{\mathrm{v}} v : \tau \mathbin{\&} 1$. (The proof is a trivial induction on the typing derivation.) We also use capture-avoiding substitution $e[x \mapsto v]$. There are no rules for reducing $\mathrm{op}\, v$. To give a complete operational semantics to an instantiation of the source language, one would augment the rules in the figure to characterize the behaviour of the operations, but we do not consider this here. (Though we do consider this for call-by-push-value; see Definition 2.7.4.)

Call-by-value reductions preserve the effect (and type) assigned by the call-by-value effect system. (Recall that $\diamond$ is the empty typing context.)

Theorem 2.3.1 (Subject reduction for call-by-value) If $\diamond \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ and $e \overset{\mathrm{v}}{\rightsquigarrow} e'$ then $\diamond \vdash_{\mathrm{v}} e' : \tau \mathbin{\&} \varepsilon$. ◀

$$\frac{}{\Gamma \vdash_{\text{moggi}} x : \tau \mathbin{\&} \varepsilon} \text{ if } (x : \tau\&\varepsilon) \in \Gamma \qquad \frac{\Gamma \vdash_{\text{moggi}} e : \mathrm{car}_{\mathrm{op}} \mathbin{\&} \varepsilon}{\Gamma \vdash_{\text{moggi}} \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \mathbin{\&} \varepsilon \cdot \mathrm{eff}_{\mathrm{op}}} \qquad \frac{}{\Gamma \vdash_{\text{moggi}} () : \mathbf{unit} \mathbin{\&} 1}$$

$$\frac{}{\Gamma \vdash_{\text{moggi}} \mathbf{true} : \mathbf{bool} \mathbin{\&} 1} \qquad \frac{}{\Gamma \vdash_{\text{moggi}} \mathbf{false} : \mathbf{bool} \mathbin{\&} 1}$$

$$\frac{\Gamma \vdash_{\text{moggi}} e_1 : \mathbf{bool} \mathbin{\&} \varepsilon \qquad \Gamma \vdash_{\text{moggi}} e_2 : \tau \mathbin{\&} \varepsilon' \qquad \Gamma \vdash_{\text{moggi}} e_3 : \tau \mathbin{\&} \varepsilon'}{\Gamma \vdash_{\text{moggi}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \mathbin{\&} \varepsilon \cdot \varepsilon'}$$

$$\frac{\Gamma, x : \tau\&\varepsilon \vdash_{\text{moggi}} e : \tau' \mathbin{\&} \varepsilon'}{\Gamma \vdash_{\text{moggi}} \lambda x{:}\tau\&\varepsilon.\, e : \tau \xrightarrow{\varepsilon,\varepsilon'} \tau' \mathbin{\&} 1} \qquad \frac{\Gamma \vdash_{\text{moggi}} e_1 : \tau \xrightarrow{\varepsilon_2,\varepsilon_3} \tau' \mathbin{\&} \varepsilon_1 \qquad \Gamma \vdash_{\text{moggi}} e_2 : \tau \mathbin{\&} \varepsilon_2}{\Gamma \vdash_{\text{moggi}} e_1\, e_2 : \tau' \mathbin{\&} \varepsilon_1 \cdot \varepsilon_3}$$

$$\frac{\Gamma \vdash_{\text{moggi}} e : \tau \mathbin{\&} \varepsilon}{\Gamma \vdash_{\text{moggi}} e : \tau \mathbin{\&} \varepsilon'} \text{ if } \varepsilon \le \varepsilon'$$

Figure 2.4: Call-by-name effect system

2.4 Moggi-style call-by-name

The effect system given in the previous section is not suitable for call-by-name evaluation. This is because it assigns the effect 1 to each variable, but in call-by-name, variables may have side-effects. Subject reduction does not hold in general if we use call-by-name reduction instead of call-by-value.

In this section, we give an effect system for call-by-name. We refer to this as *Moggi-style* call-by-name here because the effect system is partly based on Moggi's monadic semantics for call-by-name [78]. It is distinguished from *Levy-style* call-by-name (Section 2.6) in that side-effects can occur at any type, rather than just at base types.[3]

Unlike the call-by-value effect system, a call-by-name effect system should not assign the effect 1 to every variable. To assign the correct effects to variables, we change the notion of typing context to also include effects. For Moggi-style call-by-name, a typing context is an ordered list of triples of the form $x : \tau\&\varepsilon$, where ε is the effect associated with the variable x. The syntax is changed so that function types are annotated with the effect ε of the argument as well as the latent effect ε' of the function, and we also annotate lambdas with the effect of the argument:

$$\begin{aligned} \tau, \tau' &::= \mathbf{unit} \mid \mathbf{bool} \mid \tau \xrightarrow{\varepsilon,\varepsilon'} B \\ e &::= c \mid x \mid () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \lambda x{:}\tau\&\varepsilon.\, e \mid e_1\, e_2 \end{aligned}$$

A practical effect system would probably add effect polymorphism to this, so that each function can be applied to arguments of various effects. The syntax of expressions e is the same as for call-by-value.

The call-by-name effect system is parameterized by the same notion of signature (Definition 2.2.1) as the other source-language type systems. The typing judgment $\Gamma \vdash_{\text{moggi}} e : \tau \mathbin{\&} \varepsilon$ is defined by the rules in Figure 2.4. Compared to the call-by-value effect system, only the rules

[3] Benton et al. [7] refer to Levy-style call-by-name as *Algol-like*.

$$\frac{e \overset{\text{moggi}}{\rightsquigarrow} e'}{\mathrm{op}\, e \overset{\text{moggi}}{\rightsquigarrow} \mathrm{op}\, e'} \qquad \begin{array}{l} \textbf{if true then } e_2 \textbf{ else } e_3 \overset{\text{moggi}}{\rightsquigarrow} e_2 \\ \textbf{if false then } e_2 \textbf{ else } e_3 \overset{\text{moggi}}{\rightsquigarrow} e_3 \end{array} \qquad (\lambda x\!:\!\tau \& \varepsilon.\, e)\, e' \overset{\text{moggi}}{\rightsquigarrow} e[x \mapsto e']$$

$$\frac{e_1 \overset{\text{moggi}}{\rightsquigarrow} e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \overset{\text{moggi}}{\rightsquigarrow} \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3} \qquad \frac{e_1 \overset{\text{moggi}}{\rightsquigarrow} e_1'}{e_1\, e_2 \overset{\text{moggi}}{\rightsquigarrow} e_1'\, e_2}$$

Figure 2.5: Call-by-name reduction relation

for variables, lambda abstraction and application change. The rule for application requires the effect ε_2 of the argument to be the same as the annotation on the function type. The conclusion of this rule does not mention the effect ε_2, since the expression e_2 is only evaluated if the function uses its argument. If it does, the latent effect ε_3 will reflect this. The typing rule for operations has not changed: operations still evaluate their arguments eagerly.

We also define a call-by-name operational semantics, consisting of a small-step reduction relation $\overset{\text{moggi}}{\rightsquigarrow}$. Again the definition is straightforward; it is given in Figure 2.5. Call-by-name reductions preserve effects.

Theorem 2.4.1 (Subject reduction for Moggi-style call-by-name) If $\diamond \vdash_{\text{moggi}} e : \tau \,\&\, \varepsilon$ and $e \overset{\text{moggi}}{\rightsquigarrow} e'$ then $\diamond \vdash_{\text{moggi}} e' : \tau \,\&\, \varepsilon$. ◀

2.5 Graded monadic metalanguage

So far, this chapter has given two source-language effect systems: one for call-by-value and one for Moggi-style call-by-name. We now consider an intermediate language that captures both of these evaluation orders, by making (side-effecting) computations into a first-class notion.

The intermediate language we describe is the *graded monadic metalanguage* (GMM), which is essentially the monadic metalanguage [78]: it consists of a pure fragment together with a type constructor that forms types of effectful computations. The primary difference is that the type constructor is *graded* by the effect algebra. GMM is similar to languages described by Katsumata [45] and by Gaboardi et al. [29].

The syntax of GMM types A, B and terms M, N is as follows:

$$\begin{array}{rcl@{\qquad}rcl} A, B & ::= & b & M, N & ::= & c \mid x \\ & & \mid \textbf{unit} & & & \mid () \\ & & \mid A_1 \times A_2 & & & \mid (M_1, M_2) \mid \textbf{fst}\, M \mid \textbf{snd}\, M \\ & & \mid \textbf{empty} & & & \mid \textbf{case}_A\; M \textbf{ of } \{\} \\ & & \mid A_1 + A_2 & & & \mid \textbf{inl}_{A_2} M \mid \textbf{inr}_{A_1} M \mid \textbf{case } M \textbf{ of } \{\textbf{inl}\, x_1.\, N_1,\ \textbf{inr}\, x_2.\, N_2\} \\ & & \mid A \to B & & & \mid \lambda x\!:\!A.\, M \mid M\, N \\ & & \mid \langle \varepsilon \rangle A & & & \mid \mathrm{op}\, M \mid \langle M \rangle \mid \textbf{let } \langle x \rangle = M \textbf{ in } N \mid \textbf{coerce}_{\varepsilon \le \varepsilon'} M \end{array}$$

We include a richer syntax of types than we do for source languages. The non-highlighted part of the syntax is just the simply-typed lambda calculus, with base types b, unit type, products, empty type, sum types, and function types. GMM does not have general recursion built in (but it can be added). The terms include constants c, and the eliminator $\mathbf{case}_A\ M\ \mathbf{of}\ \{\}$ of the empty type. The constants c are intended to introduce elements of base types; for example, we could have a base type **nat** of natural numbers and constants **zero** : **nat** and **suc** : **nat** → **nat** for zero and successor. They are not intended to provide side-effects (this is what the operations op are for). Modulo constants, there are no closed terms of type **empty**. (This does not mean the empty type is useless: there may still be closed terms of type $\langle\varepsilon\rangle\mathbf{empty}$.) We sometimes omit the type ascriptions (e.g. writing $\mathbf{inl}\,M$ instead of $\mathbf{inl}_{A_2} M$). We also write _ for an arbitrary fresh variable.

The highlighted part of the syntax concerns effectful computations. The type $\langle\varepsilon\rangle A$ contains computations that return results of type A, and have side-effects restricted to $\varepsilon \in \mathcal{E}$. For example, using a Gifford-style effect algebra (Example 2.1.2), computations $M : \langle\{\mathsf{raise}, \mathsf{get}\}\rangle A$ may use raise and get, but do not use the put operation. The type $\langle\varepsilon\rangle A$ is the same as the monadic type $\mathbf{T}A$ of Moggi's language, except for the restriction on side-effects. We refer to the family of type constructors $\langle-\rangle$ as the *graded monad*. The intention is that *all* side-effects are encapsulated in the graded monad, including e.g. divergence.

The term $\mathsf{op}\,M$ is the application of the operation op to the argument M. The term $\langle M\rangle$ is a computation that immediately returns M, causing no side-effects (this is also commonly written $\mathbf{return}\,M$). Computations are sequenced using $\mathbf{let}\ \langle x\rangle = M\ \mathbf{in}\ N$, which runs M, binds x to the result (if any) and then runs N. (This is the same as $M \ggg\!= \lambda x.N$ in Haskell.) The variable x is bound inside N but not M. We emphasize that the evaluation order of $\mathbf{let}\ \langle X\rangle = M\ \mathbf{in}\ N$ is fixed: M is always evaluated eagerly. It is possible to express various evaluation orders in GMM because computations are first-class. For example, it is possible to duplicate a computation by passing it to a function. Computations with effect ε can be regarded as computations with a less restrictive effect ε' using $\mathbf{coerce}_{\varepsilon\le\varepsilon'}$. This has the same purpose as the subeffecting rule in the above source-language effect systems. Subeffecting in GMM is explicit so that the effect system is syntax-directed.

As usual, we instantiate GMM with different side-effects by choosing different operations. For example, defining **bool** := **unit** + **unit**, we could have the following (recall that the coarity is the type of the argument of the operation, and the arity is the type of the output).

Operation op	Coarity $\mathsf{car}_{\mathsf{op}}$	Arity $\mathsf{ar}_{\mathsf{op}}$	Effect $\mathsf{eff}_{\mathsf{op}}$
raise	**unit**	**empty**	{raise}
get	**unit**	**bool**	{get}
put	**bool**	**unit**	{put}

Again the arity and coarity of each operation are required to be ground types. In GMM, ground types G are types that do not contain function types or the graded monad:

$$G \ ::= \ b \mid \mathbf{unit} \mid G_1 \times G_2 \mid \mathbf{empty} \mid G_1 + G_2$$

GMM is parameterized by the base types b, operations, effect algebra and constants c.

Definition 2.5.1 A *GMM signature* consists of the following data:

- A preordered monoid $(\mathcal{E}, \le, \cdot, 1)$ of *effects.*
- A set $\mathcal{B}$ of *base types.*
- A type-indexed family of pairwise disjoint sets $\mathcal{K}_A$ of *constants of type A.*
- A set Σ of *operations.*
- For each operation $\mathsf{op} \in \Sigma$, ground types $\mathsf{car}_{\mathsf{op}}$ and $\mathsf{ar}_{\mathsf{op}}$, respectively called the *coarity* and *arity* of op, and an effect $\mathsf{eff}_{\mathsf{op}} \in \mathcal{E}$. ◀

$$\frac{}{\Gamma \vdash_{\text{mon}} c : A}\ \text{if } c \in \mathcal{K}_A \qquad \frac{}{\Gamma \vdash_{\text{mon}} x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash_{\text{mon}} () : \textbf{unit}}$$

$$\frac{\Gamma \vdash_{\text{mon}} M_1 : A_1 \qquad \Gamma \vdash_{\text{mon}} M_2 : A_2}{\Gamma \vdash_{\text{mon}} (M_1, M_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash_{\text{mon}} M : A_1 \times A_2}{\Gamma \vdash_{\text{mon}} \textbf{fst}\, M : A_1} \qquad \frac{\Gamma \vdash_{\text{mon}} M : A_1 \times A_2}{\Gamma \vdash_{\text{mon}} \textbf{snd}\, M : A_2}$$

$$\frac{\Gamma \vdash_{\text{mon}} M : \textbf{empty}}{\Gamma \vdash_{\text{mon}} \textbf{case}_A\ M\ \textbf{of}\ \{\} : A}$$

$$\frac{\Gamma \vdash_{\text{mon}} M : A_1}{\Gamma \vdash_{\text{mon}} \textbf{inl}_{A_2} M : A_1 + A_2} \qquad \frac{\Gamma \vdash_{\text{mon}} M : A_2}{\Gamma \vdash_{\text{mon}} \textbf{inr}_{A_1} M : A_1 + A_2}$$

$$\frac{\Gamma \vdash_{\text{mon}} M : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_{\text{mon}} N_1 : B \qquad \Gamma, x_2 : A_2 \vdash_{\text{mon}} N_2 : B}{\Gamma \vdash_{\text{mon}} \textbf{case}\ M\ \textbf{of}\ \{\textbf{inl}\, x_1 .\, N_1,\ \textbf{inr}\, x_2 .\, N_2\} : B}$$

$$\frac{\Gamma, x : A \vdash_{\text{mon}} M : B}{\Gamma \vdash_{\text{mon}} \lambda x{:}A.\, M : A \to B} \qquad \frac{\Gamma \vdash_{\text{mon}} M : A \to B \qquad \Gamma \vdash_{\text{mon}} N : A}{\Gamma \vdash_{\text{mon}} M\, N : B}$$

$$\frac{\Gamma \vdash_{\text{mon}} M : \mathrm{car}_{\mathrm{op}}}{\Gamma \vdash_{\text{mon}} \mathrm{op}\, M : \langle \varepsilon \rangle \mathrm{ar}_{\mathrm{op}}} \qquad \frac{\Gamma \vdash_{\text{mon}} M : A}{\Gamma \vdash_{\text{mon}} \langle M \rangle : \langle 1 \rangle A}$$

$$\frac{\Gamma \vdash_{\text{mon}} M : \langle \varepsilon \rangle A \qquad \Gamma, x : A \vdash_{\text{mon}} N : \langle \varepsilon' \rangle B}{\Gamma \vdash_{\text{mon}} \textbf{let}\ \langle x \rangle = M\ \textbf{in}\ N : \langle \varepsilon \cdot \varepsilon' \rangle B} \qquad \frac{\Gamma \vdash_{\text{mon}} M : \langle \varepsilon \rangle A}{\Gamma \vdash_{\text{mon}} \textbf{coerce}_{\varepsilon \le \varepsilon'}\, M : \langle \varepsilon' \rangle A}\ \text{if } \varepsilon \le \varepsilon'$$

Figure 2.6: GMM typing rules

We assume a fixed GMM signature for the rest of this section.

Typing contexts Γ are ordered lists of pairs of variable names and types such that each variable appears at most once. The typing judgment has the form $\Gamma \vdash_{\text{mon}} M : A$. By contrast with the above effect systems, it is not annotated with an extra effect: in GMM, effects only appear inside types. The typing rules are given in Figure 2.6. Each of the rules that add variables to the typing context implicitly assume the variable is fresh. The rules are syntax-directed, so given a context Γ and term M, there is at most one type A such that $\Gamma \vdash_{\text{mon}} M : A$, and if $\Gamma \vdash_{\text{mon}} M : A$ holds then it has exactly one derivation.

The non-highlighted part consists of the standard typing rules of the simply-typed lambda calculus. The rules for the graded monad are highlighted. For operations, we assume a pure term M of type $\mathrm{car}_{\mathrm{op}}$. In the source language effect systems, the arguments of operations are computations, and the effect system reflects the fact that they are eagerly evaluated. Here, if we wish to use a computation M as an argument to an operation, it must be evaluated first. If M has type $\langle \varepsilon \rangle \mathrm{car}_{\mathrm{op}}$, we can do this by writing

$$\textbf{let}\ \langle x \rangle = M\ \textbf{in}\ \mathrm{op}\, x$$

which has type $\langle \varepsilon \cdot \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}$.

$$\mathbf{fst}\,(M_1, M_2) \equiv M_1 \qquad \mathbf{snd}\,(M_1, M_2) \equiv M_2$$

$$\begin{pmatrix}\mathbf{case}\ \mathbf{inl}_{A_2} M\ \mathbf{of} \\ \quad \{\mathbf{inl}\ x_1.\, N_1 \\ \quad ,\ \mathbf{inr}\ x_2.\, N_2\}\end{pmatrix} \equiv N_1[x_1 \mapsto M] \qquad \begin{pmatrix}\mathbf{case}\ \mathbf{inr}_{A_1} M\ \mathbf{of} \\ \quad \{\mathbf{inl}\ x_1.\, N_1 \\ \quad ,\ \mathbf{inr}\ x_2.\, N_2\}\end{pmatrix} \equiv N_2[x_2 \mapsto M]$$

$$(\lambda x{:}A.\, M)\, N \equiv M[x \mapsto N] \qquad \mathbf{let}\ \langle x \rangle = \langle M \rangle\ \mathbf{in}\ N \equiv N[x \mapsto M]$$

$$\begin{aligned} M &\equiv () \\ M &\equiv \mathbf{case}_A\ N\ \mathbf{of}\ \{\} \\ M &\equiv \lambda x{:}A.\, M\, x \\ M &\equiv \mathbf{let}\ \langle x \rangle = M\ \mathbf{in}\ \langle x \rangle \end{aligned} \qquad \begin{aligned} M &\equiv (\mathbf{fst}\, M, \mathbf{snd}\, M) \\ M[x \mapsto N] &\equiv \begin{pmatrix}\mathbf{case}\ N\ \mathbf{of} \\ \quad \{\mathbf{inl}\ y_1.\, M[x \mapsto \mathbf{inl}_{A_2} y_1] \\ \quad ,\ \mathbf{inr}\ y_2.\, M[x \mapsto \mathbf{inr}_{A_1} y_2]\}\end{pmatrix} \end{aligned}$$

$$\begin{aligned} \mathbf{let}\ \langle y \rangle = (\mathbf{let}\ \langle x \rangle = M_1\, \mathbf{in}\, M_2)\, \mathbf{in}\, M_3 &\equiv \mathbf{let}\ \langle x \rangle = M_1\, \mathbf{in}\, \mathbf{let}\ \langle y \rangle = M_2\, \mathbf{in}\, M_3 \\ \mathbf{coerce}_{\varepsilon_1 \cdot \varepsilon_2 \leq \varepsilon_1' \cdot \varepsilon_2'}\, (\mathbf{let}\ \langle x \rangle = M\ \mathbf{in}\ N) &\equiv \mathbf{let}\ \langle x \rangle = \mathbf{coerce}_{\varepsilon_1 \leq \varepsilon_1'}\, M\ \mathbf{in}\ \mathbf{coerce}_{\varepsilon_2 \leq \varepsilon_2'}\, N \\ \mathbf{coerce}_{\varepsilon \leq \varepsilon}\, M &\equiv M \\ \mathbf{coerce}_{\varepsilon' \leq \varepsilon''}\, (\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M) &\equiv \mathbf{coerce}_{\varepsilon \leq \varepsilon''}\, M \end{aligned}$$

Figure 2.7: Axioms of the GMM equational theory

Each of our other three highlighted rules corresponds to one part of the structure of the effect algebra: we use 1 for returning a value, the binary operation · for sequencing two computations, and the order ≤ for coercions from a more restrictive effect to a less restrictive effect.

2.5.1 Equational theory

GMM is designed to prove validity of program transformations. We therefore require a notion of *contextual equivalence*, which specifies when it is correct to replace one program with another. This is defined in terms of an *equational theory* (rather than an operational semantics). The equational theory for GMM consists of a judgment of the form $\Gamma \vdash_{\mathrm{mon}} M \equiv N : A$, which means M and N both have type A in context Γ, and they have the same semantics. We sometimes write this just as $M \equiv N$.

The axioms of the equational theory are listed in Figure 2.7. Since we relate well-typed terms, each of the axioms should be read as assuming that both sides have the same type in the same typing context (but not that they are closed). For example, the axiom $M \equiv ()$ holds when $\Gamma \vdash_{\mathrm{mon}} M : \mathbf{unit}$. The necessary constraints can be derived by looking at the typing rules (because they are syntax-directed). As usual, the non-highlighted part of the figure is standard from the simply-typed lambda calculus (it consists of the usual β- and η- laws), and the highlighted part describes the behaviour of computations. Each of the axioms of the highlighted part corresponds to one of the requirements in the definition of preordered monoid.

The first group of axioms consists of β-laws. The highlighted β-law states that sequencing can be eliminated if the first computation immediately returns a value. The constraints necessary to ensure that both sides of the axiom have the same type are $\Gamma \vdash_{\mathrm{mon}} M : A$ and

$\Gamma, x : A \vdash_{\text{mon}} N : \langle\varepsilon\rangle B$. When these hold both sides have the same type because of the left unit law of the preordered monoid: $1 \cdot \varepsilon = \varepsilon$.

The second group consists of one η-law for each of the type formers of GMM (excluding base types). The highlighted axiom states that sequencing of computations can be added without changing behaviour by returning the value of the variable x. It corresponds to the right unit law $\varepsilon \cdot 1 = \varepsilon$.

The third group contains the remaining axioms for computations. The first axiom states that sequencing of computations is associative, and corresponds to associativity of $\cdot$. The second states that coercion commutes with sequencing of computations. The two terms have the same type because $\cdot$ is monotone. (A similar equation can be stated for commutativity of **coerce** with the eliminator for sum types. This equation follows from the other axioms in the figure, so we do not need it as an additional axiom.) The final two axioms of the figure allow us to remove coercions that do not change the effect, and to combine adjacent coercions. They correspond to reflexivity and transitivity of the preorder $\leq$.

In addition to the core axioms, we also close the equational theory under congruence. To define this, we use a notion of *term context*. A GMM term context $C[]$ consists of a term with a single *hole* (which we write as $\square$):

$$\begin{aligned}
C[] \;::=\; & \square \mid (C[], M_2) \mid (M_1, C[]) \mid \mathbf{fst}\, C[] \mid \mathbf{snd}\, C[] \mid \mathbf{case}_A\, C[]\;\mathbf{of}\;\{\} \\
& \mid \mathbf{inl}_{A_2} C[] \mid \mathbf{inr}_{A_1} C[] \mid \mathbf{case}\; C[]\;\mathbf{of}\;\{\mathbf{inl}\, x_1.\, N_1,\ \mathbf{inr}\, x_2.\, N_2\} \\
& \mid \mathbf{case}\; M\;\mathbf{of}\;\{\mathbf{inl}\, x_1.\, C[],\ \mathbf{inr}\, x_2.\, N_2\} \mid \mathbf{case}\; M\;\mathbf{of}\;\{\mathbf{inl}\, x_1.\, N_1,\ \mathbf{inr}\, x_2.\, C[]\} \\
& \mid \lambda x{:}A.\, C[] \mid C[]\, N \mid M\, C[] \\
& \mid \mathrm{op}\, C[] \mid \langle C[] \rangle \mid \mathbf{let}\;\langle x\rangle = C[]\;\mathbf{in}\; N \mid \mathbf{let}\;\langle x\rangle = M\;\mathbf{in}\; C[] \mid \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, C[]
\end{aligned}$$

We write $C[M]$ for the term formed by replacing $\square$ with M. The term context may capture some of the free variables of M: for example, if $C[]$ is $\mathbf{let}\;\langle x\rangle = M\;\mathbf{in}\;\langle\square\rangle$ then $C[(x, y)]$ is $\mathbf{let}\;\langle x\rangle = M\;\mathbf{in}\;\langle(x, y)\rangle$; in the latter term, only y is free.

This notion of term context is used in the definition of the GMM equational theory.

Definition 2.5.2 We define $\Gamma \vdash_{\text{mon}} M \equiv N : A$ inductively by:

- Preorder: if $\Gamma \vdash_{\text{mon}} M : A$ then $\Gamma \vdash_{\text{mon}} M \equiv M : A$, and if $\Gamma \vdash_{\text{mon}} M_1 \equiv M_2 : A$ and $\Gamma \vdash_{\text{mon}} M_2 \equiv M_3 : A$ then $\Gamma \vdash_{\text{mon}} M_1 \equiv M_3 : A$.
- Congruence: if $\Gamma \vdash_{\text{mon}} M \equiv N : A$ and both $\Gamma' \vdash_{\text{mon}} C[M] : B$ and $\Gamma' \vdash_{\text{mon}} C[N] : B$ then $\Gamma' \vdash_{\text{mon}} C[M] \equiv C[N] : B$.
- Axioms: If $\Gamma \vdash_{\text{mon}} M : A$ and $\Gamma \vdash_{\text{mon}} N : A$, and $M \equiv N$ is an instance of an axiom in Figure 2.7, then $\Gamma \vdash_{\text{mon}} M \equiv N : A$ and $\Gamma \vdash_{\text{mon}} N \equiv M : A$. ◀

For a fixed typing context Γ and type A, the equational theory is an equivalence relation on terms of type A in context Γ. It is also closed under substitution: if $x_1 : A_1, \ldots, x_n : A_n \vdash_{\text{mon}} N \equiv N' : B$, and $\Gamma \vdash_{\text{mon}} M_i \equiv M_i' : A_i$ for each i, then

$$\Gamma \vdash_{\text{mon}} N[x_1 \mapsto M_1, \ldots, x_n \mapsto M_n] \;\equiv\; N'[x_1 \mapsto M_1', \ldots, x_n \mapsto M_n'] : B$$

Two terms that are syntactically equal up to the placement of coercions are also usually related by the equational theory. Given a term M, define $\lfloor M \rfloor$ to be the same as M but with all uses of **coerce** deleted (for example, $\lfloor \lambda x{:}A.\, \mathbf{coerce}_{1 \leq \varepsilon}\, \langle x\rangle \rfloor = \lambda x{:}A.\, \langle x\rangle$).

Lemma 2.5.3 Suppose that the effect algebra is a partially ordered monoid with bounded binary joins. Given two terms M_1, M_2 such that $\Gamma \vdash_{\text{mon}} M_1 : A$ and $\Gamma \vdash_{\text{mon}} M_2 : A$, if $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$ then $\Gamma \vdash_{\text{mon}} M_1 \equiv M_2 : A$. ◀

$$\begin{aligned} \llparenthesis \mathbf{unit} \rrparenthesis &:= \mathbf{unit} \\ \llparenthesis \mathbf{bool} \rrparenthesis &:= \mathbf{unit} + \mathbf{unit} \\ \llparenthesis \tau \xrightarrow{\varepsilon} \tau' \rrparenthesis &:= \llparenthesis \tau \rrparenthesis \to \langle \varepsilon \rangle \llparenthesis \tau' \rrparenthesis \end{aligned} \qquad \begin{aligned} \llparenthesis \diamond \rrparenthesis &:= \diamond \\ \llparenthesis \Gamma, x : \tau \rrparenthesis &:= \llparenthesis \Gamma \rrparenthesis,\ x : \llparenthesis \tau \rrparenthesis \end{aligned}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} x : \tau \mathbin{\&} 1 \rrparenthesis := \langle x \rangle} \qquad \frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \mathrm{car}_{\mathrm{op}} \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \mathbin{\&} \varepsilon \cdot \mathrm{eff}_{\mathrm{op}} \rrparenthesis = \mathbf{let}\ \langle x \rangle = M\ \mathbf{in}\ \mathrm{op}\, x}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} () : \mathbf{unit} \mathbin{\&} 1 \rrparenthesis := \langle () \rangle}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{true} : \mathbf{bool} \mathbin{\&} 1 \rrparenthesis := \langle \mathbf{inl}_{\mathbf{unit}}\, () \rangle} \qquad \frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{false} : \mathbf{bool} \mathbin{\&} 1 \rrparenthesis := \langle \mathbf{inr}_{\mathbf{unit}}\, () \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1 : \mathbf{bool} \mathbin{\&} \varepsilon \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon' \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_3 : \tau \mathbin{\&} \varepsilon' \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \mathbin{\&} \varepsilon \cdot \varepsilon' \rrparenthesis \ :=\ \mathbf{let}\ \langle x \rangle = M_1\ \mathbf{in}\ \mathbf{case}\ x\ \mathbf{of}\ \{\mathbf{inl}\ \_.\, M_2,\ \mathbf{inr}\ \_.\, M_3\}}$$

$$\frac{\llparenthesis \Gamma, x : \tau \vdash_{\mathrm{v}} e : \tau' \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \lambda x{:}\tau.\, e : \tau \xrightarrow{\varepsilon} \tau' \mathbin{\&} 1 \rrparenthesis \ :=\ \langle \lambda x{:}\llparenthesis \tau \rrparenthesis.\, M \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1 : \tau \xrightarrow{\varepsilon_3} \tau' \mathbin{\&} \varepsilon_1 \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon_2 \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1\, e_2 : \tau' \mathbin{\&} \varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3 \rrparenthesis \ :=\ \mathbf{let}\ \langle f \rangle = M_1\ \mathbf{in}\ \mathbf{let}\ \langle x \rangle = M_2\ \mathbf{in}\ f\, x}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon' \rrparenthesis \ :=\ \mathbf{coerce}_{\varepsilon \le \varepsilon'}\, M}$$

Figure 2.8: Translation of call-by-value types (top left), contexts (top right), and typing derivations (bottom) into GMM.

A proof of a similar fact for our call-by-push-value effect system (Lemma 2.7.5) is given in Appendix B.1, where we also conjecture a counterexample for an effect algebra that does not have bounded binary joins. The proof for GMM is similar to the proof for call-by-push-value.

2.5.2 Translations into GMM

GMM captures both call-by-value and Moggi-style call-by-name via two translations (one for each evaluation order) from source-language expressions e into GMM terms. They are based on similar translations given by Moggi [77, 78], Wadler [98] and Benton et al. [7]. Translations like these allow us to use intermediate languages to reason about source languages. Having multiple evaluation-order-directed translations enables reasoning about changes in evaluation order in the source language using the intermediate language.

In this section, we assume a source-language signature and GMM signature that are compatible. Recall that the source-language ground types are **unit** and **bool**. We translate these into GMM ground types:

$$\llparenthesis \mathbf{unit} \rrparenthesis := \mathbf{unit} \qquad \llparenthesis \mathbf{bool} \rrparenthesis := \mathbf{unit} + \mathbf{unit}$$

(The two translations we give below agree on ground types with this definition.) We require that both signatures have the same effect algebra, and that for each operation op in the source-language signature: op is also included in the GMM signature; both signatures agree on the effect of op; and if τ and τ' are respectively the coarity and arity assigned to op by the source-language signature, then $(\!|\tau|\!)$ and $(\!|\tau'|\!)$ are the coarity and arity assigned to op by the GMM signature. We allow the GMM signature to contain additional operations, and place no restrictions on base types or constants.

The first translation maps call-by-value source-language types τ to GMM types $(\!|\tau|\!)^{\mathrm{v}}_{\mathrm{mon}}$ and call-by-value contexts Γ to GMM contexts $(\!|\Gamma|\!)^{\mathrm{v}}_{\mathrm{mon}}$. It maps *derivations* of $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ to GMM *terms* $(\!|\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon|\!)^{\mathrm{v}}_{\mathrm{mon}}$. Since the call-by-value effect system is not syntax-directed, derivations are not unique (there is a choice of where to use the subeffecting rule), and different derivations are translated into syntactically different GMM terms. The term $(\!|\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon|\!)^{\mathrm{v}}_{\mathrm{mon}}$ is a computation with effect ε: in context $(\!|\Gamma|\!)^{\mathrm{v}}_{\mathrm{mon}}$, it has type $\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\mathrm{mon}}$. The definition of the call-by-value translation is given in Figure 2.8, where we omit the sub- and superscripts. The important part is the translation of functions. Source-language function types are translated into GMM function types where the result is a computation (so may have side-effects), but the argument is not. Function applications first evaluate the function itself, then the argument, and then the function body (this matches the operational semantics in Figure 2.3).

Call-by-name types τ and contexts Γ are translated into GMM types $(\!|\tau|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$ and contexts $(\!|\Gamma|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$ respectively. The translation of derivations of $\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$ has similar typing to the call-by-value translation: $(\!|\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$ is a GMM term of type $\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$ in context $(\!|\Gamma|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$. The definition of the call-by-name translation is in Figure 2.9. Many of the cases are the same as for call-by-value. The key differences are for typing contexts and functions. The translation of typing contexts uses the graded monad because variables have side-effects. Similarly, arguments of functions have effects, and therefore are encapsulated in the graded monad. The translation of a function application does not evaluate the argument immediately; instead, the computation is passed to the function, and is evaluated each time the function uses its argument (matching Figure 2.5).

The call-by-value and Moggi-style call-by-name translations have the desired typing:

Lemma 2.5.4

1. If M is the call-by-value translation of a derivation of $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$, then
$$(\!|\Gamma|\!)^{\mathrm{v}}_{\mathrm{mon}} \vdash_{\mathrm{mon}} M : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\mathrm{mon}}$$
2. If M is the call-by-name translation of a derivation of $\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$, then
$$(\!|\Gamma|\!)^{\mathrm{moggi}}_{\mathrm{mon}} \vdash_{\mathrm{mon}} M : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{moggi}}_{\mathrm{mon}}$$ ◀

Although it is important that we translate derivations rather than just well-typed expressions, for suitable effect algebras it does not matter which derivation is chosen up to the equational theory:

Lemma 2.5.5 Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. If M and M' are call-by-value translations of derivations of $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$, then $M \equiv M'$.
2. If M and M' are call-by-name translations of derivations of $\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$, then $M \equiv M'$.

◀

This follows immediately from Lemma 2.5.3. Hence when considering GMM terms up to $\equiv$, we can refer to *the* translation of a well-typed expression. We do this in the following theorem.

$$\begin{aligned}
\llparenthesis \mathbf{unit} \rrparenthesis &:= \mathbf{unit} \\
\llparenthesis \mathbf{bool} \rrparenthesis &:= \mathbf{unit} + \mathbf{unit} \\
\llparenthesis \tau \xrightarrow{\varepsilon,\varepsilon'} \tau' \rrparenthesis &:= \langle \varepsilon \rangle \llparenthesis \tau \rrparenthesis \to \langle \varepsilon' \rangle \llparenthesis \tau' \rrparenthesis
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis \diamond \rrparenthesis &:= \diamond \\
\llparenthesis \Gamma, x : \tau \& \varepsilon \rrparenthesis &:= \llparenthesis \Gamma \rrparenthesis,\ x : \langle \varepsilon \rangle \llparenthesis \tau \rrparenthesis
\end{aligned}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} x : \tau \;\&\; \varepsilon \rrparenthesis := x}
\qquad
\frac{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e : \mathrm{car}_{\mathrm{op}} \;\&\; \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \;\&\; \varepsilon \cdot \mathrm{eff}_{\mathrm{op}} \rrparenthesis := \mathbf{let}\ \langle x \rangle = M\ \mathbf{in}\ \mathrm{op}\, x}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} () : \mathbf{unit} \;\&\; 1 \rrparenthesis := \langle () \rangle}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} \mathbf{true} : \mathbf{bool} \;\&\; 1 \rrparenthesis := \langle \mathbf{inl}_{\mathbf{unit}}\, () \rangle}
\qquad
\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} \mathbf{false} : \mathbf{bool} \;\&\; 1 \rrparenthesis := \langle \mathbf{inr}_{\mathbf{unit}}\, () \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_1 : \mathbf{bool} \;\&\; \varepsilon \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_2 : \tau \;\&\; \varepsilon' \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_3 : \tau \;\&\; \varepsilon' \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \;\&\; \varepsilon \cdot \varepsilon' \rrparenthesis \;:=\; \mathbf{let}\ \langle x \rangle = M_1\ \mathbf{in}\ \mathbf{case}\ x\ \mathbf{of}\ \{\mathbf{inl}\,\_.\, M_2,\ \mathbf{inr}\,\_.\, M_3\}}$$

$$\frac{\llparenthesis \Gamma, x : \tau \& \varepsilon \vdash_{\mathrm{moggi}} e : \tau' \;\&\; \varepsilon' \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} \lambda x {:} \tau \& \varepsilon.\, e : \tau \xrightarrow{\varepsilon,\varepsilon'} \tau' \;\&\; 1 \rrparenthesis \;:=\; \langle \lambda x {:} \langle \varepsilon \rangle \llparenthesis \tau \rrparenthesis .\, M \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_1 : \tau \xrightarrow{\varepsilon_2,\varepsilon_3} \tau' \;\&\; \varepsilon_1 \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_2 : \tau \;\&\; \varepsilon_2 \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e_1\, e_2 : \tau' \;\&\; \varepsilon_1 \cdot \varepsilon_3 \rrparenthesis \;:=\; \mathbf{let}\ \langle f \rangle = M_1\ \mathbf{in}\ f\, M_2}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e : \tau \;\&\; \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{moggi}} e : \tau \;\&\; \varepsilon' \rrparenthesis \;:=\; \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M}$$

Figure 2.9: Translation of Moggi-style call-by-name types (top left), contexts (top right), and typing derivations (bottom) into GMM.

Theorem 2.5.6 (Soundness) Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. If $\diamond \vdash_{\mathrm{v}} e : \tau \;\&\; \varepsilon$ and $e \overset{\mathrm{v}}{\rightsquigarrow} e'$, then $\llparenthesis e \rrparenthesis^{\mathrm{v}}_{\mathrm{mon}} \equiv \llparenthesis e' \rrparenthesis^{\mathrm{v}}_{\mathrm{mon}}$.
2. If $\diamond \vdash_{\mathrm{moggi}} e : \tau \;\&\; \varepsilon$ and $e \overset{\mathrm{moggi}}{\rightsquigarrow} e'$, then $\llparenthesis e \rrparenthesis^{\mathrm{moggi}}_{\mathrm{mon}} \equiv \llparenthesis e' \rrparenthesis^{\mathrm{moggi}}_{\mathrm{mon}}$. ◂

Both of the translations described in this section are *compositional*: the translation of a typing derivation depends only on the translations of its subderivations (rather than the actual structure of the derivations). This property is crucial for reasoning, because we look at parts of programs rather than complete programs. Compositionality implies it is valid to do so.

2.6 Levy-style call-by-name

As we mentioned above, Moggi-style call-by-name is not the only form of call-by-name. We also discuss *Levy-style* call-by-name.

First we explain why there are two forms of call-by-name. Suppose we are in a language (such as PCF) that has divergence as the only side-effect, and that Ω_τ is a closed diverging term of type τ. Each program M of type **bool** will either diverge or return a boolean. In a domain-theoretic denotational semantics, the denotation of M is an element of $2_\bot$, where $2 = \{\mathrm{true}, \mathrm{false}\}$ is discrete, and the operation $(-)_\bot$ freely adds a least element. What should the denotation of a closed term of type $\mathbf{bool} \to \mathbf{bool}$ be? It could be an element of either of the following (where $\Rightarrow$ means function space):

$$(\mathbf{bool}_\bot \Rightarrow \mathbf{bool}_\bot)_\bot \qquad\qquad \mathbf{bool}_\bot \Rightarrow \mathbf{bool}_\bot$$

The left distinguishes between $\Omega_{\mathbf{bool}\to\mathbf{bool}}$ (which has interpretation $\bot$) and $\lambda x.\, \Omega_{\mathbf{bool}}$. On the right, there is no separate least element $\bot$ for diverging terms of function type, so $[\![\Omega_{\mathbf{bool}\to\mathbf{bool}}]\!] = [\![\lambda x.\, \Omega_{\mathbf{bool}}]\!]$.

In PCF there is no *observable* difference between $\Omega_{\mathbf{bool}\to\mathbf{bool}}$ and $\lambda x.\, \Omega_{\mathbf{bool}}$, because programs can only use functions by applying them. Hence a denotational semantics does not need to distinguish between these terms, and both interpretations of $\mathbf{bool} \to \mathbf{bool}$ are correct. This breaks down if we add other ways to use functions: in Haskell, M `` `seq` `` N roughly means evaluate M, discard the result, and then evaluate N; by passing these two terms as M we can distinguish between them. In languages with `seq`, only the interpretation on the left is valid. However, in languages without `seq`, the interpretation on the right more closely reflects the observable behaviours of terms. In particular, η-expansion preserves the interpretation of functions. The interpretation on the right also has other useful properties: for example, the evident function $\mathrm{swap} : (\tau_1 \to \tau_2 \to \tau') \to (\tau_2 \to \tau_1 \to \tau')$ is an isomorphism $(\tau_1 \to \tau_2 \to \tau') \cong (\tau_2 \to \tau_1 \to \tau')$ (by which we mean $[\![\mathrm{swap}\,(\mathrm{swap}\, e)]\!] = [\![e]\!]$). Both of the two interpretations of call-by-name have advantages.

The Moggi-style call-by-name translation we define in Figure 2.9 takes the option on the left (think of $\langle\varepsilon\rangle$ as being $(-)_\bot$). It assumes that it might be possible to observe side-effects at any type. *Levy-style* call-by-name corresponds to the option on the right. It is the form of call-by-name primarily considered by Levy [56].[4] For Levy-style, we assume that side-effects can only be observed at base types. This is the case for the source language we consider (and so Levy-style is suitable) because we have not included any `seq`-like constructs.

To describe a source-language effect system for Levy-style call-by-name, we attach the effects ε to base types (rather than having an extra parameter in the typing judgment). The syntax of types is therefore

$$\tau, \tau' \;::=\; \langle\varepsilon\rangle\mathbf{unit} \mid \langle\varepsilon\rangle\mathbf{bool} \mid \tau \to \tau'$$

The similarity with the graded monad $\langle-\rangle$ of GMM (Section 2.5) is not accidental. For example, $\langle\varepsilon\rangle\mathbf{bool}$ should be thought of as the type of computations of effect ε that return booleans. Typing contexts Γ are lists of pairs of the form $x : \tau$. The syntax of expressions is similar to the other source languages:

$$e \;::=\; x \mid \mathrm{op}\, e \mid () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2$$

The effect system is again parameterized by a source-language signature (Definition 2.2.1). The coarity and arity of each operation is either **unit** or **bool**; these are not annotated with effects, and hence are not Levy-style call-by-name types (but $\langle\varepsilon\rangle\tau$ is for $\tau \in \{\mathbf{unit}, \mathbf{bool}\}$). The

[4] And this is why we call it *Levy-style*, though it has been considered previously. For example, it is used for PCF and idealized Algol [90], and is discussed by Benton et al. [7].

$$\frac{}{\Gamma \vdash_{\mathrm{n}} x : \tau}\ \text{if } (x : \tau) \in \Gamma \qquad \frac{\Gamma \vdash_{\mathrm{n}} e : \langle \varepsilon \rangle \mathrm{car}_{\mathrm{op}}}{\Gamma \vdash_{\mathrm{n}} \mathrm{op}\, e : \langle \varepsilon \cdot \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}} \qquad \frac{}{\Gamma \vdash_{\mathrm{n}} () : \langle 1 \rangle \mathbf{unit}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{n}} \mathbf{true} : \langle 1 \rangle \mathbf{bool}} \qquad \frac{}{\Gamma \vdash_{\mathrm{n}} \mathbf{false} : \langle 1 \rangle \mathbf{bool}} \qquad \frac{\Gamma \vdash_{\mathrm{n}} e_1 : \langle \varepsilon \rangle \mathbf{bool} \qquad \Gamma \vdash_{\mathrm{n}} e_2 : \tau \qquad \Gamma \vdash_{\mathrm{n}} e_3 : \tau}{\Gamma \vdash_{\mathrm{n}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \langle\!\langle \varepsilon \rangle\!\rangle \tau}$$

$$\frac{\Gamma, x : \tau \vdash_{\mathrm{n}} e : \tau'}{\Gamma \vdash_{\mathrm{n}} \lambda x{:}A.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash_{\mathrm{n}} e_1 : \tau \to \tau' \qquad \Gamma \vdash_{\mathrm{n}} e_2 : \tau}{\Gamma \vdash_{\mathrm{n}} e_1\, e_2 : \tau'} \qquad \frac{\Gamma \vdash_{\mathrm{n}} e : \langle\!\langle \varepsilon \rangle\!\rangle \tau}{\Gamma \vdash_{\mathrm{n}} e : \langle\!\langle \varepsilon' \rangle\!\rangle \tau}\ \text{if } \varepsilon \leq \varepsilon'$$

Figure 2.10: Effect system for Levy-style call-by-name

typing judgment has the form $\Gamma \vdash_{\mathrm{n}} e : \tau$. To define it, we use a function $\langle\!\langle \varepsilon \rangle\!\rangle$ that adds the effect ε to types. The type $\langle\!\langle \varepsilon \rangle\!\rangle \tau$ is given by:

$$\langle\!\langle \varepsilon \rangle\!\rangle (\langle \varepsilon' \rangle \mathbf{unit}) \coloneqq \langle \varepsilon \cdot \varepsilon' \rangle \mathbf{unit} \qquad \langle\!\langle \varepsilon \rangle\!\rangle (\langle \varepsilon' \rangle \mathbf{bool}) \coloneqq \langle \varepsilon \cdot \varepsilon' \rangle \mathbf{bool} \qquad \langle\!\langle \varepsilon \rangle\!\rangle (\tau \to \tau') \coloneqq \tau \to \langle\!\langle \varepsilon \rangle\!\rangle \tau'$$

The typing judgment is inductively defined by the rules of Figure 2.10. Most of the rules are standard; in particular, the rules for λ-abstraction and application are the same as for the simply-typed λ-calculus. Hence η-expansion preserves types: if $\Gamma \vdash_{\mathrm{n}} e : \tau \to \tau'$ then $\Gamma \vdash_{\mathrm{n}} \lambda x : \tau.\, e\, x : \tau \to \tau'$. (This is the key property that we would expect if we can only observe effects at base types. It is not true for the Moggi-style effect system.) In the rule for **if** e_1 **then** e_2 **else** e_3, the operation $\langle\!\langle \varepsilon \rangle\!\rangle$ is used because the expression e_1 (with effect ε) is evaluated before evaluating either e_2 or e_3. The subeffecting rule is similar to those in our other effect systems.

The Levy-style call-by-name operational semantics consists of a small-step reduction relation $\overset{\mathrm{n}}{\rightsquigarrow}$. The rules defining this are exactly the same as for Moggi-style call-by-name (Figure 2.5), except for the slightly different λ-abstraction syntax. The Levy-style call-by-name effect system also satisfies the following subject reduction theorem. (Recall that $\diamond$ is the empty typing context.)

Theorem 2.6.1 (Subject reduction for Levy-style call-by-name) If $\diamond \vdash_{\mathrm{n}} e : \tau$ and $e \overset{\mathrm{n}}{\rightsquigarrow} e'$ then $\diamond \vdash_{\mathrm{n}} e' : \tau$. ◀

A natural question to ask is whether we can add Levy-style call-by-name to the translations into the graded monadic metalanguage described in Section 2.5.2. Filinski [24] defines a *generalized let* construct that can be used to do this. However, as explained by Levy [56], the result is difficult to reason about because the translation of **if** depends on the type τ. This motivates *call-by-push-value*, which is designed to admit a more suitable translation of Levy-style call-by-name.

2.7 Graded call-by-push-value

The language we use for the remainder of this thesis is Levy's *call-by-push-value* (CBPV) [56]. CBPV has several advantages over other intermediate languages such as the monadic metalanguage. The main advantage for our purposes is that is allows us to express more evaluation orders. There is a compositional translation from Moggi's monadic metalanguage (and therefore from call-by-value and Moggi-style call-by-name), and a compositional translation from Levy-style call-by-name. As for the other languages in this chapter, we wish to use an effect system to enable effect-dependent reasoning. We therefore present *graded call-by-push-value* (GCBPV), which is similar to CBPV except that the type system tracks effect information. Before doing this, we take a closer look at how CBPV allows programs to control their evaluation order.

CBPV stratifies terms into *values*, which do not have side-effects, and *computations*, which might.[5] Evaluation order is irrelevant for values (because of the lack of side-effects), so we only need to be careful about how computations are sequenced. There is exactly one CBPV primitive that causes the evaluation of more than one computation. The computation $M\ \mathbf{to}\ x.\,N$ runs the computation M, binds x to the result, and then runs the computation N. (It is similar to $\mathbf{let}\ \langle x\rangle = M\ \mathbf{in}\ N$ in GMM.) The evaluation order is fixed: M is always evaluated eagerly. To allow more control over evaluation order, CBPV allows computations to be thunked. The term $\mathbf{thunk}\,M$ is a value that contains the thunk of the computation M. Thunks can be duplicated (to allow a single computation to be evaluated more than once), and discarded (so that the computation is not evaluated). If V is a thunk, it can be converted back into a computation with $\mathbf{force}\,V$.

2.7.1 Syntax

The syntax of graded call-by-push-value is as follows. It is similar to the syntax of ordinary call-by-push-value, except that it tracks effect information.[6]

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
A, B & ::= & b & V, W & ::= & c \mid x \\
 & & \mid \mathbf{unit} & & & \mid () \\
 & & \mid A_1 \times A_2 & & & \mid (V_1, V_2) \mid \mathbf{fst}\,V \mid \mathbf{snd}\,V \\
 & & \mid \mathbf{empty} & & & \mid \mathbf{case}_{A}\ V\ \mathbf{of}\ \{\} \\
 & & \mid A_1 + A_2 & & & \mid \mathbf{inl}_{A_2} V \mid \mathbf{inr}_{A_1} V \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\,x_1.\,W_1,\ \mathbf{inr}\,x_2.\,W_2\} \\
 & & \mid \mathbf{U}\,\underline{C} & & & \mid \mathbf{thunk}\,M \\[1ex]
\underline{C}, \underline{D} & ::= & \underline{\mathbf{unit}} & M, N & ::= & \lambda\{\} \\
 & & \mid \underline{C}_1 \,\underline{\times}\, \underline{C}_2 & & & \mid \lambda\{1.\,M_1, 2.\,M_2\} \mid 1\text{‘}M \mid 2\text{‘}M \\
 & & \mid A \to \underline{C} & & & \mid \lambda x{:}A.\,M \mid V\text{‘}M \\
 & & \mid \langle\varepsilon\rangle A & & & \mid \mathrm{op}\,V \mid \langle V\rangle \mid M\ \mathbf{to}\ x.\,N \mid \mathbf{coerce}_{\varepsilon\leq\varepsilon'}\,M \\
 & & & & & \mid \mathbf{force}\,V
\end{array}
$$

Types are stratified into *value types* A, B and *computation types* $\underline{C}, \underline{D}$; terms are stratified into *value terms* V, W and *computation terms* M, N. Value types include base types b, the unit type, product types, the empty type, and sum types. As for GMM, we include a collection of constants

[5] There are several calculi that similarly capture various evaluation orders by stratifying their syntax into two parts, such as *polarized* calculi [102]. These have various viewpoints on the distinction between the two parts, and these affect the design of the language. We return to this issue when we add call-by-need in Chapter 5.

[6] Other than tracking effect information, the only difference with ordinary CBPV is that eliminators of product and sum types are value terms rather than computation terms (which makes value terms slightly more general). Levy [56] calls this CBPV with *complex values.* This influences how we think about the split between values and computations: with complex values, value terms do not have side-effects when they are executed; without complex values, values do not reduce at all.

c. All GCBPV constants are value terms; we do not need constants on the computation level (this is discussed further below). Values also include *thunks.* The value type $\mathbf{U}\,\underline{C}$ contains thunks of computations of type $\underline{C}$. The value term $\mathbf{thunk}\,V$ suspends a computation M; the computation term $\mathbf{force}\,V$ runs a suspended computation V. Computation types include a unit type $\underline{\mathbf{unit}}$ (introduced by the computation term $\lambda\{\}$), and *lazy* binary products $\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2$. Pairs of computation terms are written $\lambda\{1.\,M_1, 2.\,M_2\}$, and the first and second projections are $1`M$ and $2`M$. Laziness means $i`M$ evaluates only the $i$th component of the pair; the other component (and its side-effects) are ignored. Binary products of value types give us strict pairing. Functions send values to computations, and are computations themselves. Application is written $V`M$, where V is the argument and M is the function to apply.

The highlighted computation type $\langle\varepsilon\rangle A$ is a *returner type.* It contains computations which have side-effects restricted to the effect ε, before potentially returning a value (which must be an element of A). The side-effects could include e.g. divergence (and hence such a computation might never return a value). The corresponding type in ordinary CBPV is usually written $\mathbf{F}\,A$; we add grading by the effect ε. Application of an operation op to the argument V is written $\mathrm{op}\,V$. As for GMM, the argument is not a computation; to apply an operation to the result of a computation M we can use $(M\ \mathbf{to}\ x.\,\mathrm{op}\,x)$. Returners (i.e. elements of returner types) are introduced by $\langle V\rangle$, which is a computation that immediately returns V (with no side-effects). The eliminator is $M\ \mathbf{to}\ x.\,N$ (which requires M to have returner type). As mentioned above, this runs M, then binds x to the result and runs N. Notationally, **to** is right-associative (so $M_1\ \mathbf{to}\ x.\,M_2\ \mathbf{to}\ y.\,M_3$ means $M_1\ \mathbf{to}\ x.\,(M_2\ \mathbf{to}\ y.\,M_3)$), and variable bindings extend as far to the right as possible (so $\lambda x : A.\,M_1\ \mathbf{to}\ y.\,M_2$ means $\lambda x : A.\,(M_1\ \mathbf{to}\ y.\,M_2)$). Finally, we can coerce computations of returner type from effect ε to a less restrictive effect ε' with $\mathbf{coerce}_{\varepsilon\leq\varepsilon'}$.

The eliminators of the empty and sum types are value terms. We also have syntactic sugar for eliminators of value terms on the computation level:

$$\begin{aligned}\underline{\mathbf{case}}_{\underline{C}}\ V\ \mathbf{of}\ \{\} &:= \mathbf{force}\,(\mathbf{case}_{\mathbf{U}\,\underline{C}}\ V\ \mathbf{of}\ \{\})\\ \underline{\mathbf{case}}\ V\ \mathbf{of}\ \{\mathbf{inl}\,x_1.\,M_1,\ \mathbf{inr}\,x_2.\,M_2\} &:= \mathbf{force}\,(\mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\,x_1.\,\mathbf{thunk}\,M_1,\ \mathbf{inr}\,x_2.\,\mathbf{thunk}\,M_2\})\end{aligned}$$

In GCBPV, all side-effects occur at returner types. This is similar to Levy-style call-by-name, where side-effects occur at base types. Lambda abstraction therefore does not thunk effects, so η-expansion is valid for function types. Similarly, products of computation types are *lazy.* The computation $1`\lambda\{1.\,M_1, 2.\,M_2\}$ has the same semantics as $M_1$ (it does not evaluate $M_2$), and $\eta$-expansion is valid (if $M$ is a product of computations, it has the same semantics as $\lambda\{1.\,1`M,\ 2.\,2`M\}$).

Ground types G in GCBPV are value types that do not contain thunks.

$$G \ ::=\ b \mid \mathbf{unit} \mid G_1 \times G_2 \mid \mathbf{empty} \mid G_1 + G_2$$

The notion of signature for graded call-by-push-value is almost identical to that of the graded monadic metalanguage (Definition 2.5.1).

Definition 2.7.1 A *GCBPV signature* consists of the following data:

- A preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ of *effects.*
- A set $\mathcal{B}$ of *base types.*
- A family of pairwise disjoint sets $\mathcal{K}_A$ of *constants of type* A, indexed by value types A.
- A set Σ of *operations.*
- For each operation $\mathrm{op} \in \Sigma$, ground types $\mathrm{car}_{\mathrm{op}}$ and $\mathrm{ar}_{\mathrm{op}}$, respectively called the *coarity* and *arity* of op, and an effect $\mathrm{eff}_{\mathrm{op}} \in \mathcal{E}$. ◀

$$\boxed{\Gamma \vdash V : A}$$

$$\frac{}{\Gamma \vdash c : A}\ \text{if } c \in \mathcal{K}_A \qquad \frac{}{\Gamma \vdash x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash () : \textbf{unit}}$$

$$\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \textbf{fst}\, V : A_1} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \textbf{snd}\, V : A_2} \qquad \frac{\Gamma \vdash V : \textbf{empty}}{\Gamma \vdash \textbf{case}_A\ V\ \textbf{of}\ \{\} : A}$$

$$\frac{\Gamma \vdash V : A_1}{\Gamma \vdash \textbf{inl}_{A_2} V : A_1 + A_2} \qquad \frac{\Gamma \vdash V : A_2}{\Gamma \vdash \textbf{inr}_{A_1} V : A_1 + A_2}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash W_1 : B \qquad \Gamma, x_2 : A_2 \vdash W_2 : B}{\Gamma \vdash \textbf{case}\ V\ \textbf{of}\ \{\textbf{inl}\, x_1.\, W_1,\ \textbf{inr}\, x_2.\, W_2\} : B}$$

$$\frac{\Gamma \,\underline{\vdash}\, M : \underline{C}}{\Gamma \vdash \textbf{thunk}\, M : \textbf{U}\underline{C}}$$

$$\boxed{\Gamma \,\underline{\vdash}\, M : \underline{C}}$$

$$\frac{}{\Gamma \,\underline{\vdash}\, \lambda\{\} : \underline{\textbf{unit}}} \qquad \frac{\Gamma \,\underline{\vdash}\, M_1 : \underline{C}_1 \qquad \Gamma \,\underline{\vdash}\, M_2 : \underline{C}_2}{\Gamma \,\underline{\vdash}\, \lambda\{1.\, M_1, 2.\, M_2\} : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2} \qquad \frac{\Gamma \,\underline{\vdash}\, M : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2}{\Gamma \,\underline{\vdash}\, 1\text{‘}M : \underline{C}_1} \qquad \frac{\Gamma \,\underline{\vdash}\, M : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2}{\Gamma \,\underline{\vdash}\, 2\text{‘}M : \underline{C}_2}$$

$$\frac{\Gamma, x : A \,\underline{\vdash}\, M : \underline{C}}{\Gamma \,\underline{\vdash}\, \lambda x{:}A.\, M : A \rightarrow \underline{C}} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma \,\underline{\vdash}\, M : A \rightarrow \underline{C}}{\Gamma \,\underline{\vdash}\, V\text{‘}M : \underline{C}}$$

$$\frac{\Gamma \vdash V : \mathrm{car}_{\mathrm{op}}}{\Gamma \,\underline{\vdash}\, \mathrm{op}\, V : \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}} \qquad \frac{\Gamma \vdash V : A}{\Gamma \,\underline{\vdash}\, \langle V \rangle : \langle 1 \rangle A} \qquad \frac{\Gamma \,\underline{\vdash}\, M : \langle \varepsilon \rangle A \qquad \Gamma, x : A \,\underline{\vdash}\, N : \underline{C}}{\Gamma \,\underline{\vdash}\, M\ \textbf{to}\ x.\, N : \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}}$$

$$\frac{\Gamma \,\underline{\vdash}\, M : \langle \varepsilon \rangle A}{\Gamma \,\underline{\vdash}\, \textbf{coerce}_{\varepsilon \le \varepsilon'}\, M : \langle \varepsilon' \rangle A}\ \text{if } \varepsilon \le \varepsilon' \qquad \frac{\Gamma \vdash V : \textbf{U}\underline{C}}{\Gamma \,\underline{\vdash}\, \textbf{force}\, V : \underline{C}}$$

Figure 2.11: Graded call-by-push-value typing rules

Typing contexts Γ are ordered lists mapping variable names to value types A; as usual, we assume no variable appears more than once. Variables do not have computation types (but they may be thunks of computations). There are two typing judgments: $\Gamma \vdash V : A$ means the value term V has value type A in context Γ, and similarly $\Gamma \underline{\vdash} M : \underline{C}$ means the computation term M has computation type $\underline{C}$ in context Γ. To define them, we use a function $\langle\!\langle \varepsilon \rangle\!\rangle$ that adds the effect ε to computation types, similar to the function on types used for Levy-style call-by-name. The computation type $\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}$ is defined by recursion on the structure of $\underline{C}$.

$$\begin{aligned} \langle\!\langle \varepsilon \rangle\!\rangle \underline{\mathbf{unit}} &:= \underline{\mathbf{unit}} & \qquad \langle\!\langle \varepsilon \rangle\!\rangle (\underline{C}_1 \underline{\times} \underline{C}_2) &:= \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_1 \underline{\times} \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_2 \\ \langle\!\langle \varepsilon \rangle\!\rangle (A \to \underline{C}) &:= A \to \langle\!\langle \varepsilon \rangle\!\rangle \underline{C} & \langle\!\langle \varepsilon \rangle\!\rangle (\langle \varepsilon' \rangle A) &:= \langle \varepsilon \cdot \varepsilon' \rangle A \end{aligned}$$

The base case of the recursion is returner types, where we collect the two effects ε and ε' together by multiplying them. We do not look at the structure of A in this case. The definition of the typing judgments is given inductively by the rules in Figure 2.11. As for our other type systems, rules that extend typing contexts implicitly assume the extra variable is fresh. We use $\langle\!\langle \varepsilon \rangle\!\rangle$ in the typing rule for **to** because it evaluates more than one computation.

We also have a substitution lemma for GCBPV. Since we need it later, we first define a notion of well-typed substitution. *Substitutions* are given by the following grammar:

$$\sigma \ ::= \ \diamond \mid \sigma, x \mapsto V$$

where $\diamond$ is the empty substitution. Variables are associated with value types (not computation types) in typing contexts, and are therefore mapped to value terms by substitutions. We have a typing judgment $\Gamma \vdash \sigma : \Delta$ for substitutions, meaning in the context Γ the terms in σ have the types given in the context Δ. This is defined as follows:

$$\frac{}{\Gamma \vdash \diamond : \diamond} \qquad\qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash V : A}{\Gamma \vdash (\sigma,\, x \mapsto V) : (\Delta, x : A)}$$

We write $V[\sigma]$ and $M[\sigma]$ for the applications of the substitution σ to the value term V and to the computation term M. These are defined by induction on the structure of the terms, and are capture-avoiding. The key property of the substitution typing judgment is the substitution lemma:

Lemma 2.7.2 (Substitution) Suppose that $\Gamma \vdash \sigma : \Delta$.

1. (Values) If $\Delta \vdash V : B$ then $\Gamma \vdash V[\sigma] : B$.
2. (Computations) If $\Delta \underline{\vdash} M : \underline{C}$ then $\Gamma \underline{\vdash} M[\sigma] : \underline{C}$.

Proof sketch. There is a weakening lemma on value terms, which states that if $\Gamma, \Gamma' \vdash W : A$ then $\Gamma, x : A', \Gamma' \vdash W : A$. This is proved by induction on the typing derivation mutually with a similar weakening lemma for computation terms. The weakening lemma for value terms can then be extended to substitutions. Finally, the substitution lemma is proved by induction on the typing derivation for V (or M), using weakening. □

GCBPV allows subeffecting on returner types using **coerce**. We could have instead included a general form of sub*typing*, by including terms of the form $\mathbf{coerce}_{\underline{C} <: \underline{D}}\, M$ in the syntax, but this turns out to be unnecessary. Subtyping $\underline{C} <: \underline{D}$ for computation types and $A <: B$ for value types is defined inductively by the rules on the left of Figure 2.12. Both subtyping relations are provably reflexive and transitive. Subtyping on terms is then syntactic sugar: value terms $\mathbf{coerce}_{A <: B}\, V$ and computation terms $\mathbf{coerce}_{\underline{C} <: \underline{D}}\, M$ are given by induction on the derivation

Subtyping rule	Syntactic sugar
$\boxed{A <: B}$	$\boxed{\Gamma \vdash \mathbf{coerce}_{A<:B}\, V : B}$
$\dfrac{}{b <: b}$	$\mathbf{coerce}_{b<:b}\, V := V$
$\dfrac{}{\mathbf{unit} <: \mathbf{unit}}$	$\mathbf{coerce}_{\mathbf{unit}<:\mathbf{unit}}\, V := V$
$\dfrac{A_1 <: B_1 \qquad A_2 <: B_2}{A_1 \times A_2 <: B_1 \times B_2}$	$\mathbf{coerce}_{A_1\times A_2<:B_1\times B_2}\, V :=$ $\quad (\mathbf{coerce}_{A_1<:B_1}\, (\mathbf{fst}\, V), \mathbf{coerce}_{A_2<:B_2}\, (\mathbf{snd}\, V))$
$\dfrac{}{\mathbf{empty} <: \mathbf{empty}}$	$\mathbf{coerce}_{\mathbf{empty}<:\mathbf{empty}}\, V := V$
$\dfrac{A_1 <: B_1 \qquad A_2 <: B_2}{A_1 + A_2 <: B_1 + B_2}$	$\mathbf{coerce}_{A_1+A_2<:B_1+B_2}\, V :=$ $\quad \mathbf{case}\ V\ \mathbf{of}$ $\qquad \{\mathbf{inl}\ x_1.\, \mathbf{inl}_{B_2}\, (\mathbf{coerce}_{A_1<:B_1}\, x_1)$ $\qquad , \mathbf{inr}\ x_2.\, \mathbf{inr}_{B_1}\, (\mathbf{coerce}_{A_2<:B_2}\, x_2)\}$
$\dfrac{\underline{C} <: \underline{D}}{\mathbf{U}\,\underline{C} <: \mathbf{U}\,\underline{D}}$	$\mathbf{coerce}_{\mathbf{U}\,\underline{C}<:\mathbf{U}\,\underline{D}}\, V := \mathbf{thunk}\, (\mathbf{coerce}_{\underline{C}<:\underline{D}}\, (\mathbf{force}\, V))$
$\boxed{\underline{C} <: \underline{D}}$	$\boxed{\Gamma \mathrel{\underline{\vdash}} \mathbf{coerce}_{\underline{C}<:\underline{D}}\, M : \underline{D}}$
$\dfrac{}{\underline{\mathbf{unit}} <: \underline{\mathbf{unit}}}$	$\mathbf{coerce}_{\underline{\mathbf{unit}}<:\underline{\mathbf{unit}}}\, M := M$
$\dfrac{\underline{C}_1 <: \underline{D}_1 \qquad \underline{C}_2 <: \underline{D}_2}{\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 <: \underline{D}_1 \mathbin{\underline{\times}} \underline{D}_2}$	$\mathbf{coerce}_{\underline{C}_1\underline{\times}\underline{C}_2<:\underline{D}_1\underline{\times}\underline{D}_2}\, M :=$ $\quad \lambda\{1.\, \mathbf{coerce}_{\underline{C}_1<:\underline{D}_1}\, (1`M),\ 2.\, \mathbf{coerce}_{\underline{C}_2<:\underline{D}_2}\, (2`M)\}$
$\dfrac{B <: A \qquad \underline{C} <: \underline{D}}{(A \to \underline{C}) <: (B \to \underline{D})}$	$\mathbf{coerce}_{(A\to\underline{C})<:(B\to\underline{D})}\, M :=$ $\quad \lambda x : B.\, \mathbf{coerce}_{\underline{C}<:\underline{D}}\, ((\mathbf{coerce}_{B<:A}\, x)\ `\ M)$
$\dfrac{\varepsilon \leq \varepsilon' \qquad A <: B}{\langle\varepsilon\rangle A <: \langle\varepsilon'\rangle B}$	$\mathbf{coerce}_{\langle\varepsilon\rangle A<:\langle\varepsilon'\rangle B}\, M := \mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, (M\ \mathbf{to}\ x.\, \langle\mathbf{coerce}_{A<:B}\, x\rangle)$

Figure 2.12: Subtyping in graded call-by-push-value

of subtyping on the right of Figure 2.12. In general, the definition η-expands the terms (for example, coercions between function types introduce lambdas) and propagates coercions down to returner types. Recall that η-expansion preserves the semantics of GCBPV terms. Coercions have the expected typing: if $A <: B$ then $\Gamma \vdash V : A$ implies $\Gamma \vdash \mathbf{coerce}_{A<:B}\, V : B$, and if $\underline{C} <: \underline{D}$ then $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ implies $\Gamma \mathrel{\underline{\vdash}} \mathbf{coerce}_{\underline{C}<:\underline{D}}\, M : \underline{D}$.

A useful property is that the functions $\langle\!\langle \varepsilon \rangle\!\rangle$ on computation types form an *action* of the preordered monoid of effects on the preorder of computation types (ordered by subtyping).

Definition 2.7.3 An *action* of a preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ on a preordered set $(X, \sqsubseteq)$ is a function $* : \mathcal{E} \times X \to X$ that is monotone in both arguments and satisfies

$$1 * x \;=\; x \qquad \varepsilon * (\varepsilon' * x) \;=\; (\varepsilon \cdot \varepsilon') * x$$

for all $\varepsilon, \varepsilon' \in \mathcal{E}$ and $x \in X$. ◄

2.7.2 (In)equational theories

As for GMM (Section 2.5) we give an equational theory for determining when it is correct to replace one program with another. However, there are two key differences in this section. The first is that we define a general notion of *theory* for GCBPV, consisting of some core axioms possibly augmented with additional axioms, rather than constraining ourselves to one particular equational theory. The reason for this is that we add extra axioms to capture the behaviour of side-effects included in the signature. The second difference is that we do not constrain ourselves to *symmetric* theories (hence we prefer the term *in*equational theory rather than equational theory). We give examples of non-symmetric theories in Chapter 4.

For GCBPV an inequational theory consists of two judgment forms, one for values and one for computations:

$$\Gamma \vdash V \preccurlyeq W : A \qquad \Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$$

These mean that the terms are well-typed, and every behaviour of V (or M) is a possible behaviour of W (or N). (For this chapter, we can use the same intuition as for $\equiv$ in GMM. The lack of symmetry only becomes important in Chapter 4.) As before, we omit the typing context and type when they are obvious or unimportant.

Each inequational theory must satisfy a core set of symmetric axioms, which are listed in Figure 2.13. We use the symbol $\equiv$ to indicate that they should be read symmetrically (we define $\equiv$ to be the intersection of $\preccurlyeq$ and its opposite $\succcurlyeq$). They are just the usual axioms for call-by-push-value (with complex values), plus some additional axioms for coercions. Each axiom is subject to suitable restrictions on free variables and typing (for example, $M \equiv \lambda\{\}$ can hold only if M has type $\underline{\mathbf{unit}}$).

The first group consists of β laws. The second consists of η laws; there is an η law for every GCBPV type (excluding base types). In the third group, the first two equations state that sequencing of computations commutes with formation of binary products and functions. A consequence of these two axioms and the η laws is that it is not necessary to allow arbitrary computation types on the right-hand side of **to**; returner types suffice. This is the case because:

$$\begin{array}{lcll}
M \,\mathbf{to}\, x.\, N & \equiv & \lambda\{\} & \text{if } N : \underline{\mathbf{unit}} \\
M \,\mathbf{to}\, x.\, N & \equiv & \lambda\{1.\, M \,\mathbf{to}\, x.\, 1\text{‘}N,\ 2.\, M \,\mathbf{to}\, x.\, 2\text{‘}N\} & \text{if } N : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 \\
M \,\mathbf{to}\, x.\, N & \equiv & \lambda y{:}A.\, M \,\mathbf{to}\, x.\, y\text{‘}N & \text{if } N : A \to \underline{C}
\end{array}$$

where y is not free in N. (These are instances of the equational theory defined below, but not axioms.) The third axiom of the third group is associativity of **to**. The fourth is commutativity

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{fst}\,(V_1, V_2) & \equiv & V_1 & \mathbf{snd}\,(V_1, V_2) & \equiv & V_2 \\
\mathbf{case}\ \mathbf{inl}_{A_2} V\ \mathbf{of} & & & \mathbf{case}\ \mathbf{inr}_{A_1} V\ \mathbf{of} & & \\
\{\mathbf{inl}\ x_1.\, W_1 & \equiv & W_1[x_1 \mapsto V] & \{\mathbf{inl}\ x_1.\, W_1 & \equiv & W_2[x_2 \mapsto V] \\
,\ \mathbf{inr}\ x_2.\, W_2\} & & & ,\ \mathbf{inr}\ x_2.\, W_2\} & & \\
1\,\text{‘}\,\lambda\{1.\, M_1, 2.\, M_2\} & \equiv & M_1 & 2\,\text{‘}\,\lambda\{1.\, M_1, 2.\, M_2\} & \equiv & M_2 \\
V\,\text{‘}\,\lambda x{:}A.\, M & \equiv & M[x \mapsto V] & \langle V \rangle\ \mathbf{to}\ x.\, M & \equiv & M[x \mapsto V] \\
\mathbf{force}\,(\mathbf{thunk}\, M) & \equiv & M & & &
\end{array}
$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
V & \equiv & () & V & \equiv & (\mathbf{fst}\, V, \mathbf{snd}\, V) \\
V & \equiv & \mathbf{case}_A\ W\ \mathbf{of}\ \{\} & & & \mathbf{case}\ W\ \mathbf{of} \\
V & \equiv & \mathbf{thunk}\,(\mathbf{force}\, V) & V[x \mapsto W] & \equiv & \quad \{\mathbf{inl}\ y_1.\, V[x \mapsto \mathbf{inl}_{A_2} y_1] \\
 & & & & & \quad ,\ \mathbf{inr}\ y_2.\, V[x \mapsto \mathbf{inr}_{A_1} y_2]\} \\
M & \equiv & \lambda\{\} & M & \equiv & (1\text{‘}M, 2\text{‘}M) \\
M & \equiv & \lambda x{:}A.\, x\text{‘}M & M & \equiv & M\ \mathbf{to}\ x.\, \langle x \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\lambda\{1.\, M\ \mathbf{to}\ x.\, N_1,\ 2.\, M\ \mathbf{to}\ x.\, N_2\} & \equiv & M\ \mathbf{to}\ x.\, \lambda\{1.\, N_1,\ 2.\, N_2\} \\
\lambda y{:}A.\, M\ \mathbf{to}\ x.\, N & \equiv & M\ \mathbf{to}\ x.\, \lambda y{:}A.\, N \\
(M_1\ \mathbf{to}\ x.\, M_2)\ \mathbf{to}\ y.\, M_3 & \equiv & M_1\ \mathbf{to}\ x.\, M_2\ \mathbf{to}\ y.\, M_3 \\
\mathbf{coerce}_{\varepsilon_1 \cdot \varepsilon_2 \leq \varepsilon'_1 \cdot \varepsilon'_2}\,(M\ \mathbf{to}\ x.\, N) & \equiv & (\mathbf{coerce}_{\varepsilon_1 \leq \varepsilon'_1}\, M)\ \mathbf{to}\ x.\, \mathbf{coerce}_{\varepsilon_2 \leq \varepsilon'_2}\, N \\
\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M & \equiv & M \\
\mathbf{coerce}_{\varepsilon' \leq \varepsilon''}\,(\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M) & \equiv & \mathbf{coerce}_{\varepsilon \leq \varepsilon''}\, M
\end{array}
$$

Figure 2.13: Core axioms of GCBPV inequational theories

of **coerce** with **to**. This axiom can only be applied if N has returner type; this is not an issue because we can write all uses of **to** in terms of those that only use returner types. The equation

$$\mathbf{coerce}_{\langle\!\langle \varepsilon \rangle\!\rangle \underline{C} <: \langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}'}\,(M\ \mathbf{to}\ x.\, N) \ \equiv\ (\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M)\ \mathbf{to}\ x.\, \mathbf{coerce}_{\underline{C} <: \underline{C}'}\, N$$

follows. Finally, there are two equations corresponding to reflexivity and transitivity of the order $\leq$ on effects. The equations

$$\mathbf{coerce}_{\underline{C} <: \underline{C}}\, M \ \equiv\ M \qquad \mathbf{coerce}_{\underline{C}' <: \underline{C}''}\,(\mathbf{coerce}_{\underline{C} <: \underline{C}'}\, M) \ \equiv\ \mathbf{coerce}_{\underline{C} <: \underline{C}''}\, M$$

follow from these.

Each inequational theory is also required to be closed under congruence. Congruence involves two kinds of *term context*: we have $C[]$ for value terms with a single hole, and $\underline{C}[]$ for computation terms with a single hole. We write □ for a hole where a value term is expected, and $\underline{\square}$ for a hole where a computation term is expected.

$$
\begin{array}{rcl}
C[] & ::= & \square \mid (C[], V_2) \mid (V_1, C[]) \mid \mathbf{fst}\, C[] \mid \mathbf{snd}\, C[] \mid \mathbf{case}_A\, C[]\ \mathbf{of}\ \{\} \\
 & & \mid \mathbf{inl}_{A_2} C[] \mid \mathbf{inr}_{A_1} C[] \mid \mathbf{case}\ C[]\ \mathbf{of}\ \{\mathbf{inl}\, x_1 .\, W_1,\ \mathbf{inr}\, x_2 .\, W_2\} \\
 & & \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1 .\, C[],\ \mathbf{inr}\, x_2 .\, W_2\} \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1 .\, W_1,\ \mathbf{inr}\, x_2 .\, C[]\} \\
 & & \mid \mathbf{thunk}\, \underline{C}[] \\
\underline{C}[] & ::= & \underline{\square} \mid \lambda\{1 .\, M_1, 2 .\, \underline{C}[]\} \mid \lambda\{1 .\, \underline{C}[], 2 .\, M_2\} \mid 1`\underline{C}[] \mid 2`\underline{C}[] \mid \lambda x{:}A .\, \underline{C}[] \mid C[]`M \mid V`\underline{C}[] \\
 & & \mid \mathrm{op}\, C[] \mid \langle C[] \rangle \mid \underline{C}[]\ \mathbf{to}\ x .\, N \mid M\ \mathbf{to}\ x .\, \underline{C}[] \mid \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, \underline{C}[] \mid \mathbf{force}\, C[]
\end{array}
$$

If $C[]$ is a term context that expects a value term, we write $C[V]$ for the term formed by replacing $\square$ with the value term V, and similarly for the other kinds of term context. The term context may capture some of the free variables of V.

Both judgments of an inequational theory are also required to be closed under substitution (see also Lemma 2.7.2). To state closure under substitution, we extend the value judgment

$$\Gamma \vdash V \preccurlyeq W : A$$

of each inequational theory to substitutions componentwise: given a typing context Γ, as well as another typing context and two substitutions

$$\Delta = x_1 : A_1, \ldots x_n : A_n \qquad \sigma = x_1 \mapsto V_1, \ldots, x_n \mapsto V_n \qquad \sigma' = x_1 \mapsto V'_1, \ldots, x_n \mapsto V'_n$$

we write $\Gamma \vdash \sigma \preccurlyeq \sigma' : \Delta$ if $\Gamma \vdash \sigma : \Delta$, $\Gamma \vdash \sigma' : \Delta$, and for all i,

$$\Gamma \vdash V_i \preccurlyeq V'_i : A_i$$

We now collect together our requirements on inequational theories into a single definition:

Definition 2.7.4 (Inequational theory) An *inequational theory* consists of a GCBPV signature and two judgments

$$\Gamma \vdash V \preccurlyeq W : A \qquad \Gamma \underline{\vdash} M \preccurlyeq N : \underline{C}$$

such that:

- Preorder:
  - Values: if $\Gamma \vdash V : A$ then $\Gamma \vdash V \preccurlyeq V : A$, and if $\Gamma \vdash V_1 \preccurlyeq V_2 : A$ and $\Gamma \vdash V_2 \preccurlyeq V_3 : A$ then $\Gamma \vdash V_1 \preccurlyeq V_3 : A$.
  - Computations: if $\Gamma \underline{\vdash} M : \underline{C}$ then $\Gamma \underline{\vdash} M \preccurlyeq M : \underline{C}$, and if $\Gamma \underline{\vdash} M_1 \preccurlyeq M_2 : \underline{C}$ and $\Gamma \underline{\vdash} M_2 \preccurlyeq M_3 : \underline{C}$ then $\Gamma \underline{\vdash} M_1 \preccurlyeq M_3 : \underline{C}$.
- Congruence: if $\Gamma \vdash V \preccurlyeq W : A$ then for term contexts with hole $\square$

  $$
  \begin{array}{rcl}
  \Gamma' \vdash C[V] : B \wedge \Gamma' \vdash C[W] : B & \Rightarrow & \Gamma' \vdash C[V] \preccurlyeq C[W] : B \\
  \Gamma' \underline{\vdash} \underline{C}[V] : \underline{D} \wedge \Gamma' \underline{\vdash} \underline{C}[W] : \underline{D} & \Rightarrow & \Gamma' \underline{\vdash} \underline{C}[V] \preccurlyeq \underline{C}[W] : \underline{D}
  \end{array}
  $$

  If $\Gamma \underline{\vdash} M \preccurlyeq N : \underline{C}$ then for term contexts with hole $\underline{\square}$

  $$
  \begin{array}{rcl}
  \Gamma' \vdash C[M] : B \wedge \Gamma' \vdash C[N] : B & \Rightarrow & \Gamma' \vdash C[M] \preccurlyeq C[N] : B \\
  \Gamma' \underline{\vdash} \underline{C}[M] : \underline{D} \wedge \Gamma' \underline{\vdash} \underline{C}[N] : \underline{D} & \Rightarrow & \Gamma' \underline{\vdash} \underline{C}[M] \preccurlyeq \underline{C}[N] : \underline{D}
  \end{array}
  $$

- Substitution: If $\Gamma \vdash \sigma \preccurlyeq \sigma' : \Delta$ then

  $$
  \begin{array}{rcl}
  \Delta \vdash V \preccurlyeq W : B & \Rightarrow & \Gamma \vdash V[\sigma] \preccurlyeq W[\sigma'] : B \\
  \Delta \underline{\vdash} M \preccurlyeq N : \underline{C} & \Rightarrow & \Gamma \underline{\vdash} M[\sigma] \preccurlyeq N[\sigma'] : \underline{C}
  \end{array}
  $$

$$\begin{array}{rcl}
\mathrm{get}\,()\ \mathbf{to}\ x.\,(\mathrm{put}\,x;\ \langle x\rangle) & \equiv & \mathbf{coerce}_{\{\mathrm{get}\}\leq\{\mathrm{get},\mathrm{put}\}}\,(\mathrm{get}\,())\\
\mathrm{put}\,V;\ \mathrm{get}\,() & \equiv & \mathrm{put}\,V;\ \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle V\rangle\\
\mathrm{put}\,V_1;\ \mathrm{put}\,V_2 & \equiv & \mathrm{put}\,V_2\\
\mathrm{get}\,()\ \mathbf{to}\ x.\,\mathrm{get}\,()\ \mathbf{to}\ y.\,\langle(x,y)\rangle & \equiv & \mathrm{get}\,()\ \mathbf{to}\ x.\,\langle(x,x)\rangle\\
\mathrm{get}\,();\ \langle()\rangle & \equiv & \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle()\rangle
\end{array}$$

Figure 2.14: Signature axioms for global state

- Core axioms:
  - Values: if $\Gamma \vdash V : A$ and $\Gamma \vdash W : A$, and $V \equiv W$ is an instance of an axiom in Figure 2.13, then $\Gamma \vdash V \preccurlyeq W : A$ and $\Gamma \vdash W \preccurlyeq V : A$.
  - Computations: if $\Gamma \underline{\vdash} M : \underline{C}$ and $\Gamma \underline{\vdash} N : \underline{C}$, and $M \equiv N$ is an instance of an axiom in Figure 2.13, then $\Gamma \underline{\vdash} M \preccurlyeq N : \underline{C}$ and $\Gamma \underline{\vdash} N \preccurlyeq M : \underline{C}$. ◄

For a fixed typing context Γ and type A, each inequational theory is a preorder on terms of type A in context Γ, and similarly for computations. We use the symbol $\equiv$ for the symmetric part of the inequational theory (e.g. on values, $\Gamma \vdash V \equiv W : A$ means both $\Gamma \vdash V \preccurlyeq W : A$ and $\Gamma \vdash V \preccurlyeq W : A$ hold).

The smallest possible inequational theory (i.e. the one that relates the fewest terms), is given by taking the core axioms, and closing them under reflexivity, transitivity and congruence. (Closure under substitution holds automatically in this case.) Since all of the core axioms are symmetric, the smallest inequational theory is also symmetric.

In general, to specify an inequational theory, we give a list of (possibly non-symmetric) *signature axioms* (which characterize the side-effects in the signature) to be taken in addition to the core axioms. The $\preccurlyeq$ judgments are then given by closing under congruence, reflexivity and transitivity. We do not assume that inequational theories are given by a finite set of axioms (or axiom schemas). Nor do we forbid infinitary rules in the definition. This allows us to consider inequational theories for recursion (using an infinitary *rational continuity* rule [19]).

We give one example here, for global state (further examples are given in Chapter 4). For this example, we use the Gifford-style effect algebra with two operations: get with coarity **unit** and arity **bool**, and put with coarity **unit** and arity **bool**. Effects are subsets of $\{\mathrm{get}, \mathrm{put}\}$. We write $M;\ N$ as syntactic sugar for M **to** $x.\,N$, where x is fresh. The signature axioms for this example are listed in Figure 2.14; they characterize the behaviour of get and put. The signature axioms for global state imply more general equations. For example, the fourth axiom, which allows us to merge two adjacent reads, implies the following general equation:

$$\mathrm{get}\,()\ \mathbf{to}\ x.\,\mathrm{get}\,()\ \mathbf{to}\ y.\,M \ \equiv\ \mathrm{get}\,()\ \mathbf{to}\ z.\,M[x \mapsto z, y \mapsto z]$$

As an aside, we mention that these are not the same as the axioms given by Plotkin and Power [88]. If we forget about the effect system (more formally, if we use the trivial effect algebra), then our axioms are equivalent to Plotkin and Power's. However, the proof that our fifth axiom (which is just $\mathrm{get}\,();\ \langle()\rangle \equiv \langle()\rangle$ for the trivial effect algebra) follows from theirs is a sequence of equalities involving put. We cannot use that proof with a Gifford-style effect algebra because the effect for this axiom does not contain $\{\mathrm{put}\}$. In general, when adding an effect system it may be necessary to specify additional axioms.

In the rest of this chapter we consider arbitrary inequational theories $\preccurlyeq$. Given a value term V we define $\lfloor V \rfloor$ to be V but with all uses of **coerce** deleted, and similarly for computation terms. We have:

Lemma 2.7.5 Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. Given two value terms V_1, V_2 such that $\Gamma \vdash V_1 : A$ and $\Gamma \vdash V_2 : A$, if $\lfloor V_1 \rfloor = \lfloor V_2 \rfloor$ then $\Gamma \vdash V_1 \equiv V_2 : A$.
2. Given two computation terms M_1, M_2 such that $\Gamma \underline{\vdash} M_1 : \underline{C}$ and $\Gamma \underline{\vdash} M_2 : \underline{C}$, if $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$ then $\Gamma \underline{\vdash} M_1 \equiv M_2 : \underline{C}$. ◄

We use this fact when considering the translations from source-language effect systems. A proof is given in Appendix B.1. We conjecture that this lemma does not hold when the assumption about the effect algebra is dropped. A conjectured counterexample is given along with the proof.

When we consider equivalences between terms in graded call-by-push-value, we ask for the terms to have the same observable behaviour. We therefore define a *contextual preorder* for GCBPV, by considering closed computations that return elements of ground types.

Definition 2.7.6 (Contextual preorder) Given an inequational theory, the *contextual preorder* consists of two judgment forms.

1. Between value terms: $\Gamma \vdash V \preccurlyeq_{\mathrm{ctx}} W : A$ if $\Gamma \vdash V : A$, $\Gamma \vdash W : A$, and for all ground types G, effects ε, and term contexts $\underline{C}[]$ with hole $\square$ such that $\diamond \underline{\vdash} \underline{C}[V] : \langle\varepsilon\rangle G$ and $\diamond \underline{\vdash} \underline{C}[W] : \langle\varepsilon\rangle G$ we have
$$\diamond \underline{\vdash} \underline{C}[V] \preccurlyeq \underline{C}[W] : \langle\varepsilon\rangle G$$
2. Between computation terms: $\Gamma \vdash M \preccurlyeq_{\mathrm{ctx}} N : \underline{C}$ if $\Gamma \underline{\vdash} M : \underline{C}$, $\Gamma \underline{\vdash} N : \underline{C}$, and for all ground types G, effects ε, and term contexts $\underline{C}[]$ with hole $\underline{\square}$ such that $\diamond \underline{\vdash} \underline{C}[M] : \langle\varepsilon\rangle G$ and $\diamond \underline{\vdash} \underline{C}[N] : \langle\varepsilon\rangle G$ we have
$$\diamond \underline{\vdash} \underline{C}[M] \preccurlyeq \underline{C}[N] : \langle\varepsilon\rangle G$$ ◄

As for inequational theories, we often omit the typing context and type. The inequational theory is stronger than contextual equivalence: if $M \preccurlyeq N$ then $M \preccurlyeq_{\mathrm{ctx}} N$ (and similarly for value terms), because $\preccurlyeq$ is a congruence. For closed terms of $\langle\varepsilon\rangle G$, where G is a ground type, the converse also holds: if $\diamond \underline{\vdash} M \preccurlyeq_{\mathrm{ctx}} N : \langle\varepsilon\rangle G$ then $\diamond \underline{\vdash} M \preccurlyeq N : \langle\varepsilon\rangle G$. This is not generally the case for open terms or for other types. For example, with global state

$$x : \langle\{\mathrm{get}\}\rangle\mathbf{unit} \,\underline{\vdash}\; x \;\preccurlyeq\; \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle()\rangle \;:\langle\{\mathrm{get}\}\rangle\mathbf{unit}$$

does not hold, but the two computations have the same observable behaviour.

2.7.3 Translations into GCBPV

To reason about evaluation orders using graded call-by-push-value, we give compositional translations from our source-language effect systems, similar to the translations into the graded monadic metalanguage described in Section 2.5.2. In addition to call-by-value and Moggi-style call-by-name we give a translation from Levy-style call-by-name into GCBPV.[7] We could not do this satisfactorily for GMM. The call-by-value and Levy-style call-by-name translations we give here are based on those of Levy [56], with some minor modifications to deal with effects.

[7] We can also translate most of GMM into GCBPV by decomposing the graded monad $\langle\varepsilon\rangle-$ as $\mathbf{U}\,(\langle\varepsilon\rangle-)$. (Function types cannot be translated.)

$$\begin{aligned} \llparenthesis \mathbf{unit} \rrparenthesis &:= \mathbf{unit} \\ \llparenthesis \mathbf{bool} \rrparenthesis &:= \mathbf{unit} + \mathbf{unit} \\ \llparenthesis \tau \xrightarrow{\varepsilon} \tau' \rrparenthesis &:= \mathbf{U}\,(\llparenthesis \tau \rrparenthesis \to \langle \varepsilon \rangle \llparenthesis \tau' \rrparenthesis) \end{aligned} \qquad \begin{aligned} \llparenthesis \diamond \rrparenthesis &:= \diamond \\ \llparenthesis \Gamma, x : \tau \rrparenthesis &:= \llparenthesis \Gamma \rrparenthesis,\ x : \llparenthesis \tau \rrparenthesis \end{aligned}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} x : \tau \mathbin{\&} 1 \rrparenthesis := \langle x \rangle} \qquad \frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \mathrm{car}_{\mathrm{op}} \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \mathbin{\&} \varepsilon \cdot \mathrm{eff}_{\mathrm{op}} \rrparenthesis = M \ \mathbf{to}\ x.\, \mathrm{op}\, x}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} () : \mathbf{unit} \mathbin{\&} 1 \rrparenthesis := \langle () \rangle}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{true} : \mathbf{bool} \mathbin{\&} 1 \rrparenthesis := \langle \mathbf{inl}_{\mathbf{unit}}\, () \rangle} \qquad \frac{}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{false} : \mathbf{bool} \mathbin{\&} 1 \rrparenthesis := \langle \mathbf{inr}_{\mathbf{unit}}\, () \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1 : \mathbf{bool} \mathbin{\&} \varepsilon \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon' \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_3 : \tau \mathbin{\&} \varepsilon' \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \mathbin{\&} \varepsilon \cdot \varepsilon' \rrparenthesis \ := \ M_1\ \mathbf{to}\ x.\, \underline{\mathbf{case}}\ x\ \mathbf{of}\ \{\mathbf{inl}\,\_.\, M_2,\ \mathbf{inr}\,\_.\, M_3\}}$$

$$\frac{\llparenthesis \Gamma, x : \tau \vdash_{\mathrm{v}} e : \tau' \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} \lambda x{:}\tau.\, e : \tau \xrightarrow{\varepsilon} \tau' \mathbin{\&} 1 \rrparenthesis \ := \ \langle \mathbf{thunk}\,(\lambda x{:}\llparenthesis \tau \rrparenthesis.\, M) \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1 : \tau \xrightarrow{\varepsilon_3} \tau' \mathbin{\&} \varepsilon_1 \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{v}} e_2 : \tau \mathbin{\&} \varepsilon_2 \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash_{\mathrm{v}} e_1\, e_2 : \tau' \mathbin{\&} \varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3 \rrparenthesis \ := \ M_1\ \mathbf{to}\ f.\, M_2\ \mathbf{to}\ x.\ x\,\text{‘}\,\mathbf{force}\, f}$$

$$\frac{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon' \rrparenthesis \ := \ \mathbf{coerce}_{\varepsilon \le \varepsilon'}\, M}$$

Figure 2.15: Translation of call-by-value types (top left), contexts (top right), and typing derivations (bottom) into GCBPV.

The translations we give in this section assume a fixed source-language signature (Definition 2.2.1) that is used to instantiate the source-language effect systems, and a fixed GCBPV signature (Definition 2.7.1). The two signatures are required to be compatible. We translate source-language ground types $\tau \in \{\mathbf{unit}, \mathbf{bool}\}$ into GCBPV as follows:

$$\llparenthesis \mathbf{unit} \rrparenthesis := \mathbf{unit} \qquad \llparenthesis \mathbf{bool} \rrparenthesis := \mathbf{unit} + \mathbf{unit}$$

For compatibility we assume that the two signatures have the same effect algebra, and that for each source-language operation op: the GCBPV signature also includes op; the two signatures assign the same effect $\mathrm{eff}_{\mathrm{op}}$ to op; and if τ and τ' are respectively the coarity and arity of op in the source-language signature, then $\llparenthesis \tau \rrparenthesis$ and $\llparenthesis \tau' \rrparenthesis$ are the coarity and arity of op in the GCBPV signature. The GCBPV signature is allowed to contain operations that are not in the source-language signature, and we place no requirements on base types or constants.

Call-by-value types τ are translated into GCBPV value types $\llparenthesis \tau \rrparenthesis^{\mathrm{v}}$, which represent values of type τ (with no side-effects), and call-by-value contexts Γ are translated into GCBPV contexts $\llparenthesis \Gamma \rrparenthesis^{\mathrm{v}}$. Call-by-value *derivations* $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ are translated into GCBPV *computations* $\llparenthesis \Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon \rrparenthesis^{\mathrm{v}}$ that return elements of $\llparenthesis \tau \rrparenthesis$, so $\llparenthesis \Gamma \rrparenthesis^{\mathrm{v}} \mathrel{\underline{\vdash}} \llparenthesis e \rrparenthesis^{\mathrm{v}} : \langle \varepsilon \rangle \llparenthesis \tau \rrparenthesis^{\mathrm{v}}$, where we omit the context and type in $\llparenthesis e \rrparenthesis^{\mathrm{v}}$. (Recall that we have to translate *derivations* rather than just well-typed

$$\begin{aligned} \llparenthesis \mathbf{unit} \rrparenthesis &:= \mathbf{unit} \\ \llparenthesis \mathbf{bool} \rrparenthesis &:= \mathbf{unit} + \mathbf{unit} \\ \llparenthesis \tau \xrightarrow{\varepsilon,\varepsilon'} \tau' \rrparenthesis &:= \mathbf{U}\,(\mathbf{U}\,(\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis) \to \langle\varepsilon'\rangle\llparenthesis\tau'\rrparenthesis) \end{aligned}$$

$$\begin{aligned} \llparenthesis \diamond \rrparenthesis &:= \diamond \\ \llparenthesis \Gamma, x : \tau \& \varepsilon \rrparenthesis &:= \llparenthesis\Gamma\rrparenthesis,\ x : \mathbf{U}\,(\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis) \end{aligned}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\text{moggi}} x : \tau \,\&\, \varepsilon \rrparenthesis := \mathbf{force}\, x} \qquad \frac{\llparenthesis \Gamma \vdash_{\text{moggi}} e : \text{car}_{\text{op}} \,\&\, \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\text{moggi}} \text{op}\, e : \text{ar}_{\text{op}} \,\&\, \varepsilon \cdot \text{eff}_{\text{op}} \rrparenthesis = M \mathbin{\mathbf{to}} x.\, \text{op}\, x}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\text{moggi}} () : \mathbf{unit} \,\&\, 1 \rrparenthesis := \langle () \rangle}$$

$$\frac{}{\llparenthesis \Gamma \vdash_{\text{moggi}} \mathbf{true} : \mathbf{bool} \,\&\, 1 \rrparenthesis := \langle \mathbf{inl}_{\mathbf{unit}}\,() \rangle} \qquad \frac{}{\llparenthesis \Gamma \vdash_{\text{moggi}} \mathbf{false} : \mathbf{bool} \,\&\, 1 \rrparenthesis := \langle \mathbf{inr}_{\mathbf{unit}}\,() \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\text{moggi}} e_1 : \mathbf{bool} \,\&\, \varepsilon \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\text{moggi}} e_2 : \tau \,\&\, \varepsilon' \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash_{\text{moggi}} e_3 : \tau \,\&\, \varepsilon' \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash_{\text{moggi}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \,\&\, \varepsilon \cdot \varepsilon' \rrparenthesis \ := \ M_1 \mathbin{\mathbf{to}} x.\, \underline{\mathbf{case}}\ x\ \mathbf{of}\ \{\mathbf{inl}\,\_.\, M_2,\ \mathbf{inr}\,\_.\, M_3\}}$$

$$\frac{\llparenthesis \Gamma, x : \tau \& \varepsilon \vdash_{\text{moggi}} e : \tau' \,\&\, \varepsilon' \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\text{moggi}} \lambda x{:}\tau\&\varepsilon.\, e : \tau \xrightarrow{\varepsilon,\varepsilon'} \tau' \,\&\, 1 \rrparenthesis \ := \ \langle \mathbf{thunk}\,(\lambda x{:}\mathbf{U}\,(\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis).\, M) \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash_{\text{moggi}} e_1 : \tau \xrightarrow{\varepsilon_2,\varepsilon_3} \tau' \,\&\, \varepsilon_1 \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\text{moggi}} e_2 : \tau \,\&\, \varepsilon_2 \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash_{\text{moggi}} e_1\, e_2 : \tau' \,\&\, \varepsilon_1 \cdot \varepsilon_3 \rrparenthesis \ := \ M_1 \mathbin{\mathbf{to}} f.\, (\mathbf{thunk}\, M_2)\text{ ‘ }(\mathbf{force}\, f)}$$

$$\frac{\llparenthesis \Gamma \vdash_{\text{moggi}} e : \tau \,\&\, \varepsilon \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\text{moggi}} e : \tau \,\&\, \varepsilon' \rrparenthesis \ := \ \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M}$$

Figure 2.16: Translation of Moggi-style call-by-name types (top left), contexts (top right), and typing derivations (bottom) into GCBPV.

expressions because the call-by-value effect system is not syntax directed.) The side-effects of e are encapsulated in the returner type.

The definition of the call-by-value translation is in Figure 2.15, where we omit the superscript on $\llparenthesis - \rrparenthesis^{\text{v}}$. For function types we use computations of returner type with effect ε (for the latent effect of the function), and the side-effects of the function are thunked by wrapping it in $\mathbf{U}$. This means that call-by-value expressions of function type are translated into computation terms that return thunks of GCBPV functions. For operations op, the expression e is first eagerly evaluated, and then op is applied to the result. In the translation of **if**-expressions we use the eliminator of GCBPV sum types on the computation level (recall that this is defined in terms of the eliminator on the value level). In the translation of lambda abstraction and function application we explicitly specify where side-effects are thunked, and where thunks are forced.

Most of the translation of Moggi-style call-by-name is similar to the call-by-value translation. Types τ are translated into GCBPV value types $\llparenthesis\tau\rrparenthesis^{\text{moggi}}$, which represent values of type τ with no side-effects. Contexts Γ become GCBPV contexts $\llparenthesis\Gamma\rrparenthesis^{\text{moggi}}$. Derivations of $\Gamma \vdash_{\text{moggi}} e : \tau \,\&\, \varepsilon$ become computation terms $\llparenthesis \Gamma \vdash_{\text{moggi}} e : \tau \,\&\, \varepsilon \rrparenthesis^{\text{moggi}}$ of type $\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\text{moggi}}$.

The definition is in Figure 2.16. The primary difference is in the translation of function types,

$$
\begin{aligned}
\llparenthesis \langle\varepsilon\rangle\mathbf{unit} \rrparenthesis &:= \langle\varepsilon\rangle\mathbf{unit}\\
\llparenthesis \langle\varepsilon\rangle\mathbf{bool} \rrparenthesis &:= \langle\varepsilon\rangle(\mathbf{unit} + \mathbf{unit})\\
\llparenthesis \tau \longrightarrow \tau' \rrparenthesis &:= \mathbf{U}\llparenthesis \tau \rrparenthesis \longrightarrow \llparenthesis \tau' \rrparenthesis
\end{aligned}
\qquad\qquad
\begin{aligned}
\llparenthesis \diamond \rrparenthesis &:= \diamond\\
\llparenthesis \Gamma, x : \tau \rrparenthesis &:= \llparenthesis \Gamma \rrparenthesis,\ x : \mathbf{U}\llparenthesis \tau \rrparenthesis
\end{aligned}
$$

$$
\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{n}} x : \tau \rrparenthesis \ := \ \mathbf{force}\, x}
\qquad
\frac{\llparenthesis \Gamma \vdash_{\mathrm{n}} e : \langle\varepsilon\rangle\mathrm{car}_{\mathrm{op}} \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{n}} \mathrm{op}\, e : \langle\varepsilon \cdot \mathrm{eff}_{\mathrm{op}}\rangle\mathrm{ar}_{\mathrm{op}} \rrparenthesis = M \ \mathbf{to}\ x.\, \mathrm{op}\, x}
$$

$$
\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{n}} () : \langle 1\rangle\mathbf{unit} \rrparenthesis \ := \ \langle () \rangle}
$$

$$
\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{n}} \mathbf{true} : \langle 1\rangle\mathbf{bool} \rrparenthesis \ := \ \langle \mathbf{inl}_{\mathbf{unit}}\, () \rangle}
\qquad
\frac{}{\llparenthesis \Gamma \vdash_{\mathrm{n}} \mathbf{false} : \langle 1\rangle\mathbf{bool} \rrparenthesis \ := \ \langle \mathbf{inr}_{\mathbf{unit}}\, () \rangle}
$$

$$
\frac{\llparenthesis \Gamma \vdash_{\mathrm{n}} e_1 : \langle\varepsilon\rangle\mathbf{bool} \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{n}} e_2 : \tau \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash_{\mathrm{n}} e_3 : \tau \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash_{\mathrm{n}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \langle\!\langle \varepsilon \rangle\!\rangle\tau \rrparenthesis \ := \ M_1\ \mathbf{to}\ x.\, \underline{\mathrm{case}}\ x\ \mathbf{of}\ \{\mathbf{inl}\ \_.\, M_2,\ \mathbf{inr}\ \_.\, M_3\}}
$$

$$
\frac{\llparenthesis \Gamma, x : \tau \vdash_{\mathrm{n}} e : \tau' \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{n}} \lambda x{:}\tau.\, e : \tau \longrightarrow \tau' \rrparenthesis \ := \ \lambda x{:}\mathbf{U}\llparenthesis \tau \rrparenthesis.\, M}
\qquad
\frac{\llparenthesis \Gamma \vdash_{\mathrm{n}} e_1 : \tau \longrightarrow \tau' \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash_{\mathrm{n}} e_2 : \tau \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash_{\mathrm{n}} e_1\, e_2 : \tau' \rrparenthesis \ := \ (\mathbf{thunk}\, M_2)\ {}^{\backprime} M_1}
$$

$$
\frac{\llparenthesis \Gamma \vdash_{\mathrm{n}} e : \langle\!\langle \varepsilon \rangle\!\rangle\tau \rrparenthesis = M}{\llparenthesis \Gamma \vdash_{\mathrm{n}} e : \langle\!\langle \varepsilon' \rangle\!\rangle\tau \rrparenthesis \ := \ \mathbf{coerce}_{\langle\!\langle \varepsilon \rangle\!\rangle \llparenthesis \tau \rrparenthesis <: \langle\!\langle \varepsilon' \rangle\!\rangle \llparenthesis \tau \rrparenthesis}\ M}
$$

Figure 2.17: Translation of Levy-style call-by-name types (top left), contexts (top right), and typing derivations (bottom) into GCBPV.

where arguments are thunked computations rather than just values. Similarly, typing contexts contain thunks. Variables x force the corresponding thunk; hence we reevaluate a computation each time x is used. Operations eagerly evaluate their argument. For function application, the argument is not evaluated immediately; instead a thunk is passed to the function.

Both the call-by-value and Moggi-style call-by-name translations into GCBPV are somewhat similar to the corresponding translations into GMM. In this section, we add a translation for Levy-style call-by-name. Unlike the previous two, Levy-style call-by-name types τ are translated into GCBPV *computation* types $\llparenthesis \tau \rrparenthesis^{\mathrm{n}}$. The elements of $\llparenthesis \tau \rrparenthesis^{\mathrm{n}}$ are therefore computations, and might have side-effects. A derivation $\Gamma \vdash_{\mathrm{n}} e : \tau$ is translated into a computation term $\llparenthesis \Gamma \vdash_{\mathrm{n}} e : \tau \rrparenthesis^{\mathrm{n}}$ of type $\llparenthesis \tau \rrparenthesis^{\mathrm{n}}$. We do not introduce an additional returner type (in contrast to the previous translations), because the side-effects occur at returner types that appear in $\llparenthesis \tau \rrparenthesis^{\mathrm{n}}$. Typing contexts Γ are translated into GCBPV typing contexts $\llparenthesis \Gamma \rrparenthesis^{\mathrm{n}}$ as before.

The definition of the translation from Levy-style call-by-name into GCBPV is given in Figure 2.17. Returner types are attached to **unit** and **bool** because these are the types at which side-effects occur for Levy-style call-by-name. As for Moggi-style call-by-name, functions take thunks of computations as arguments. Otherwise, they just use the GCBPV function types directly. The translation of types satisfies $\llparenthesis \langle\!\langle \varepsilon \rangle\!\rangle\tau \rrparenthesis^{\mathrm{n}} = \langle\!\langle \varepsilon \rangle\!\rangle \llparenthesis \tau \rrparenthesis^{\mathrm{n}}$. Contexts are translated using thunks. Most of the cases in the translation of typing derivations are similar to the Moggi-style call-by-name translation. The case for function application is simpler because the side-effects associated with computing the function are not thunked. Like the other two translations in this section, the translation from Levy-style call-by-name into GCBPV is compositional.

Each of the three translations in this section satisfies some useful properties. First, the terms that arise from translating typing derivations into graded call-by-push-value have the desired types.

Lemma 2.7.7

1. If M is the translation of a derivation of $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ in call-by-value then
$$(\!|\Gamma|\!)^{\mathrm{v}} \mathrel{\underline{\vdash}} M : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}$$
2. If M is the translation of a derivation of $\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$ in Moggi-style call-by-name then
$$(\!|\Gamma|\!)^{\mathrm{moggi}} \mathrel{\underline{\vdash}} M : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{moggi}}$$
3. If M is the translation of a derivation of $\Gamma \vdash_{\mathrm{n}} e : \tau$ in Levy-style call-by-name then
$$(\!|\Gamma|\!)^{\mathrm{n}} \mathrel{\underline{\vdash}} M : (\!|\tau|\!)^{\mathrm{n}}$$ ◀

Next, we note that any two typing derivations with the same conclusion in our source-language effect systems can differ only in where subeffecting is used. Hence by Lemma 2.7.5, for typical effect algebras, translations of derivations with the same conclusion are the same up to the GCBPV inequational theory.

Lemma 2.7.8 Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. If M and M' are call-by-value translations of derivations of $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ then $M \equiv M'$.
2. If M and M' are Moggi-style call-by-name translations of derivations of $\Gamma \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$ then $M \equiv M'$.
3. If M and M' are Levy-style call-by-name translations of derivations of $\Gamma \vdash_{\mathrm{n}} e : \tau$ then $M \equiv M'$. ◀

In particular, when considering GCBPV terms up to the (symmetric part of) the inequational theory, we can consider translations of well-typed expressions; the choice of derivation is irrelevant. (For example, for each Γ, τ and ε, the call-by-value translation is a function from expressions e such that $\Gamma \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ to equivalence classes of computation terms up to $\equiv$.) We use this in the statement of soundness for the three translations. Recall that $\diamond$ is the empty typing context. In the statement of soundness we omit the typing context, type and effect when writing translations of well-typed expressions.

Lemma 2.7.9 Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. If $\diamond \vdash_{\mathrm{v}} e : \tau \mathbin{\&} \varepsilon$ and $e \overset{\mathrm{v}}{\rightsquigarrow} e'$ then $(\!|e|\!)^{\mathrm{v}} \equiv (\!|e'|\!)^{\mathrm{v}}$.
2. If $\diamond \vdash_{\mathrm{moggi}} e : \tau \mathbin{\&} \varepsilon$ and $e \overset{\mathrm{moggi}}{\rightsquigarrow} e'$ then $(\!|e|\!)^{\mathrm{moggi}} \equiv (\!|e'|\!)^{\mathrm{moggi}}$.
3. If $\diamond \vdash_{\mathrm{n}} e : \tau$ and $e \overset{\mathrm{n}}{\rightsquigarrow} e'$ then $(\!|e|\!)^{\mathrm{n}} \equiv (\!|e'|\!)^{\mathrm{n}}$. ◀

2.8 Related work

Previous work on effect systems has focused on call-by-value languages. Historically these used Gifford-style effect algebras [63], which are well-known. The more general form of effect algebra we use here (preordered monoids) was first described by Katsumata [45]. Effect systems can be extended, for example by adding operations to the effect algebras to support more language constructs [80], or by adding support for regions [81]. The Moggi-style call-by-name effect presented here was first described by McDermott and Mycroft [69], but is somewhat similar to a system described by Abadi et al. [1]. The Levy-style effect system is new here. GMM is similar to languages described by Katsumata [45] and Gaboardi et al. [29].

A version of the call-by-push-value effect system we present here was first described by McDermott and Mycroft [70]. Kammar and Plotkin [42] describe a Gifford-style effect system for call-by-push-value, called MAIL (multi-adjunctive intermediate language). Unlike GCBPV, in MAIL thunk types, returner types, and the typing judgment are all annotated with effects. MAIL does not generalize to other effect algebras: if the effect algebra is allowed to be an arbitrary preordered monoid then it is unsound, and some computations that are typable in GCBPV are not typable in MAIL. It also overapproximates effects more than ours, even for Gifford-style. Hence our effect system has several advantages, though in many cases Kammar and Plotkin's suffices.

Our translations into GMM and GCBPV are partly based on translations given by Moggi [78], Wadler [98] and Levy [56, 59]. We could have presented these translations differently. For example, the translation of Moggi-style call-by-name into GMM can be presented by first translating into call-by-value [35, 34]. We choose to translate directly from source languages into GMM and GCBPV for simplicity. We could also have considered languages other than GMM or GCBPV, such as Filinski's [24].

2.9 Summary

One of the themes of this thesis is reasoning about programs using knowledge of *local* restrictions on side-effects. That is, by supposing that parts of programs are restricted to particular side-effects, rather than every part of every program. A standard method for doing this is to use effect systems. We take this approach here. This chapter provides the effect systems we use to do effect-dependent reasoning.

In later chapters, we reason about different evaluation orders for the source language, in the presence of side-effects. The method we use relies on having an intermediate language together with translations, one for each evaluation order of interest, from the source language. This chapter supplies these. The intermediate language we use is graded call-by-push-value (Section 2.7), and we have effect systems for call-by-value (Section 2.3), Moggi-style call-by-name (Section 2.4) and Levy-style call-by-name (Section 2.6), each with a translation (Section 2.7.3) into GCBPV. We are therefore in a position to consider equivalences between evaluation orders in the next chapter.

Chapter 3

Call-by-value and call-by-name

One of the goals of this thesis is to relate evaluation orders. Specifically, suppose we have some program M (for example in GCBPV) that uses a mix of several evaluation orders and we construct a new program M' by replacing one evaluation order with another for some subterm. We want to reason formally about the relationship between M and M'. These relationships generally depend on restrictions on side-effects. For example, for replacing call-by-value with call-by-name:

- If there are no side-effects at all (in particular, all programs are strongly normalizing), the choice of call-by-value or call-by-name does not affect the semantics of the program: M and M' terminate with the same result.
- If there are diverging terms (for instance, via recursion), then the behaviour may change: a program might diverge under call-by-value and return a result under call-by-name. However, we can say something about how the behaviour changes: if M terminates with some result, then M' terminates with the same result.
- If nondeterminism is the only side-effect, every possible result of M is a possible result of M'.

These properties are intuitively obvious, and can be proved for specific examples by reasoning at the meta-level.

In this chapter, we show how to relate call-by-value (Section 2.3) and Levy-style call-by-name (Section 2.6). We obtain a reasoning principle (Theorem 3.4.2) for relating these two evaluation orders. This reasoning principle can be applied to various effects. As an example, we show that, in GCBPV instantiated with global state, call-by-value and Levy-style call-by-name have the same observable behaviour for subterms restricted to no side-effects. (Of course, is not the case if we drop the restriction.)

Rather than reasoning at the meta-level, we use a common intermediate language (here GCBPV) that captures both evaluation orders via translations from source-language effect systems. We relate the observable behaviours of the call-by-value translation $(\!|e|\!)^{\mathrm{v}}$ and the Levy-style call-by-name translation $(\!|e|\!)^{\mathrm{n}}$ of each source-language expression e (the two translations are the subterms we refer to above). Intuitively, we want to show $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} (\!|e|\!)^{\mathrm{n}}$ (recall that $\preccurlyeq_{\mathrm{ctx}}$ is the contextual preorder defined in Definition 2.7.6). However, there are two subtleties we need to deal with.

The first is that the two translations have different types, and hence we cannot relate them directly using the contextual preorder. The way around this is inspired by work relating direct and continuation interpretations of languages [89]: we identify maps between the call-by-value and call-by-name interpretations[1], and then compose these with the translations of expressions to arrive at two terms that can be compared directly. We show that, under certain conditions on the side-effects, the maps between call-by-value and call-by-name form a *Galois connection*, and this fact allows us to derive our reasoning principle.

The second subtlety arises when we consider effect systems, which we do so that we only need to restrict subterms to particular side-effects, rather than the entire source language. We have various effect systems, each for a different evaluation order, and these assign different effects to each source language expression e. We deal with this by only considering effect algebras in which, if e is typable without tracking effects (i.e. our first source-language type system, defined in Section 2.2), then it is typable under both the call-by-value and call-by-name effect systems. In this chapter, we consider only Gifford-style effect algebras, for which this holds.

The technique we propose is intended to be *general*, in that it should work for other pairs of evaluation orders, such as call-by-need and call-by-name (even though we only concentrate on one pair here). We also work abstractly and identify properties of side-effects that enable us to relate call-by-value and call-by-name, rather than just considering some fixed collection of side-effects. An advantage of the technique of using Galois connections is that the properties required are derived from the structure of the two maps between evaluation orders.

To derive our reasoning principle, we need a method of proving instances of the GCBPV contextual preorder. We use *logical relations* to do this. We therefore discuss logical relations for graded call-by-push-value in this chapter before considering call-by-value and call-by-name.

This chapter has two main contributions:

- We describe a notion of logical relation for graded call-by-push-value (Section 3.1), based on previous work on logical relations for other languages. We also provide a method of constructing such a logical relation, called the *free lifting* (Section 3.1.1).
- We derive a reasoning principle for relating call-by-value and Levy-style call-by-name (Section 3.4). This is formulated in terms of a Galois connection between call-by-value and call-by-name computations (Section 3.3).

In this chapter, we work exclusively with syntax. In particular, we do not use denotational semantics. Chapter 4 shows (amongst other things) how to derive a similar reasoning principle using denotational semantics, and argues that the denotational approach has several advantages over the approach we use in this chapter. However, Chapter 4 requires significantly more machinery than this one.

3.1 Logical relations for graded call-by-push-value

To prove instances of GCBPV contextual preorders, we use *logical relations*. We consider binary logical relations on GCBPV terms in general, before considering evaluation orders in later sections of this chapter.

Recall that an inequational theory for GCBPV consists of a signature (Definition 2.7.1) and two judgments

$$\Gamma \vdash V \preccurlyeq W : A \qquad \Gamma \underline{\vdash} M \preccurlyeq N : \underline{C}$$

satisfying certain properties (see Definition 2.7.4). Each inequational theory induces a contextual preorder $\preccurlyeq_{\mathrm{ctx}}$ (Definition 2.7.6). As usual, we write

$$\Gamma \vdash V \equiv W : A \qquad \Gamma \underline{\vdash} M \equiv N : \underline{C}$$

[1] Specifically, these map between computations of type $\langle\varepsilon\rangle(\!|\tau^{\mathrm{v}}_{\varepsilon}|\!)^{\mathrm{v}}$ and computations of type $(\!|\tau^{\mathrm{n}}_{\varepsilon}|\!)^{\mathrm{n}}$, where $\tau^{\mathrm{v}}_{\varepsilon}$ and $\tau^{\mathrm{n}}_{\varepsilon}$ are source-language types annotated with the effect ε.

$$
\begin{aligned}
\mathcal{R}\llbracket \mathbf{unit} \rrbracket &= \mathrm{Term}_{\mathbf{unit}} \times \mathrm{Term}_{\mathbf{unit}} \\
\mathcal{R}\llbracket A_1 \times A_2 \rrbracket &= \{(V, V') \mid (\mathbf{fst}\, V, \mathbf{fst}\, V') \in \mathcal{R}\llbracket A_1 \rrbracket \wedge (\mathbf{snd}\, V, \mathbf{snd}\, V') \in \mathcal{R}\llbracket A_2 \rrbracket\} \\
\mathcal{R}\llbracket \mathbf{empty} \rrbracket &= \emptyset \\
\mathcal{R}\llbracket A_1 + A_2 \rrbracket &= \{(\mathbf{inl}_{A_2} V, \mathbf{inl}_{A_2} V') \mid (V, V') \in \mathcal{R}\llbracket A_1 \rrbracket\} \cup \{(\mathbf{inr}_{A_1} V, \mathbf{inr}_{A_1} V') \mid (V, V') \in \mathcal{R}\llbracket A_2 \rrbracket\} \\
\mathcal{R}\llbracket \mathrm{U}\,\underline{C} \rrbracket &= \{(V, V') \mid (\mathbf{force}\, V, \mathbf{force}\, V') \in \mathcal{R}\llbracket \underline{C} \rrbracket\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}\llbracket \underline{\mathbf{unit}} \rrbracket &= \underline{\mathrm{Term}_{\mathbf{unit}}} \times \underline{\mathrm{Term}_{\mathbf{unit}}} \\
\mathcal{R}\llbracket \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 \rrbracket &= \{(M, M') \mid (1‘M, 1‘M') \in \mathcal{R}\llbracket \underline{C}_1 \rrbracket \wedge (2‘M, 2‘M') \in \mathcal{R}\llbracket \underline{C}_2 \rrbracket\} \\
\mathcal{R}\llbracket A \to \underline{C} \rrbracket &= \{(M, M') \mid \forall (V, V') \in \mathcal{R}\llbracket A \rrbracket.\, (V‘M, V'‘M') \in \mathcal{R}\llbracket \underline{C} \rrbracket\}
\end{aligned}
$$

Figure 3.1: Logical relations for graded call-by-push-value

for the symmetric part of the equational theory (i.e. $V \equiv W$ if $V \preccurlyeq W$ and $W \preccurlyeq V$). For each value type A we define Term_A as the set of $\equiv$-equivalence classes of closed terms of type A, and similarly define $\underline{\mathrm{Term}_{\underline{C}}}$ for computation types:

$$
\mathrm{Term}_A := \{[V]_\equiv \mid \diamond \vdash V : A\} \qquad \underline{\mathrm{Term}_{\underline{C}}} := \{[M]_\equiv \mid \diamond \mathrel{\underline{\vdash}} M : \underline{C}\}
$$

Throughout this chapter, we consider terms up to $\equiv$. All of the definitions we use are invariant under $\equiv$ (including for example the contextual preorder $\preccurlyeq_{\mathrm{ctx}}$). We omit the square brackets when writing equivalence classes.

In GCBPV, a logical relation consists of two families of types

$$
A \;\mapsto\; \mathcal{R}\llbracket A \rrbracket \subseteq \mathrm{Term}_A \times \mathrm{Term}_A \qquad\qquad \underline{C} \;\mapsto\; \mathcal{R}\llbracket \underline{C} \rrbracket \subseteq \underline{\mathrm{Term}_{\underline{C}}} \times \underline{\mathrm{Term}_{\underline{C}}}
$$

indexed by value types A and by computation types $\underline{C}$. These are usually defined by induction on the structure of the types. They are *logical* in the sense that they respect the equations in Figure 3.1. These equations uniquely determine the logical relation on every type former, except for base types and returner types. For ground types these are standard. Note that for **unit** the set $\mathrm{Term}_{\mathbf{unit}}$ contains only one equivalence class. Since the only way to use a thunk is to force it, the definition on thunk types just requires the two forced computations to be related. For products of computation types the definition is similar to products of value types: we require that each of the projections are related. For function types, we require that related arguments are sent to related results.

The only type formers that are omitted from the figure are base types and returner types. For these, the logical relation must be chosen based on the constants and side-effects included in the language. In particular, the definition of the logical relation on returner types depends on the choice of operations $\mathrm{op} \in \Sigma$. In the definition of logical relations, we impose some additional requirements on these:

Definition 3.1.1 (Logical relation) A *logical relation* consists of a relation $\mathcal{R}\llbracket A \rrbracket \subseteq \mathrm{Term}_A \times \mathrm{Term}_A$ for each value type A and a relation $\mathcal{R}\llbracket \underline{C} \rrbracket \subseteq \underline{\mathrm{Term}_{\underline{C}}} \times \underline{\mathrm{Term}_{\underline{C}}}$ for each computation type $\underline{C}$, such that:

- The relations satisfy the equations in Figure 3.1.
- Closure under operations: for each operation $\mathrm{op} \in \Sigma$, if $(V, V') \in \mathcal{R}\llbracket \mathrm{car}_{\mathrm{op}} \rrbracket$ then $(\mathrm{op}\, V, \mathrm{op}\, V') \in \mathcal{R}\llbracket \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}} \rrbracket$.

- Closure under pure computations: if $(V, V') \in \mathcal{R}[\![A]\!]$ then $(\langle V\rangle, \langle V'\rangle) \in \mathcal{R}[\![\langle 1\rangle A]\!]$.
- Closure under **to**: if $(M, M') \in \mathcal{R}[\![\langle\varepsilon\rangle A]\!]$, and $x : A \vdash N : \langle\varepsilon'\rangle B$ and $x : A \vdash N' : \langle\varepsilon'\rangle B$ satisfy
$$\forall (V, V') \in \mathcal{R}[\![A]\!].\, (N[x \mapsto V], N'[x \mapsto V']) \in \mathcal{R}[\![\langle\varepsilon'\rangle B]\!]$$
then
$$(M \mathbf{\ to\ } x.\, N, M' \mathbf{\ to\ } x.\, N') \in \mathcal{R}[\![\langle\varepsilon\cdot\varepsilon'\rangle B]\!]$$
- Closure under **coerce**: if $(M, M') \in \mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ and $\varepsilon \leq \varepsilon'$ then
$$(\mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, M, \mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, M') \in \mathcal{R}[\![\langle\varepsilon'\rangle A]\!]$$
- For each constant $c \in \mathcal{K}_A$ we have $(c, c) \in \mathcal{R}[\![A]\!]$. ◀

In the definition, we require closure under **to** only when N and N' have returner type. This suffices to get closure under **to** for arbitrary computation types $\underline{C}$, because, as we noted in Section 2.7.2, having **to** only for returner types on the right-hand side suffices.

Lemma 3.1.2 Suppose that $\mathcal{R}[\![-]\!]$ is a logical relation. If $(M, M') \in \mathcal{R}[\![\langle\varepsilon\rangle A]\!]$, and $x : A \vdash N : \underline{C}$ and $x : A \vdash N' : \underline{C}$ are well-typed computations such that
$$\forall (V, V') \in \mathcal{R}[\![A]\!].\, (N[x \mapsto V], N'[x \mapsto V']) \in \mathcal{R}[\![\underline{C}]\!]$$
then
$$(M \mathbf{\ to\ } x.\, N, M' \mathbf{\ to\ } x.\, N') \in \mathcal{R}[\![\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}]\!]$$

Proof. By induction on the structure of $\underline{C}$. For $\underline{\mathbf{unit}}$ this holds by the equation for $\mathcal{R}[\![\underline{\mathbf{unit}}]\!]$. For $\underline{C}_1 \underline{\times} \underline{C}_2$, it suffices to show that
$$(i`(M \mathbf{\ to\ } x.\, N),\ i`(M' \mathbf{\ to\ } x.\, N')) \in \mathcal{R}[\![\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}_i]\!]$$
for each $i \in \{1, 2\}$. We have
$$\begin{aligned} i`(M \mathbf{\ to\ } x.\, N) &\equiv i`(\lambda\{1.\, M \mathbf{\ to\ } x.\, 1`N,\ 2.\, M \mathbf{\ to\ } x.\, 2`N\}) \\ &\equiv M \mathbf{\ to\ } x.\, i`N \end{aligned}$$
and similarly for $M' \mathbf{\ to\ } x.\, N'$. The result then follows from the inductive hypothesis, using the fact that
$$\forall (V, V') \in \mathcal{R}[\![A]\!].\, ((i`N)[x \mapsto V],\ (i`N')[x \mapsto V']) \in \mathcal{R}[\![\underline{C}_i]\!]$$
For $B \rightarrow \underline{C}$ we show that
$$(W`(M \mathbf{\ to\ } x.\, N),\ W'`(M' \mathbf{\ to\ } x.\, N')) \in \mathcal{R}[\![\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}]\!]$$
for each $(W, W') \in \mathcal{R}[\![B]\!]$. This is the case because
$$\begin{aligned} W`(M \mathbf{\ to\ } x.\, N) &\equiv W`(\lambda y{:}A.\, M \mathbf{\ to\ } x.\, y`N) \\ &\equiv M \mathbf{\ to\ } x.\, W`N \end{aligned}$$
and similarly for $M' \mathbf{\ to\ } x.\, N'$, so we can apply the inductive hypothesis using
$$\forall (V, V') \in \mathcal{R}[\![A]\!].\, ((W`N)[x \mapsto V],\ (W'`N')[x \mapsto V']) \in \mathcal{R}[\![\underline{C}]\!]$$
Finally, for returner types we have closure under **to** by assumption. □

$$
\begin{aligned}
\mathcal{R}[\![\langle\emptyset\rangle A]\!] &:= \{(\langle V\rangle, \langle V'\rangle) \mid (V, V') \in \mathcal{R}[\![A]\!]\} \\
\mathcal{R}[\![\langle\{\mathrm{get}\}\rangle A]\!] &:= \left\{\begin{pmatrix}\mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \langle W_1\rangle\ \mathbf{else}\ \langle W_2\rangle, \\ \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \langle W_1'\rangle\ \mathbf{else}\ \langle W_2'\rangle\end{pmatrix} \,\middle|\, (W_1, W_1'), (W_2, W_2') \in \mathcal{R}[\![A]\!]\right\} \\
\mathcal{R}[\![\langle\{\mathrm{put}\}\rangle A]\!] &:= \{(\mathbf{coerce}_{\emptyset\leq\{\mathrm{put}\}}\langle W\rangle, \mathbf{coerce}_{\emptyset\leq\{\mathrm{put}\}}\langle W'\rangle) \mid (W, W') \in \mathcal{R}[\![A]\!]\} \cup \mathrm{MustPut}\,A \\
\mathcal{R}[\![\langle\{\mathrm{get},\mathrm{put}\}\rangle A]\!] &:= \left\{\begin{pmatrix}\mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2, \\ \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ M_1'\ \mathbf{else}\ M_2'\end{pmatrix} \,\middle|\, (M_1, M_1'), (M_2, M_2') \in \mathrm{MustPut}\,A\right\}
\end{aligned}
$$

where $(\mathrm{MustPut}\,A) \subseteq \underline{\mathrm{Term}}_{\langle\{\mathrm{put}\}\rangle A} \times \underline{\mathrm{Term}}_{\langle\{\mathrm{put}\}\rangle A}$ is given by

$$\mathrm{MustPut}\,A := \left\{((\mathrm{put}\,V;\ \langle W\rangle), (\mathrm{put}\,V;\ \langle W'\rangle)) \,\middle|\, V \in \{\mathbf{true}, \mathbf{false}\} \wedge (W, W') \in \mathcal{R}[\![A]\!]\right\}$$

Figure 3.2: Logical relation for global state

For many concrete examples, it is difficult to show that closure under **to** holds for all computation types directly. We instead show that it holds for returner types.

Example 3.1.3 As an example, consider global state. We have no base types and a Gifford-style effect algebra with two operations: get with coarity **unit** and arity **bool** = **unit** + **unit**, and put with coarity **bool** and arity **unit**. We gave a (symmetric) inequational theory for this example in Figure 2.14. To define a logical relation, we only need to give $\mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ in terms of $\mathcal{R}[\![A]\!]$ for each $\varepsilon \subseteq \{\mathrm{get}, \mathrm{put}\}$. We do this in Figure 3.2. The idea behind the definition is to think about how computations with each effect could behave. For example, a computation with effect $\{\mathrm{put}\}$ can either just return a result (without setting the value of the state), or set the state once and then return. Computations that use put more than once are equivalent to computations that use put exactly once because $\mathrm{put}\,V_1;\ \mathrm{put}\,V_2 \equiv \mathrm{put}\,V_2$ (lack of recursion implies that computations are finite).

The proof that the requirements in the definition of logical relation are met in this case uses the inequational theory. In particular, to show that the logical relations are closed under operations, we need to show that $(\mathrm{get}\,(), \mathrm{get}\,()) \in \mathcal{R}[\![\langle\{\mathrm{get}\}\rangle\mathbf{bool}]\!]$ and if $V \in \{\mathbf{true}, \mathbf{false}\}$ then $(\mathrm{put}\,V, \mathrm{put}\,V) \in \mathcal{R}[\![\langle\{\mathrm{put}\}\rangle\mathbf{unit}]\!]$. These follow from the η-laws:

$$
\begin{aligned}
\mathrm{get}\,() &\equiv \mathrm{get}\,()\ \mathbf{to}\ b.\, \langle b\rangle \equiv \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \langle\mathbf{true}\rangle\ \mathbf{else}\ \langle\mathbf{false}\rangle \\
\mathrm{put}\,V &\equiv \mathrm{put}\,V\ \mathbf{to}\ x.\, \langle x\rangle \equiv \mathrm{put}\,V\ \mathbf{to}\ x.\, \langle()\rangle \equiv \mathrm{put}\,V;\ \langle()\rangle
\end{aligned}
$$

So the terms have the correct form to be related by the logical relation. The signature axioms for global state come into play when showing that the logical relation is closed under pure computations and under **to**. To show closure under **to** for $\varepsilon = \varepsilon' = \{\mathrm{get}\}$, we use the axiom that merges adjacent uses of get:

$$
\begin{aligned}
&(\mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \langle V_1\rangle\ \mathbf{else}\ \langle V_2\rangle)\ \mathbf{to}\ x.\, \mathrm{get}\,()\ \mathbf{to}\ b'.\, \mathbf{if}\ b'\ \mathbf{then}\ \langle W_1\rangle\ \mathbf{else}\ \langle W_2\rangle \\
\equiv\ & \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathrm{get}\,()\ \mathbf{to}\ b'.\, \mathbf{if}\ b\ \mathbf{then}\ \mathbf{if}\ b'\ \mathbf{then}\ \langle W_1[x \mapsto V_1]\rangle\ \mathbf{else}\ \langle W_2[x \mapsto V_1]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{if}\ b'\ \mathbf{then}\ \langle W_1[x \mapsto V_2]\rangle\ \mathbf{else}\ \langle W_2[x \mapsto V_2]\rangle \\
\equiv\ & \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \mathbf{if}\ b\ \mathbf{then}\ \langle W_1[x \mapsto V_1]\rangle\ \mathbf{else}\ \langle W_2[x \mapsto V_1]\rangle \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{if}\ b\ \mathbf{then}\ \langle W_1[x \mapsto V_2]\rangle\ \mathbf{else}\ \langle W_2[x \mapsto V_2]\rangle \\
\equiv\ & \mathrm{get}\,()\ \mathbf{to}\ b.\, \mathbf{if}\ b\ \mathbf{then}\ \langle W_1[x \mapsto V_1]\rangle\ \mathbf{else}\ \langle W_2[x \mapsto V_2]\rangle
\end{aligned}
$$

We return to our global state example in Section 3.1.1. ◀

We extend logical relations to typing contexts Γ, for which we relate closed substitutions. Recall from Section 2.7 that substitutions σ satisfying $\diamond \vdash \sigma : \Gamma$ map variables $(x : A) \in \Gamma$ to closed terms of type A. Also recall that we extend the inequational theory to substitutions componentwise. The set of equivalence classes of closed substitutions is

$$\mathrm{Subst}_\Gamma := \{[\sigma]_\equiv \mid \diamond \vdash \sigma : \Gamma\}$$

In this chapter we identify substitutions that are related by $\equiv$, and omit the square brackets on equivalence classes.

We extend logical relations $\mathcal{R}[\![-]\!]$ to typing contexts Γ by defining a relation $\mathcal{R}[\![\Gamma]\!] \subseteq \mathrm{Subst}_\Gamma \times \mathrm{Subst}_\Gamma$ on closed substitutions componentwise:

$$\mathcal{R}[\![\diamond]\!] := \{(\diamond, \diamond)\} \quad \mathcal{R}[\![\Gamma, x : A]\!] := \{((\sigma, x \mapsto V), (\sigma', x \mapsto V')) \mid (\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!] \wedge (V, V') \in \mathcal{R}[\![A]\!]\}$$

The key property that follows from the requirements in the definition of logical relation is the *fundamental lemma.*

Lemma 3.1.4 (Fundamental) Suppose that $\mathcal{R}[\![-]\!]$ is a logical relation and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]$.

1. (Values) If $\Gamma \vdash V : A$ then $(V[\sigma], V[\sigma']) \in \mathcal{R}[\![A]\!]$.
2. (Computations) If $\Gamma \underline{\vdash} M : \underline{C}$ then $(M[\sigma], M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]$.

Proof sketch. By induction on the derivations of $\Gamma \vdash V : A$ and $\Gamma \underline{\vdash} M : \underline{C}$. Most of the cases are standard, so we do not give them. For $\mathrm{op}\, V$ and $\langle V \rangle$ we use the assumptions that the logical relation is closed under operations and pure computations. For M **to** $x.\, N$ we use closure under **to** for arbitrary computation types (Lemma 3.1.2): by the inductive hypothesis we have $(M[\sigma], M[\sigma']) \in \mathcal{R}[\![\langle\varepsilon\rangle A]\!]$, and

$$\forall (V, V') \in \mathcal{R}[\![A]\!].\, (N[\sigma, x \mapsto V],\ N[\sigma', x \mapsto V'])$$

Hence

$$((M[\sigma]\ \mathbf{to}\ x.\, N[\sigma]),\ (M[\sigma']\ \mathbf{to}\ x.\, N'[\sigma'])) \in \mathcal{R}[\![\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}]\!]$$

For $\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M$ we use closure under **coerce**. For constants we use the assumption about constants. □

A consequence of the fundamental lemma is that each relation is reflexive (using the empty context for Γ and the empty substitution for σ and σ'). Hence if $M \equiv M'$ for closed, well-typed terms $M, M' : \underline{C}$, then $(M, M') \in \mathcal{R}[\![\underline{C}]\!]$ (recall that we relate $\equiv$-equivalence classes of terms). In general the converse $(M, M') \in \mathcal{R}[\![\underline{C}]\!] \Rightarrow M \equiv M'$ does not hold. Nor does $M \preccurlyeq M' \Rightarrow (M, M') \in \mathcal{R}[\![\underline{C}]\!]$.

In the fundamental lemma the logical relation is applied to open terms by quantifying over pairs of related substitutions. We do this frequently, and so define abbreviations for this notion. For values we write $(V, V') : \mathcal{R}[\![\Gamma]\!] \dot{\rightarrow} \mathcal{R}[\![A]\!]$ if $\Gamma \vdash V : A$ and $\Gamma \vdash V' : A$, and $(V[\sigma], V'[\sigma']) \in \mathcal{R}[\![A]\!]$ for all $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]$. Similarly, for computations we write $(M, M') : \mathcal{R}[\![\Gamma]\!] \dot{\rightarrow} \mathcal{R}[\![\underline{C}]\!]$ if $\Gamma \underline{\vdash} M : \underline{C}$ and $\Gamma \underline{\vdash} M' : \underline{C}$, and $(M[\sigma], M'[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]$ for all $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]$.

Recall that the contextual preorder $\preccurlyeq_{\mathrm{ctx}}$ is defined by considering closed term-contexts of type $\langle\varepsilon\rangle G$, where G is a ground type. In this chapter, we use logical relations to prove instances of $\preccurlyeq_{\mathrm{ctx}}$. We show that we can do this for logical relations that satisfy an additional assumption at the types $\langle\varepsilon\rangle G$. (This assumption is satisfied for our global state example.)

Lemma 3.1.5 Suppose that $\mathcal{R}[\![-]\!]$ is a logical relation, and that for each ground type G, effect ε, and pair of computations N, N' we have:

$$(N, N') \in \mathcal{R}[\![\langle\varepsilon\rangle G]\!] \quad \Rightarrow \quad N \preccurlyeq N'$$

1. If $(V, V') : \mathcal{R}[\![\Gamma]\!] \dot{\to} \mathcal{R}[\![A]\!]$ then $\Gamma \vdash V \preccurlyeq_{\mathrm{ctx}} V' : A$.
2. If $(M, M') : \mathcal{R}[\![\Gamma]\!] \dot{\to} \mathcal{R}[\![\underline{C}]\!]$ then $\Gamma \,\underline{\vdash}\, M \preccurlyeq_{\mathrm{ctx}} M' : \underline{C}$.

Proof sketch. For (1) we show for all typing contexts Γ', types B and term contexts $C[\,]$ with hole $\square$ that if $\Gamma' \vdash C[V] : B$ and $\Gamma' \vdash C[V'] : B$ then $(C[V], C[V']) : \mathcal{R}[\![\Gamma']\!] \dot{\to} \mathcal{R}[\![B]\!]$, and similarly for computation-typed term contexts $\underline{C}[\,]$. These are by induction on the typing derivations, and are similar to the proof of the fundamental lemma. For $C[\,] = \square$ we use the assumption about (V, V').

Then given a ground type G, effect ε and term context $\underline{C}[\,]$ with hole $\square$ such that $\diamond \,\underline{\vdash}\, \underline{C}[V] : \langle\varepsilon\rangle G$ and $\diamond \,\underline{\vdash}\, \underline{C}[V'] : \langle\varepsilon\rangle G$, we have $(C[V], C[V']) \in \mathcal{R}[\![\langle\varepsilon\rangle G]\!]$ (using the above with the empty context for Γ'). By the assumption about ground types, it follows that $\underline{C}[V] \preccurlyeq \underline{C}[V']$.

The proof of (2) is similar. □

3.1.1 Free lifting

Logical relations must be compatible with the side-effects and effect system that GCBPV is instantiated with. The main difficulty in constructing a logical relation is *lifting* a relation $\mathcal{R}[\![A]\!]$ for the value type A, to form relations $\mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ on computations for each effect ε. There are several existing techniques for lifting relations to computations, such as ⊤⊤-lifting [62, 43] and codensity lifting [46]. Here we give a further lifting technique, called the *free lifting*. This technique is folklore, and is described for category-theoretic monads by Kammar and McDermott [41]. The free lifting has the important property that it is *initial*: it relates only the computations that must be related in order to form a logical relation (see Lemma 3.1.7). This does not mean it is the best possible lifting for all applications (a logical relation can relate too few terms), but it is for some (e.g. Example 3.4.5). The example logical relation we give above (for global state) is an instance of the free lifting.

The insight used to define the free lifting is that every closed computation of returner type, when evaluated, will execute a sequence of operations (get, put, raise, etc.), before possibly returning a result. This means that, up to $\equiv$, every closed computation has one of three forms: it is either a return $\langle V\rangle$, a coercion $\mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, M$ where M is a closed computation of returner type, or a sequencing $\mathrm{op}\, V$ **to** $x.\, M$ of an operation op followed by a closed computation M. The free lifting is constructed as the smallest relation that relates computations of all three forms.

Given a relation $\mathcal{R}_{\mathrm{free}}[\![b]\!] \in \mathrm{Term}_b \times \mathrm{Term}_b$ for each base type b, we form a logical relation $\mathcal{R}_{\mathrm{free}}[\![-]\!]$, by defining $\mathcal{R}_{\mathrm{free}}[\![A]\!]$ on non-base types A by induction on the structure of A. We use the equations in Figure 3.1 as the definition on the rest of the type formers except returner types. It remains to provide, given a type A for which the logical relation is defined, a family of relations

$$\mathcal{R}_{\mathrm{free}}[\![\langle\varepsilon\rangle A]\!] \subseteq \underline{\mathrm{Term}}_{\langle\varepsilon\rangle A} \times \underline{\mathrm{Term}}_{\langle\varepsilon\rangle A}$$

indexed by $\varepsilon \in \mathcal{E}$. We define this family inductively by the following rules:

- If $(V, V') \in \mathcal{R}_{\mathrm{free}}[\![A]\!]$ then $(\langle V\rangle, \langle V'\rangle) \in \mathcal{R}_{\mathrm{free}}[\![\langle 1\rangle A]\!]$.
- If $(M, M') \in \mathcal{R}_{\mathrm{free}}[\![\langle\varepsilon\rangle A]\!]$ and $\varepsilon \leq \varepsilon'$ then $(\mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, M, \mathbf{coerce}_{\varepsilon\leq\varepsilon'}\, M') \in \mathcal{R}_{\mathrm{free}}[\![\langle\varepsilon'\rangle A]\!]$.

- For each op $\in \Sigma$, if $(M, M') : \mathcal{R}_{\text{free}}[\![x : \text{ar}_{\text{op}}]\!] \dot{\rightarrow} \mathcal{R}_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$ and $(V, V') \in \mathcal{R}_{\text{free}}[\![\text{car}_{\text{op}}]\!]$ then $$((\text{op}\, V \textbf{ to } x.\, M), (\text{op}\, V' \textbf{ to } x.\, M')) \in \mathcal{R}_{\text{free}}[\![\langle \text{eff}_{\text{op}} \cdot \varepsilon\rangle A]\!]$$

In this definition we use the logical relation on the arity and coarity of operations. The arity and coarity are ground types (in particular do not contain returner types), so the definition is well-founded.

Lemma 3.1.6 If $(c, c) \in \mathcal{R}_{\text{free}}[\![A]\!]$ for each constant $c \in \mathcal{K}_A$, then $\mathcal{R}_{\text{free}}[\![-]\!]$ is a logical relation.

Proof. See Appendix B.2. □

Hence the free lifting does give us a way of constructing logical relations. Logical relations constructed in this way are *initial* in the following sense:

Lemma 3.1.7 Suppose that $\mathcal{R}[\![-]\!]$ is a logical relation and $\mathcal{R}_{\text{free}}[\![b]\!] = \mathcal{R}[\![b]\!]$ for all base types b. Then $\mathcal{R}_{\text{free}}[\![A]\!] \subseteq \mathcal{R}[\![A]\!]$ implies $\mathcal{R}_{\text{free}}[\![\langle\varepsilon\rangle A]\!] \subseteq \mathcal{R}[\![\langle\varepsilon\rangle A]\!]$.

Proof. See Appendix B.2. □

This lemma does not imply that $\mathcal{R}_{\text{free}}[\![-]\!]$ is included in $\mathcal{R}[\![-]\!]$ at *every* type, because of the contravariance of the logical relation at function types. However, it does imply that $\mathcal{R}_{\text{free}}[\![\langle\varepsilon\rangle G]\!] \subseteq \mathcal{R}[\![\langle\varepsilon\rangle G]\!]$ for ground types G. In particular, this means if any logical relation satisfies the assumptions in Lemma 3.1.5 (which we use to prove contextual equivalence), then a logical relation based on the free lifting does.

The definition of the free lifting slightly simplifies when we restrict to Gifford-style effect algebras. In particular, in the above definition of the free lifting we define the entire ε-indexed family of relations $\mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ by an induction. For Gifford-style effect algebras, we can instead define each relation $\mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ separately by induction.

Suppose that the effect algebra is the preordered monoid

$$(\mathcal{P}\Sigma, \subseteq, \cup, \emptyset)$$

so that each effect ε is just a set of operations, and that for each operation op $\in \Sigma$ we have $\text{eff}_{\text{op}} = \{\text{op}\}$. Given $\mathcal{R}_{\text{free}}[\![A]\!]$, we define relations $\mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$ (our alternative definition). We show that the alternative definition coincides with our general definition below.

For each effect $\varepsilon \subseteq \Sigma$, the relation $\mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!] \subseteq \underline{\text{Term}}_{\langle\varepsilon\rangle A} \times \underline{\text{Term}}_{\langle\varepsilon\rangle A}$ is defined inductively by the following rules:

- If $(V, V') \in \mathcal{R}_{\text{free}}[\![A]\!]$ then $(\textbf{coerce}_{\emptyset \leq \varepsilon} \langle V\rangle, \textbf{coerce}_{\emptyset \leq \varepsilon} \langle V'\rangle) \in \mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$.
- For each op $\in \varepsilon$, if $(M, M') : \mathcal{R}_{\text{free}}[\![x : \text{ar}_{\text{op}}]\!] \dot{\rightarrow} \mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$ and $(V, V') \in \mathcal{R}_{\text{free}}[\![\text{car}_{\text{op}}]\!]$ then $$((\text{op}\, V \textbf{ to } x.\, M), (\text{op}\, V' \textbf{ to } x.\, M')) \in \mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$$

Lemma 3.1.8 For each value type A and effect $\varepsilon \subseteq \Sigma$,

$$\mathcal{R}'_{\text{free}}[\![\langle\varepsilon\rangle A]\!] = \mathcal{R}_{\text{free}}[\![\langle\varepsilon\rangle A]\!]$$

Proof. See Appendix B.2. □

This alternative definition is closer to the one given by Kammar and McDermott [41].

3.2 Restricting side-effects in call-by-value and call-by-name

As we mentioned in the introduction to this chapter, there are two major difficulties in relating the call-by-value and call-by-name translations of expressions when restricting side-effects. The first is that the side-effects of expressions depend on the evaluation order. This is why different evaluation orders have different effect systems (each of which can be instantiated with the same effect algebra). In particular, we have one effect system for call-by-value (Section 2.3) and another for Levy-style call-by-name. This means we cannot in general use the same restriction on side-effects for *both* call-by-value and call-by-name.

For the rest of this chapter, we therefore restrict to Gifford-style effect algebras (Example 2.1.2). Hence we take the preordered monoid to be $(\mathcal{P}\Sigma, \subseteq, \cup, \emptyset)$, where Σ is the set of operations from the signature. The two key properties that Gifford-style effect algebras satisfy are that the unit $\emptyset$ is the least element and that the multiplication $\cup$ is idempotent. Hence it does not matter that call-by-name might discard or duplicate side-effects; the call-by-value and call-by-name effect systems assign the same effects.

To impose restrictions on side-effects for both call-by-value and call-by-name we do the following. Recall that the syntax of source-language types τ does not have any effect annotations. For call-by-value and Levy-style call-by-name, we use types that do have effect annotations. Given any source-language type τ and effect ε, we write $\tau^{\mathrm{v}}_{\varepsilon}$ and $\tau^{\mathrm{n}}_{\varepsilon}$ respectively for the call-by-value and call-by-name types constructed by annotating τ with ε everywhere:

$$\begin{array}{llll} \boxed{\tau^{\mathrm{n}}_{\varepsilon}} & \mathbf{unit}^{\mathrm{n}}_{\varepsilon} := \langle\varepsilon\rangle\mathbf{unit} & \mathbf{bool}^{\mathrm{n}}_{\varepsilon} := \langle\varepsilon\rangle\mathbf{bool} & (\tau \to \tau')^{\mathrm{n}}_{\varepsilon} := \tau^{\mathrm{n}}_{\varepsilon} \to \tau'^{\mathrm{n}}_{\varepsilon} \\ \boxed{\tau^{\mathrm{v}}_{\varepsilon}} & \mathbf{unit}^{\mathrm{v}}_{\varepsilon} := \mathbf{unit} & \mathbf{bool}^{\mathrm{v}}_{\varepsilon} := \mathbf{bool} & (\tau \to \tau')^{\mathrm{v}}_{\varepsilon} := \tau^{\mathrm{v}}_{\varepsilon} \xrightarrow{\varepsilon} \tau'^{\mathrm{v}}_{\varepsilon} \end{array}$$

We annotate source-language typing contexts Γ pointwise to get $\Gamma^{\mathrm{v}}_{\varepsilon}$ for call-by-value and $\Gamma^{\mathrm{n}}_{\varepsilon}$ for call-by-name. (Since expressions contain types we similarly have expressions $e^{\mathrm{v}}_{\varepsilon}$ and $e^{\mathrm{n}}_{\varepsilon}$, although we write both of these just as e to avoid cluttering the notation. It can always be determined which we mean from context.) Define $\mathit{ops}\, e \subseteq \Sigma$ to be the set of operations that appear syntactically in the (unannotated) expression e. If we annotate types with an effect $\varepsilon \supseteq \mathit{ops}\, e$, then e is well-typed in both the call-by-value and call-by-name effect systems:

Lemma 3.2.1 If $\Gamma \vdash e : \tau$ and $\mathit{ops}\, e \subseteq \varepsilon$ then $\Gamma^{\mathrm{v}}_{\varepsilon} \vdash_{\mathrm{v}} e : \tau^{\mathrm{v}}_{\varepsilon} \;\&\; \varepsilon$ and $\Gamma^{\mathrm{n}}_{\varepsilon} \vdash_{\mathrm{n}} e : \tau^{\mathrm{n}}_{\varepsilon}$. ◀

Hence we can consider expressions restricted to ε in both evaluation orders simultaneously. Even though we translate derivations of expressions in this lemma, it does not matter which derivations we choose. This is because we identify GCBPV terms up to $\equiv$ throughout this chapter, and can apply Lemma 2.7.8.

This way of restricting side-effects of source-language expressions, where we just annotate everything with the same effect, is slightly ad hoc. There may be better ways of restricting side-effects across evaluation orders. In particular, effect polymorphism may enable a more principled way of restricting side-effects.

3.3 A Galois connection between call-by-value and call-by-name

The second problem with relating the call-by-value and call-by-name translations of expressions is that they have different types. Given an unannotated source-language type τ and effect ε

we write $\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ for $\llparenthesis\tau^{\mathrm{v}}_{\varepsilon}\rrparenthesis^{\mathrm{v}}$ and $\llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ for $\llparenthesis\tau^{\mathrm{n}}_{\varepsilon}\rrparenthesis^{\mathrm{n}}$, i.e. the call-by-value and call-by-name translations of τ into GCBPV, where all of the effect annotations are ε. Similarly for typing contexts. If $\Gamma \vdash e : \tau$ and $\mathit{ops}\ e \subseteq \varepsilon$ then we also write $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ and $\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ for the call-by-value and Levy-style call-by-name translations of e with ε for the effect annotations. These are typable in the two effect systems by Lemma 3.2.1. Pictorially, we have:

$$\begin{tikzcd}
& \Gamma^{\mathrm{v}}_{\varepsilon} \vdash_{\mathrm{v}} e : \tau^{\mathrm{v}}_{\varepsilon} \mathbin{\&} \varepsilon \arrow[r, mapsto] & \llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon} \mathrel{\underline{\vdash}} \llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon} : \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon} \\
\Gamma \vdash e : \tau \arrow[ur, mapsto] \arrow[dr, mapsto] & & \\
& \Gamma^{\mathrm{n}}_{\varepsilon} \vdash_{\mathrm{n}} e : \tau^{\mathrm{n}}_{\varepsilon} \arrow[r, mapsto] & \llparenthesis\Gamma\rrparenthesis^{\mathrm{n}}_{\varepsilon} \mathrel{\underline{\vdash}} \llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon} : \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}
\end{tikzcd}$$

The two computations on the right are the ones that we want to relate using the contextual preorder. However, we cannot ask whether $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon} \preccurlyeq_{\mathrm{ctx}} \llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ because one computation has a call-by-value type and one has a call-by-name type. It does not make sense to replace $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ with $\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ inside a GCBPV program, because the result would not be well-typed.

A similar problem arises when comparing two different denotational semantics of the same language. When comparing direct and continuation semantics of the lambda calculus, Reynolds [89], solves this problem by defining maps between the two semantics, so that a denotation in the direct semantics can be viewed as a denotation in the continuation semantics and vice versa. We use a similar idea here.

Specifically, we define maps Φ from call-by-value computations to call-by-name computations, and Ψ from call-by-name to call-by-value:

$$\Gamma \mathrel{\underline{\vdash}} M : \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon} \quad\mapsto\quad \Gamma \mathrel{\underline{\vdash}} \Phi_{\tau,\varepsilon}M : \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon} \qquad\qquad \Gamma \mathrel{\underline{\vdash}} N : \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon} \quad\mapsto\quad \Gamma \mathrel{\underline{\vdash}} \Psi_{\tau,\varepsilon}N : \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$$

Then instead of replacing $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ with $\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ directly, we use Φ and Ψ to convert $\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ to a computation of the correct type (defined formally in Section 3.4). The result looks something like the following composition of the translation of the expression e with two maps:

$$\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon} \longrightarrow \llparenthesis\Gamma\rrparenthesis^{\mathrm{n}}_{\varepsilon} \xrightarrow{\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}} \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon} \longrightarrow \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$$

This behaves like a call-by-name computation, but has the same type as a call-by-value computation. We could instead have chosen to convert $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ into a computation of the same type as $\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}$. This choice is arbitrary, because of the properties of Galois connections.

We do not want just *any* maps between call-by-value and call-by-name. We show that under certain conditions (which is where the choice of side-effects becomes important) the maps we define form *Galois connections* [73]. This is crucial for the correctness of our reasoning principle. It also intuitively implies that it does not matter where we use Φ and Ψ inside programs.

The maps $\Phi_{\tau,\varepsilon}$ and $\Psi_{\tau,\varepsilon}$ are defined by induction on the structure of τ in Figure 3.3. We use some extra variables in the definition, which are assumed to be fresh. To go from a call-by-value computation to a call-by-name computation we first evaluate the call-by-value computation, and then map the result to call-by-name using $\hat{\Phi}_{\tau,\varepsilon}$, which has the following typing:

$$\Gamma \vdash V : \llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon} \quad\mapsto\quad \Gamma \mathrel{\underline{\vdash}} \hat{\Phi}_{\tau,\varepsilon}V : \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}$$

On the base types **bool** and **unit**, the maps Φ and Ψ do nothing. On function types, they convert the argument to the other evaluation order, apply the function, and then convert back. (Due to the contravariance of arguments the two maps are mutually defined.) These have the types stated because the unit effect $\emptyset$ is the least element of the effect algebra, and the multiplication $\cup$ is idempotent.

$$\begin{aligned}
\Phi_{\tau,\varepsilon} M &:= M\ \mathbf{to}\ x.\, \hat{\Phi}_{\tau,\varepsilon} x \\[1ex]
\hat{\Phi}_{\mathbf{unit},\varepsilon} V &:= \mathbf{coerce}_{\emptyset\le\varepsilon}\,\langle V\rangle \\
\hat{\Phi}_{\mathbf{bool},\varepsilon} V &:= \mathbf{coerce}_{\emptyset\le\varepsilon}\,\langle V\rangle \\
\hat{\Phi}_{\tau\to\tau',\varepsilon} V &:= \lambda x : \mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}_{\varepsilon}.\ \Psi_{\tau,\varepsilon}(\mathbf{force}\,x)\ \mathbf{to}\ z.\ \Phi_{\tau',\varepsilon}(z\,\text{`}\,\mathbf{force}\,V) \\[1ex]
\Psi_{\mathbf{unit},\varepsilon} N &:= N \\
\Psi_{\mathbf{bool},\varepsilon} N &:= N \\
\Psi_{\tau\to\tau',\varepsilon} N &:= \mathbf{coerce}_{\emptyset\le\varepsilon}\,\Big\langle \mathbf{thunk}\,\lambda x : (\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}.\ \Psi_{\tau',\varepsilon}\big((\mathbf{thunk}\,(\hat{\Phi}_{\tau,\varepsilon} x))\,\text{`}\,N\big)\Big\rangle
\end{aligned}$$

Figure 3.3: Syntactic maps Φ from call-by-value to Levy-style call-by-name and Ψ from Levy-style call-by-name to call-by-value

In the rest of this section, we show that for side-effects satisfying certain conditions, the two maps form *Galois connections.* We do this with respect to some given logical relation $\mathcal{R}\llbracket - \rrbracket$, with the aim that we can use the logical relation to show instances of the contextual preorder (by applying Lemma 3.1.5). In addition to the usual requirements on logical relations (Definition 3.1.1), we assume that the relations $\mathcal{R}\llbracket A\rrbracket$ and $\mathcal{R}\llbracket \underline{C}\rrbracket$ are transitive for each A and $\underline{C}$. This implies (using Lemma 3.1.4) that each relation is a preorder.[2] We emphasize that the assumptions we have made about the logical relation so far are weak: it should be possible to define a useful logical relation satisfying these assumptions for every collection of side-effects (in particular, they hold for our global state example). We add further constraints below that allow us to derive our reasoning principle.

Definition 3.3.1 (Galois connection) A *Galois connection* (Φ, Ψ) from $\underline{C}$ to $\underline{D}$ is pair of maps

$$M : \underline{C} \;\mapsto\; \Phi M : \underline{D} \qquad\qquad N : \underline{D} \;\mapsto\; \Psi N : \underline{C}$$

such that for all closed $M : \underline{C}$ and $N : \underline{D}$,

$$(\Phi M, N) \in \mathcal{R}\llbracket \underline{D}\rrbracket \qquad \Leftrightarrow \qquad (M, \Psi N) \in \mathcal{R}\llbracket \underline{C}\rrbracket \qquad ◀$$

In this definition we restrict to closed terms. Although $\Phi_{\tau,\varepsilon}$ and $\Psi_{\tau,\varepsilon}$ are defined for open terms, we do not need to consider open terms when showing that they form a Galois connection, because $\mathcal{R}\llbracket \Gamma\rrbracket \dot{\to} \mathcal{R}\llbracket \underline{C}\rrbracket$ is defined in terms of $\mathcal{R}\llbracket \underline{C}\rrbracket$ on closed computations, and this is what we need to prove instances of.

For $(\Phi_{\tau,\varepsilon}, \Psi_{\tau,\varepsilon})$ to be a Galois connection the following must hold:

$$(\Phi_{\tau,\varepsilon}(\Psi_{\tau,\varepsilon} M),\ M) \in \mathcal{R}\llbracket (\!|\tau|\!)^{\mathrm{n}}_{\varepsilon}\rrbracket \qquad (N,\ \Psi_{\tau,\varepsilon}(\Phi_{\tau,\varepsilon} N)) \in \mathcal{R}\llbracket \langle\varepsilon\rangle (\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}\rrbracket$$

The right cannot hold in general when τ is a function type $\tau' \to \tau''$, because the term $\Psi_{\tau,\varepsilon}(\Phi_{\tau,\varepsilon} N)$ has no side-effects, even if N does. If we convert a computation N from call-by-value to call-by-name and then back using these maps then the side-effects of N are *thunked.* The side-effects do not occur until the function is applied. For $\tau = \mathbf{unit} \to \mathbf{unit}$ we get

$$\Psi_{\mathbf{unit}\to\mathbf{unit},\varepsilon}(\Phi_{\mathbf{unit}\to\mathbf{unit},\varepsilon} N) \;\equiv\; \mathbf{coerce}_{\emptyset\le\varepsilon}\,\langle \mathbf{thunk}\,\lambda x : \mathbf{unit}.\ N\ \mathbf{to}\ z.\ x\,\text{`}\,\mathbf{force}\,z\rangle$$

[2] Galois connections are normally defined for partial orders. We do not have antisymmetry, and hence need to generalize to preorders, but this is unimportant.

Based on this observation, we require a further constraint on the logical relation to ensure that we get Galois connections. This is where it becomes important which side-effects we have: it is only satisfied in certain cases (see Example 3.3.5). The constraint is that computations are *thunkable*:

Definition 3.3.2 (Thunkable) A closed computation $M : \langle\varepsilon\rangle A$ is *thunkable* if

$$(M\ \mathbf{to}\ x.\ \langle\mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\ \langle x\rangle\rangle,\ \mathbf{coerce}_{\emptyset\leq\varepsilon}\ \langle\mathbf{thunk}\,M\rangle) \in \mathcal{R}\llbracket\langle\varepsilon\rangle\mathbf{U}\,\langle\varepsilon\rangle A\rrbracket$$

An effect ε is *thunkable* if for every value type A, every closed computation $M : \langle\varepsilon\rangle A$ is thunkable. ◂

This property was first defined in a symmetric setting (non-enriched Kleisli categories) by Führmann [27]. Our setting is not in general symmetric (even though we only give a symmetric example in this chapter), so we give a directed version (an alternative name might be *lax* thunkable). We only need one direction to hold for the proofs.

For intuition, suppose we have a computation $y : \mathbf{U}\,\langle\varepsilon\rangle A \vdash N : \underline{C}$. Then

$$M\ \mathbf{to}\ x.\ N[y \mapsto \mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\ \langle x\rangle] \qquad\qquad \mathbf{coerce}_{\underline{C}<:\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}}\ (N[y \mapsto \mathbf{thunk}\,M])$$

are two closed computations of type $\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}$. The left computation evaluates M exactly once (like function arguments in call-by-value); the right potentially evaluates M several times (like call-by-name). If M is thunkable, then the left is related to the right by $R\llbracket\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}\rrbracket$. In other words, it is correct to replace a computation that evaluates M once with one that evaluates M zero or more times.

A key example is thunking the side-effects of call-by-value functions. Every computation $M : \langle\varepsilon\rangle\llparenthesis\tau' \to \tau''\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ has two opportunities to have side-effects: when evaluating M to a function (immediate), and after applying the function (latent). If M is thunkable, the immediate side-effects can be turned into latent side-effects, because M is related to the following computation by $\mathcal{R}\llbracket\langle\varepsilon\rangle\llparenthesis\tau' \to \tau''\rrparenthesis^{\mathrm{v}}_{\varepsilon}\rrbracket$:

$$\mathbf{coerce}_{\emptyset\leq\varepsilon}\ \langle\mathbf{thunk}\,\lambda x : \llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}.\ M\ \mathbf{to}\ f.\ x`\mathbf{force}\,f\rangle$$

where the M is now inside the lambda.

To solve the issue with $\Psi_{\mathbf{unit}\to\mathbf{unit},\varepsilon}(\Phi_{\mathbf{unit}\to\mathbf{unit},\varepsilon}M)$ above it suffices for M to be thunkable. In fact, if enough computations are thunkable then the maps defined above do in fact form Galois connections:

Theorem 3.3.3 Suppose that the effect ε is thunkable. For every source-language type τ the pair $(\Phi_{\tau,\varepsilon}, \Psi_{\tau,\varepsilon})$ is a Galois connection from $\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ to $\llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}$.

Proof. By induction on the structure on τ. This is trivial for **unit** and **bool**, because the maps are just identities.

The interesting case is when τ is a function type $\tau' \to \tau''$. We first show that $(M, \Psi_{\tau,\varepsilon}N) \in \mathcal{R}\llbracket\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}\rrbracket$ implies $(\Phi_{\tau,\varepsilon}M, N) \in \mathcal{R}\llbracket\llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}\rrbracket$. It suffices to show for arbitrary $(P, P') \in \mathcal{R}\llbracket\llparenthesis\tau'\rrparenthesis^{\mathrm{n}}_{\varepsilon}\rrbracket$ that we have

$$(\mathbf{thunk}\,P)\ `\ \Phi_{\tau,\varepsilon}M \quad \mathcal{R} \quad (\mathbf{thunk}\,P')\ `\ N$$

We reason as follows:

$$\begin{array}{lll}
 & M\ \mathbf{to}\ x.\ \Psi_{\tau',\varepsilon}P\ \mathbf{to}\ z.\ z`\mathbf{force}\,x & \\
\mathcal{R} & \Psi_{\tau,\varepsilon}N\ \mathbf{to}\ x.\ \Psi_{\tau',\varepsilon}P\ \mathbf{to}\ z.\ z`\mathbf{force}\,x & \text{(assumption)} \\
\equiv & \Psi_{\tau',\varepsilon}P\ \mathbf{to}\ z.\ \Psi_{\tau'',\varepsilon}\left((\mathbf{thunk}\,\Phi_{\tau',\varepsilon}(\mathbf{coerce}_{\emptyset\leq\varepsilon}\ \langle z\rangle))\ `\ N\right) & \\
\mathcal{R} & \Psi_{\tau'',\varepsilon}\left((\mathbf{thunk}\,\Phi_{\tau',\varepsilon}(\Psi_{\tau',\varepsilon}P))\ `\ N\right) & (\varepsilon\text{ is thunkable}) \\
\mathcal{R} & \Psi_{\tau'',\varepsilon}\left((\mathbf{thunk}\,P')\ `\ N\right) & \text{(Galois connection on }\tau')
\end{array}$$

This and the fact that we have a Galois connection on τ'' then imply

$$\mathbf{thunk}\,P\,\text{`}\,\Phi_{\tau,\varepsilon}M \quad\equiv\quad \Phi_{\tau'',\varepsilon}(M\ \mathbf{to}\ x.\,\Psi_{\tau',\varepsilon}P\ \mathbf{to}\ z.\,z\text{`}\mathbf{force}\,x) \quad \mathcal{R} \quad (\mathbf{thunk}\,P')\,\text{`}\,N$$

as required.

Second, we show that $(\Phi_{\tau,\varepsilon}M, N) \in \mathcal{R}[\![(\!|\tau|\!)^{\mathrm{n}}_{\varepsilon}]\!]$ implies $(M, \Psi_{\tau,\varepsilon}N) \in \mathcal{R}[\![\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}]\!]$. We have already mentioned that M is related to

$$\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle\mathbf{thunk}\,\lambda x : (\!|\tau'|\!)^{\mathrm{v}}_{\varepsilon}.\,M\ \mathbf{to}\ f.\,x\text{`}\mathbf{force}\,f\rangle$$

(using the fact that M is thunkable) so it suffices to show that this computation is related to $\Psi_{\tau,\varepsilon}N$ by $\mathcal{R}[\![\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}]\!]$. By looking at the definitions of $\Psi_{\tau,\varepsilon}$ and this relation, and using the fact that logical relations are closed under **coerce**, it is therefore enough to show that, for all $(V, V') \in \mathcal{R}[\![(\!|\tau'|\!)^{\mathrm{v}}_{\varepsilon}]\!]$, the following computations are related by $\mathcal{R}[\![\langle\varepsilon\rangle(\!|\tau''|\!)^{\mathrm{v}}_{\varepsilon}]\!]$:

$$M\ \mathbf{to}\ f.\,V\text{`}\mathbf{force}\,f \qquad\qquad \Psi_{\tau',\varepsilon}\big((\mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon}V'))\,\text{`}\,N\big)$$

We prove this by first reasoning as follows:

$$\begin{aligned}
\Phi_{\tau'',\varepsilon}(M\ \mathbf{to}\ f.\,V\text{`}\mathbf{force}\,f) \quad &\equiv \quad \Phi_{\tau'',\varepsilon}(M\ \mathbf{to}\ f.\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle V\rangle\ \mathbf{to}\ z.\,z\text{`}\mathbf{force}\,f)\\
&\mathcal{R} \quad \Phi_{\tau'',\varepsilon}(M\ \mathbf{to}\ f.\,\Psi_{\tau',\varepsilon}(\Phi_{\tau',\varepsilon}(\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle V'\rangle))\ \mathbf{to}\ z.\,z\text{`}\mathbf{force}\,f)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Galois connection on } \tau')\\
&\equiv \quad (\mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon}V'))\,\text{`}\,\Phi_{\tau,\varepsilon}M\\
&\mathcal{R} \quad (\mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon}V'))\,\text{`}\,N \qquad\qquad\qquad\qquad\qquad \text{(assumption)}
\end{aligned}$$

and then using the fact that we have a Galois connection for τ''. □

This theorem also has a partial converse: if the maps form Galois connections, then computations that return elements of base types are thunkable:

Lemma 3.3.4 If the pair $(\Phi_{\mathbf{unit}\to\mathbf{unit},\varepsilon}, \Psi_{\mathbf{unit}\to\mathbf{unit},\varepsilon})$ is a Galois connection then every closed computation $M : \langle\varepsilon\rangle\tau$, where $\tau \in \{\mathbf{unit}, \mathbf{bool}\}$, is thunkable.

Proof. Define the closed computation $M' : \langle\varepsilon\rangle(\!|\mathbf{unit} \to \tau|\!)^{\mathrm{v}}_{\varepsilon}$ by

$$M' \ := \ M\ \mathbf{to}\ x.\,\langle\mathbf{thunk}\,\lambda\_ : \mathbf{unit}.\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle x\rangle\rangle$$

By a standard property of Galois connections, M' is related by $\mathcal{R}[\![\langle\varepsilon\rangle(\!|\mathbf{unit} \to \tau|\!)^{\mathrm{v}}_{\varepsilon}]\!]$ to the computation $\Psi_{\mathbf{unit}\to\tau,\varepsilon}(\Phi_{\mathbf{unit}\to\tau,\varepsilon}M')$. Hence:

$$\begin{aligned}
M\ \mathbf{to}\ x.\,\langle\mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle x\rangle\rangle \quad &\equiv \quad M'\ \mathbf{to}\ t.\,\langle\mathbf{thunk}\,(()\,\text{`}\,\mathbf{force}\,t)\rangle\\
&\mathcal{R} \quad \Psi_{\mathbf{unit}\to\tau,\varepsilon}(\Phi_{\mathbf{unit}\to\tau,\varepsilon}M')\ \mathbf{to}\ t.\,\langle\mathbf{thunk}\,(()\,\text{`}\,\mathbf{force}\,t)\rangle\\
&\equiv \quad \mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle\mathbf{thunk}\,M\rangle
\end{aligned}$$

□

This proof works for base types τ because the two maps $\Phi_{\tau,\varepsilon}$ and $\Psi_{\tau,\varepsilon}$ are identities.

We end this section by returning to our main example.

Example 3.3.5 Recall our global state example. There are four possible effects: $\emptyset$, $\{\mathrm{get}\}$, $\{\mathrm{put}\}$ and $\{\mathrm{get}, \mathrm{put}\}$. As expected, the effect $\emptyset$ is thunkable. This is the case because if $M : \langle\emptyset\rangle A$ then by the fundamental lemma (Lemma 3.1.4) there is some $V : A$ such that $M \equiv \langle V\rangle$. We therefore have:

$$M\ \mathbf{to}\ x.\,\langle\mathbf{thunk}\,\langle x\rangle\rangle \ \equiv\ \langle\mathbf{thunk}\,\langle V\rangle\rangle \ \equiv\ \mathbf{coerce}_{\emptyset\leq\emptyset}\,\langle\mathbf{thunk}\,M\rangle$$

and these terms are related by $\mathcal{R}[\![\langle\emptyset\rangle A]\!]$.

The effect $\{\mathrm{put}\}$ is not thunkable, because the computation $\mathrm{put}\,\mathbf{true}$ is not. This is also what we would expect: if $\mathrm{put}\,\mathbf{true}$ were thunkable then we would have

$$\mathrm{put}\,\mathbf{true}\ \mathbf{to}\ x.\,\langle\mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\{\mathrm{put}\}}\,\langle x\rangle\rangle\ \cong_{\mathrm{ctx}}\ \mathbf{coerce}_{\emptyset\leq\{\mathrm{put}\}}\,\langle\mathbf{thunk}\,(\mathrm{put}\,\mathbf{true})\rangle$$

which implies (using the computation context $\square\ \mathbf{to}\ t.\ \mathrm{put}\,\mathbf{false};\ \mathbf{force}\,t$) that $\mathrm{put}\,\mathbf{false}\preccurlyeq\mathrm{put}\,\mathbf{true}$. Clearly we do not want this to be the case (and it is not). The effect $\{\mathrm{get},\mathrm{put}\}$ similarly is not thunkable.

The most interesting case is $\{\mathrm{get}\}$. This effect is not thunkable because the computation $\mathrm{get}\,()$ is not (this can be verified trivially by looking at the definition of the logical relation). We would not expect get to be thunkable because the state is not read-only: if get were thunkable then we would have

$$\mathrm{get}\,()\ \mathbf{to}\ x.\,\langle\mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle x\rangle\rangle\ \cong_{\mathrm{ctx}}\ \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle\mathbf{thunk}\,(\mathrm{get}\,())\rangle \tag{3.1}$$

and using the context $\mathrm{put}\,\mathbf{false};\ \square\ \mathbf{to}\ t.\ (\mathrm{put}\,\mathbf{true};\ \mathbf{force}\,t)$ this implies

$$\mathbf{coerce}_{\{\mathrm{put}\}\leq\{\mathrm{get},\mathrm{put}\}}\,(\mathrm{put}\,\mathbf{true};\ \langle\mathbf{false}\rangle)\ \equiv\ \mathbf{coerce}_{\{\mathrm{put}\}\leq\{\mathrm{get},\mathrm{put}\}}\,(\mathrm{put}\,\mathbf{true};\ \langle\mathbf{true}\rangle)$$

which is not the case because the results of the two computations differ.[3]

To summarize: the only effect that is thunkable for this example is $\emptyset$, and hence $\emptyset$ is the only effect for which we can apply our reasoning principle (more interesing examples are discussed in Section 4.4). In other words, we can only replace call-by-value with call-by-name in our global state example for subterms that have no side-effects. (Though the rest of the program might still have side-effects.) ◀

3.4 Reasoning principle for call-by-value and call-by-name

We now use the Galois connections defined in the previous section to relate the call-by-value and Levy-style call-by-name translations of expressions, and arrive at our reasoning principle.

Recall that we compose the call-by-name translation of each expression e with the maps Φ and Ψ defined above, to arrive at a GCBPV computation of the same type as the call-by-value translation[4]:

$$(\!|\Gamma|\!)^{\mathrm{v}}_{\varepsilon}\longrightarrow(\!|\Gamma|\!)^{\mathrm{n}}_{\varepsilon}\xrightarrow{(\!|e|\!)^{\mathrm{n}}_{\varepsilon}}(\!|\tau|\!)^{\mathrm{n}}_{\varepsilon}\longrightarrow\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}$$

We first give a precise definition of this composition. The arrow on the right is just given by applying $\Psi_{\tau,\varepsilon}$. The arrow on the left is a substitution $\hat{\Phi}_{\Gamma,\varepsilon}$ from terms in call-by-name contexts $(\!|\Gamma|\!)^{\mathrm{n}}_{\varepsilon}$ to terms in call-by-value contexts $(\!|\Gamma|\!)^{\mathrm{v}}_{\varepsilon}$. We define $\hat{\Phi}_{\Gamma,\varepsilon}$ for the source-language typing context $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$ as

$$\hat{\Phi}_{\Gamma,\varepsilon}\ :=\ x_1\mapsto\mathbf{thunk}\,\big(\hat{\Phi}_{\tau_1,\varepsilon}x_1\big),\ldots,x_n\mapsto\mathbf{thunk}\,\big(\hat{\Phi}_{\tau_n,\varepsilon}x_n\big)$$

[3] However, even if we removed put from the language, and used the same definition of the logical relation (but omitting the $\{\mathrm{put}\}$ and $\{\mathrm{get},\mathrm{put}\}$ cases, the computation $\mathrm{get}\,()$ would still not be thunkable with respect to the logical relation. This suggests that the definition of $\mathcal{R}[\![\mathrm{get}]\!]$ above (and hence the free lifting) might not be suitable in that case. We might be able to use a logical relation that relates more computations.

[4] Also recall that we identify GCBPV terms up to $\equiv$ throughout this chapter, and therefore, because the effect algebra is Gifford-style, it does not matter which typing derivation for e we choose to translate. We use this fact implicitly in this section.

The composition above is then the term $\Psi_{\tau,\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)$. If $\Gamma \vdash e : \tau$ and $\mathit{ops}\, e \subseteq \varepsilon$ then $\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon} \,\underline{\vdash}\, \Psi_{\tau,\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big) : \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$, so this computation has the same typing as $\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}$.

The reasoning principle we derive proves instances of the contextual preorder that have the form

$$\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon} \;\preccurlyeq_{\mathrm{ctx}}\; \Psi_{\tau,\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)$$

This works by first showing the two terms are related by the logical relation:

$$\big(\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon},\; \Psi_{\tau,\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)\big) \;:\; \mathcal{R}[\![\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!] \mathrel{\dot{\rightarrow}} \mathcal{R}[\![\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$$

and then applying Lemma 3.1.5, which allows us to promote this to an instance of the contextual preorder. We prove that the two terms are related using the properties of Galois connections, which allow us to push composition with $\Psi_{\tau,\varepsilon}$ into the structure of terms.

Lemma 3.4.1 Suppose that $(\Phi_{\tau',\varepsilon}, \Psi_{\tau',\varepsilon})$ is a Galois connection from $\langle\varepsilon\rangle\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ to $\llparenthesis\tau'\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ for every source language type τ'. If $\Gamma \vdash e : \tau$ and $\mathit{ops}\, e \subseteq \varepsilon$ then

$$\big(\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon},\; \Psi_{\tau,\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)\big) \;:\; \mathcal{R}[\![\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!] \mathrel{\dot{\rightarrow}} \mathcal{R}[\![\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$$

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. In each case of the induction we consider an arbitrary pair $(\sigma, \sigma') \in \mathcal{R}[\![\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$ (recall that this is how $\dot{\rightarrow}$ is defined). We define σ'' to be the composition of $\hat{\Phi}_{\Gamma,\varepsilon}$ with σ'. In each case we are required to show $(\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}[\sigma],\; \Psi_{\tau,\varepsilon}(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma''])) \in \mathcal{R}[\![\langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$. We only give three representative cases.

- If e is a variable x then, writing V for $\sigma\, x$ and V' for $\sigma'\, x$, we have:

$$\llparenthesis x\rrparenthesis^{\mathrm{v}}_{\varepsilon}[\sigma] \;\equiv\; \mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle V\rangle \quad \mathcal{R} \quad \Psi_{\tau,\varepsilon}(\Phi_{\tau,\varepsilon'}(\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle V'\rangle)) \;\equiv\; \Psi_{\tau,\varepsilon}(\llparenthesis x\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma''])$$

- If e is a λ-abstraction $\lambda x{:}\tau'.\, e$ and $\tau = \tau' \rightarrow \tau''$ then (expanding the definition of $\Psi_{\tau,\varepsilon}$) it suffices to show that $(\lambda x{:}\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}.\, \llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon})[\sigma]$ is related by $\mathcal{R}[\![\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon} \rightarrow \langle\varepsilon\rangle\llparenthesis\tau''\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$ to

$$\big(\lambda x{:}\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}.\, \Psi_{\tau'',\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[x \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon} x)]\big)\big)[\sigma'']$$

Consider arbitrary $(W, W') \in \mathcal{R}[\![\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}]\!]$. We have:

$$\begin{array}{rcl} W \text{ ‘ } (\lambda x{:}\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}.\, \llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon})[\sigma] & \equiv & \llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon}[\sigma, x \mapsto W] \\ & \mathcal{R} & \Psi_{\tau'',\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'', x \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon} W')]\big) \\ & \equiv & W' \text{ ‘ } \big(\lambda x{:}\llparenthesis\tau'\rrparenthesis^{\mathrm{v}}_{\varepsilon}.\, \Psi_{\tau'',\varepsilon}\big(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[x \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau',\varepsilon} x)]\big)\big)[\sigma''] \end{array}$$

- If e is a function application $e_1\, e_2$, where e_1 has type $\tau' \rightarrow \tau$, then:

$$\begin{array}{rcl} \llparenthesis e_1\, e_2\rrparenthesis^{\mathrm{v}}_{\varepsilon}[\sigma] & \equiv & \llparenthesis e_1\rrparenthesis^{\mathrm{v}}_{\varepsilon}\ \mathbf{to}\ f.\, \llparenthesis e_2\rrparenthesis^{\mathrm{v}}_{\varepsilon}\ \mathbf{to}\ x.\, x \text{ ‘ } \mathbf{force}\, f \\ & \mathcal{R} & \Psi_{\tau'\rightarrow\tau,\varepsilon}\big(\llparenthesis e_1\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big)\ \mathbf{to}\ f.\, \Psi_{\tau',\varepsilon}\big(\llparenthesis e_2\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big)\ \mathbf{to}\ x.\, x \text{ ‘ } \mathbf{force}\, f \\ & \mathcal{R} & \Psi_{\tau,\varepsilon}\big(\Phi_{\tau,\varepsilon}\big(\Psi_{\tau'\rightarrow\tau,\varepsilon}\big(\llparenthesis e_1\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big)\ \mathbf{to}\ f.\, \Psi_{\tau',\varepsilon}\big(\llparenthesis e_2\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big)\ \mathbf{to}\ x.\, x \text{ ‘ } \mathbf{force}\, f\big)\big) \\ & & \hfill\text{(Galois connection on } \tau\text{)} \\ & \equiv & \Psi_{\tau,\varepsilon}\big((\mathbf{thunk}\,(\llparenthesis e_2\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma''])) \text{ ‘ } \Phi_{\tau'\rightarrow\tau,\varepsilon}(\Psi_{\tau'\rightarrow\tau,\varepsilon}(\llparenthesis e_1\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']))\big) \\ & \mathcal{R} & \Psi_{\tau,\varepsilon}\big((\mathbf{thunk}\,(\llparenthesis e_2\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma''])) \text{ ‘ } \llparenthesis e_1\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big) \\ & & \hfill\text{(Galois connection on } \tau' \rightarrow \tau\text{)} \\ & \equiv & \Psi_{\tau,\varepsilon}\big(\llparenthesis e_1\, e_2\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\sigma'']\big) \end{array}$$

□

We showed in the previous section (Theorem 3.3.3) that the maps between the two evaluation orders form Galois connections if the effect ε is thunkable. Hence we arrive at our reasoning principle, which we state formally as Theorem 3.4.2. Recall that each inequational theory $\preccurlyeq$ (Definition 2.7.4) induces a contextual preorder $\preccurlyeq_{\mathrm{ctx}}$ (Definition 2.7.6). Given any such inequational theory, to show that the call-by-value and call-by-name translations of source-language expressions restricted to the effect ε are related by $\preccurlyeq_{\mathrm{ctx}}$ it is enough to define a suitable logical relation, and show that the effect ε is thunkable.

Theorem 3.4.2 (Call-by-value and call-by-name) Suppose we given some GCBPV inequational theory and a logical relation such that:

- Each relation $\mathcal{R}[\![A]\!]$ and each relation $\mathcal{R}[\![\underline{C}]\!]$ is transitive.
- For each ground type G and effect ε' we have:

$$(N, N') \in \mathcal{R}[\![\langle\varepsilon\rangle G]\!] \quad\Rightarrow\quad N \preccurlyeq N'$$

  for all N, N'.
- The effect ε is thunkable.

If $\Gamma \vdash e : \tau$ and *ops* $e \subseteq \varepsilon$ then

$$(\!|e|\!)^{\mathrm{v}}_{\varepsilon} \preccurlyeq_{\mathrm{ctx}} \Psi_{\tau,\varepsilon}\big((\!|e|\!)^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)$$

Proof. By Theorem 3.3.3, the maps between call-by-value and call-by-name computations form Galois connections. Hence we can apply Lemma 3.4.1, which tells us that

$$\big((\!|e|\!)^{\mathrm{v}}_{\varepsilon},\ \Psi_{\tau,\varepsilon}\big((\!|e|\!)^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)\big) \ :\ \mathcal{R}[\![(\!|\Gamma|\!)^{\mathrm{v}}_{\varepsilon}]\!] \dot{\to} \mathcal{R}[\![\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}]\!]$$

Finally, Lemma 3.1.5 implies the corresponding instance of the contextual preorder, which is the result we want. □

The generality of this theorem comes from two sources. First, we consider arbitrary inequational theories $\preccurlyeq$ in which the effect algebra is Gifford-style. (It may be possible to prove a similar reasoning principle for other effect algebras on a case-by-case basis.) The only other requirement is the existence of a suitable logical relation in which enough computations are thunkable. Second, this theorem applies to terms that are open and have higher types, using the maps between the two evaluation orders. We obtain a result about source-language *programs* (closed expressions of base types) as a corollary. The corollary is closer to the standard results that are proved for specific side-effects, because the maps are trivial.

Corollary 3.4.3 Suppose that the assumptions of Theorem 3.4.2 hold. If e is a closed source-language expression of type $\tau \in \{\mathbf{unit}, \mathbf{bool}\}$ and *ops* $e \subseteq \varepsilon$ then $(\!|e|\!)^{\mathrm{v}}_{\varepsilon} \preccurlyeq (\!|e|\!)^{\mathrm{n}}_{\varepsilon}$. ◀

Like our result about Galois connections, the reasoning principle has a partial converse. If call-by-value can be replaced with call-by-name, then computations of certain types are thunkable:

Lemma 3.4.4 Suppose that for each $\Gamma \vdash e : \tau$ with *ops* $e \subseteq \varepsilon$ we have

$$\big((\!|e|\!)^{\mathrm{v}}_{\varepsilon},\ \Psi_{\tau,\varepsilon}\big((\!|e|\!)^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}]\big)\big) \ :\ \mathcal{R}[\![(\!|\Gamma|\!)^{\mathrm{v}}_{\varepsilon}]\!] \dot{\to} \mathcal{R}[\![\langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}]\!]$$

For every base type $A \in \{\mathbf{unit}, \mathbf{bool}\}$, all closed computations $M : \langle\varepsilon\rangle A$ are thunkable.

Proof. First we show that every closed computation $N : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon}$ is related to $\Psi_{\tau,\varepsilon}(\Phi_{\tau,\varepsilon}N)$ (this is one of the properties of Galois connections). To do this, define the following source-language expression e and GCBPV value V:

$$\begin{aligned} e &\coloneqq x\,() && (x : \mathbf{unit} \longrightarrow \tau \vdash e : \tau) \\ V &\coloneqq \mathbf{thunk}\,(\lambda y\!:\!\mathbf{unit}.\, N) && (\diamond \vdash V : \mathbf{U}\,(\mathbf{unit} \longrightarrow \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_{\varepsilon})) \end{aligned}$$

Now we apply the assumption to e:

$$N \;\equiv\; (\!|e|\!)^{\mathrm{v}}_{\varepsilon}[x \mapsto V] \quad \mathcal{R} \quad \Psi_{\tau,\varepsilon}\big((\!|e|\!)^{\mathrm{n}}_{\varepsilon}[x \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\mathbf{unit}\to\tau,\varepsilon}V)]\big) \;\equiv\; \Psi_{\tau,\varepsilon}(\Phi_{\tau,\varepsilon}N)$$

We use this to prove the result as follows. Let τ' be either **unit** or **bool**, so that $(\!|\tau'|\!)^{\mathrm{v}}_{\varepsilon} = A$. Define the closed computation $N : \langle\varepsilon\rangle(\!|\mathbf{unit} \longrightarrow \tau'|\!)^{\mathrm{v}}_{\varepsilon}$ by:

$$N \coloneqq M\;\mathbf{to}\;x.\,\langle\mathbf{thunk}\,\lambda y\!:\!\mathbf{unit}.\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle x\rangle\rangle$$

Using the above with $\tau = \mathbf{unit} \longrightarrow \tau'$ we have:

$$\begin{aligned} M\;\mathbf{to}\;x.\,\langle\mathbf{thunk}\,\mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle x\rangle\rangle \quad &\equiv \quad N\;\mathbf{to}\;f.\,\langle\mathbf{thunk}\,(()\text{‘}\mathbf{force}\,f)\rangle \\ &\mathcal{R} \quad \Psi_{\mathbf{unit}\to\tau',\varepsilon}(\Phi_{\mathbf{unit}\to\tau',\varepsilon}N)\;\mathbf{to}\;f.\,\langle\mathbf{thunk}\,(()\text{‘}\mathbf{force}\,f)\rangle \\ &\equiv \quad \mathbf{coerce}_{\emptyset\leq\varepsilon}\,\langle\mathbf{thunk}\,M\rangle \end{aligned}$$

so M is thunkable. □

We again return to our main example, and apply our reasoning principle to it.

Example 3.4.5 Recall that for our global state example the effect $\emptyset$ is thunkable, but the other three effects $\{\mathrm{get}\}$, $\{\mathrm{put}\}$, and $\{\mathrm{get}, \mathrm{put}\}$ are not. (So the maps $\Phi_{\tau,\varepsilon}$ and $\Psi_{\tau,\varepsilon}$ are Galois connections for $\varepsilon = \emptyset$.) We have already mentioned that the logical relation for global state satisfies the assumption about ground types that allows us to use it to prove instances of the contextual preorder (this is not difficult to show). We can therefore apply our reasoning principle to expressions e that do not use either of the operations get and put. (This restriction also applies to the free variables of e. Their types are annotated with the effect $\emptyset$ in both the call-by-value and call-by-name effect systems, and hence we cannot bind them to expressions that have side-effects.)

If we have any program with a call-by-value subterm that does not have side-effects then the reasoning principle shows that we can replace the evaluation order of the subterm with call-by-name. For this example, the contextual preorder is symmetric, so we could also replace call-by-name with call-by-value: the reasoning principle shows they have identical behaviour. In both cases, the program itself may have side-effects. The restriction on side-effects applies only on the subterm. The corollary shows that if we have an entire program e that does not have any side-effects, then call-by-value and call-by-name are equivalent: we have $(\!|e|\!)^{\mathrm{v}}_{\emptyset} \;\equiv\; (\!|e|\!)^{\mathrm{n}}_{\emptyset}$.◄

3.5 Related work

Comparing evaluation orders Plotkin [84] and many others relate call-by-value and call-by-name. Crucially, they consider lambda-calculi with no side-effects other than divergence. This makes a significant difference to the techniques that can be used, in particular because in this case the equational theory for call-by-name is strictly weaker than for call-by-value. This is not necessarily true for other side-effects. Other evaluation orders (such as call-by-need) have also been compared in similarly restricted settings [67, 70, 33].

It might also be possible to recast some of our work in terms of the *duality* between call-by-value and call-by-name [23, 18, 99].

Relating semantics of languages The technique we use here to relate call-by-value and call-by-name is based on the idea used first by Reynolds [89] to relate direct and continuation semantics of the lambda calculus, and later used by others (e.g. [75, 52, 16, 24]). There are several differences with our approach. Reynolds first constructs a logical relation *between the two semantics*, and uses this to establish a relationship between direct and continuation semantics using the two maps. Our logical relations do not relate call-by-value and call-by-name, they instead relate arbitrary GCBPV terms. Reynolds also relies on continuations with a large-enough domain of answers (e.g. a solution to a particular recursive domain equation) to construct the maps between the two semantics. In our case we do not need to impose such a restriction for the maps to exist.

There has been previous work [92, 54, 93] on soundness and completeness properties of translations (similar to the translations into GCBPV). Galois connections (and similar structures) in which the order is reduction of programs play a significant role in these. A key difference in our case is that we consider the observable behaviour of programs. We cannot consider reductions in our case, because moving from call-by-value to call-by-name can add, remove, and reorder reductions.

Axiomatic properties of side-effects In this chapter, we emphasize the use of axiomatic properties of side-effects [27] for formal reasoning about programs. Many of these properties have been used for this and similar purposes, and have been studied for particular side-effects [14, 42, 60]. We contribute a new use for them.

3.6 Summary

This chapter has two primary goals. The first is to show that we can use GCBPV for formal reasoning about effect-dependent program transformations, including those that involve source languages. We do this primarily by defining a notion of logical relation for GCBPV (Definition 3.1.1) and relating it to the contextual preorder (Lemma 3.1.5). We also describe a technique for constructing these logical relations (the free lifting in Section 3.1.1).

The second is to derive a reasoning principle (Theorem 3.4.2) that relates call-by-value and call-by-name. The principle shows that it is correct to replace call-by-value with call-by-name for subterms restricted to *thunkable* effects (Definition 3.3.2). It is about open expressions, and allows us to change evaluation order *within* programs. We obtain a result about call-by-value and call-by-name evaluations of *programs* as a corollary (Corollary 3.4.3). The reasoning principle is not restricted to a particular collection of side-effects; we instead identify the axiomatic property of side-effects (thunkable) that gives rise to a relationship between the two evaluation orders.

We expect that the technique we use can be applied to other evaluation orders. Two evaluation orders can be related by giving translations into some common intermediate language (here we use GCBPV), constructing maps between the two translations, and showing that (for some models) these maps form Galois connections. In Chapter 5 we give a specific example of a relationship between call-by-name and call-by-need (with nontermination as the only side-effect), but not a *general* reasoning principle.

In this chapter we work exclusively at the level of syntax. For example, the notion of logical relation we use relates pairs of GCBPV terms, and we define thunkable in terms of the inequational theory. The advantage of using syntax rather than a denotational semantics is that the syntax requires less machinery. However, it also has disadvantages. Sticking to syntax makes it more difficult to work with some side-effects (such as recursion) syntactically, and many of the proofs (e.g. Theorem 3.3.3 and Lemma 3.4.1) are more complex. This partly motivates the next chapter, where we switch to semantics. We redevelop the relationship between call-by-value and call-by-name in terms of the semantics, and give more examples of side-effects that it can be applied to. At a high-level, the principle does not change much, and the technique used to derive it is the same. Both versions have merits.

Chapter 4

Noninvertible program transformations

Most of the previous work on proving the correctness of effect-dependent transformations has been restricted to cases where the transformations are *invertible* in the sense that both directions are valid, since they rely on the expressions having the same behaviour. However, there are many situations in which only one direction of a program transformation is valid. Consider the following two expressions:

$$\begin{array}{ll}
\mathbf{let}\ x = e\ \mathbf{in} & \qquad \mathbf{let}\ x = e\ \mathbf{in} \\
\mathbf{let}\ y = e\ \mathbf{in} & \\
(x, y) & \qquad (x, x)
\end{array}$$

In some cases these have the same behaviour, e.g. if the only side-effect of e is to write to some state that is not shared between threads. When this is true and the expression on the left appears inside some program, then it can safely be replaced with the expression on the right, perhaps as part of a compiler optimization.

Suppose that e above is allowed to write to state that is shared between threads. In this case, the two expressions are not equivalent, because some other thread may concurrently use the state between the two executions of e on the left. The observable behaviours of the expression on the right are an (in general proper) superset of those on the left. Replacing the left with the right would be a valid transformation, but the reverse is not. We cannot verify all effect-dependent transformations by proving only equivalences between expressions.

There are also other examples. Consider a language with boxing (i.e. conversion of primitive values into immutable references), and an equality operator on such references. We might wish to validate a transformation that merges boxes of the same value. This transformation would change uses of equality that could have evaluated to either true or false into ones that always give true. The reverse transformation is not valid because it could replace a program that is guaranteed to give true with one that might give false. Another example is undefined behaviour (e.g. in the C language). The C standard allows each expression that has undefined behaviour to be replaced with any other (even one that causes arbitrary effects), but the reverse is not true. Including undefined behaviour when considering program transformations is useful even if the source language does not have undefined behaviour [55].

These examples explain why we defined *in*equational theories in Section 2.7.2. This chapter develops machinery that we can use for reasoning about these *noninvertible* effect-dependent transformations.

Contributions We show that previous work on equational reasoning about effects can be adapted to *noninvertible* effect-dependent transformations:

- We describe a general *order-enriched* categorical semantics for GCBPV that allows us to state and prove noninvertible transformations semantically (Section 4.2).
- We *relate the syntax to the semantics* by describing how to prove adequacy of models of GCBPV (Section 4.3). We do this by developing an abstract notion of logical relation that generalizes the one given in Section 3.1.
- We apply the framework to examples by verifying effect-dependent transformations involving undefined behaviour, nondeterminism, and mutable state shared between threads. The examples are introduced in Section 4.1.
- We use the denotational semantics to redevelop our reasoning principle for relating call-by-value and call-by-name (Section 4.4).

Much of this chapter simply shows that techniques that have previously been applied to reasoning about effect-dependent transformations can be adapted to non-invertible cases, allowing support for previously neglected applications. To do this we have to redevelop some of the definitions that appear in the previous chapter. There we worked exclusively with syntax (in particular, the notion of logical relation we used relates terms to other terms). Many of the definitions we give here generalize those in the previous chapter.

4.1 Examples of noninvertible transformations

We first instantiate GCBPV (by choosing suitable inequational theories) to characterize three different collections of side-effects that we use as examples. For each example, we also state instances of the contextual preorder $\preccurlyeq_{\mathrm{ctx}}$ (and its symmetric counterpart $\cong_{\mathrm{ctx}}$) that represent effect-dependent program transformations. We defer the proofs that these instances hold to later sections of this chapter.

We use the following syntactic sugar for booleans (with eliminators for values and for computations), let bindings, and sequencing of computations throughout.

$$\mathbf{bool} \coloneqq \mathbf{unit} + \mathbf{unit} \qquad \mathbf{true} \coloneqq \mathbf{inl}\,() \qquad \mathbf{false} \coloneqq \mathbf{inr}\,()$$

$$\begin{array}{rcll}
\mathbf{if}\ V\ \mathbf{then}\ W_1\ \mathbf{else}\ W_2 & \coloneqq & \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\,x_1.\,W_1,\ \mathbf{inr}\,x_2.\,W_2\} & (x_i \text{ not free in } W_i) \\
\underline{\mathbf{if}}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 & \coloneqq & \underline{\mathbf{case}}\ V\ \mathbf{of}\ \{\mathbf{inl}\,x_1.\,M_1,\ \mathbf{inr}\,x_2.\,M_2\} & (x_i \text{ not free in } M_i) \\
\mathbf{let}\ x = V\ \mathbf{in}\ M & \coloneqq & \langle V \rangle\ \mathbf{to}\ x.\,M & \\
M;\ N & \coloneqq & M\ \mathbf{to}\ y.\,N & (y \text{ not free in } N)
\end{array}$$

The syntactic sugar for let-bindings can be written equivalently using lambdas or substitution:

$$\mathbf{let}\ x = V\ \mathbf{in}\ M \ \equiv\ V\,\text{‘}\,\lambda x.\,M \ \equiv\ M[x \mapsto V]$$

4.1.1 Undefined behaviour

As a simple example, we consider C-style *undefined behaviour*. Computations with undefined behaviour are allowed to do anything (including arbitrary side-effects) at runtime. Here we consider undefined behaviour in isolation, without any other side-effects in the language. We contribute a denotational semantics for undefined behaviour.

Recall that to instantiate GCBPV we need to choose a signature (see Definition 2.7.1), which consists of an effect algebra, base types, constants, and operations (with coarities, arities and effects). For this example, the signature consists of the following data:

- The effect algebra is the trivial preordered monoid $\{\star\}$. (So we do not actually track any effect information, and none of the transformations we consider are effect-dependent. This example is intended to be as simple as possible.)
- The only base type is **int**, which contains signed 32-bit integers.
- We have a constant $\underline{n} : \mathbf{int}$ for each n in the set $[\![\mathbf{int}]\!] := \{-2^{31}, \ldots, 2^{31}-1\}$, and two constants

  $$\mathbf{geq} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool} \qquad \mathbf{add} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$$

  The first determines if its first argument is greater than or equal to its second argument, and the second adds two integers, with the result defined to wrap around on overflow.
- There are two operations. The first is undef, which has coarity **unit**, arity **empty** and effect $\star$. Hence $\mathsf{undef}\,()$ is a computation of type $\langle\star\rangle\mathbf{empty}$. Semantically, it has undefined behaviour (and the inequational theory we define below captures this property.) As syntactic sugar we have an undefined computation $\mathbf{undef}_{\underline{C}}$ of type $\underline{C}$ for every computation type $\underline{C}$:

  $$\mathbf{undef}_{\underline{C}} \;:=\; \mathsf{undef}\,()\ \mathbf{to}\ x.\, \underline{\mathbf{case}}_{\underline{C}}\ x\ \mathbf{of}\ \{\}$$

  (We have $\mathsf{undef}\,() \equiv \mathbf{undef}_{\langle\star\rangle\mathbf{empty}}$.) The second operation is $\mathsf{add}_{\mathsf{nsw}}$ (no signed wrap), which adds two integers with undefined behaviour on overflow. Its coarity is $\mathbf{int} \times \mathbf{int}$, its arity is **int**, and its effect is $\star$, so if $V_1, V_2 : \mathbf{int}$ then $\mathsf{add}_{\mathsf{nsw}}\,(V_1, V_2) : \langle\star\rangle\mathbf{int}$ adds V_1 and V_2.

As we explain in Section 2.7.2, we can give an inequational theory by specifying a list of signature axioms, which characterize the behaviour of the operations and constants.

The main axiom we have is that undefined behaviour can be replaced with any computation (for all $\underline{C}$ and M):

$$\mathbf{undef}_{\underline{C}} \preccurlyeq M$$

(Recall our usual convention on typing of equations: the axiom holds only if both sides have the same type in the same typing context.)

We also have axioms that specify the behaviour of the three functions on constant integers. These are (for all $m, n \in [\![\mathbf{int}]\!]$):

$$\begin{aligned}
\mathbf{add}(\underline{m}, \underline{n}) &\equiv \underline{m+n+2^{32}} && \text{if } (m+n) < -2^{31}\\
\mathbf{add}(\underline{m}, \underline{n}) &\equiv \underline{m+n} && \text{if } (m+n) \in [\![\mathbf{int}]\!]\\
\mathbf{add}(\underline{m}, \underline{n}) &\equiv \underline{m+n-2^{32}} && \text{if } (m+n) \geq 2^{31}\\[1ex]
\mathbf{geq}(\underline{m}, \underline{n}) &\equiv \mathbf{true} && \text{if } m \geq n\\
\mathbf{geq}(\underline{m}, \underline{n}) &\equiv \mathbf{false} && \text{if } m < n\\[1ex]
\mathsf{add}_{\mathsf{nsw}}(\underline{m}, \underline{n}) &\equiv \langle \underline{m+n} \rangle && \text{if } (m+n) \in [\![\mathbf{int}]\!]\\
\mathsf{add}_{\mathsf{nsw}}(\underline{m}, \underline{n}) &\equiv \mathbf{undef}_{\mathbf{int}} && \text{if } (m+n) \notin [\![\mathbf{int}]\!]
\end{aligned}$$

where + is the usual addition of integers in $\mathbb{Z}$. (Recall that we write $\equiv$ to mean both directions are taken as axioms.) This completes the definition of the inequational theory.

A consequence of these signature axioms is that if undefined behaviour occurs as part of a larger computation, then the whole computation is undefined:

$$\mathbf{undef}_{\underline{C}}\ \mathbf{to}\ x.\, M \;\equiv\; \mathbf{undef}_{\underline{D}}$$

We give some valid program transformations for the inequational theory for undefined behaviour. The first is replacing $\mathsf{add}_{\mathsf{nsw}}$ with **add** inside programs:

$$\mathsf{add}_{\mathsf{nsw}}(v_1, v_2) \ \preccurlyeq_{\mathrm{ctx}}\ \langle \mathbf{add}(v_1, v_2)\rangle$$

Validity of this transformation implies it is correct to compile $\mathsf{add}_{\mathsf{nsw}}$ as **add**. It is not invertible in general (since the reverse can introduce undefined behaviour).

A common optimization is to simplify comparisons of integers by using the fact that some operations on signed integers have undefined behaviour. One example is the following transformation:

$$\begin{array}{lcl} \mathsf{add}_{\mathsf{nsw}}(V_1, V_2)\ \mathbf{to}\ x. & & \mathsf{add}_{\mathsf{nsw}}(V_1, V_2)\ \mathbf{to}\ x. \\ \mathbf{let}\ b = \mathbf{geq}(x, V_1)\ \mathbf{in} & \preccurlyeq_{\mathrm{ctx}} & \mathbf{let}\ b = \mathbf{geq}(V_2, \underline{0})\ \mathbf{in} \\ N & & N \end{array}$$

(This is especially useful when V_2 is a constant, because the comparison on the right can be evaluated statically.) Perhaps surprisingly, the reverse direction is also valid. This is the case because if the addition overflows, both terms have undefined behaviour, so the result of the comparison is irrelevant. It would not be correct to apply either direction of this transformation if **add** was used instead of $\mathsf{add}_{\mathsf{nsw}}$.

4.1.2 Nondeterminism

We also consider binary nondeterminism [11], so that programs choose from finite nonempty sets of results via a binary choice operation flip (that “flips a coin”). Unlike previous work (such as [11]), we allow nondeterministic choices to be made statically by program transformations, reducing the amount of runtime nondeterminism. These transformations are correct because they restrict the behaviour of programs; they do not add new behaviours. This is not the case for the reverse direction, and hence these transformations are not invertible. Finite nondeterminism is superficially similar to undefined behaviour, but not the same since there is no equation that corresponds to $\mathbf{undef}_{\underline{C}}\ \mathbf{to}\ x.\, M \ \equiv\ \mathbf{undef}_{\underline{D}}$.

The GCBPV signature for this example consists of the following data:

- The effect algebra tracks nondeterminism. It is the preordered monoid $\{1 \leq +\}$ with least upper bound as the multiplication and 1 as the unit. In this case, 1 means deterministic, and + means potentially nondeterministic.
- There are no base types or constants.
- There is a single operation flip with coarity **unit**, arity **bool** and effect +. The computation $\mathsf{flip}\,()$ has type $\langle + \rangle\mathbf{bool}$, and nondeterministically chooses either **true** or **false**.

We use the operation flip to choose between arbitrary computations M_1, M_2 by defining

$$M_1\ \mathbf{or}\ M_2\ :=\ \mathsf{flip}\,()\ \mathbf{to}\ x.\, \underline{\mathbf{if}}\ x\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2$$

If $\Gamma \vDash M_1 : \underline{C}$ and $\Gamma \vDash M_2 : \underline{C}$ then $\Gamma \vDash M_1\ \mathbf{or}\ M_2 : \langle\!\langle + \rangle\!\rangle\underline{C}$. The η-law for **bool** implies that $\mathsf{flip}\,() \ \equiv\ \langle\mathbf{true}\rangle\ \mathbf{or}\ \langle\mathbf{false}\rangle$.

Again we specify the inequational theory by giving signature axioms. These are listed in Figure 4.1. The first is the crucial one: it allows us to replace $\mathsf{flip}\,()$ with **true** or **false** statically (a similar axiom with **false** instead of **true** is derivable). The other direction of this axiom (with $\succcurlyeq$ rather than $\preccurlyeq$) is not derivable, so we cannot replace constant booleans with $\mathsf{flip}\,()$. The final three axioms in this group ensure that nondeterministic choice is idempotent, commutative and associative. Although these three axioms are non-symmetric (we only assume $\preccurlyeq$ rather than $\equiv$) the other direction of each is derivable.

$$
\begin{aligned}
\mathsf{flip}\,() &\preccurlyeq \mathbf{coerce}_{1\leq+}\,\langle\mathbf{true}\rangle \\
\mathbf{coerce}_{1\leq+}\,\langle()\rangle &\preccurlyeq \mathsf{flip}\,();\ \langle()\rangle \\
\mathsf{flip}\,() &\preccurlyeq \mathsf{flip}\,()\ \mathbf{to}\ x.\,\langle\mathbf{if}\ x\ \mathbf{then}\ \mathbf{false}\ \mathbf{else}\ \mathbf{true}\rangle
\end{aligned}
$$

$$
\begin{pmatrix}
\mathsf{flip}\,()\ \mathbf{to}\ x. \\
\underline{\mathbf{if}}\ x\ \mathbf{then}\ \mathsf{flip}\,()\ \mathbf{to}\ y.\,\langle\mathbf{inl}_{\mathbf{unit}}\,()\rangle \\
\quad \mathbf{else}\ \mathbf{coerce}_{1\leq+}\,\langle\mathbf{inr}_{\mathbf{bool}}\,()\rangle
\end{pmatrix}
\preccurlyeq
\begin{pmatrix}
\mathsf{flip}\,()\ \mathbf{to}\ x. \\
\underline{\mathbf{if}}\ x\ \mathbf{then}\ \mathbf{coerce}_{1\leq+}\,\langle\mathbf{inl}_{\mathbf{unit}}\mathbf{true}\rangle \\
\quad \mathbf{else}\ \mathsf{flip}\,()\ \mathbf{to}\ y. \\
\qquad \langle\mathbf{if}\ y\ \mathbf{then}\ \mathbf{inl}_{\mathbf{unit}}\mathbf{false}\ \mathbf{else}\ \mathbf{inr}_{\mathbf{bool}}\,()\rangle
\end{pmatrix}
$$

Figure 4.1: Signature axioms for nondeterminism

Lemma 4.1.1

1. If $\Gamma \vdash M_1 : \underline{C}$ and $\Gamma \vdash M_2 : \underline{C}$ then $M_1\ \mathbf{or}\ M_2 \ \equiv\ M_2\ \mathbf{or}\ M_1$.
2. If $\Gamma \vdash M : \underline{C}$ then $M\ \mathbf{or}\ M \ \equiv\ \mathbf{coerce}_{\underline{C}<:\langle\!\langle+\rangle\!\rangle\underline{C}}\,M$.
3. If $\Gamma \vdash M_1 : \underline{C}$ and $\Gamma \vdash M_2 : \underline{C}$ then
$$M_1\ \mathbf{or}\ M_2 \ \preccurlyeq\ \mathbf{coerce}_{\underline{C}<:\langle\!\langle+\rangle\!\rangle\underline{C}}\,M_1 \qquad M_1\ \mathbf{or}\ M_2 \ \preccurlyeq\ \mathbf{coerce}_{\underline{C}<:\langle\!\langle+\rangle\!\rangle\underline{C}}\,M_2$$
4. If $\Gamma \vdash M_1 : \underline{C}$, $\Gamma \vdash M_2 : \underline{C}$ and $\Gamma \vdash M_3 : \underline{C}$ then $(M_1\ \mathbf{or}\ M_2)\ \mathbf{or}\ M_3 \ \equiv\ M_1\ \mathbf{or}\ (M_2\ \mathbf{or}\ M_3)$.
5. If $\Gamma \vdash M_1 : \langle\varepsilon\rangle A$, $\Gamma \vdash M_2 : \langle\varepsilon\rangle A$ and $\Gamma, x : A \vdash N : \underline{C}$ then
$$(M_1\ \mathbf{or}\ M_2)\ \mathbf{to}\ x.\,N \ \equiv\ (M_1\ \mathbf{to}\ x.\,N)\ \mathbf{or}\ (M_2\ \mathbf{to}\ x.\,N) \qquad ◀$$

This completes the inequational theory we use for our nondeterminism example.

We consider a transformation that reuses the result of a duplicated computation:

$$M\ \mathbf{to}\ x.\,M\ \mathbf{to}\ y.\,N \ \preccurlyeq_{\mathrm{ctx}}\ M\ \mathbf{to}\ x.\,N[y \mapsto x]$$

Validity of this transformation is shown in later sections of this chapter. Benton et al. [11] consider a similar transformation when M is deterministic; in this case, the reverse direction also holds (if M has effect 1, then $\cong_{\mathrm{ctx}}$ holds). Their equational framework cannot validate this duplicated computation transformation when nondeterminism is allowed because it is not invertible (the right-to-left direction is not correct in general). Our framework can. We can also validate other transformations involving nondeterminism, for example, dead code elimination:

$$M;\ N \ \cong_{\mathrm{ctx}}\ N$$

4.1.3 Shared global state

We also consider a concurrency example. We again use mutable global state as a source of side-effects, but in this case assume the state can be accessed by multiple threads, which requires us to use directed versions of the signature axioms for global state. We do not attempt to fully characterize a concurrent language in this example. Instead, we focus only on a single thread (assuming there might be other threads that can arbitrarily access and mutate the state), and do not include any concurrency primitives (e.g. creation of new threads). The idea is that we can validate transformations on subprograms that use the shared state but do not use any concurrency primitives. For simplicity, the state is the only source of side-effects in this example.

The signature for this example consists of the following data:

$$\begin{array}{rcl}
\mathrm{get}\,()\ \mathbf{to}\ x.\,(\mathrm{put}\,x;\ \langle x\rangle) & \preccurlyeq & \mathbf{coerce}_{\{\mathrm{get}\}\leq\{\mathrm{get},\mathrm{put}\}}\,(\mathrm{get}\,()) \\
\mathrm{put}\,V;\ \mathrm{get}\,() & \preccurlyeq & \mathrm{put}\,V;\ \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle V\rangle \\
\mathrm{put}\,V_1;\ \mathrm{put}\,V_2 & \preccurlyeq & \mathrm{put}\,V_2 \\
\mathrm{get}\,()\ \mathbf{to}\ x.\,\mathrm{get}\,()\ \mathbf{to}\ y.\,\langle(x,y)\rangle & \preccurlyeq & \mathrm{get}\,()\ \mathbf{to}\ x.\,\langle(x,x)\rangle \\
\mathrm{get}\,();\ \langle()\rangle & \equiv & \mathbf{coerce}_{\emptyset\leq\{\mathrm{get}\}}\,\langle()\rangle
\end{array}$$

Figure 4.2: Signature axioms for shared global state

- We use a Gifford-style effect algebra (Example 2.1.2) with set of operations $\Sigma := \{\mathrm{get}, \mathrm{put}\}$, so that effects $\varepsilon \subseteq \Sigma$ specify the operations that may be used.
- There are no base types or constants.
- There are two operations:

Operation op	Coarity $\mathrm{car}_{\mathrm{op}}$	Arity $\mathrm{ar}_{\mathrm{op}}$	Effect $\mathrm{eff}_{\mathrm{op}}$
get	**unit**	**bool**	$\{\mathrm{get}\}$
put	**bool**	**unit**	$\{\mathrm{put}\}$

The signature axioms of the inequational theory are given in Figure 4.2. The axioms are the same as those for non-shared global state (Figure 2.14), except that four of them are directed. The reverse directions introduce data races if another thread is accessing the state concurrently and hence they are not included.

We validate similar program transformations to the previous example: subject to suitable restrictions on effects, it is sound to reuse the result of a duplicated computation, and eliminate dead computations. Specifically, if M has type $\langle\varepsilon\rangle A$ where $\varepsilon \subseteq \{\mathrm{put}\}$, and M' has type $\langle\varepsilon'\rangle A'$ where $\varepsilon' \subseteq \{\mathrm{get}\}$, then the following hold:

$$M\ \mathbf{to}\ x.\,M\ \mathbf{to}\ y.\,N\ \preccurlyeq_{\mathrm{ctx}}\ M\ \mathbf{to}\ x.\,N[y\mapsto x] \qquad\qquad M';\ N\ \cong_{\mathrm{ctx}}\ N$$

The instance on the left does not hold when the effect of M is $\{\mathrm{get}\}$, because we do not allow reordering of consecutive uses of get. We could instead capture a weaker memory model by including a signature axiom that allows this reordering; in this case, the program transformation would be valid. The instance on the left is also not symmetric (so the transformation is not invertible). This is unlike non-shared state, in which it is symmetric.

4.2 Order-enriched semantics of GCBPV

The goal of this chapter is to prove instances of contextual preorders $\preccurlyeq_{\mathrm{ctx}}$. It is difficult to prove these directly so, as in the previous chapter, we develop some extra machinery. Unlike the previous chapter, here we focus on denotational semantics. This is primarily because denotational models are easier to reason about than syntactic logical relations.

A common way of proving contextual equivalences is to give an adequate model of the chosen side-effects, so that it suffices to show that denotations of terms are equal in the model. In our case, $\preccurlyeq_{\mathrm{ctx}}$ is not symmetric, so equality between denotations is not suitable. We therefore use *order-enriched* models, which come with partial orders $\sqsubseteq$ between denotations.

We have three examples, which need to be modelled in different settings. There are also other examples of side-effects (e.g. local state [88] and probability [37]) that require other settings. We aim to provide a general *categorical* semantics for GCBPV that can be applied to as many of these as possible. Our semantics is based on *adjunction models* for CBPV [58], which we extend with grading and with order-enrichment.[1] The order $\sqsubseteq$ on morphisms is the semantic counterpart of the contextual preorder.

We use some order-enriched category theory to describe the semantics (see Appendix A for background).

4.2.1 Graded adjunctions

The data required for a model of GCBPV closely mirrors the syntax. There are two kinds of types, and therefore we use two **Poset**-categories: a *value category* **C** and a *computation category* **D**. Value types are interpreted as objects of **C**, and computation types are interpreted as objects of **D**. We use $X, Y, \ldots$ to refer to objects of **C** and $\underline{X}, \underline{Y}, \ldots$ for objects of **D**.

The key component in each model is the data required to interpret thunk types and returner types. In ordinary CBPV, there is a single type former **F** for returner types (because there is no grading), and in the semantics this is left adjoint to the type former **U** for thunk types:

$$\begin{tikzcd} \mathbf{C} \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] & \mathbf{D} \arrow[l, bend left, "U"] \end{tikzcd}$$

(Of course, we actually need a *strong* adjunction; we add strength below.) We have an adjunction because for each value of type A we can form a computation of type $\mathbf{F}\,A$ (by returning it), and any computation $\Gamma, x : A \underline{\vdash} M : \underline{C}$ can be *extended* to a computation

$$\Gamma, y : \mathbf{U}\,\mathbf{F}\,A \underline{\vdash} \mathbf{force}\, y \ \mathbf{to}\ x.\, M : \underline{C}$$

that satisfies certain properties. To add grading, we note that this is still the case with the returner type $\langle 1 \rangle A$ (i.e. with the unit effect). We therefore use the left adjoint F as the interpretation of $\langle 1 \rangle A$. For returner types annotated with other effects, we note that they can be written using $\langle 1 \rangle$ and the *action* of the effect algebra on computation types (see Definition 2.7.3), as $\langle \varepsilon \rangle A = \langle\!\langle \varepsilon \rangle\!\rangle \langle 1 \rangle A$. Hence the remaining structure we need to interpret the other returner types is an action $\varepsilon \circledast -$ of the preordered monoid of effects on the computation category **D**.

Definition 4.2.1 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid, and **D** is a **Poset**-category. A *strict* **Poset**-*action* of $\mathcal{E}$ on **D** is a **Poset**-functor $\circledast : \mathcal{E} \times \mathbf{D} \to \mathbf{D}$ that respects the unit and multiplication:

$$1 \circledast - \ = \ Id_{\mathbf{D}} \qquad (\varepsilon \cdot \varepsilon') \circledast - \ = \ \varepsilon \circledast (\varepsilon' \circledast -)$$ ◀

Here, and throughout this chapter, we treat the preordered monoid as a **Poset**-category with set of objects $\mathcal{E}$, and a single morphism from ε to ε' if $\varepsilon \leq \varepsilon'$ (we write this morphism as $\varepsilon \leq \varepsilon'$). The ordering on morphisms is equality.

Each strict **Poset**-action comes in particular with morphisms $(\varepsilon \leq \varepsilon') \circledast \underline{X}$ from $\varepsilon \circledast \underline{X}$ to $\varepsilon' \circledast \underline{X}$ whenever $\varepsilon \leq \varepsilon'$. This corresponds to the subtyping $\langle\!\langle \varepsilon \rangle\!\rangle \underline{C} <: \langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}$.

We call the combination of an adjunction and an action a *graded* **Poset**-*adjunction*. This definition is implicit in [28] (which considers *resolutions* of graded monads).

Definition 4.2.2 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid, and that **C** and **D** are **Poset**-categories. An *($\mathcal{E}$-)graded* **Poset**-*adjunction* is a triple $(F, U, \circledast)$ where:

[1] Though unlike Levy, we do not define adjunction models in terms of *locally indexed* categories, instead we use more a more elementary (but less elegant) description.

- $F : \mathbf{C} \longrightarrow \mathbf{D}$ and $U : \mathbf{D} \longrightarrow \mathbf{C}$ form a **Poset**-adjunction $F \dashv U$, with unit $\eta : Id_{\mathbf{C}} \longrightarrow U \circ F$ and counit $\varpi : F \circ U \longrightarrow Id_{\mathbf{D}}$.
- $\circledast : \mathcal{E} \times \mathbf{D} \longrightarrow \mathbf{D}$ is a strict $\mathcal{E}$-action on $\mathbf{D}$. ◀

The counit induces an extension operator: every **C**-morphism $f : X \longrightarrow U\underline{Y}$ can be extended to a **C**-morphism

$$U(\varepsilon \circledast FX) \xrightarrow{U(\varepsilon \circledast Ff)} U(\varepsilon \circledast F(U\underline{Y})) \xrightarrow{U(\varepsilon \circledast \varpi)} U\underline{Y}$$

This is similar to the extension of computations in context $\Gamma, x : A$ to computations in context $\Gamma, x : \mathbf{U}\,\mathbf{F}\,A$ mentioned above, except for the extra context Γ. This extension operator is enough to interpret *closed* computations M **to** $x.\,N$, but because there is no Γ, it cannot interpret this computation if it is open. We require a more general extension operator than this, and to define it we need a strong adjunction. (Strength appears for exactly the same reason as it does for models of the monadic metalanguage [78].)

Definition 4.2.3 (Graded strong Poset-adjunction) Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid, **C** is a cartesian **Poset**-category, and **D** is a **Poset**-category. An *graded strong* **Poset**-*adjunction* consists of a graded **Poset**-adjunction $(F, U, \circledast)$ and a family of morphisms

$$str_{\varepsilon,X,Y} : X \times U(\varepsilon \circledast FY) \longrightarrow U(\varepsilon \circledast F(X \times Y))$$

indexed by $\varepsilon \in \mathcal{E}$ and $X, Y \in \mathbf{C}$, such that:

- For each ε, the pair $(U(\varepsilon \circledast F-), str_\varepsilon)$ is a strong **Poset**-functor $\mathbf{C} \longrightarrow \mathbf{C}$.
- The unit η is a strong natural transformation $Id_{\mathbf{C}} \longrightarrow (U \circ F, str_1)$.
- For each $\varepsilon, \varepsilon'$ the family of morphisms $U(\varepsilon \circledast \varpi_{(\varepsilon' \circledast F-)})$ is a strong natural transformation
$$(U(\varepsilon \circledast F-),\ str_\varepsilon) \circ (U(\varepsilon' \circledast F-),\ str_{\varepsilon'}) \longrightarrow (U((\varepsilon \cdot \varepsilon') \circledast F-),\ str_{\varepsilon \cdot \varepsilon'})$$
- The strength is natural in ε: if $\varepsilon \leq \varepsilon'$ then
$$\begin{tikzcd} X \times U(\varepsilon \circledast FY) \arrow[r, "str_{\varepsilon,X,Y}"] \arrow[d, "X \times U((\varepsilon \leq \varepsilon') \circledast FY)"'] & U(\varepsilon \circledast F(X \times Y)) \arrow[d, "U((\varepsilon \leq \varepsilon') \circledast F(X \times Y))"] \\ X \times U(\varepsilon' \circledast FY) \arrow[r, "str_{\varepsilon',X,Y}"'] & U(\varepsilon' \circledast F(X \times Y)) \end{tikzcd}$$
commutes for all X, Y. ◀

The strength enables us to define a more general version of the extension operator above. Given a **C**-morphism $f : Z \times X \longrightarrow U\underline{Y}$, define its extension $f^\dagger : Z \times U(\varepsilon \circledast FX) \longrightarrow U(\varepsilon \circledast \underline{Y})$ as

$$Z \times U(\varepsilon \circledast FX) \xrightarrow{str_{\varepsilon,Z,X}} U(\varepsilon \circledast F(Z \times X)) \xrightarrow{U(\varepsilon \circledast Ff)} U(\varepsilon \circledast F(U\underline{Y})) \xrightarrow{U(\varepsilon \circledast \varpi_{\underline{Y}})} U(\varepsilon \circledast \underline{Y})$$

In the semantics, we use this extension operator to interpret open computations M **to** $x.\,N$ in typing context Γ. The object Z is the interpretation of Γ.

Each graded **Poset**-adjunction with $\mathbf{C} = \mathbf{Set}$ or $\mathbf{C} = \mathbf{Poset}$ forms a graded strong **Poset**-adjunction in exactly one way. The strength in both of these cases is given by:

$$str_{\varepsilon,X,Y}(x, t) = U(\varepsilon \circledast (F(y \mapsto (x, y))))\, t$$

For $\mathbf{C} = \omega\mathbf{Cpo}$, if a strength exists it is also given by this formula, and this formula defines a strength when the functions $U(\varepsilon \circledast F-) : \mathbf{C}(X, Y) \longrightarrow \mathbf{D}(U(\varepsilon \circledast FX), U(\varepsilon \circledast FY))$ preserve least upper bounds.[2] These facts are useful when defining models based on **Set**, **Poset** or ω**Cpo**.

[2] This characterization of strengths for **Set**, **Poset** and ω**Cpo** follows from the fact that tensorial strengths correspond to enrichments of functors [50]. One can also show that if **C** is *well-pointed*, then strengths are uniquely determined by the formula above.

4.2.2 Models of GCBPV

We are almost ready to give the definition of GCBPV *model*. We do this formally below (Definition 4.2.7), but first give an informal description. In addition to the graded adjunction, we need data to interpret the rest of the type formers. On the value level we have product and sum types, so we ask for the value category $\mathbf{C}$ to be bicartesian, and require distributivity. For computations we have product types, and so we ask for the computation category $\mathbf{D}$ to be cartesian. Since for types we have $\langle\!\langle \varepsilon \rangle\!\rangle \underline{\mathbf{unit}} = \underline{\mathbf{unit}}$ and $\langle\!\langle \varepsilon \rangle\!\rangle (\underline{C}_1 \underline{\times} \underline{C}_2) = \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_1 \underline{\times} \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_2$ we also require the action $\circledast$ to *strictly preserve* the cartesian structure of $\mathbf{D}$. In particular, we require

$$\varepsilon \circledast \underline{1} \;=\; \underline{1} \qquad \varepsilon \circledast (\underline{X}_1 \underline{\times} \underline{X}_2) \;=\; (\varepsilon \circledast \underline{X}_1) \underline{\times} (\varepsilon \circledast \underline{X}_1)$$

where we write the cartesian structure of $\mathbf{D}$ with an underline.

Functions in GCBPV map values to computations, and are computations themselves. To interpret these we use a **Poset**-functor $X \Rightarrow - : \mathbf{D} \to \mathbf{D}$ for each object $X \in \mathbf{C}$, with *currying* functions Λ and *uncurrying* functions Λ^{-1}:

$$\Lambda \;:\; \mathbf{C}(Z \times X, U\underline{Y}) \;\cong\; \mathbf{C}(Z, U(X \Rightarrow \underline{Y})) \;:\; \Lambda^{-1}$$

As for products, we require exponentials to strictly preserve the action, in particular,

$$\varepsilon \circledast (X \Rightarrow \underline{Y}) \;=\; X \Rightarrow (\varepsilon \circledast \underline{Y})$$

We also impose an additional requirement on the exponentials $X \Rightarrow -$. To explain this we define a notion of *linearity*[3] for morphisms of $\mathbf{C}$. The definition uses the *right* strength $\mathit{str}^{\mathrm{r}}$ (which is the *left* strength str with the products reversed):

$$\mathit{str}^{\mathrm{r}}_{\varepsilon,X,Y} \;:=\; U(\varepsilon \circledast F\langle \pi_2, \pi_1 \rangle) \circ \mathit{str}_{\varepsilon,Y,X} \circ \langle \pi_2, \pi_1 \rangle \;:\; U(\varepsilon \circledast FX) \times Y \to U(\varepsilon \circledast F(X \times Y))$$

Definition 4.2.4 A natural transformation $\alpha_\varepsilon : U(\varepsilon \circledast \underline{X}) \to U(\varepsilon \circledast \underline{Y})$ is *linear* if the following diagram commutes

$$\begin{tikzcd}
U(\varepsilon \circledast FU(\varepsilon' \circledast \underline{X})) \arrow[r, "U(\varepsilon \circledast F\alpha_{\varepsilon'})"] \arrow[d, "U(\varepsilon \circledast \varpi_{\varepsilon' \circledast \underline{X}})"'] & U(\varepsilon \circledast FU(\varepsilon' \circledast \underline{Y})) \arrow[d, "U(\varepsilon \circledast \varpi_{\varepsilon' \circledast \underline{Y}})"] \\
U((\varepsilon \cdot \varepsilon') \circledast \underline{X}) \arrow[r, "\alpha_{\varepsilon \cdot \varepsilon'}"'] & U((\varepsilon \cdot \varepsilon') \circledast \underline{Y})
\end{tikzcd}$$

for all $\varepsilon, \varepsilon' \in \mathcal{E}$.

A natural transformation $\alpha_\varepsilon : U(\varepsilon \circledast \underline{X}) \times Z \to U(\varepsilon \circledast \underline{Y})$ is *left-linear* if the following diagram commutes

$$\begin{tikzcd}
U(\varepsilon \circledast FU(\varepsilon' \circledast \underline{X})) \times Z \arrow[r, "\mathit{str}^{\mathrm{r}}"] \arrow[d, "U(\varepsilon \circledast \varpi_{\varepsilon' \circledast \underline{X}}) \times Z"'] & U(\varepsilon \circledast F(U(\varepsilon' \circledast \underline{X}) \times Z)) \arrow[r, "U(\varepsilon \circledast F\alpha_{\varepsilon'})"] & U(\varepsilon \circledast FU(\varepsilon' \circledast \underline{Y})) \arrow[d, "U(\varepsilon \circledast \varpi_{\varepsilon' \circledast \underline{Y}})"] \\
U((\varepsilon \cdot \varepsilon') \circledast \underline{X}) \times Z \arrow[rr, "\alpha_{\varepsilon \cdot \varepsilon'}"'] & & U((\varepsilon \cdot \varepsilon') \circledast \underline{Y})
\end{tikzcd}$$

for all $\varepsilon, \varepsilon' \in \mathcal{E}$. ◂

[3] The word linear comes from Kock [49]. A similar property for syntax (see Section 4.2.3.5) is also called linearity by Munch-Maccagnoni [79] (and by Levy [60] for call-by-push-value).

$$
\boxed{\llbracket A \rrbracket \in \mathbf{C}}
\qquad
\boxed{\llbracket \underline{C} \rrbracket \in \mathbf{D}}
\qquad
\boxed{\llbracket \Gamma \rrbracket \in \mathbf{C}}
$$

$$
\begin{array}{rcl}
\llbracket \mathbf{unit} \rrbracket & := & 1 \\
\llbracket A_1 \times A_2 \rrbracket & := & \llbracket A_1 \rrbracket \times \llbracket A_2 \rrbracket \\
\llbracket \mathbf{empty} \rrbracket & := & 0 \\
\llbracket A_1 + A_2 \rrbracket & := & \llbracket A_1 \rrbracket + \llbracket A_2 \rrbracket \\
\llbracket \mathbf{U}\,\underline{C} \rrbracket & := & U\llbracket \underline{C} \rrbracket
\end{array}
\qquad
\begin{array}{rcl}
\llbracket \underline{\mathbf{unit}} \rrbracket & := & \underline{1} \\
\llbracket \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 \rrbracket & := & \llbracket \underline{C}_1 \rrbracket \mathbin{\underline{\times}} \llbracket \underline{C}_2 \rrbracket \\
\llbracket A \to \underline{C} \rrbracket & := & \llbracket A \rrbracket \Rightarrow \llbracket \underline{C} \rrbracket \\
\llbracket \langle \varepsilon \rangle A \rrbracket & := & \varepsilon \circledast F\llbracket A \rrbracket
\end{array}
\qquad
\begin{array}{rcl}
\llbracket \diamond \rrbracket & := & 1 \\
\llbracket \Gamma, x : A \rrbracket & := & \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket
\end{array}
$$

$$
\boxed{\llbracket \Gamma \vdash V : A \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket}
\qquad
\boxed{\llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket : \llbracket \Gamma \rrbracket \to U\llbracket \underline{C} \rrbracket}
$$

$$
\begin{array}{rcl}
\llbracket c \rrbracket & := & \llbracket c \rrbracket \circ \langle\rangle_{\llbracket \Gamma \rrbracket} \\
\llbracket x \rrbracket & := & \pi_x \\
\llbracket () \rrbracket & := & \langle\rangle_{\llbracket \Gamma \rrbracket} \\
\llbracket (V_1, V_2) \rrbracket & := & \langle \llbracket V_1 \rrbracket, \llbracket V_2 \rrbracket \rangle \\
\llbracket \mathbf{fst}\, V \rrbracket & := & \pi_1 \circ \llbracket V \rrbracket \\
\llbracket \mathbf{snd}\, V \rrbracket & := & \pi_2 \circ \llbracket V \rrbracket \\
\llbracket \mathbf{case}_A\ V\ \mathbf{of}\ \{\} \rrbracket & := & []_{\llbracket A \rrbracket} \circ \llbracket V \rrbracket \\
\llbracket \mathbf{inl}\, V \rrbracket & := & \mathit{inl} \circ \llbracket V \rrbracket \\
\llbracket \mathbf{inr}\, V \rrbracket & := & \mathit{inr} \circ \llbracket V \rrbracket \\
\left\llbracket \begin{array}{l} \mathbf{case}\ V\ \mathbf{of} \\ \quad \{\mathbf{inl}\ x_1.\, W_1 \\ \quad ,\ \mathbf{inr}\ x_2.\, W_2\} \end{array} \right\rrbracket & := & \left(\begin{array}{l} [\llbracket W_1 \rrbracket, \llbracket W_2 \rrbracket] \\ \circ\ \mathit{dist} \\ \circ\ \langle \mathit{id}, \llbracket V \rrbracket \rangle \end{array} \right) \\
\llbracket \mathbf{thunk}\, M \rrbracket & := & \llbracket M \rrbracket
\end{array}
\qquad
\begin{array}{rcl}
\llbracket \lambda\{\} \rrbracket & := & \cong \circ \langle\rangle_{\llbracket \Gamma \rrbracket} \\
\llbracket \lambda\{1.\, M_1, 2.\, M_2\} \rrbracket & := & \cong \circ \langle \llbracket M_1 \rrbracket, \llbracket M_2 \rrbracket \rangle \\
\llbracket i`M \rrbracket & := & U\pi_i \circ \llbracket M \rrbracket \qquad (i \in \{1, 2\}) \\
\llbracket \lambda x {:} A.\, M \rrbracket & := & \Lambda \llbracket M \rrbracket \\
\llbracket V`M \rrbracket & := & \Lambda^{-1} \llbracket M \rrbracket \circ \langle \mathit{id}, \llbracket V \rrbracket \rangle \\
\llbracket \mathrm{op}\, V \rrbracket & := & \llbracket \mathrm{op} \rrbracket \circ \llbracket V \rrbracket \\
\llbracket \langle V \rangle \rrbracket & := & \eta \circ \llbracket V \rrbracket \\
\llbracket M\ \mathbf{to}\ x.\, N \rrbracket & := & \llbracket N \rrbracket^{\dagger} \circ \langle \mathit{id}, \llbracket M \rrbracket \rangle \\
\llbracket \mathbf{coerce}_{\varepsilon \le \varepsilon'}\, M \rrbracket & := & U((\varepsilon \le \varepsilon') \circledast F\llbracket A \rrbracket) \circ \llbracket M \rrbracket \\
\llbracket \mathbf{force}\, V \rrbracket & := & \llbracket V \rrbracket
\end{array}
$$

Figure 4.3: Denotational semantics of GCBPV

The intuition is that (left-)linearity means α uses the $U(\varepsilon \circledast \underline{X})$ argument exactly once, and does not add any other side-effects.

Using the data for exponentials above we define a natural transformation *ev* for function application:

$$ev_{\varepsilon, X, \underline{Y}} \ := \ \Lambda^{-1} id \ : \ U(\varepsilon \circledast (X \Rightarrow \underline{Y})) \times X \to U(\varepsilon \circledast \underline{Y})$$

The extra requirement is that this natural transformation is left-linear. This is equivalent to requiring $\Lambda^{-1}\alpha_\varepsilon$ to be left-linear for all linear α.

The remaining data required to interpret GCBPV is an interpretation $\llbracket b \rrbracket \in \mathbf{C}$ of each base type b, an interpretation $\llbracket c \rrbracket : 1 \to \llbracket A \rrbracket$ of each constant c of type A, and an interpretation $\llbracket \mathrm{op} \rrbracket : \mathrm{car}_{\mathrm{op}} \to U(\mathrm{eff}_{\mathrm{op}} \circledast F\mathrm{ar}_{\mathrm{op}})$ of each operation op.

Using this data, we give the denotational semantics of GCBPV in Figure 4.3. Each value type A is interpreted as an object $\llbracket A \rrbracket$ is the value category $\mathbf{C}$, and each computation type $\underline{C}$ is interpreted as an object $\llbracket \underline{C} \rrbracket$ of the computation category $\mathbf{D}$. The strict preservation requirements we impose imply that $\llbracket \langle\!\langle \varepsilon \rangle\!\rangle \underline{C} \rrbracket = \varepsilon \circledast \llbracket D \rrbracket$, and so $\llbracket M\ \mathbf{to}\ x.\, N \rrbracket$ is well-defined. Typing contexts Γ are interpreted as objects $\llbracket \Gamma \rrbracket \in \mathbf{C}$ using the terminal object and binary products. If $(x : A) \in \Gamma$ then we write $\pi_x : \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket$ for the corresponding projection.

The interpretation of terms is given on the bottom of Figure 4.3. Values $\Gamma \vdash V : A$ are interpreted as morphisms $[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ in the value category **C**. Computations $\Gamma \underline{\vdash} M : \underline{C}$ are also interpreted in the value category, as morphisms $[\![\Gamma \underline{\vdash} M : \underline{C}]\!] : [\![\Gamma]\!] \to U[\![\underline{C}]\!]$. (Intuitively, we interpret computations as morphisms of **C** because morphisms in the computation category **D** are effect-preserving, while computations are not. The interpretation of a computation M is also not in general linear.) We often omit the typing context and type from denotations of terms. The two isomorphisms $\cong$ in Figure 4.3 are the inverses of the following morphisms:

$$\langle\rangle_{U\underline{1}} : U\underline{1} \to 1 \qquad \langle U\pi_1, U\pi_2\rangle : U(\underline{X}_1 \mathbin{\underline{\times}} \underline{X}_2) \to U\underline{X}_1 \times U\underline{X}_2$$

These inverses exist because U is a right adjoint. In adjunction models, **thunk** and **force** are invisible because computations are interpreted in **C**. The interpretation of **to** uses the general extension operator $(-)^\dagger$ defined in Section 4.2.1. Computations of returner type are coerced using $(\varepsilon \leq \varepsilon')\circledast(-)$.

We collect together all of the data required for the semantics into the notion of GCBPV *structure.*

Definition 4.2.5 (GCBPV structure) Given some GCBPV signature, a *structure* consists of

- A distributive **Poset**-category **C** and a cartesian **Poset**-category **D**.
- A **Poset**-functor $X \Rightarrow - : \mathbf{D} \to \mathbf{D}$ for each $X \in \mathbf{C}$, together with a family of bijections

$$\Lambda \ : \ \mathbf{C}(Z \times X, U\underline{Y}) \ \cong \ \mathbf{C}(Z, U(X \Rightarrow \underline{Y})) \ : \ \Lambda^{-1}$$

  natural in $Z \in \mathbf{C}$ and $\underline{Y} \in \mathbf{D}$.
- A graded strong **Poset**-adjunction $(F, U, \circledast)$, where $F : \mathbf{C} \to \mathbf{D}$ and $U : \mathbf{D} \to \mathbf{C}$, such that

$$\begin{aligned}
\varepsilon\circledast\underline{1} &= \underline{1} \\
\varepsilon\circledast(\underline{X}_1 \mathbin{\underline{\times}} \underline{X}_2) &= (\varepsilon\circledast\underline{X}_1) \mathbin{\underline{\times}} (\varepsilon\circledast\underline{X}_2) \qquad && \langle \varepsilon\circledast\pi_1, \varepsilon\circledast\pi_2\rangle = id_{\varepsilon\circledast(\underline{X}_1 \underline{\times} \underline{X}_2)} \\
\varepsilon\circledast(X \Rightarrow \underline{Y}) &= X \Rightarrow (\varepsilon\circledast\underline{Y}) && ev \text{ is left-linear}
\end{aligned}$$

- An object $[\![b]\!] \in \mathbf{C}$ for each base type b.
- A morphism $[\![c]\!] : 1 \to [\![A]\!]$ for each constant $c \in \mathcal{K}_A$, and a morphism $[\![\mathrm{op}]\!] : [\![\mathrm{car}_{\mathrm{op}}]\!] \to U(\mathrm{eff}_{\mathrm{op}}\circledast F[\![\mathrm{ar}_{\mathrm{op}}]\!])$ for each operation op $\in \Sigma$. ◀

Given any such structure, the denotational semantics defined in Figure 4.3 satisfies some useful properties. Recall that by definition, each GCBPV inequational theory is required to satisfy congruence and substitution properties, and also to respect the core axioms given in Figure 2.13. The denotational semantics satisfies similar properties. In the following lemma, we interpret well-typed substitutions $\Gamma \vdash \sigma : \Delta$, as **C**-morphisms $[\![\Gamma \vdash \sigma : \Delta]\!] : [\![\Gamma]\!] \to [\![\Delta]\!]$, given by tupling their components.

Lemma 4.2.6 Given any GCBPV structure, the following properties hold:

1. Compositionality:
   - If $[\![\Gamma \vdash V : A]\!] \sqsubseteq [\![\Gamma \vdash W : A]\!]$ then for term contexts with hole $\square$,

$$\begin{aligned}
\Gamma' \vdash \mathcal{C}[V] : B \wedge \Gamma' \vdash \mathcal{C}[W] : B \quad &\Rightarrow \quad [\![\Gamma \vdash \mathcal{C}[V] : B]\!] \sqsubseteq [\![\Gamma \vdash \mathcal{C}[W] : B]\!] \\
\Gamma' \underline{\vdash} \underline{\mathcal{C}}[V] : \underline{D} \wedge \Gamma' \underline{\vdash} \underline{\mathcal{C}}[W] : \underline{D} \quad &\Rightarrow \quad [\![\Gamma' \underline{\vdash} \underline{\mathcal{C}}[V] : \underline{D}]\!] \sqsubseteq [\![\Gamma' \underline{\vdash} \underline{\mathcal{C}}[W] : \underline{D}]\!]
\end{aligned}$$

- If $\llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket \sqsubseteq \llbracket \Gamma \mathrel{\underline{\vdash}} N : \underline{C} \rrbracket$ then for term contexts with hole $\underline{\square}$,

$$\begin{array}{lcl} \Gamma' \vdash \mathcal{C}[M] : B \land \Gamma' \vdash \mathcal{C}[N] : B & \Rightarrow & \llbracket \Gamma \vdash \mathcal{C}[M] : B \rrbracket \sqsubseteq \llbracket \Gamma \vdash \mathcal{C}[N] : B \rrbracket \\ \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] : \underline{D} \land \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[N] : \underline{D} & \Rightarrow & \llbracket \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] : \underline{D} \rrbracket \sqsubseteq \llbracket \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[N] : \underline{D} \rrbracket \end{array}$$

2. Substitution: if $\Gamma \vdash \sigma : \Delta$ then:

$$\begin{array}{lcl} \Delta \vdash V : A & \Rightarrow & \llbracket \Gamma \vdash V[\sigma] : A \rrbracket = \llbracket \Delta \vdash V : A \rrbracket \circ \llbracket \Gamma \vdash \sigma : \Delta \rrbracket \\ \Delta \mathrel{\underline{\vdash}} M : \underline{C} & \Rightarrow & \llbracket \Gamma \mathrel{\underline{\vdash}} M[\sigma] : \underline{C} \rrbracket = \llbracket \Delta \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket \circ \llbracket \Gamma \vdash \sigma : \Delta \rrbracket \end{array}$$

Hence if $\llbracket \Gamma \vdash \sigma : \Delta \rrbracket \sqsubseteq \llbracket \Gamma \vdash \sigma' : \Delta \rrbracket$ then

$$\begin{array}{lcl} \llbracket \Delta \vdash V : A \rrbracket \sqsubseteq \llbracket \Delta \vdash W : A \rrbracket & \Rightarrow & \llbracket \Gamma \vdash V[\sigma] : A \rrbracket \sqsubseteq \llbracket \Gamma \vdash W[\sigma'] : A \rrbracket \\ \llbracket \Delta \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket \sqsubseteq \llbracket \Delta \mathrel{\underline{\vdash}} N : \underline{C} \rrbracket & \Rightarrow & \llbracket \Gamma \mathrel{\underline{\vdash}} M[\sigma] : \underline{C} \rrbracket \sqsubseteq \llbracket \Gamma \mathrel{\underline{\vdash}} N[\sigma'] : \underline{C} \rrbracket \end{array}$$

3. Core axioms:
   - Values: if $\Gamma \vdash V : A$ and $\Gamma \vdash W : A$, and $V \equiv W$ is an instance of an axiom in Figure 2.13, then $\llbracket \Gamma \vdash V : A \rrbracket = \llbracket \Gamma \vdash W : A \rrbracket$.
   - Computations: if $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ and $\Gamma \mathrel{\underline{\vdash}} N : \underline{C}$, and $M \equiv N$ is an instance of an axiom in Figure 2.13, then $\llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket = \llbracket \Gamma \mathrel{\underline{\vdash}} N : \underline{C} \rrbracket$.

Proof sketch. Each part of this lemma is proved separately.

For compositionality, the proof is by induction on the structure of the term contexts $\mathcal{C}[]$ and $\underline{\mathcal{C}}[]$. Each case follows immediately from monotonicity (for example of the copairing operations $\langle -, - \rangle$).

For substitution, we first show a weakening lemma: there is an evident morphism

$$\llbracket \Gamma, x : A, \Gamma' \rrbracket \to \llbracket \Gamma, \Gamma' \rrbracket$$

and if $\Gamma, \Gamma' \mathrel{\underline{\vdash}} M : \underline{C}$ then the interpretation of $\Gamma, x : A, \Gamma' \mathrel{\underline{\vdash}} M : \underline{C}$ is equal to

$$\llbracket \Gamma, x : A, \Gamma' \rrbracket \longrightarrow \llbracket \Gamma, \Gamma' \rrbracket \xrightarrow{\llbracket \Gamma, \Gamma' \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket} \llbracket \underline{C} \rrbracket$$

There is a similar fact for values. We then show the substitution lemma by induction on V and M, weakening substitutions where necessary. The second part of substitution follows immediately from the first part and monotonicity of $\circ$.

For the core axioms we go case-by-case. Most follow from the universal properties of the structure used to interpret each type former. For the axioms that involve substitution we use the previous part of this lemma. For the axiom $\lambda y : A.\, M$ **to** $x.\, N \equiv M$ **to** $x.\, \lambda y : A.\, N$ we use left-linearity of *ev*. □

This lemma implies that the denotational semantics is sound with respect to the smallest inequational theory, in which the core axioms are the only axioms. It is not in general sound with respect to an arbitrary inequational theory. For our three examples in Section 4.1, we give signature axioms in addition to the core axioms, and define the inequational theory by closing under reflexivity, transitivity and congruence. The above lemma implies that to check soundness with respect to an inequational theory defined in this way, we need only check the signature axioms.

We call a structure a *model* if it is sound with respect to some given inequational theory.

Definition 4.2.7 (GCBPV model) A *model* of some GCBPV inequational theory $\preccurlyeq$ is a structure such that the interpretation of terms is *sound*: if $\Gamma \vdash V \preccurlyeq W : A$ then $[\![\Gamma \vdash V : A]\!] \sqsubseteq [\![\Gamma \vdash W : A]\!]$, and if $\Gamma \,\underline{\vdash}\, M \preccurlyeq N : \underline{C}$ then $[\![\Gamma \,\underline{\vdash}\, M : \underline{C}]\!] \sqsubseteq [\![\Gamma \,\underline{\vdash}\, N : \underline{C}]\!]$. ◀

Each model induces a semantic notion of validity of program transformations: a transformation is valid in a model if it only replaces M with N when $[\![M]\!] \sqsubseteq [\![N]\!]$.

4.2.3 Constructing models

We give various models of GCBPV. First we present the three models we use to verify the example program transformations in Section 4.1, one each for undefined behaviour, nondeterminism and shared global state. We then describe how to construct *monadic* models of GCBPV. Finally we describe the term model, which we use for adequacy proofs in Section 4.3.

4.2.3.1 Undefined behaviour

Our undefined behaviour example has the simplest model. We interpret value types as posets (in the value category $\mathbf{C} = \mathbf{Poset}$), and computation types as pointed posets (in the computation category $\mathbf{D} = \mathbf{Poset}_\perp$). The least element of each object of $\mathbf{Poset}_\perp$ represents undefined behaviour. We use the usual bicartesian structure of $\mathbf{C}$ and cartesian structure of $\mathbf{D}$.

For the graded adjunction, the left adjoint sends each poset X to the pointed poset $X_\perp$ obtained by freely adding a least element, and sends each monotone function $f : X \longrightarrow Y$ to its strict extension $f_\perp : X_\perp \longrightarrow Y_\perp$ (so $f_\perp\, \perp := \perp$ and $f_\perp\, x := x$ for $x \in X$). The right adjoint is the forgetful functor, which sends each pointed poset to itself. The unit $\eta_X : X \longrightarrow X_\perp$ maps each element of X to itself, and the counit $\varpi_X : (X_\perp)_\perp \longrightarrow X_\perp$ merges the two bottom elements. For this example, the effect algebra is the trivial preordered monoid with underlying set $\{\star\}$. Hence the action $\star \circledast -$ is just the identity. Since the value category is $\mathbf{Poset}$, strength is also trivial in this case: the strength is given as above, and all natural transformations are automatically strong. With this graded adjunction, we can show that the computation M **to** $x.\, N$ eagerly evaluates M in the semantics:

$$[\![M \textbf{ to } x.\, N]\!]\rho \;=\; \begin{cases} \perp & \text{if } [\![M]\!]\rho = \perp \\ [\![N]\!](\rho, [\![M]\!]\rho) & \text{otherwise} \end{cases}$$

In this case, a natural transformation $\alpha_\varepsilon : U(\varepsilon \circledast \underline{X}) \longrightarrow U(\varepsilon \circledast \underline{Y})$ is just a monotone function $\alpha_\star : \underline{X} \longrightarrow \underline{Y}$ between the pointed posets $\underline{X}$ and $\underline{Y}$. Such a natural transformation is linear if it is strict (i.e. is a morphism in $\mathbf{Poset}_\perp$). Similarly, left-linearity just means strict in the left argument.

The function space $X \Rightarrow \underline{Y}$ is the set of monotone functions, ordered pointwise. The least element of the function space is the function that maps every element of X to the least element of Y. Currying and uncurrying are the usual operations on functions. The evaluation map is given by

$$ev_{\star, X, \underline{Y}}(f, x) = f\, x$$

and is clearly left-linear.

The interpretation of the only base type **int** is just $[\![\mathbf{int}]\!] := \{-2^{31}, \ldots, 2^{31} - 1\}$ (ordered by equality). Recall that constants $c \in \mathcal{K}_A$ are interpreted as morphisms $[\![c]\!] : 1 \longrightarrow [\![A]\!]$. On $\mathbf{Poset}$, these are just elements $[\![c]\!] \in [\![A]\!]$ by abuse of notation. For undefined behaviour, **geq** and **add** have obvious interpretations (as functions $[\![\mathbf{int}]\!] \longrightarrow [\![\mathbf{int}]\!] \longrightarrow 2$ and $[\![\mathbf{int}]\!] \longrightarrow [\![\mathbf{int}]\!] \longrightarrow [\![\mathbf{int}]\!]$ respectively). The constant integers are interpreted as $[\![\underline{n}]\!] := n \in [\![\mathbf{int}]\!]$.

The operation undef is interpreted as a morphism $[\![\mathsf{undef}]\!] : 1 \longrightarrow 0_\bot$ (recall that 0 is the empty poset). For this we take the least element $\bot$. Finally, the operation $\mathsf{add}_{\mathsf{nsw}}$ is interpreted as a morphism $[\![\mathsf{add}_{\mathsf{nsw}}]\!] : [\![\mathbf{int}]\!] \times [\![\mathbf{int}]\!] \longrightarrow [\![\mathbf{int}]\!]_\bot$, which is given by:

$$[\![\mathsf{add}_{\mathsf{nsw}}]\!]\,(m, n) := \begin{cases} m + n & \text{if } (m + n) \in [\![\mathbf{int}]\!] \\ \bot & \text{otherwise} \end{cases}$$

It is easy to see that this interpretation is sound, and hence a model in the sense of Definition 4.2.7, by checking the signature axioms. In particular, we have $[\![\mathbf{undef}_{\underline{C}}]\!] \sqsubseteq [\![M]\!]$ because the left-hand side is the least element.

We validate each program transformation $M \preccurlyeq_{\mathrm{ctx}} N$ by showing that $[\![M]\!] \sqsubseteq [\![N]\!]$. This is sufficient because the model of undefined behaviour is *adequate*; we prove adequacy in Section 4.3.2.1. Hence, to validate the example transformations we show

$$[\![\mathsf{add}_{\mathsf{nsw}}(V_1, V_2)]\!] \sqsubseteq [\![\langle \mathbf{add}(V_1, V_2)\rangle]\!]$$

$$\left[\!\!\left[\begin{array}{l} \mathsf{add}_{\mathsf{nsw}}(V_1, V_2)\ \mathbf{to}\ x. \\ \mathbf{let}\ b = \mathbf{geq}(x, V_1)\ \mathbf{in} \\ N \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \mathsf{add}_{\mathsf{nsw}}(V_1, V_2)\ \mathbf{to}\ x. \\ \mathbf{let}\ b = \mathbf{geq}(V_2, \underline{0})\ \mathbf{in} \\ N \end{array}\right]\!\!\right]$$

To do this we just use cases on whether the addition overflows. For the first transformation if the addition does not overflow then the two sides are equal by definition, otherwise the left-hand side is the least element. The second is similar.

4.2.3.2 Nondeterminism

For our nondeterminism example, we again interpret value types as posets (in the value category **Poset**). For computation types we define a new **Poset**-category **D**. Recall that a meet-semilattice is a poset X such that each pair of elements $x_1, x_2 \in X$ has a *meet* (greatest lower bound) $x_1 \sqcap x_2$, i.e. an element of $x_1 \sqcap x_2 \in X$ that satisfies

$$\forall y \in X.\, y \sqsubseteq x_1 \sqcap x_2 \iff y \sqsubseteq x_1 \land y \sqsubseteq x_2$$

Objects of **D** are pairs of a meet-semilattice A_+ and a subset A_1 (which is not required to be closed under meets). We write such an object as $(A_1 \subseteq A_+)$. Morphisms $f : (A_1 \subseteq A_+) \longrightarrow (B_1 \subseteq B_+)$ in **D** are monotone functions $f : A_+ \longrightarrow B_+$ that preserve meets ($f(x_1 \sqcap x_2) = f x_1 \sqcap f x_2$), and such that if $x \in A_1$ then $f x \in B_1$.

The idea is that the elements of A_+ are nondeterministic, and elements of A_1 are deterministic. We use the usual bicartesian structure of **Poset**. The terminal object of **D** is $\{\star\} \subseteq \{\star\}$ and the binary product $(A_1 \subseteq A_+) \underline{\times} (B_1 \subseteq B_+)$ is $(A_1 \times B_1) \subseteq (A_+ \times B_+)$. The exponential $X \Rrightarrow (A_1 \subseteq A_+)$ consists of the meet-semilattice of monotone functions $f : X \longrightarrow A_+$. The subset contains the monotone functions whose images are subsets of A_1.

We next define the graded adjunction. The right adjoint $U : \mathbf{D} \longrightarrow \mathbf{Poset}$ sends $(A_1 \subseteq A_+)$ to A_1 (the order on A_1 is the restriction of the order on A_+), and sends each morphism $f : (A_1 \subseteq A_+) \longrightarrow (B_1 \subseteq B_+)$ to its restriction to $A_1 \longrightarrow B_1$. The monoid action is defined for the effect $+$ by:

$$(+) \circledast (A_1 \subseteq A_+) := (A_+ \subseteq A_+) \qquad (+) \circledast f := f$$

and on the effect 1 it is the identity.

For the left adjoint, recall that binary nondeterminism is traditionally modelled using the monad $\mathcal{P}^+_{\mathrm{fin}}$ on **Set**, where $\mathcal{P}^+_{\mathrm{fin}} X$ is the set of nonempty finite subsets of X. A subset $S \in \mathcal{P}^+_{\mathrm{fin}} X$

is the set of possible results of a computation. Here we also have to take into account the ordering on elements of X. If $x \in X$ is a possible result of a computation and $x \sqsubseteq x'$ then x' should also be viewed as a possible result. Hence instead of using subsets $S \in \mathcal{P}^+_{\text{fin}}X$ we use their *upwards closures* $\uparrow S := \{x' \in X \mid \exists x \in S.\, x \sqsubseteq x'\}$.

Formally, the left adjoint $F : \mathbf{Poset} \longrightarrow \mathbf{D}$ sends each poset X to $(A_1 \subseteq A_+)$, where

$$A_1 := \{\uparrow\{x\} \mid x \in X\} \qquad A_+ := \{\uparrow S \mid S \in \mathcal{P}^+_{\text{fin}}X\}$$

and A_+ has superset as the order.[4] The meet of $S_1, S_2 \in A_+$ is the union $S_1 \cup S_2$. On monotone functions $f : X \longrightarrow Y$, the **Poset**-functor F is given by

$$Ff := \lambda S.\, \uparrow\{f\,x \mid x \in S\}$$

The unit of the adjunction is $\eta_X\, x := \uparrow\{x\}$. For the counit we note that for any $(A_1 \subseteq A_+) \in \mathbf{D}$ each set $S' \in \mathcal{P}^+_{\text{fin}}A_+$ has a meet $\bigsqcap S' \in A_+$, and hence if $S \subseteq A_+$ is the upwards closure of some $S' \in \mathcal{P}^+_{\text{fin}}A_+$ then S has a meet $\bigsqcap S = \bigsqcap S'$. The counit $\varpi_{(A_1 \subseteq A_+)} : FA_1 \longrightarrow (A_1 \subseteq A_+)$ is given by

$$\varpi_{(A_1 \subseteq A_+)} S := \bigsqcap S$$

For this graded adjunction, sequencing of computations takes the meet of all possible results:

$$[\![M \textbf{ to } x.\, N]\!] = \bigsqcap_{a \in [\![M]\!]\,\rho} [\![N]\!](\rho, a)$$

If M is deterministic (has type $\langle 1 \rangle A$ for some A), then $[\![M]\!]\,\rho \;=\; \uparrow\{a\}$ for some a, and so $[\![M \textbf{ to } x.\, N]\!] = [\![N]\!](\rho, a)$ by monotonicity of $[\![N]\!]$.

To complete the model we give the interpretation of flip, which is a function $[\![\mathsf{flip}]\!] : 1 \longrightarrow \{\uparrow S \mid S \in \mathcal{P}^+_{\text{fin}}\{\text{true}, \text{false}\}\}$. This maps the unique element of 1 to the set $\{\text{true}, \text{false}\}$, so that the two possible results of $\mathsf{flip}\,()$ are true and false. Under this interpretation we can show that binary nondeterministic choice takes the meet:

$$[\![M_1 \textbf{ or } M_2]\!]\,\rho = [\![M_1]\!]\,\rho \sqcap [\![M_2]\!]\,\rho$$

It is not difficult to check soundness of the signature axioms (Figure 4.1), and hence that this data forms a model of the inequational theory for nondeterminism. For example, for the axiom $\mathsf{flip}\,() \;\preccurlyeq\; \textbf{coerce}_{1 \le +}\,\langle \textbf{true} \rangle$ we have

$$[\![\mathsf{flip}\,()]\!]\,\rho \;=\; \{\text{true}, \text{false}\} \;\supseteq\; \{\text{true}\} \;=\; [\![\textbf{coerce}_{1 \le +}\,\langle \textbf{true} \rangle]\!]\,\rho$$

We show that this model is adequate in Section 4.3.2.2.

We verify that both of the program transformations given in Section 4.1.2 hold. For reusing the result of a duplicated computation this means showing

$$[\![M \textbf{ to } x.M \textbf{ to } y.N]\!] \;\sqsubseteq\; [\![M \textbf{ to } x.N[y \mapsto x]]\!]$$

Using the interpretation of **to** above this inequality becomes

$$\bigsqcap_{x \in [\![M]\!]\,\rho}\; \bigsqcap_{y \in [\![M]\!]\,\rho} [\![N]\!]((\rho, x), y) \;\sqsubseteq\; \bigsqcap_{x \in [\![M]\!]\,\rho} [\![N]\!]((\rho, x), x)$$

[4] For the left adjoint, A_+ is the *free meet-semilattice* on the poset X. (Similarly, $\mathcal{P}^+_{\text{fin}}Y$ ordered by $\supseteq$ is the free meet-semilattice on the *set* Y.)

and this holds, so the left-to-right transformation is valid. If M is deterministic (has type $\langle 1\rangle A$ for some A), then $[\![M]\!]\,\rho = \uparrow\{a\}$ for some a, and the two sides of this inequality are just $[\![N]\!]((\rho, a), a)$. So in this case, we have an equality, and the right-to-left direction of the transformation is also valid.

The other example we give is eliminating a dead computation, which is valid because

$$[\![M;\, N]\!]\,\rho \;=\; \bigsqcap_{x\in[\![M]\!]\,\rho} [\![N]\!]\,\rho \;=\; [\![N]\!]\,\rho$$

4.2.3.3 Shared global state

We give a model of shared global state on **Poset**, in the style of *algebraic effects* [88]. We do not give a general description of algebraic effects on **Poset**; instead we consider only get and put. A general description of algebraic models of GCBPV on **Poset** would be similar to [42, 64, 47], but adapted to include grading.

We interpret value types as posets (in the value category **Poset**). The computation category is more complicated, and somewhat similar to the computation category we use for nondeterminism. Objects are roughly posets, together with interpretations of the get and put operations.

Recall that the set of operations is $\Sigma := \{\mathrm{get}, \mathrm{put}\}$, and effects are subsets $\varepsilon \subseteq \Sigma$.

Definition 4.2.8 A *graded shared mnemoid*[5] (A, g, p) consists of a family of sets $(A_\varepsilon)_{\varepsilon\subseteq\Sigma}$ such that $\varepsilon \subseteq \varepsilon'$ implies $A_\varepsilon \subseteq A_{\varepsilon'}$, a partial order $\sqsubseteq$ on A_Σ, and monotone functions

$$g : A_\Sigma \times A_\Sigma \to A_\Sigma \qquad p_{\mathrm{true}} : A_\Sigma \to A_\Sigma \qquad p_{\mathrm{false}} : A_\Sigma \to A_\Sigma$$

such that if $a_{\mathrm{true}}, a_{\mathrm{false}} \in A_\varepsilon$ then $g(a_{\mathrm{true}}, a_{\mathrm{false}}) \in A_{\varepsilon\cup\{\mathrm{get}\}}$, if $a \in A_\varepsilon$ then $p_b\, a \in A_{\varepsilon\cup\{\mathrm{put}\}}$, and the following axioms hold:

$$\begin{gathered} g(p_{\mathrm{true}}\, a_{\mathrm{true}}, p_{\mathrm{false}}\, a_{\mathrm{false}}) \sqsubseteq g(a_{\mathrm{true}}, a_{\mathrm{false}}) \qquad p_b(g(a_{\mathrm{true}}, a_{\mathrm{false}})) \sqsubseteq p_b\, a_b \\ g(g(a_{\mathrm{true,true}}, a_{\mathrm{true,false}}), g(a_{\mathrm{false,true}}, a_{\mathrm{false,false}})) \sqsubseteq g(a_{\mathrm{true,true}}, a_{\mathrm{false,false}}) \\ p_{b_1}(p_{b_2}\, a) \sqsubseteq p_{b_2}\, a \qquad g(a, a) = a \end{gathered}$$

The computation category **GSMnem** has graded shared mnemoids as objects. Morphisms $f : (A, g, p) \to (A', g', p')$ are monotone functions $f : A_\Sigma \to A'_\Sigma$ such that

$$f(g(a_{\mathrm{true}}, a_{\mathrm{false}})) = g'(f\, a_{\mathrm{true}}, f\, a_{\mathrm{false}}) \qquad f(p_b a) = p'_b(f\, a)$$

and $a \in A_\varepsilon$ implies $f\, a \in A'_\varepsilon$. Morphisms are ordered pointwise. ◀

The set A_ε contains the interpretations of computations with effect ε. The functions provide the operations: $g(a_{\mathrm{true}}, a_{\mathrm{false}})$ means get the value b of the state and then run a_b, and $p_b a$ means put the value b and then run a. The axioms are similar to those of the inequational theory in Figure 4.2.

We use the usual bicartesian structure on **Poset**. The cartesian structure on **GSMnem**, and exponentials, are straightforward. In particular, the family of sets in the product $(A, g, p) \,\underline{\times}\, (A', g', p')$ is $(A_\varepsilon \times A'_\varepsilon)_{\varepsilon\subseteq\Sigma}$, and the functions are given componentwise. For the exponential $X \Rightarrow (A, g, p)$ we take the set of monotone functions $X \to A_\Sigma$.

[5] These are based on Melliès's [72] *mnemoids*. We adapt the definition to *shared* global state, and also add grading (given by the subsets). The usual global state monad on a fixed set forms a graded shared mnemoid (just as it forms a mnenoid); this implies consistency of our axioms.

For the graded adjunction the right adjoint $U : \mathbf{GSMnem} \longrightarrow \mathbf{Poset}$ sends (A, g, p) to $A_\emptyset$, and morphisms f to their restrictions to $A_\emptyset$. The monoid action is given by:

$$\varepsilon \circledast (A, g, p) \;:=\; ((A_{\varepsilon \cup \varepsilon'})_{\varepsilon' \subseteq \Sigma}, g, p) \qquad \varepsilon \circledast f \;:=\; f$$

and $(\varepsilon \subseteq \varepsilon') \circledast -$ is the identity on A_Σ.

For the left adjoint, given a poset $(X, \sqsubseteq_X)$, *terms t over X* are given by the following formal grammar, where x ranges over elements of X and b over elements of $2 = \{\mathsf{true}, \mathsf{false}\}$:

$$t ::= x \mid \mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}}) \mid \mathsf{put}_b\, t$$

The term x just returns x; the term $\mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}})$ gets the value b of the state and then runs the term t_b; and $\mathsf{put}_b\, t$ puts b and then runs the term t. For example, $\mathsf{get}(\mathsf{put}_{\mathsf{false}}\, x, \mathsf{put}_{\mathsf{true}}\, x)$ is a term that gets the value of the state, puts its negation, and then returns x. We capture the required behaviour of these terms using a (non-antisymmetric) preorder $\trianglelefteq$ on terms. It is defined as the smallest preorder that is closed under congruence

$$\frac{x \sqsubseteq_X x'}{x \trianglelefteq x'} \qquad \frac{t_{\mathsf{true}} \trianglelefteq t'_{\mathsf{true}} \qquad t_{\mathsf{false}} \trianglelefteq t'_{\mathsf{false}}}{\mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}}) \trianglelefteq \mathsf{get}(t'_{\mathsf{true}}, t'_{\mathsf{false}})} \qquad \frac{t \trianglelefteq t' \qquad b \in 2}{\mathsf{put}_b\, t \trianglelefteq \mathsf{put}_b\, t'}$$

and under the following six axioms (which should be compared with the shared mnemoid axioms above):

$$\begin{gathered}
\mathsf{get}(\mathsf{put}_{\mathsf{true}}\, t_{\mathsf{true}}, \mathsf{put}_{\mathsf{false}}\, t_{\mathsf{false}}) \trianglelefteq \mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}}) \qquad \mathsf{put}_b(\mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}})) \trianglelefteq \mathsf{put}_b\, t_b \\
\mathsf{get}(\mathsf{get}(t_{\mathsf{true,true}}\,, t_{\mathsf{true,false}}), \mathsf{get}(t_{\mathsf{false,true}}\,, t_{\mathsf{false,false}})) \;\trianglelefteq\; \mathsf{get}(t_{\mathsf{true,true}}\,, t_{\mathsf{false,false}}) \\
\mathsf{put}_{b_1}(\mathsf{put}_{b_2}\, t) \trianglelefteq \mathsf{put}_{b_2}\, t \qquad\qquad \mathsf{get}(t, t) \trianglelefteq t \qquad t \trianglelefteq \mathsf{get}(t, t)
\end{gathered}$$

The intersection of $\trianglelefteq$ and its converse $\trianglerighteq$ is an equivalence relation on computations. We quotient by this equivalence relation, so that $\trianglelefteq$ induces a partial order on equivalence classes $[t]$ of computations t. (We do not know of a desciption that does not involve a quotient.) From this point onwards, we always consider equivalence classes, and suppress the square brackets in the notation.

We construct a graded shared mnemoid $FX := (A, g, p)$ as follows: A_ε is the set of equivalence classes of terms t over X that contain only the operations in ε. The functions g and p_b are straightforward:

$$g(t_{\mathsf{true}}, t_{\mathsf{false}}) := \mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}}) \qquad p_b\, t := \mathsf{put}_b\, t$$

This defines the left adjoint on objects. To define it on morphisms, we first define the *interpretations* of terms in graded shared mnemoids $(B.g, p)$. Given a term t over X, and a monotone function $\rho : X \longrightarrow B_\Sigma$, we define $\mathcal{I}\llbracket t \rrbracket \rho \in B_\Sigma$ by:

$$\mathcal{I}\llbracket x \rrbracket \rho := \rho\, x \qquad \mathcal{I}\llbracket \mathsf{get}(t_{\mathsf{true}}, t_{\mathsf{false}}) \rrbracket \rho := g(\mathcal{I}\llbracket t_{\mathsf{true}} \rrbracket \rho, \mathcal{I}\llbracket t_{\mathsf{false}} \rrbracket \rho) \qquad \mathcal{I}\llbracket \mathsf{put}_b\, t \rrbracket \rho := p_b(\mathcal{I}\llbracket t \rrbracket \rho)$$

(This is well-defined and monotone by the axioms required for graded shared mnemoids.) Any monotone function $f : X \longrightarrow Y$ can be viewed as a monotone function $f : X \longrightarrow B_\Sigma$ where B_Σ is the set of equivalence classes of terms over Y, and so we define $Ff := \mathcal{I}\llbracket - \rrbracket f$. The unit of the graded adjunction maps each x to the trivial computation x. The counit $\varpi_{(A,g,p)} : FA_\emptyset \longrightarrow (A, g, p)$ is given by $\varpi_{(A,g,p)} := \mathcal{I}\llbracket - \rrbracket\, id$.

The constants are interpreted as follows:

$$\llbracket \mathsf{get} \rrbracket \star := (\mathsf{get}(\mathsf{true}, \mathsf{false})) \qquad \llbracket \mathsf{put} \rrbracket\, b := (\mathsf{put}_b \star)$$

This defines a model of the inequational theory for shared global state, in which

$$\llbracket M \textbf{ to } x.\, N\rrbracket\rho \coloneqq \mathcal{I}\llbracket\llbracket M\rrbracket\rho\rrbracket(a \mapsto \llbracket N\rrbracket(\rho, a))$$

We show adequacy in Section 4.3.2.3.

Finally, we verify that the example transformations in Section 4.1.3 are valid in this model. For reusing the result of a duplicated computation we show

$$\llbracket M \textbf{ to } x.\, M \textbf{ to } y.\, N\rrbracket \sqsubseteq \llbracket M \textbf{ to } x.\, N[y \mapsto x]\rrbracket$$

for computations M of type $\langle\varepsilon\rangle A$ with $\varepsilon \subseteq \{\mathrm{put}\}$. Given any $\rho \in \llbracket\Gamma\rrbracket$, the term $\llbracket M\rrbracket\rho$ has the form $\mathrm{put}_{b_1}(\mathrm{put}_{b_2}(\cdots(\mathrm{put}_{b_n}\, a)\cdots))$ for some $b_1, \ldots, b_n \in 2$ and $a \in \llbracket A\rrbracket$, because of the restriction on the effect ε. By expanding the interpretations of the two computations, the goal becomes

$$p_{b_1}(\cdots(p_{b_n}(p_{b_1}(\cdots(p_{b_n}(\llbracket N\rrbracket(\rho,(a,a))))\cdots)))\cdots) \;\sqsubseteq\; p_{b_1}(\cdots(p_{b_n}(\llbracket N\rrbracket(\rho,(a,a))))\cdots)$$

If $n = 0$ then this holds automatically; if $n > 0$ then we can show by induction on n that

$$p_{b_1}(\cdots(p_{b_n}(p_b a))\cdots) \sqsubseteq p_b a$$

for all a, b, using the axiom $p_{b'}(p_{b''}a') \sqsubseteq p_{b''}a'$, and instantiate this to get the inequality we need.

For eliminating a dead computation we show

$$\llbracket M';\, N'\rrbracket = \llbracket N'\rrbracket$$

for computations M' of type $\langle\varepsilon'\rangle A'$ where $\varepsilon' \subseteq \{\mathrm{get}\}$. This uses a similar induction, except that we use the axiom $g(a, a) = a$.

4.2.3.4 Monadic models

The three models above suffice for our examples. We additionally give a *general* source of models. It is well-known that monads can be used to model various side-effects, and strong monads are used for models of the monadic metalanguage [78]. Monads can also be used as models for CBPV by taking the adjunction to be the *Eilenberg-Moore resolution*, as shown by Levy [57, Chapter 12]. Here we generalize the notion of monadic model of CBPV to the graded, **Poset**-enriched case. This allows to use previous monadic models of side-effects as models of GCBPV. The construction in this chapter is a relatively straightforward generalization, and the proofs are almost identical.

Monadic models of GCBPV are based on Katsumata's [45] *graded monads*. We again need tensorial strengths (to interpret open terms) and, in our case, need **Poset**-enrichment. Hence we use *graded strong* **Poset**-*monads*. In the following definition, **C** should be thought of as the value category of a model of GCBPV.

Definition 4.2.9 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid and **C** is a cartesian **Poset**-category. A *($\mathcal{E}$-)graded strong* **Poset**-*monad*[6] $(T, \mathit{str}, \eta, \mu)$ on **C** consists of:

- A strong **Poset**-functor $(T\varepsilon, \mathit{str}_\varepsilon)$ on **C** for each $\varepsilon \in \mathcal{E}$.
- A strong natural transformation $T(\varepsilon \leq \varepsilon') : (T\varepsilon, \mathit{str}_\varepsilon) \longrightarrow (T\varepsilon', \mathit{str}_{\varepsilon'})$ for each $\varepsilon \leq \varepsilon' \in \mathcal{E}$.
- A strong natural transformation $\eta : \mathit{Id}_{\mathbf{C}} \longrightarrow (T1, \mathit{str}_1)$ from the identity strong **Poset**-functor.

- A strong natural transformation $\mu_{\varepsilon,\varepsilon'} : (T\varepsilon, str_\varepsilon) \circ (T\varepsilon', str_{\varepsilon'}) \longrightarrow (T(\varepsilon{\cdot}\varepsilon'), str_{\varepsilon\cdot\varepsilon'})$ for each $\varepsilon, \varepsilon' \in \mathcal{E}$.

Such that:

- T is functorial: $T(\varepsilon \leq \varepsilon) = id_{T\varepsilon}$ and if $\varepsilon \leq \varepsilon' \leq \varepsilon''$ then $T(\varepsilon \leq \varepsilon'') = T(\varepsilon' \leq \varepsilon'') \circ T(\varepsilon \leq \varepsilon')$.
- μ is natural in ε and ε': if $\varepsilon_1 \leq \varepsilon_2$ and $\varepsilon'_1 \leq \varepsilon'_2$ then the following diagrams commute

$$\begin{tikzcd}
T\varepsilon_1 \circ T\varepsilon' \arrow[r, "(T(\varepsilon_1\leq\varepsilon_2))_{T\varepsilon'}"] \arrow[d, "\mu_{\varepsilon_1,\varepsilon'}"'] & T\varepsilon_2 \circ T\varepsilon' \arrow[d, "\mu_{\varepsilon_2,\varepsilon'}"] \\
T(\varepsilon_1{\cdot}\varepsilon') \arrow[r, "T(\varepsilon_1\cdot\varepsilon'\leq\varepsilon_2\cdot\varepsilon')"'] & T(\varepsilon_2{\cdot}\varepsilon')
\end{tikzcd}
\qquad
\begin{tikzcd}
T\varepsilon \circ T\varepsilon'_1 \arrow[r, "T\varepsilon(T(\varepsilon'_1\leq\varepsilon'_2))"] \arrow[d, "\mu_{\varepsilon,\varepsilon'_1}"'] & T\varepsilon \circ T\varepsilon'_2 \arrow[d, "\mu_{\varepsilon,\varepsilon'_2}"] \\
T(\varepsilon{\cdot}\varepsilon'_1) \arrow[r, "T(\varepsilon\cdot\varepsilon'_1\leq\varepsilon\cdot\varepsilon'_2)"'] & T(\varepsilon{\cdot}\varepsilon'_2)
\end{tikzcd}$$

- The (graded) monad laws hold: the following diagrams commute

$$\begin{tikzcd}
 & T\varepsilon \arrow[dl, bend right, "\eta_{T\varepsilon}"'] \arrow[d, equal] \arrow[dr, bend left, "T\varepsilon\eta"] & \\
T1 \circ T\varepsilon \arrow[r, "\mu_{1,\varepsilon}"'] & T\varepsilon & T\varepsilon \circ T1 \arrow[l, "\mu_{\varepsilon,1}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
T\varepsilon \circ T\varepsilon' \circ T\varepsilon'' \arrow[r, "\mu_{\varepsilon,\varepsilon',T\varepsilon''}"] \arrow[d, "T\varepsilon\mu_{\varepsilon',\varepsilon''}"'] & T(\varepsilon{\cdot}\varepsilon') \circ T\varepsilon'' \arrow[d, "\mu_{\varepsilon\cdot\varepsilon',\varepsilon''}"] \\
T\varepsilon \circ T(\varepsilon'{\cdot}\varepsilon'') \arrow[r, "\mu_{\varepsilon,\varepsilon'\cdot\varepsilon''}"'] & T(\varepsilon{\cdot}\varepsilon'{\cdot}\varepsilon'')
\end{tikzcd}$$

◀

Graded strong **Poset**-functors can be used as models of our graded version of the monadic metalanguage (GMM in Section 2.5). The **Poset**-functors $T\varepsilon$ are used to interpret the type constructors $\langle\varepsilon\rangle$. The strong natural transformations $T(\varepsilon \leq \varepsilon')$ are used for coercions. The *unit* η is used to interpret pure computations, and the *multiplication* μ is used for sequencing of computations.

If T is a graded strong **Poset**-monad then it forms a functor $\mathcal{E} \longrightarrow [\mathbf{C}, \mathbf{C}]$ (where as usual we view $\mathcal{E}$ as a category). The strength is trivial in many cases in the same way that it is for adjunctions. If $\mathbf{C} = \mathbf{Set}$ or $\mathbf{C} = \mathbf{Poset}$, there is always a unique strength, and every natural transformation is strong. If $\mathbf{C} = \omega\mathbf{Cpo}$ the strength is unique if it exists, and it does exist if $T\,\varepsilon$ preserves least upper bounds of morphisms for each ε.

It is well-known that every adjunction induces a monad, and there is a analogous fact in our situation:

Lemma 4.2.10 Suppose that $(F, U, \circledast)$ is a graded strong **Poset**-adjunction with $F : \mathbf{C} \longrightarrow \mathbf{D}$. Define $T : \mathcal{E} \longrightarrow [\mathbf{C}, \mathbf{C}]$ by:

$$T\varepsilon := U(\varepsilon{\circledast}F-) \qquad T(\varepsilon \leq \varepsilon') := U((\varepsilon \leq \varepsilon'){\circledast}F-)$$

Then T forms a graded strong **Poset**-monad. Each strength str_ε is the strength of the adjunction, the unit η is the unit of the adjunction, and the multiplication is given by:

$$\mu_{\varepsilon,\varepsilon',X} \;:=\; U(\varepsilon{\circledast}\varpi_{\varepsilon'\circledast X}) \;:\; U(\varepsilon{\circledast}F(U(\varepsilon'{\circledast}FX))) \longrightarrow U((\varepsilon{\cdot}\varepsilon'){\circledast}FX)$$

◀

It is possible to continue in this direction and show that each GCBPV structure induces most of a GMM structure (the data required to interpret the graded monadic metalanguage (Section 2.5), which includes a graded strong **Poset**-monad; the only missing part is that we do not quite

[6] Similar to the non-enriched case, graded strong **Poset**-monads are just *lax monoidal functors* $\mathcal{E} \longrightarrow [\mathbf{C}, \mathbf{C}]_s$, where the preordered monoid $\mathcal{E}$ is viewed as a thin monoidal category, and $[\mathbf{C}, \mathbf{C}]_s$ is the monoidal category of strong **Poset**-functors on $\mathbf{C}$, with composition as the tensor product. Graded strong **Poset**-monads can be equivalently presented as *Kleisli triples*, which have morphisms $(\gg\!=) : (X \Rightarrow T\varepsilon' Y) \longrightarrow (T\varepsilon X \Rightarrow T(\varepsilon{\cdot}\varepsilon')Y)$.

have exponentials on **C**). This should not be surprising: we mentioned in Section 2.7.3 that we can translate most of GMM into GCBPV (by interpreting the graded monad $\langle\varepsilon\rangle-$ as $\mathbf{U}\,\langle\varepsilon\rangle-$).

In this section we go in the other direction, and construct models of GCBPV from models of GMM. Given a graded strong **Poset**-monad on **C**, we construct a graded strong **Poset**-adjunction. We use **C** as the value category of the model, so the first step is to construct the computation category **D**.

Definition 4.2.11 (Graded algebra [76, 28]) Suppose that $T : \mathcal{E} \to [\mathbf{C}, \mathbf{C}]$ is a graded strong **Poset**-monad. A *T-algebra* is a pair (A, a) of a functor $A : \mathcal{E} \to \mathbf{C}$ and a natural transformation

$$a_{\varepsilon,\varepsilon'} : T\varepsilon(A\varepsilon') \to A(\varepsilon\cdot\varepsilon')$$

such that the following diagrams commute for all $\varepsilon, \varepsilon', \varepsilon''$:

$$\begin{tikzcd}
A\varepsilon \arrow[r, "\eta"] \arrow[dr, equal] & T1(A\varepsilon) \arrow[d, "a_{1,\varepsilon}"] \\
& A\varepsilon
\end{tikzcd}
\qquad
\begin{tikzcd}
T\varepsilon(T\varepsilon'(A\varepsilon'')) \arrow[r, "T\varepsilon a"] \arrow[d, "\mu"'] & T\varepsilon(A(\varepsilon'\cdot\varepsilon'')) \arrow[d, "a"] \\
T(\varepsilon\cdot\varepsilon')(A\varepsilon'') \arrow[r, "a"'] & A(\varepsilon\cdot\varepsilon'\cdot\varepsilon'')
\end{tikzcd}$$

We call the functor A the *carrier* of the algebra.

A *homomorphism* from (A, a) to (A', a') is a natural transformation $h : A \to A'$ such that

$$\begin{tikzcd}
T\varepsilon(A\varepsilon') \arrow[r, "T\varepsilon h"] \arrow[d, "a"'] & T\varepsilon(A'\varepsilon') \arrow[d, "a'"] \\
A(\varepsilon\cdot\varepsilon') \arrow[r, "h"'] & A'(\varepsilon\cdot\varepsilon')
\end{tikzcd}$$

commutes.

The *Eilenberg-Moore category* T-**Alg** of the graded monad T has T-algebras as objects and graded algebra homomorphisms as morphisms. It is a **Poset**-category: $h \sqsubseteq h'$ if $h_\varepsilon \sqsubseteq h'_\varepsilon$ for all ε. ◀

(This definition does not actually require strength, but we assume strength throughout this section for consistency.)

We take T-**Alg** as the computation category **D**, and construct a graded adjunction

$$\begin{tikzcd}
\mathbf{C} \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] & T\text{-}\mathbf{Alg} \arrow[l, bend left, "U"] \arrow[loop right, "\varepsilon\circledast-"]
\end{tikzcd}$$

The right adjoint U is the *forgetful* **Poset**-functor, which is given by

$$U(A, a) := A1 \qquad Uh := h_1$$

The left adjoint F maps each object X to the *free T-algebra on X*:

$$F\,X := (T\,(-)\,X, \mu) \qquad (Ff)_\varepsilon := T\,\varepsilon\,f$$

So the carrier of $F\,X$ on the effect ε is $T\,\varepsilon\,X$. The action $\varepsilon\circledast-$ multiplies by the effect ε: given a T-algebra (A, a), the T-algebra $\varepsilon\circledast(A, a)$ is (B, b), where:

$$B\,\varepsilon' := A(\varepsilon\cdot\varepsilon') \qquad B(\varepsilon' \leq \varepsilon'') := A(\varepsilon\cdot\varepsilon' \leq \varepsilon\cdot\varepsilon'') \qquad b_{\varepsilon',\varepsilon''} := a_{\varepsilon',\varepsilon\cdot\varepsilon''}$$

This defines the action on objects. On homomorphisms $h : (A, a) \longrightarrow (A', a')$, the homomorphism $\varepsilon \circledast h : \varepsilon \circledast (A, a) \longrightarrow \varepsilon \circledast (A', a')$ is given by

$$(\varepsilon \circledast h)_{\varepsilon'} \coloneqq h_{\varepsilon \cdot \varepsilon'}$$

and if $\varepsilon \leq \varepsilon'$, then the homomorphism $(\varepsilon \leq \varepsilon') \circledast (A, a) : \varepsilon \circledast (A, a) \longrightarrow \varepsilon' \circledast (A, a)$ is given by:

$$((\varepsilon \leq \varepsilon') \circledast (A, a))_{\varepsilon''} \coloneqq A(\varepsilon \cdot \varepsilon'' \leq \varepsilon' \cdot \varepsilon'')$$

Finally, the unit of the graded adjunction is just the unit η of the monad. The counit is given by

$$(\varpi_{(A,a)})_{\varepsilon} \;\coloneqq\; a_{1,\varepsilon} \;:\; T1(A\varepsilon) \longrightarrow A\varepsilon$$

and the strength is just the strength of the monad.

The data that we construct here satisfies $U(\varepsilon \circledast F-) = T\varepsilon$. If we apply Lemma 4.2.10 to it then we recover the graded monad we started with. Showing this construction does give a graded strong **Poset**-adjunction is straightforward.

Lemma 4.2.12 Given any graded strong **Poset**-monad, the construction above defines a graded strong **Poset**-adjunction. ◀

We still have to construct the rest of the data required for a model, in particular, we need the computation category T-**Alg** to be cartesian, and also require exponentials. We can construct these out of similar structure on **C**.

Lemma 4.2.13 Suppose that $T : \mathcal{E} \longrightarrow [\mathbf{C}, \mathbf{C}]$ is a graded strong **Poset**-monad on a cartesian closed **Poset**-category **C**. Then:

1. The terminal object $\underline{1}$ of T-**Alg** is the unique T-algebra with the constantly-1 functor as carrier, and satisfies $\varepsilon \circledast \underline{1} = \underline{1}$.
2. The binary product of (A_1, a_1) and (A_2, a_2) is the T-algebra with carrier $A_1 \times A_2$ and natural transformation

$$T\varepsilon(A_1\varepsilon' \times A_2\varepsilon') \xrightarrow{\langle T\varepsilon\pi_1, T\varepsilon\pi_2 \rangle} T\varepsilon(A_1\varepsilon') \times T\varepsilon(A_2\varepsilon') \xrightarrow{a_{1,\varepsilon,\varepsilon'} \times a_{2,\varepsilon,\varepsilon'}} A_1(\varepsilon \cdot \varepsilon') \times A_2(\varepsilon \cdot \varepsilon')$$

   together with $\pi_i : A_1\varepsilon \times A_2\varepsilon \longrightarrow A_i\varepsilon$ as the ith projection. It satisfies

$$\varepsilon \circledast ((A_1, a_1) \underline{\times} (A_2, a_2)) = (\varepsilon \circledast (A_1, a_1)) \underline{\times} (\varepsilon \circledast (A_2, a_2)) \quad \langle \varepsilon \circledast \pi_1, \varepsilon \circledast \pi_2 \rangle = id_{\varepsilon \circledast ((A_1, a_1) \underline{\times} (A_2, a_2))}$$

3. The exponential $X \Rightarrow (A, a)$ is the T-algebra with carrier $X \Rightarrow A$ and natural transformation

$$T\varepsilon(X \Rightarrow A\varepsilon') \xrightarrow{\Lambda(ev \circ str^{r})} X \Rightarrow T\varepsilon(A\varepsilon') \xrightarrow{X \Rightarrow a_{\varepsilon,\varepsilon'}} X \Rightarrow A(\varepsilon \cdot \varepsilon')$$

   with currying given by currying for exponentials in **C**. It satisfies

$$\varepsilon \circledast (X \Rightarrow (A, a)) \;=\; X \Rightarrow (\varepsilon \circledast (A, a)) \qquad ev \text{ is left-linear}$$ ◀

The proof is essentially the same as for the non-graded, non-enriched case.

To give a GCBPV structure, it therefore suffices to give:

- A bicartesian closed **Poset**-category **C**.
- A graded strong **Poset**-monad T on **C**.
- An object $[\![b]\!] \in \mathbf{C}$ for each base type b.

- A morphism $[\![c]\!] : 1 \to [\![A]\!]$ for each constant $c \in \mathcal{K}_A$ and a morphism $[\![\mathrm{op}]\!] : [\![\mathrm{ar}_{\mathrm{op}}]\!] \to T\varepsilon[\![\mathrm{car}_{\mathrm{op}}]\!]$ for each operation $\mathrm{op} \in \Sigma$.

The amount of data required and number of diagrams to check in this definition can be large (for Gifford-style effect algebras, it is exponential in the number of operations). We end this section by giving a lemma that allows us to reduce the amount of effort required to specify monadic models.

A graded strong monad *structure* consists of the same data as a graded strong **Poset**-monad, but none of the laws.

Definition 4.2.14 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid and $\mathbf{C}$ is a cartesian **Poset**-category. A *graded strong monad structure* on $\mathbf{C}$ consists of:

- For each $\varepsilon \in \mathcal{E}$: an object $T\varepsilon X \in \mathbf{C}$ for each $X \in \mathbf{C}$, a morphism $T\varepsilon f : T\varepsilon X \to T\varepsilon Y$ for each $f : X \to Y$, and a morphism $str_{\varepsilon,X,Y} : X \times T\varepsilon Y \to T\varepsilon(X \times Y)$ for each $X, Y \in \mathbf{C}$.
- A morphism $(T(\varepsilon \leq \varepsilon'))_X : T\varepsilon X \to T\varepsilon' X$ for each $\varepsilon \leq \varepsilon' \in \mathcal{E}$ and $X \in \mathbf{C}$.
- A morphism $\eta_X : X \to T1X$ for each $X \in \mathbf{C}$.
- A morphism $\mu_{\varepsilon,\varepsilon',X} : T\varepsilon(T\varepsilon' X) \to T(\varepsilon \cdot \varepsilon')X$ for each $\varepsilon, \varepsilon' \in \mathcal{E}$ and $X \in \mathbf{C}$. ◀

Each graded strong monad structure either is or is not a graded strong **Poset**-monad. We show that such a structure is a graded strong **Poset**-monad if it forms a *grading* of a strong **Poset**-monad S, which means that it comes with morphisms $m_{\varepsilon,X} : T\varepsilon X \to SX$. These morphisms allow us to view computations in $T\varepsilon X$ as computations in S. They are required to preserve the structure (multiplication, etc.), and also to be *full monomorphisms*:

Definition 4.2.15 (Meseguer [74]) A morphism $m : X \to Y$ in a **Poset**-category is a *full monomorphism* if $m \circ f \sqsubseteq m \circ g$ implies $f \sqsubseteq g$ for all $f, g : Y \to Z$. ◀

Each full monomorphism is in particular a monomorphism ($m \circ f = m \circ g \Rightarrow f = g$). In **Set** the full monomorphisms are the injections, in **Poset** they are the monotone functions such that $m\,x \sqsubseteq m\,y \Rightarrow x \sqsubseteq y$.

Definition 4.2.16 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid, $\mathbf{C}$ is a cartesian **Poset**-category and that S is a strong **Poset**-monad on $\mathbf{C}$. A *grading* of S consists of a graded strong monad structure T on $\mathbf{C}$ together with a full monomorphism

$$m_{\varepsilon,X} : T\varepsilon X \to SX$$

for each $\varepsilon \in \mathcal{E}$ and $X \in \mathbf{C}$, such that the following diagrams commute:

$$\begin{tikzcd}
T\varepsilon X \arrow[r, "T\varepsilon f"] \arrow[d, "m_{\varepsilon,X}"'] & T\varepsilon Y \arrow[d, "m_{\varepsilon,Y}"] \\
SX \arrow[r, "Sf"'] & SY
\end{tikzcd}
\qquad
\begin{tikzcd}
X \times T\varepsilon Y \arrow[r, "str_{\varepsilon,X,Y}"] \arrow[d, "X \times m_{\varepsilon,Y}"'] & T\varepsilon(X \times Y) \arrow[d, "m_{\varepsilon,X \times Y}"] \\
X \times SY \arrow[r, "str_{X,Y}"'] & S(X \times Y)
\end{tikzcd}
\qquad
\begin{tikzcd}
T\varepsilon X \arrow[d, "(T(\varepsilon \leq \varepsilon'))_X"'] \arrow[dr, "m_{\varepsilon,X}"] & \\
T\varepsilon' X \arrow[r, "m_{\varepsilon',X}"'] & SX
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[d, "\eta_X"'] \arrow[dr, "\eta_X"] & \\
T1X \arrow[r, "m_{1,X}"'] & SX
\end{tikzcd}
\qquad
\begin{tikzcd}
T\varepsilon(T\varepsilon' X) \arrow[r, "m_{\varepsilon,T\varepsilon' X}"] \arrow[d, "\mu_{\varepsilon,\varepsilon',X}"'] & S(T\varepsilon' X) \arrow[r, "Sm_{\varepsilon',X}"] & S(SX) \arrow[d, "\mu_X"] \\
T(\varepsilon \cdot \varepsilon')X \arrow[rr, "m_{\varepsilon \cdot \varepsilon',X}"'] & & SX
\end{tikzcd}$$

◀

The lemma we use to reduce the amount of effort required to specify graded strong **Poset**-monads is as follows.

Lemma 4.2.17 Suppose that $(\mathcal{E}, \leq, \cdot, 1)$ is a preordered monoid, **C** is a cartesian **Poset**-category, and S is a strong **Poset**-monad on **C**. The graded monad structure T of any grading of S is a graded strong **Poset**-monad.

Proof. We reduce each of the requirements in the definition of graded strong **Poset**-monad to the corresponding requirement on S by composing with m, and then using the diagrams in the definition of grading. This suffices because m is a full monomorphism. We do not give all of the cases since they are all similar to each other, we just give a few representative examples.

We must show that each $T\varepsilon$ forms a **Poset**-functor. Monotonicity holds because if $f \sqsubseteq g$ then:

$$m_{\varepsilon,Y} \circ T\varepsilon f \;=\; Sf \circ m_{\varepsilon,X} \;\sqsubseteq\; Sg \circ m_{\varepsilon,X} \;=\; m_{\varepsilon,Y} \circ T\varepsilon g$$

Composition is preserved because:

$$m_{\varepsilon,Z} \circ T\varepsilon g \circ T\varepsilon f \;=\; Sg \circ Sf \circ m_{\varepsilon,X} \;=\; S(g \circ f) \circ m_{\varepsilon,X} \;=\; m_{\varepsilon,Z} \circ T\varepsilon(g \circ f)$$

The associativity law for the strength holds because of the following diagram chase

$$\begin{tikzcd}
(X \times Y) \times T\varepsilon Z \arrow[rrr, "str_{\varepsilon,X\times Y,Z}"] \arrow[dr, "(X\times Y)\times m_{\varepsilon,Z}"] \arrow[dd, "assoc"'] & & & T\varepsilon((X \times Y) \times Z) \arrow[dl, "m_{\varepsilon,(X\times Y)\times Z}"'] \arrow[d, "T\varepsilon assoc"] \\
& (X \times Y) \times SZ \arrow[r, "str_{X\times Y,Z}"] \arrow[d, "assoc"] & S((X \times Y) \times Z) \arrow[dr, "Sassoc"] & T\varepsilon(X \times (Y \times Z)) \arrow[d, "m_{\varepsilon,X\times(Y\times Z)}"] \\
X \times (Y \times T\varepsilon Z) \arrow[r, "X\times(Y\times m_{\varepsilon,Z})"] \arrow[dr, bend right, "X\times str_{\varepsilon,Y,Z}"'] & X \times (Y \times SZ) \arrow[r, "X\times str_{Y,Z}"] & X \times S(Y \times Z) \arrow[r, "str_{X,Y\times Z}"] & S(X \times (Y \times Z)) \\
& X \times T\varepsilon(Y \times Z) \arrow[ur, bend right, "X\times m_{\varepsilon,Y\times Z}"] \arrow[r, "str_{\varepsilon,X,Y\times Z}"'] & T\varepsilon(X \times (Y \times Z)) \arrow[ur, bend right, "m_{\varepsilon,X\times(Y\times Z)}"'] &
\end{tikzcd}$$

where the centre is the associativity law for the strength of S, and other parts of the diagram are from the definition of grading.

The left- and right-unit laws hold because of the following diagram chase

$$\begin{tikzcd}
T\varepsilon X \arrow[rrr, "m_{\varepsilon,X}"] \arrow[dr, "\eta_{T\varepsilon X}"] \arrow[dd, "\eta_{T\varepsilon X}"'] & & & SX \arrow[dl, "\eta_{SX}"] \arrow[dr, "S\eta_X"] \arrow[dd, equal] & & & T\varepsilon X \arrow[lll, "m_{\varepsilon,X}"'] \arrow[dl, "T\varepsilon\eta_X"] \arrow[dd, "T\varepsilon\eta_X"] \\
& S(T\varepsilon X) \arrow[r, "Sm_{\varepsilon,X}"] & S(SX) \arrow[dr, "\mu_X"] & & S(SX) \arrow[dl, "\mu_X"'] & T\varepsilon(SX) \arrow[l, "m_{\varepsilon,SX}"'] & \\
T1(T\varepsilon X) \arrow[ur, "m_{\varepsilon,T\varepsilon X}"] \arrow[rr, "\mu_{1,\varepsilon,X}"'] & & T\varepsilon X \arrow[r, "m_{\varepsilon,X}"'] & SX & T\varepsilon X \arrow[l, "m_{\varepsilon,X}"] & & T\varepsilon(T1X) \arrow[ll, "\mu_{\varepsilon,1,X}"] \arrow[ul, "T\varepsilon m_{1,X}"']
\end{tikzcd}$$

The centre is the left- and right-unit laws for S; the other parts are diagrams from the definition of grading (other than the pentagon on the bottom-right, which combines two of them). □

To further reduce the amount of work required to construct graded monads, we could also attempt to construct gradings from monads. One method of doing this is given for Gifford-style effect algebras by Kammar and McDermott [41].

4.2.3.5 Term model

The last model we describe is the *term model* of GCBPV, which is a purely syntactic model, for example, the morphisms of the value category are (equivalence classes of) GCBPV values. Terms are interpreted essentially as themselves. We do not use this to model any of our examples, and it is not itself useful for proving the validity of program transformations. However, it is useful for showing that models (such as the examples we give above) are *adequate*. In Section 4.3 we use the term model to prove adequacy.

Recall that for any given inequational theory $\preccurlyeq$ we write $\equiv$ for the intersection of $\preccurlyeq$ and its converse. For each typing context Γ and value type A, this is an equivalence relation on closed values of type A in context Γ, and similarly for computation types.

We first define the value category $\mathbf{C}_\lambda$. Objects are value types A. Morphisms from A to B are equivalence classes of values $x : A \vdash V : B$. We write $[x : A \vdash V : B]$, or just $[V]$, for such a morphism. The ordering on morphisms is $\preccurlyeq$, The identity on A is $[x]$, and the composition of $[V]$ and $[W]$ is $[V[x \mapsto W]]$.

The computation category $\mathbf{D}_\lambda$ is more difficult because contexts contain only value types, and because we need to be able to define the action $\circledast$. First we define a notion of *linearity* for computations, similar to linearity for natural transformations (Definition 4.2.4).

Definition 4.2.18 A family $(M^\varepsilon)_{\varepsilon \in \mathcal{E}}$ of computations

$$y : \mathbf{U}\,(\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}) \mathrel{\underline{\vdash}} M^\varepsilon : \langle\!\langle \varepsilon \rangle\!\rangle \underline{D}$$

is *natural* if

$$y : \mathbf{U}\,(\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}) \mathrel{\underline{\vdash}} \mathbf{coerce}_{\langle\!\langle \varepsilon \rangle\!\rangle \underline{C} <: \langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}}\, M^\varepsilon \equiv M^{\varepsilon'}[y \mapsto \mathbf{coerce}_{\mathbf{U}\,\langle\!\langle \varepsilon \rangle\!\rangle \underline{C} <: \mathbf{U}\,\langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}}\, y] : \langle\!\langle \varepsilon' \rangle\!\rangle \underline{D}$$

for each $\varepsilon \leq \varepsilon' \in \mathcal{E}$, and is *linear* if

$$z : \mathbf{U}\,(\langle \varepsilon \rangle (\mathbf{U}\,(\langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}))) \mathrel{\underline{\vdash}} \mathbf{force}\, z\ \mathbf{to}\ y.\, M^{\varepsilon'} \equiv M^{\varepsilon \cdot \varepsilon'}[y \mapsto \mathbf{thunk}\,(\mathbf{force}\, z\ \mathbf{to}\ w.\, \mathbf{force}\, w)] : \langle\!\langle \varepsilon \cdot \varepsilon' \rangle\!\rangle \underline{D}$$

for each $\varepsilon, \varepsilon' \in \mathcal{E}$. ◀

(This definition of linearity is based on the one given by Munch-Maccagnoni [79] and Levy [60], but here we need natural *families* of computations because of the effect system.) Objects of the computation category $\mathbf{D}_\lambda$ are computation types $\underline{C}$. A morphism from $\underline{C}$ to $\underline{D}$ is a family $([M^\varepsilon])_{\varepsilon \in \mathcal{E}}$ of equivalence classes of computations

$$y : \mathbf{U}\,(\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}) \mathrel{\underline{\vdash}} M^\varepsilon : \langle\!\langle \varepsilon \rangle\!\rangle \underline{D}$$

such that $(M^\varepsilon)_{\varepsilon \in \mathcal{E}}$ is natural and linear. The order on morphisms is

$$([M^\varepsilon])_{\varepsilon \in \mathcal{E}} \sqsubseteq ([N^\varepsilon])_{\varepsilon \in \mathcal{E}} \quad \Leftrightarrow \quad \forall \varepsilon \in \mathcal{E}.\, M^\varepsilon \preccurlyeq N^\varepsilon$$

We sometimes leave the effect ε implicit and write $[y : \mathbf{U}\,\underline{C} \mathrel{\underline{\vdash}} M : \underline{D}]$, or just $[M]$, for these morphisms. The identity on $\underline{C}$ at ε is $[y : \mathbf{U}\,(\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}) \mathrel{\underline{\vdash}} \mathbf{force}\, y : \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}]$ and the composition of $[y : \mathbf{U}\,\underline{C}_2 \mathrel{\underline{\vdash}} M : \underline{C}_3]$ and $[y : \mathbf{U}\,\underline{C}_1 \mathrel{\underline{\vdash}} N : \underline{C}_2]$, for each effect, is $[y : \mathbf{U}\,\underline{C}_1 \mathrel{\underline{\vdash}} M[y \mapsto \mathbf{thunk}\, N] : \underline{C}_3]$.

For every morphism in the computation category that we define, each of the equivalence classes of terms is the same up to typing annotations. They can therefore be thought of as single terms that satisfy $y : \mathbf{U}\,\underline{C} \mathrel{\underline{\vdash}} M : \underline{D}$ and, by replacing typing annotations, can be typed as $y : \mathbf{U}\,(\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}) \mathrel{\underline{\vdash}} M^\varepsilon : \langle\!\langle \varepsilon \rangle\!\rangle \underline{D}$. They are essentially polymorphic in the effect ε. Of course, this is not the case for every computation (just as not every computation is linear), but this is not a problem because computations are interpreted as morphisms in the value category.

Bicartesian structure of $\mathbf{C}_\lambda$:

$$\begin{array}{rll}
\text{Terminal object} & \mathbf{unit} & \langle\rangle_A \coloneqq [()] \\
\text{Binary products} & A_1 \times A_2 & \pi_1 \coloneqq [\mathbf{fst}\,x] \quad \pi_2 \coloneqq [\mathbf{snd}\,x] \\
\text{Initial object} & \mathbf{empty} & []_A \coloneqq [\mathbf{case}_A\; x\; \mathbf{of}\; \{\}] \\
\text{Binary coproducts} & A_1 + A_2 & \mathit{inl} \coloneqq [\mathbf{inl}_{A_2} x] \quad \mathit{inr} \coloneqq [\mathbf{inr}_{A_1} x]
\end{array}$$

Cartesian structure of $\mathbf{D}_\lambda$:

$$\begin{array}{rll}
\text{Terminal object} & \underline{\mathbf{unit}} & \langle\rangle_{\underline{C}} \coloneqq [\lambda\{\}] \\
\text{Binary products} & \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 & \pi_1 \coloneqq [1`\mathbf{force}\,y] \quad \pi_2 \coloneqq [2`\mathbf{force}\,y]
\end{array}$$

Exponentials:

$$(A \Rightarrow \underline{C}) \coloneqq (A \to \underline{C}) \qquad (A \Rightarrow [M]) \coloneqq [\lambda z{:}A.\, M[y \mapsto \mathbf{thunk}\,(z`\mathbf{force}\,y)]]$$
$$\Lambda[V] \coloneqq [\mathbf{thunk}\,(\lambda z{:}A.\, \mathbf{force}\,(V[x \mapsto (x, z)]))]$$

Graded strong **Poset**-adjunction:

$$\begin{array}{lll}
U : \mathbf{D}_\lambda \to \mathbf{C}_\lambda & U\underline{C} \coloneqq \mathbf{U}\,\underline{C} & U[y : \mathbf{U}\,\underline{C} \mathrel{\underline{\vdash}} M : \underline{C}'] \coloneqq [x : \mathbf{U}\,\underline{C} \vdash \mathbf{thunk}\,M[y \mapsto x] : \mathbf{U}\,\underline{C}'] \\
F : \mathbf{C}_\lambda \to \mathbf{D}_\lambda & FA \coloneqq \langle 1\rangle A & F[x : A \vdash V : B] \coloneqq [y : \mathbf{U}\,\langle\varepsilon\rangle A \mathrel{\underline{\vdash}} \mathbf{force}\,y\; \mathbf{to}\; x.\, \langle V\rangle : \langle\varepsilon\rangle B]
\end{array}$$

$$\varepsilon' \circledast \underline{C} \coloneqq \langle\!\langle\varepsilon'\rangle\!\rangle\underline{C} \quad (\varepsilon' \le \varepsilon'') \circledast \underline{C} \coloneqq [\mathbf{coerce}_{\langle\!\langle\varepsilon\cdot\varepsilon'\rangle\!\rangle\underline{C} <: \langle\!\langle\varepsilon\cdot\varepsilon'\rangle\!\rangle\underline{C}}\,(\mathbf{force}\,y)] \quad \varepsilon' \circledast ([M^{\varepsilon}])_{\varepsilon\in\mathcal{E}} \coloneqq ([M^{\varepsilon\cdot\varepsilon'}])_{\varepsilon\in\mathcal{E}}$$

$$\eta_A \coloneqq [\mathbf{thunk}\,\langle x\rangle] \qquad \varpi_{\underline{C}} \coloneqq [\mathbf{force}\,y\; \mathbf{to}\; z.\, \mathbf{force}\,z]$$
$$\mathit{str}_{\varepsilon,A,B} \coloneqq [\mathbf{thunk}\,(\mathbf{force}\,(\mathbf{fst}\,x)\; \mathbf{to}\; z.\, \langle(z, \mathbf{snd}\,x)\rangle)]$$

Base types, constants and operations:

$$[\![b]\!] \coloneqq b \qquad [\![c]\!] \coloneqq [c] \qquad [\![\mathrm{op}]\!] \coloneqq [\mathbf{thunk}\,(\mathrm{op}\,x)]$$

Figure 4.4: The GCBPV term model

The remaining structure of the term model is defined in Figure 4.4. The interesting part is the definition of the graded strong **Poset**-adjunction. In the definition of the right adjoint U on morphisms, only the equivalence class for the effect 1 is used (similar to the right adjoint in monadic models (Section 4.2.3.4)). The definition of the action $\circledast$ is where it is important that we have natural families of computations as morphisms of $\mathbf{D}_\lambda$ rather than just individual computations. In particular, the definition of $\varepsilon\circledast-$ on morphisms reindexes the family given (this is why it is the only definition in which we make the indices explicit). Naturality of the counit ϖ follows from linearity of morphisms in $\mathbf{D}_\lambda$.

The term model is in fact a model, as the following lemma shows.

Lemma 4.2.19 Given any inequational theory, the data given above forms a GCBPV structure in which the interpretation of terms satisfies

$$\Gamma \vdash V \preccurlyeq W : A \iff [\![V]\!] \sqsubseteq [\![W]\!] \qquad \Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C} \iff [\![M]\!] \sqsubseteq [\![N]\!]$$

Hence this GCBPV structure is a GCBPV model.

Proof sketch. Showing that this data forms a GCBPV structure just requires checking each of the requirements in turn. All are easy to check using the core axioms of inequational theories.

The fact about the interpretations of terms holds because terms are interpreted essentially as themselves. Formally, for each typing context Γ, there is a substitution σ_Γ such that

$$x : \llbracket \Gamma \rrbracket \vdash \sigma_\Gamma : \Gamma \qquad \llbracket \Gamma \vdash V : A \rrbracket = [V[\sigma_\Gamma]] \qquad \llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket = [\mathbf{thunk}\, M[\sigma_\Gamma]]$$

There is also a substitution σ_Γ^{-1} that behaves as an inverse to σ_Γ. It satisfies

$$\Gamma \vdash \sigma_\Gamma^{-1} : (x : \llbracket \Gamma \rrbracket) \qquad V[\sigma_\Gamma][\sigma_\Gamma^{-1}] \equiv V \qquad M[\sigma_\Gamma][\sigma_\Gamma^{-1}] \equiv M$$

Hence on computations M we have:

$$\begin{aligned} & M \preccurlyeq M' \\ \Rightarrow\ & \mathbf{thunk}\, M[\sigma_\Gamma] \preccurlyeq \mathbf{thunk}\, M[\sigma_\Gamma] \\ \Rightarrow\ & \llbracket M \rrbracket \sqsubseteq \llbracket M' \rrbracket \end{aligned} \qquad \begin{aligned} & \llbracket M \rrbracket \sqsubseteq \llbracket M' \rrbracket \\ \Rightarrow\ & \mathbf{thunk}\, M[\sigma_\Gamma] \preccurlyeq \mathbf{thunk}\, M[\sigma_\Gamma] \\ \Rightarrow\ & \mathbf{force}\,\mathbf{thunk}\, M[\sigma_\Gamma][\sigma_\Gamma^{-1}] \preccurlyeq \mathbf{force}\,\mathbf{thunk}\, M[\sigma_\Gamma][\sigma_\Gamma^{-1}] \\ \Rightarrow\ & M \sqsubseteq M' \end{aligned}$$

Values are similar. □

This lemma actually shows that the term model is *complete* as well as sound. Completeness is the crucial property of the term model: it allows us to use term model to show adequacy of our other models (Section 4.3 shows how to do this). Moreover, the fact that the syntax of GCBPV forms a model shows that we have not included too many requirements in the definition of model. The definition exactly matches the inequational theory in the following sense:

Corollary 4.2.20 Given any inequational theory:

1. $\llbracket \Gamma \vdash V : A \rrbracket \sqsubseteq \llbracket \Gamma \vdash W : A \rrbracket$ for every model of $\preccurlyeq$ if and only if $\Gamma \vdash V \preccurlyeq W : A$.
2. $\llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket \sqsubseteq \llbracket \Gamma \mathrel{\underline{\vdash}} N : \underline{C} \rrbracket$ for every model of $\preccurlyeq$ if and only if $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$. ◀

4.3 Relating syntax and semantics

The previous sections describe examples of program transformations we would like to perform by conjecturing instances $M \preccurlyeq_{\mathrm{ctx}} N$ of the contextual preorder, and also show for each of these instances that $\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$. To prove the validity of the program transformations, we show that our example models are adequate:

Definition 4.3.1 (Adequacy) A model of some GCBPV inequational theory is *adequate* if both of the following hold:

- (Values) $\llbracket \Gamma \vdash V : A \rrbracket \sqsubseteq \llbracket \Gamma \vdash W : A \rrbracket$ implies $\Gamma \vdash V \preccurlyeq_{\mathrm{ctx}} W : A$.
- (Computations) $\llbracket \Gamma \mathrel{\underline{\vdash}} M : \underline{C} \rrbracket \sqsubseteq \llbracket \Gamma \mathrel{\underline{\vdash}} N : \underline{C} \rrbracket$ implies $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq_{\mathrm{ctx}} N : \underline{C}$. ◀

Adequacy can be formulated in terms of the more general problem of *abstraction* between models of GCBPV.

Definition 4.3.2 (Abstraction) Suppose that $\mathcal{M}_1$ and $\mathcal{M}_2$ are models of the same inequational theory. We say that $\mathcal{M}_2$ *abstracts* $\mathcal{M}_1$ if for all ground types G, effects ε and closed computations $M, N : \langle \varepsilon \rangle G$ we have

$$\mathcal{M}_1\llbracket M \rrbracket \sqsubseteq \mathcal{M}_1\llbracket N \rrbracket \qquad \Rightarrow \qquad \mathcal{M}_2\llbracket M \rrbracket \sqsubseteq \mathcal{M}_2\llbracket N \rrbracket$$ ◀

Adequacy is a special case of abstraction because of the following lemma.

Lemma 4.3.3 A model $\mathcal{M}$ of some inequational theory is adequate if and only if the term model (Section 4.2.3.5) abstracts $\mathcal{M}$.

Proof. Since the term model is complete (Lemma 4.2.19), it abstracts $\mathcal{M}$ if and only if for all ground types G, effects ε and closed computations $M, N : \langle\varepsilon\rangle G$ we have

$$\mathcal{M}\llbracket M \rrbracket \sqsubseteq \mathcal{M}\llbracket N \rrbracket \qquad \Rightarrow \qquad M \preccurlyeq N$$

so it suffices to show that this property is equivalent to adequacy.

Suppose that this property holds and that $\mathcal{M}\llbracket V \rrbracket \sqsubseteq \mathcal{M}\llbracket W \rrbracket$ for well-typed values V, W. By compositionality (Lemma 4.2.6 (1)) we have $\mathcal{M}\llbracket \underline{C}[V] \rrbracket \sqsubseteq \mathcal{M}\llbracket \underline{C}[W] \rrbracket$. So if $\underline{C}[V]$ and $\underline{C}[W]$ satisfy the conditions on M and N we have $\underline{C}[V] \preccurlyeq \underline{C}[W]$. Hence $V \preccurlyeq_{\mathrm{ctx}} W$. Similarly for computations.

If $\mathcal{M}$ is adequate and $\mathcal{M}\llbracket M \rrbracket \sqsubseteq \mathcal{M}\llbracket N \rrbracket$ then $M \preccurlyeq_{\mathrm{ctx}} N$, which implies $M \preccurlyeq N$ by applying the definition of the contextual preorder to the term context $\underline{\square}$. □

We describe a method for proving instances of abstraction, and then show that our example models are adequate using Lemma 4.3.3.

4.3.1 Logical relations and abstraction

To prove abstraction between models, we use logical relations. We give a general categorical description of logical relations for GCBPV that can be used to state sufficient conditions for abstraction to hold. This is based on the work of Hermida [36] and Katsumata [44], adapted to our **Poset**-enriched models. This description of logical relations is more general than the one we gave in Section 3.1: instead of covering just binary relations between terms, our new definition covers relations of other arities (e.g. unary relations, and relations of varying arity), and are not restricted to relating terms (they can relate denotations).

These relations are formulated in terms of functors $p : \mathbf{R}_{\mathbf{C}} \to \mathbf{C}$ from an (ordinary) category $\mathbf{R}_{\mathbf{C}}$ into (the underlying ordinary category of) a **Poset**-category $\mathbf{C}$. Here $\mathbf{C}$ is either the value or computation category of a model of GCBPV (we actually require two such functors: one for the value category and one for the computation category). An object $P \in \mathbf{R}_{\mathbf{C}}$ should be thought of as a unary predicate on $pP \in \mathbf{C}$, and a morphism $\dot{f} : P \to Q$ as a witness that $p\dot{f} : pP \to pQ$ is truth-preserving. We assume that the functor p is faithful (i.e. for all $\dot{f}, \dot{g} : P \to Q$ if $p\dot{f} = p\dot{g}$ then $\dot{f} = \dot{g}$), which implies such witnesses are unique. We say that an object $P \in \mathbf{R}_{\mathbf{C}}$ is *above* $X \in \mathbf{C}$ if $pP = X$, and for $f : pP \to pQ$ we write $f : P \dot{\to} Q$ to mean there is some $\dot{f}$ such that $p\dot{f} = f$ (i.e. f is truth-preserving).

The general idea is to *lift* the structure of some GCBPV model (in **Poset**-categories $\mathbf{C}$ and $\mathbf{D}$) to categories $\mathbf{R}_{\mathbf{C}}$ and $\mathbf{R}_{\mathbf{D}}$ of relations, and hence construct a new *logical-relations* (LR) model (we define LR model below). In the LR model, value types A are interpreted as relations $\mathcal{R}\llbracket A \rrbracket \in \mathbf{R}_{\mathbf{C}}$ and computation types $\underline{C}$ are interpreted as relations $\mathcal{R}\llbracket \underline{C} \rrbracket \in \mathbf{R}_{\mathbf{D}}$.

The next definition captures much of the required structure. It is essentially the same as a definition given by Ma and Reynolds [65].

Definition 4.3.4 Suppose that $\mathbf{C}$ is a cartesian **Poset**-category and $p : \mathbf{R}_{\mathbf{C}} \to \mathbf{C}$ is a functor. A *lifting* of the cartesian structure of $\mathbf{C}$ consists of:

- An $\mathbf{R}_{\mathbf{C}}$-object $\dot{1}$ above 1, such that for each P, we have $\langle\rangle_{pP} : P \dot{\to} \dot{1}$.
- For each P_1, P_2, an $\mathbf{R}_{\mathbf{C}}$-object $P_1 \dot{\times} P_2$ above $pP_1 \times pP_2$ such that $\pi_1 : P_1 \dot{\times} P_2 \dot{\to} P_1$, $\pi_2 : P_1 \dot{\times} P_2 \dot{\to} P_2$, and for each P, f_1, f_2 if $f_1 : Q \dot{\to} P_1$ and $f_2 : Q \dot{\to} P_2$ then $\langle f_1, f_2 \rangle : Q \dot{\to} P_1 \dot{\times} P_2$.

If $\mathbf{C}$ is bicartesian, a *lifting* of the bicartesian structure additionally consists of:

- An $\mathbf{R_C}$-object $\dot{0}$ above 0, such that for each P, we have $[]_{pP} : \dot{0} \dot{\to} P$.
- For each P_1, P_2, an $\mathbf{R_C}$-object $P_1 \dot{+} P_2$ above $pP_1 + pP_2$ such that $inl : P_1 \dot{\to} P_1 \dot{+} P_2$, $inr : P_2 \dot{\to} P_1 \dot{+} P_2$, and for each P, f_1, f_2 if $f_1 : P_1 \dot{\to} Q$ and $f_2 : P_2 \dot{\to} Q$ then $[f_1, f_2] : P_1 \dot{+} P_2 \dot{\to} Q$.

Finally, if $\mathbf{C}$ is distributive, a *lifting* of the distributive structure is a lifting of the bicartesian structure such that $dist_{pP,pQ,pR} : pP \dot{\times} (pQ \dot{+} pR) \dot{\to} (pP \dot{\times} pQ) \dot{+} (pP \dot{\times} pR)$ for all P, Q, R. ◄

Example 4.3.5 For **Poset**, we define the category **Pred** as follows. Objects $P \subseteq X$ are pairs of a poset X and an arbitrary set P of elements of X; we call these *predicates*. The predicate $P \subseteq X$ is considered to be true on elements of P, and false on the other elements of X. Morphisms $f : (P \subseteq X) \to (Q \subseteq Y)$ are monotone functions $f : X \to Y$ such that $f\,x \in Q$ for all $x \in P$. There is a functor $p : \mathbf{Pred} \to \mathbf{Poset}$ that sends $(P \subseteq X)$ to X and f to itself. In this case, $f : (P \subseteq X) \dot{\to} (Q \subseteq Y)$ holds iff $x \in P$ implies $f\,x \in Q$ for all x. We lift the bicartesian structure of **Poset** as follows:

$$\begin{aligned} \dot{1} &:= \{\star\} \subseteq 1 & \qquad (P_1 \subseteq X_1) \dot{\times} (P_2 \subseteq X_2) &:= (P_1 \times P_2) \subseteq (X_1 \times X_2) \\ \dot{0} &:= \emptyset \subseteq 0 & \qquad (P_1 \subseteq X_1) \dot{+} (P_2 \subseteq X_2) &:= (P_1 + P_2) \subseteq (X_1 + X_2) \end{aligned}$$

◄

Given a faithful functor $p : \mathbf{R_C} \to \mathbf{C}$ we **Poset**-enrich $\mathbf{R_C}$ in the following way: if $\dot{f}, \dot{g} : P \to Q$ are morphisms in $\mathbf{R_C}$ then $\dot{f} \sqsubseteq \dot{g}$ if $p\dot{f} \sqsubseteq p\dot{g}$ (this is a partial order because f is faithful). Any lifting of the cartesian (or bicartesian) structure on $\mathbf{C}$ induces a cartesian (bicartesian) structure on $\mathbf{R_C}$: we use the objects and witnesses required in the definition of lifting (e.g. the product of P_1 and P_2 is $P_1 \dot{\times} P_2$, where each projection $\dot{\pi}_i : P_1 \dot{\times} P_2 \to P_i$ is the witness of $\pi_i : P_1 \dot{\times} P_2 \dot{\to} P_i$).

We also require liftings of certain **Poset**-functors.

Definition 4.3.6 Suppose that $p : \mathbf{R_C} \to \mathbf{C}$ and $q : \mathbf{R_D} \to \mathbf{D}$ are functors. A *lifting* of a **Poset**-functor $G : \mathbf{C} \to \mathbf{D}$ consists of an object $\dot{G}P \in \mathbf{R_D}$ that satisfies $q(\dot{G}P) = G(pP)$ for each $P \in \mathbf{R_C}$, such that for each f if $f : P \dot{\to} Q$ then $Gf : \dot{G}P \dot{\to} \dot{G}Q$. ◄

The data required for a lifting of a **Poset**-functor $G : \mathbf{C} \to \mathbf{D}$ forms a **Poset**-functor $\dot{G} : \mathbf{R_C} \to \mathbf{R_D}$ such that $G \circ p = q \circ \dot{G}$.

In the following definitions, we suppose we are given some model $\mathcal{M}$ of GCBPV with value category $\mathbf{C}$ and computation category $\mathbf{D}$. To form an LR model we need to be able to lift the graded adjunction.

Definition 4.3.7 Suppose that $p : \mathbf{R_C} \to \mathbf{C}$ and $q : \mathbf{R_D} \to \mathbf{D}$ are functors, and that $\mathbf{R_C}$ has an object $P_1 \dot{\times} P_2$ such that $p(P_1 \dot{\times} P_2) = pP_1 \times pP_2$ for each pair of objects $P_1, P_2 \in \mathbf{R_C}$. A *lifting* of the graded strong **Poset**-adjunction $(F, U, \circledast)$ consists of liftings of the **Poset**-functors $F : \mathbf{C} \to \mathbf{D}$, $U : \mathbf{D} \to \mathbf{C}$ and $\varepsilon \circledast - : \mathbf{D} \to \mathbf{D}$ for each $\varepsilon \in \mathcal{E}$, such that

- For each $\varepsilon, \varepsilon'$ we have $(\varepsilon \cdot \varepsilon') \dot{\circledast} - = \varepsilon \dot{\circledast} (\varepsilon' \dot{\circledast} -)$, and $1 \dot{\circledast} -$ is the identity.
- For each ε, P, Q we have $str_{\varepsilon,pP,pQ} : P \dot{\times} \dot{U}(\varepsilon \dot{\circledast} \dot{F}Q) \dot{\to} \dot{U}(\varepsilon \dot{\circledast} \dot{F}(P \dot{\times} Q))$.
- For each $\varepsilon, \varepsilon', \underline{Q}$, if $\varepsilon \leq \varepsilon'$ then $(\varepsilon \leq \varepsilon') \circledast q\underline{Q} : (\varepsilon \dot{\circledast} \underline{Q}) \dot{\to} (\varepsilon' \dot{\circledast} \underline{Q})$.
- For each P we have $\eta_{pP} : P \dot{\to} \dot{U}\dot{F}P$.
- For each $\underline{Q}$ we have $\varpi_{q\underline{Q}} : \dot{F}\dot{U}\underline{Q} \dot{\to} \underline{Q}$. ◄

Each lifting of $(F, U, \circledast)$ determines a graded strong **Poset**-adjunction $(\dot{F}, \dot{U}, \dot{\circledast})$ on $\mathbf{D}$, where each of the required morphisms is given by a witness. In general, the correct choice of lifting depends on the specific effects being modelled, though some general techniques for constructing liftings of monads (in the non-enriched case) exist [32, 43, 46]. We also expect that a version of the *free lifting* (Section 3.1.1) could be developed for graded adjunctions (this would be similar to the monadic version given by Kammar and McDermott [41]). We do not use these general techniques, but instead construct liftings specific to our examples (see Section 4.3.2).

Recall from Section 3.1 that the crucial part of the definition of a logical relation is its definition on returner types. Since left adjoints are unique, the definition on returner types is uniquely determined by the right adjoint in this case. The crucial part of the definition here is the choice of $\mathbf{R}_{\mathbf{D}}$.

We collect the data required to form an LR model $\mathcal{R}$ in the following definition. (We write $\mathcal{M}[\![-]\!]$ for interpretations in the base model $\mathcal{M}$, and $\mathcal{R}[\![-]\!]$ for interpretations in the logical-relations model $\mathcal{R}$.)

Definition 4.3.8 A *logical-relations* (LR) model above the model $\mathcal{M}$ consists of

- Categories $\mathbf{R}_{\mathbf{C}}$ and $\mathbf{R}_{\mathbf{D}}$ and functors $p : \mathbf{R}_{\mathbf{C}} \to \mathbf{C}$ and $q : \mathbf{R}_{\mathbf{D}} \to \mathbf{D}$, together with liftings of the distributive structure of $\mathbf{C}$ and the cartesian structure of $\mathbf{D}$.
- For each $P \in \mathbf{R}_{\mathbf{C}}$, a lifting $P \mathrel{\dot{\Rightarrow}} - : \mathbf{R}_{\mathbf{D}} \to \mathbf{R}_{\mathbf{D}}$ of the **Poset**-functor $pP \Rightarrow - : \mathbf{D} \to \mathbf{D}$, such that for all f, f' if $f : R \mathbin{\dot\times} P \mathrel{\dot\to} \dot{U}\underline{Q}$ then $\Lambda f : R \mathrel{\dot\to} \dot{U}(P \mathrel{\dot\Rightarrow} \underline{Q})$ and if $f' : R \mathrel{\dot\to} \dot{U}(P \mathrel{\dot\Rightarrow} \underline{Q})$ then $\Lambda^{-1} f' : R \mathbin{\dot\times} P \mathrel{\dot\to} \dot{U}\underline{Q}$.
- A lifting of the graded strong **Poset**-adjunction.
- A $\mathbf{R}_{\mathbf{C}}$-object $\mathcal{R}[\![b]\!]$ above $\mathcal{M}[\![b]\!]$ for each base type b.

such that $\mathcal{M}[\![c]\!] : \dot{1} \mathrel{\dot\to} \mathcal{R}[\![A]\!]$ for each constant $c \in \mathcal{K}_A$ and

$$\mathcal{M}[\![\mathrm{op}]\!] : \mathcal{R}[\![\mathrm{car}_{\mathrm{op}}]\!] \mathrel{\dot\to} \dot{U}(\mathrm{eff}_{\mathrm{op}} \mathbin{\dot\circledast} \dot{F}(\mathcal{R}[\![\mathrm{ar}_{\mathrm{op}}]\!]))$$

for each operation $\mathrm{op} \in \Sigma$. ◄

Soundness of any such model follows from the definition of the order on morphisms in $\mathbf{R}_{\mathbf{C}}$ and $\mathbf{R}_{\mathbf{D}}$ and soundness of the base model $\mathcal{M}$.

As usual, the logical relation satisfies a *fundamental lemma*, which states that interpretations of terms preserve relations.

Lemma 4.3.9 (Fundamental) If $\mathcal{R}$ is an LR model over $\mathcal{M}$ then

1. If $\Gamma \vdash V : A$ then $\mathcal{M}[\![V]\!] : \mathcal{R}[\![\Gamma]\!] \mathrel{\dot\to} \mathcal{R}[\![A]\!]$.
2. If $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ then $\mathcal{M}[\![M]\!] : \mathcal{R}[\![\Gamma]\!] \mathrel{\dot\to} \dot{U}(\mathcal{R}[\![\underline{C}]\!])$.

Proof. By a trivial induction, the interpretations $\mathcal{R}[\![V]\!]$ and $\mathcal{R}[\![M]\!]$ in the LR model satisfy $p(\mathcal{R}[\![V]\!]) = \mathcal{M}[\![V]\!]$ and $p(\mathcal{R}[\![M]\!]) = \mathcal{M}[\![M]\!]$, and hence are the required witnesses. □

To prove abstraction, we need to relate two models. Given two **Poset**-categories $\mathbf{C}_1$ and $\mathbf{C}_2$, define $\mathbf{C}_1 \times \mathbf{C}_2$ as the **Poset**-category in which objects are pairs (X_1, X_2) with $X_1 \in \mathbf{C}_1$ and $X_2 \in \mathbf{C}_2$, and morphisms (f_1, f_2) are similarly pairs of morphisms (ordered componentwise). Given models $\mathcal{M}_1$ (with value category $\mathbf{C}_1$ and computation category $\mathbf{D}_1$) and $\mathcal{M}_2$ (with value category $\mathbf{C}_2$ and computation category $\mathbf{D}_2$), we define their *product* $\mathcal{M}_1 \times \mathcal{M}_2$ as the model with value category $\mathbf{C}_1 \times \mathbf{C}_2$ and computation category $\mathbf{D}_1 \times \mathbf{D}_2$, where each part of the structure is given by pairing the structure in $\mathcal{M}_1$ and $\mathcal{M}_2$. The product model satisfies $(\mathcal{M}_1 \times \mathcal{M}_2)[\![-]\!] = (\mathcal{M}_1[\![-]\!], \mathcal{M}_2[\![-]\!])$.

To relate $\mathcal{M}_1$ to $\mathcal{M}_2$, we consider *binary* functors $p : \mathbf{R}_{\mathbf{C}_1 \times \mathbf{C}_2} \to \mathbf{C}_1 \times \mathbf{C}_2$, in which an object $R \in \mathbf{R}_{\mathbf{C}_1 \times \mathbf{C}_2}$ above (X_1, X_2) relates X_1 to X_2. A key part of the proof of abstraction is to show that certain relations are *monotone*:

Definition 4.3.10 Suppose that $\mathbf{C}_1$ and $\mathbf{C}_2$ are **Poset**-categories and $p : \mathbf{R}_{\mathbf{C}_1 \times \mathbf{C}_2} \to \mathbf{C}_1 \times \mathbf{C}_2$ is a faithful functor. An object $R \in \mathbf{D}$ is *monotone* if whenever $(f_1, f_2) : \dot{1} \dot{\to} R$ and $(g_1, g_2) : \dot{1} \dot{\to} R$,

$$f_1 \sqsubseteq g_1 \quad \Rightarrow \quad f_2 \sqsubseteq g_2 \qquad ◀$$

Note the similarity with the implication in the definition of abstraction (Definition 4.3.2). The relations $\dot{1}$ and $\dot{0}$ are monotone (the latter because we assume distributivity[7]), and if R_1, R_2 are monotone then so is $R_1 \mathbin{\dot\times} R_2$ (if these relations exist). We say that monotone relations are *closed under coproducts* if $R_1 \mathbin{\dot+} R_2$ is monotone for monotone R_1, R_2 (this does not hold in general).

Example 4.3.11 The functor **BRel** → **Poset** × **Poset**, where **BRel** is similar to **Pred** except that objects are binary relations $R \subseteq X_1 \times X_2$, could be used to relate two models on the value category **Poset**. In this case, monotonicity of R just means that if $(a_1, a_2), (b_1, b_2) \in R$ then $a_1 \sqsubseteq b_1$ implies $a_2 \sqsubseteq b_2$. ◀

Using this definition of monotonicity of relations, we give sufficient conditions for one model of GCBPV to abstract another. This is the key lemma that allows us to prove abstraction. We then use it to prove adequacy of our example models.

Lemma 4.3.12 (Abstraction) Suppose that $\mathcal{M}_1$ and $\mathcal{M}_2$ are models of the same inequational theory. If there exists an LR model $\mathcal{R}$ above $\mathcal{M}_1 \times \mathcal{M}_2$ such that each $\mathcal{R}[\![b]\!]$ is monotone, $\dot{U}(\varepsilon \mathbin{\dot\circledast} \dot{F} R)$ is monotone for each effect ε and monotone R, and monotone relations in $\mathbf{R}_\mathbf{C}$ are closed under coproducts, then $\mathcal{M}_2$ abstracts $\mathcal{M}_1$.

Proof. An easy induction on ground types G shows that each $\mathcal{R}[\![G]\!]$ is monotone, and hence so is $\dot{U}(\mathcal{R}[\![\langle\varepsilon\rangle G]\!]) = \dot{U}(\varepsilon \mathbin{\dot\circledast} \mathcal{R}[\![G]\!])$. By the fundamental lemma (Lemma 4.3.9), if $M, N : \langle\varepsilon\rangle G$ are closed computations then

$$(\mathcal{M}_1[\![M]\!], \mathcal{M}_2[\![M]\!]), (\mathcal{M}_1[\![N]\!], \mathcal{M}_2[\![N]\!]) : \dot{1} \dot{\to} \mathcal{R}[\![\langle\varepsilon\rangle G]\!]$$

and the result follows from monotonicity. □

4.3.2 Examples

We now show that the models we use for each of our three examples are adequate, by using Lemma 4.3.12 to show that each example model $\mathcal{M}$ abstracts the term model.

We therefore need three LR models, one for each of our example models. Recall that each example model uses the value category **Poset**, and the value category of the term model is $\mathbf{C}_\lambda$. Also recall that we write Term_A and $\underline{\mathrm{Term}_{\underline{C}}}$ for the sets of equivalence classes of closed values and computations (we also omit square brackets around equivalence classes in this section). We use a common construction for $\mathbf{R}_{\mathbf{Poset} \times \mathbf{C}_\lambda}$:

- Objects are triples (X, A, R) where X is a poset, A is a value type, and $R \subseteq X \times \mathrm{Term}_A$ is a relation.
- Morphisms $(f, V) : (X, A, R) \to (Y, B, R')$ are pairs of a morphism $f : X \to Y$ in **Poset** and a morphism $W : A \to B$ in $\mathbf{C}_\lambda$, such that if $(x, N) \in R$ then $(fx, W[x \mapsto V]) \in R'$.

[7] To show that $\dot{0}$ is monotone, we use the fact that distributivity of binary coproducts implies [15, Proposition 3.4] the initial object is *strict* (all morphisms into 0 are isomorphisms), so any two morphisms $1 \to 0$ are equal.

We have a functor $\mathbf{R}_{\mathbf{Poset}\times\mathbf{C}_\lambda} \to \mathbf{Poset}\times\mathbf{C}_\lambda$ that sends (X, A, R) to (X, A), and is the identity on morphisms. For the lifting of the bicartesian structure, we take the bicartesian structure of **Poset** and $\mathbf{C}_\lambda$, together with the following relations:

$$\begin{array}{ll} \dot{1} := \{(\star, ())\} & R_1 \mathbin{\dot\times} R_2 := \{(x, V) \mid (\pi_1 x, \mathbf{fst}\, V) \in R_1 \wedge (\pi_2 x, \mathbf{snd}\, V) \in R_2\} \\ \dot{0} := \emptyset & R_1 \mathbin{\dot+} R_2 := \{(\mathit{inl}\, x, \mathbf{inl}\, V) \mid (x, V) \in R_1\} \cup \{(\mathit{inr}\, x, \mathbf{inr}\, V) \mid (x, V) \in R_2\} \end{array}$$

Note in particular that $\mathcal{R}[\![\mathbf{bool}]\!] = \{(\mathrm{true}, \mathbf{true}), (\mathrm{false}, \mathbf{false})\}$, and that (X, A, R) is monotone (Definition 4.3.10) if $x \sqsubseteq x'$ implies $V \preccurlyeq V'$ for all $(x, V), (x', V') \in R$. Monotone relations are closed under coproducts in this case, so we have one of the requirements of the abstraction lemma (Lemma 4.3.12).

Each of our examples has a different computation category, and hence we use different categories of relations on computations in each case. We go through each example in turn.

4.3.2.1 Undefined behaviour

For undefined behaviour, recall that we use the trivial effect algebra, and hence morphisms in the computation category $\mathbf{D}_\lambda$ are linear (Definition 4.2.18) equivalence classes of computations $y : \mathbf{U}\,\underline{C} \mathrel{\underline{\vdash}} N : \underline{D}$ We define $\mathbf{R}_{\mathbf{Poset}_\bot\times\mathbf{D}_\lambda}$ by:

- Objects $(X, \underline{C}, R)$ consist of a pointed poset X, a computation type $\underline{C}$, and a relation $R \subseteq X \times \underline{\mathrm{Term}}_{\underline{C}}$, such that $(\bot, \mathbf{undef}_{\underline{C}}) \in R$.
- Morphisms $(f, N) : (X, \underline{C}, R) \to (Y, \underline{D}, R')$ are pairs of a morphism $f : X \to Y$ in $\mathbf{Poset}_\bot$ and a morphism $N : \underline{C} \to \underline{D}$ in $\mathbf{D}_\lambda$, such that if $(x, M) \in R$ then $(f x, N[y \mapsto \mathbf{thunk}\, M]) \in R'$.

The crucial part of this definition is that $\bot$ is required to be related to $\mathbf{undef}_{\underline{C}}$. This ensures the counit lifts.

The functor $\mathbf{R}_{\mathbf{Poset}_\bot\times\mathbf{D}_\lambda} \to \mathbf{Poset}_\bot \times \mathbf{D}_\lambda$ forgets the relation (maps $(X, \underline{C}, R)$ to $(X, \underline{C})$ and (f, N) to itself). The cartesian structure on $\mathbf{R}_{\mathbf{Poset}_\bot\times\mathbf{D}_\lambda}$, and exponentials, are given by the corresponding structures in the underlying models and the relations

$$R_{\underline{1}} := \{(\star, \lambda\{\})\} \qquad R_1 \mathbin{\underline{\dot\times}} R_2 := \{(x, M) \mid (\pi_1 x, 1`M) \in R_1 \wedge (\pi_2 x, 2`M) \in R_2\}$$
$$R \mathrel{\dot\Rightarrow} R' := \{(f, M) \mid \forall (x, V) \in R.\, (f\, x, V`M) \in R'\}$$

To lift the graded adjunction, we define $\dot{U}(X, \underline{C}, R_1) := (X, \mathbf{U}\,\underline{C}, R_1')$ and $\dot{F}(X, A, R_2) := (X_\bot, \langle\star\rangle A, R_2')$ where

$$R_1' := \{(x, \mathbf{thunk}\, M) \mid (x, M) \in R_1\} \qquad R_2' := \{(x, \langle V\rangle) \mid (x, V) \in R_2\} \cup \{(\bot, \mathbf{undef}_{\langle\star\rangle A})\}$$

The idea behind the definition of the left adjoint $\dot{F}$ is that each computation of returner type, when executed, will either return some value V, or have undefined behaviour. The lifting of the action $\circledast$ is trivial. To complete the LR model we define the interpretation of the base type **int** to have relation $\{(n, \underline{n}) \mid n \in [\![\mathbf{int}]\!]\}$. The constants **add** and **geq** lift because of the axioms of the inequational theory that specify their behaviour. Similarly for the operation $\mathsf{add}_{\mathsf{nsw}}$. The operation undef lifts because the computation type $\langle\star\rangle\mathbf{empty}$ is interpreted as the relation $\{(\bot, \mathsf{undef}\ ())\}$. We therefore have a LR model that we can use to relate our model of undefined behaviour to the term model.

To apply the abstraction lemma (Lemma 4.3.12), we just need to check that $\mathcal{R}[\![\mathbf{int}]\!]$ is monotone, which is trivial, and that $\dot{U}(\dot{F}R)$ is monotone for monotone R. For the latter, suppose that (x, M) and (x', M') are in

$$\{(x, \langle V\rangle) \mid (x, V) \in R\} \cup \{(\bot, \mathbf{undef}_{\underline{C}})\}$$

and $x \sqsubseteq x'$. Then if $(x, M) = (x, \langle V \rangle)$ for $(x, V) \in R$ we have $(x', M) = (x', \langle V' \rangle)$ with $(x', V') \in R$ (because $x' \neq \bot$), so $M = \langle V \rangle \preccurlyeq \langle V' \rangle = M'$ because R is monotone. If $(x, M) = (\bot, \mathbf{undef}_{\underline{C}})$ then we have $M \preccurlyeq M'$ as one of the axioms of the inequational theory. So the abstraction lemma tells us the our model of undefined behaviour is adequate, and hence that the example program transformations we give in Section 4.1.1 are valid instances of the contextual preorder.

4.3.2.2 Nondeterminism

Recall that for nondeterminism we use the effects $1 \leq +$. We define the computation category $\mathbf{R}_{\mathbf{D}\times\mathbf{D}_\lambda}$ of the logical relations by:

- Objects $(X, \underline{C}, R)$ consist of an object $X = (X_1 \subseteq X_+) \in \mathbf{D}$ (i.e. a meet-semilattice X_+ with a subset X_1), a computation type $\underline{C}$, and a family of relations $(R_\varepsilon \subseteq X_\varepsilon \times \underline{\mathrm{Term}}_{\langle\!\langle \varepsilon \rangle\!\rangle \underline{C}})_{\varepsilon \in \{1,+\}}$, such that $(x, M) \in R_1$ implies $(x, \mathbf{coerce}_{\underline{C} <: \langle\!\langle + \rangle\!\rangle \underline{C}}\, M) \in R_+$, and $(x_1, M_1), (x_2, M_2) \in R_+$ implies $(x_1 \sqcap x_2, M_1\ \mathbf{or}\ M_2) \in R_+$.
- Morphisms $(f, N) : (X, \underline{C}, R) \longrightarrow (Y, \underline{D}, R')$ are pairs of a morphism $f : X \longrightarrow Y$ in $\mathbf{D}$ and a morphism $N : \underline{C} \longrightarrow \underline{D}$ in $\mathbf{D}_\lambda$, such that $(x, M) \in R_\varepsilon$ implies $(f x, N^\varepsilon[y \mapsto \mathbf{thunk}\, M]) \in R'_\varepsilon$ for $\varepsilon \in \{1, +\}$.

The relation R_1 relates deterministic computations (of type $\underline{C}$), and R_+ relates nondeterministic computations (of type $\langle\!\langle + \rangle\!\rangle \underline{C}$. The condition that meets are related ensures that the counit of the graded adjunction lifts. The functor $\mathbf{R}_{\mathbf{D}\times\mathbf{D}_\lambda} \longrightarrow \mathbf{D} \times \mathbf{D}_\lambda$ forgets the relation (maps $(X, \underline{C}, R)$ to $(X, \underline{C})$ and (f, N) to itself). The cartesian structure on $\mathbf{R}_{\mathbf{D}\times\mathbf{D}_\lambda}$, and exponentials, are defined in the same way as for undefined behaviour (on each effect ε).

To lift the graded adjunction, we define $\dot{U}(X, \underline{C}, R) \coloneqq (X_1, \mathbf{U}\,\underline{C}, R')$ where R' is defined in terms of the relation on deterministic computations:

$$R' \coloneqq \{(x, \mathbf{thunk}\, M) \mid (n, M) \in R_1\}$$

The lifting of the left adjoint is $\dot{F}(X, A, T) \coloneqq (FX, \langle 1 \rangle A, T')$, where

$$\begin{aligned} T'_1 &\coloneqq \{(\uparrow\{x\}, \langle V \rangle) \mid (x, V) \in T\} \\ T'_+ &\coloneqq \{(\uparrow\{x_1, \ldots, x_n\}, \mathbf{coerce}_{1\leq+}\, \langle V_1 \rangle\ \mathbf{or} \cdots \mathbf{or}\ \mathbf{coerce}_{1\leq+}\, \langle V_n \rangle) \mid \forall i.\, (x_i, V_i) \in T\} \end{aligned}$$

In this definition, the order in which the computations V_i appear in the nondeterministic choice is irrelevant because **or** is commutative (and we consider computations up to $\equiv$). Similarly, duplicates and bracketing are irrelevant because of idempotence and associativity. For the lifting of the monoid action, $1 \circledast -$ is the identity, and for $(+) \circledast -$ we map the family $(\mathcal{R}_\varepsilon)_{\varepsilon \in \{1,+\}}$ to $(\mathcal{R}_+)_{\varepsilon \in \{1,+\}}$ (forgetting the relation on deterministic computations). This data forms an LR model.

To apply the abstraction lemma (Lemma 4.3.12) it suffices to show that the left adjoint preserves monotonicity of relations. We sketch the key part of this proof, which is showing that if T above is monotone, and $(S, M), (S', M') \in T'_+$ and $S \supseteq S'$, then $M \preccurlyeq M'$. Because of the definition of T'_+, there are lists of pairs (x_i, V_i) and (x'_i, V'_i) in T such that:

$$\begin{aligned} S &= \uparrow\{x_1, \ldots, x_m\} & M &\equiv \mathbf{coerce}_{1\leq+}\langle V_1 \rangle\ \mathbf{or}\ \cdots\ \mathbf{or}\ \mathbf{coerce}_{1\leq+}\langle V_m \rangle \\ S' &= \uparrow\{x'_1, \ldots, x'_n\} & M' &\equiv \mathbf{coerce}_{1\leq+}\langle V'_1 \rangle\ \mathbf{or}\ \cdots\ \mathbf{or}\ \mathbf{coerce}_{1\leq+}\langle V'_n \rangle \end{aligned}$$

Now $S \supseteq S'$ implies that for each i there is some ϕ_i such that $x_{\phi_i} \sqsubseteq x'_i$. Monotonicity of T implies $V_{\phi_i} \preccurlyeq V'_i$. Hence we have the following, where the first instance of $\preccurlyeq$ follows from M_1 **or** $M_2 \preccurlyeq M_j$ (as well as commutativity, associativity and idempotence of **or**).

$$M \quad \preccurlyeq \quad \mathbf{coerce}_{1\leq+}\langle V_{\phi_1} \rangle\ \mathbf{or}\ \cdots\ \mathbf{or}\ \mathbf{coerce}_{1\leq+}\langle V_{\phi_n} \rangle \quad \preccurlyeq \quad M'$$

4.3.2.3 Shared global state

For shared global state we use a Gifford-style effect algebra, so that effects are subsets $\varepsilon \subseteq \{\text{get}, \text{put}\}$, and objects of the computation category **GSMnem** are graded shared mnemoids. We define the computation category $\mathbf{R}_{\mathbf{GSMnem}\times\mathbf{D}_\lambda}$ of the LR model by:

- Objects $(X, \underline{C}, R)$ consist of a graded shared mnemoid X (we write X_ε for the individual sets, and g and p for the get and put functions), a computation type $\underline{C}$, and a family of relations $(R_\varepsilon \subseteq X_\varepsilon \times \underline{\text{Term}}_{\langle\!\langle\varepsilon\rangle\!\rangle\underline{C}})_{\varepsilon\subseteq\{\text{get},\text{put}\}}$, such that:
  - If $(x, M) \in R_\varepsilon$ and $\varepsilon \subseteq \varepsilon'$ then $(x, \mathbf{coerce}_{\langle\!\langle\varepsilon\rangle\!\rangle\underline{C} <: \langle\!\langle\varepsilon'\rangle\!\rangle\underline{C}}\, M) \in R_{\varepsilon'}$.
  - If $(x_{\text{true}}, M_{\text{true}}), (x_{\text{false}}, M_{\text{false}}) \in R_\varepsilon$ then
  $$(g(x_{\text{true}}, x_{\text{false}}),\ \text{get}\ ()\ \mathbf{to}\ b.\ \underline{\mathbf{if}}\ b\ \mathbf{then}\ M_{\text{true}}\ \mathbf{else}\ M_{\text{false}}) \in R_{\varepsilon\cup\{\text{get}\}}$$
  - If $(x, M) \in R_\varepsilon$ and $b \in \{\text{true}, \text{false}\}$ then $(p_b x,\ \text{put}\, b\ \mathbf{to}\ \_.\, M) \in R_{\varepsilon\cup\{\text{put}\}}$.
- Morphisms $(f, N) : (X, \underline{C}, R) \to (Y, \underline{D}, R')$ are pairs of a morphism $f : X \to Y$ in **GSMnem** and a morphism $N : \underline{C} \to \underline{D}$ in $\mathbf{D}_\lambda$, such that $(x, M) \in R_\varepsilon$ implies $(f x, N^\varepsilon[y \mapsto \mathbf{thunk}\, M]) \in R'_\varepsilon$ for $\varepsilon \subseteq \{\text{get}, \text{put}\}$.

This is very similar to the corresponding category for our nondeterminism example; much of the rest of the structure is also very similar, including the cartesian structure of $\mathbf{R}_{\mathbf{GSMnem}\times\mathbf{D}_\lambda}$ and exponentials. The relation R_ε relates computations with effect ε, and we require relations to be closed under the get and put operations so that the counit lifts. The functor $\mathbf{R}_{\mathbf{GSMnem}\times\mathbf{D}_\lambda} \to \mathbf{GSMnem} \times \mathbf{D}_\lambda$ maps each object $(X, \underline{C}, R)$ to $(X, \underline{C})$ and each morphism (f, N) to itself.

We lift the right adjoint by defining $\dot{U}(X, \underline{C}, R) := (X_\emptyset, \mathrm{U}\,\underline{C}, R')$ where

$$R' := \{(x, \mathbf{thunk}\, M) \mid (x, M) \in R_\emptyset\}$$

The lifting of the monoid action is $\varepsilon \dot{\circledast} (X, \underline{C}, R) := ((X_{\varepsilon'\cup\varepsilon})_{\varepsilon'\subseteq\{\text{get},\text{put}\}}, \langle\!\langle\varepsilon\rangle\!\rangle\underline{C}, (R_{\varepsilon'\cup\varepsilon})_{\varepsilon'\subseteq\{\text{get},\text{put}\}})$. Finally, for the lifting of the left adjoint suppose that X is a poset, A is a value type, and $R \subseteq X \times \text{Term}_A$. We take $\dot{F}(X, A, R) = (FX, \langle\emptyset\rangle A, R')$, where each relation R'_ε is defined inductively by the following rules:

$$\frac{(x, V) \in R}{(x, \mathbf{coerce}_{\emptyset\leq\varepsilon}\, \langle V\rangle) \in R'_\varepsilon} \qquad \frac{(t, M) \in R'_\varepsilon \qquad (b, V) \in \mathcal{R}[\![\mathbf{bool}]\!]}{(\text{put}_b\, t,\ \text{put}\, V;\, M) \in R'_\varepsilon}\ \text{if put} \in \varepsilon$$

$$\frac{(t_{\text{true}}, M_{\text{false}}), (t_{\text{false}}, M_{\text{false}}) \in R'_\varepsilon}{(\text{get}(t_{\text{true}}, t_{\text{false}}),\ \text{get}\ ()\ \mathbf{to}\ b.\ \underline{\mathbf{if}}\ b\ \mathbf{then}\ M_{\text{true}}\ \mathbf{else}\ M_{\text{false}}) \in R'_\varepsilon}\ \text{if get} \in \varepsilon$$

This definition is based on the free lifting from Section 3.1.1.

This defines an LR model that satisfies all of the requirements of the abstraction lemma. Hence our model of shared global state is adequate, and we have verified all of our example program transformations.

4.4 Relating call-by-value and call-by-name, semantically

The final contribution of this chapter is to redevelop our syntax-based reasoning principle for relating call-by-value and call-by-name evaluation (Chapter 3) using the denotational semantics. Working inside the semantics makes it easier to apply our reasoning principle.

$$\boxed{\phi_{\tau,\varepsilon} : F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon \to [\![\tau]\!]^{\mathrm{n}}_\varepsilon}$$

$$\begin{aligned}
\phi_{\mathbf{unit},\varepsilon} &:= id \;:\; F_\varepsilon 1 \to F_\varepsilon 1\\
\phi_{\mathbf{bool},\varepsilon} &:= id \;:\; F_\varepsilon 2 \to F_\varepsilon 2\\
\phi_{\tau\to\tau',\varepsilon} &:= \varpi \circ F_\varepsilon(\Lambda(U\phi_{\tau',\varepsilon} \circ ev^\dagger \circ (id \times \psi_{\tau,\varepsilon}))) \;:\; F_\varepsilon(U([\![\tau]\!]^{\mathrm{v}}_\varepsilon \Rightarrow F_\varepsilon[\![\tau']\!]^{\mathrm{v}}_\varepsilon)) \to U[\![\tau]\!]^{\mathrm{n}}_\varepsilon \Rightarrow [\![\tau']\!]^{\mathrm{n}}_\varepsilon
\end{aligned}$$

$$\boxed{\psi_{\tau,\varepsilon} : U[\![\tau]\!]^{\mathrm{n}}_\varepsilon \to U(F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon)}$$

$$\begin{aligned}
\psi_{\mathbf{unit},\varepsilon} &:= id \;:\; U(F_\varepsilon 1) \to U(F_\varepsilon 1)\\
\psi_{\mathbf{bool},\varepsilon} &:= id \;:\; U(F_\varepsilon 2) \to U(F_\varepsilon 2)\\
\psi_{\tau\to\tau',\varepsilon} &:= \eta_\varepsilon \circ \Lambda(\psi_{\tau',\varepsilon} \circ ev \circ (id \times (U\phi_{\tau,\varepsilon} \circ \eta_\varepsilon)))\\
&\qquad : U(U[\![\tau]\!]^{\mathrm{n}}_\varepsilon \Rightarrow [\![\tau']\!]^{\mathrm{n}}_\varepsilon) \to U(F_\varepsilon(U([\![\tau]\!]^{\mathrm{v}}_\varepsilon \Rightarrow F_\varepsilon[\![\tau']\!]^{\mathrm{v}}_\varepsilon)))
\end{aligned}$$

Figure 4.5: Semantic morphisms ϕ from call-by-value to Levy-style call-by-name and ψ from Levy-style call-by-name to call-by-value

Recall that in Chapter 3 we restrict to Gifford-style effect algebras, so that effects are subsets ε of some fixed set Σ of operations. We impose the same restriction in this section. Section 3.3 defines maps

$$\Gamma \vdash M : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_\varepsilon \;\mapsto\; \Gamma \vdash \Phi_{\tau,\varepsilon}M : (\!|\tau|\!)^{\mathrm{n}}_\varepsilon \qquad\qquad \Gamma \vdash N : (\!|\tau|\!)^{\mathrm{n}}_\varepsilon \;\mapsto\; \Gamma \vdash \Psi_{\tau,\varepsilon}N : \langle\varepsilon\rangle(\!|\tau|\!)^{\mathrm{v}}_\varepsilon$$

between call-by-value and Levy-style call-by-name computations (with effect ε), and shows that they form Galois connections under certain circumstances. In this section we show that the *interpretations* of these maps in models satisfying certain conditions (which we state later) form Galois connections.

Definition 4.4.1 Suppose that X and Y are posets. A *Galois connection* (ϕ, ψ) from X to Y is a pair of monotone functions

$$\phi : X \to Y \qquad\qquad \psi : Y \to X$$

such that for all $x \in X$ and $y \in Y$,

$$\phi\, x \sqsubseteq y \quad\Leftrightarrow\quad x \sqsubseteq \psi\, y \qquad\blacktriangleleft$$

Figure 4.5 defines *semantic* analogues of $\Phi_{\tau,\varepsilon}$ and $\Psi_{\tau,\varepsilon}$. We use some extra notation: $[\![\tau]\!]^{\mathrm{v}}_\varepsilon$ is $[\![(\!|\tau|\!)^{\mathrm{v}}_\varepsilon]\!]$ (i.e. the interpretation of the call-by-value translation of the type τ, with the effect annotation ε), and similarly $[\![\tau]\!]^{\mathrm{n}}_\varepsilon$ is $[\![(\!|\tau|\!)^{\mathrm{n}}_\varepsilon]\!]$. We also define F_ε to be $\varepsilon \circledast F-$, so that $[\![\langle\varepsilon\rangle A]\!] = F_\varepsilon[\![A]\!]$, and define $\eta_{\varepsilon,X} : X \to U(F_\varepsilon X)$ as $U((\emptyset \le \varepsilon)\circledast X) \circ \eta_X$ (using $\emptyset \subseteq \varepsilon$). The figure uses the extension operator $(-)^\dagger$ defined in Section 4.2.1, and the natural transformation ev from Section 4.2.2. The maps between call-by-value and call-by-name in the semantics are morphisms $\phi_{\tau,\varepsilon} : F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon \to [\![\tau]\!]^{\mathrm{n}}_\varepsilon$ in the computation category $\mathbf{D}$ and $\psi_{\tau,\varepsilon} : U[\![\tau]\!]^{\mathrm{n}}_\varepsilon \to U(F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon)$ in the value category $\mathbf{C}$. The definition relies on our assumption that the effect algebra is Gifford-style (which implies the multiplication is idempotent and $\varepsilon\circledast[\![\tau]\!]^{\mathrm{n}}_\varepsilon = [\![\tau]\!]^{\mathrm{n}}_\varepsilon$).

Using composition, the morphisms $\phi_{\tau,\varepsilon}$ and $\psi_{\tau,\varepsilon}$ induce functions between posets of morphisms (for each $X \in \mathbf{C}$):

$$U\phi_{\tau,\varepsilon} \circ - : \mathbf{C}(X, U(F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon)) \to \mathbf{C}(X, U[\![\tau]\!]^{\mathrm{n}}_\varepsilon) \qquad \psi_{\tau,\varepsilon} \circ - : \mathbf{C}(X, U[\![\tau]\!]^{\mathrm{n}}_\varepsilon) \to \mathbf{C}(X, U(F_\varepsilon[\![\tau]\!]^{\mathrm{v}}_\varepsilon))$$

These are the interpretations of Φ and Ψ in the following sense.

Lemma 4.4.2 If $\Gamma \vdash M : \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$ then $\llbracket\Phi_{\tau,\varepsilon}M\rrbracket = U\phi_{\tau,\varepsilon} \circ \llbracket M\rrbracket$, and if $\Gamma \vdash N : \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ then $\llbracket\Psi_{\tau,\varepsilon}N\rrbracket = \psi_{\tau,\varepsilon} \circ \llbracket N\rrbracket$. ◀

Our goal is to show that the pair $(U\phi_{\tau,\varepsilon} \circ -, \psi_{\tau,\varepsilon} \circ -)$ is a Galois connection for each τ. For the same reason as in Section 3.3, we cannot expect this to hold in every case; it will only hold if enough computations are thunkable. We defined thunkable for syntax in Definition 3.3.2; we now define thunkable for semantics.

Definition 4.4.3 (Thunkable) A morphism $f : X \longrightarrow U(F_\varepsilon Y)$ in $\mathbf{C}$ is *thunkable* if

$$U(F_\varepsilon\eta_{\varepsilon,Y}) \circ f \;\sqsubseteq\; \eta_{\varepsilon,UF_\varepsilon Y} \circ f$$

An effect ε is *thunkable* if $U(F_\varepsilon\eta_{\varepsilon,Z}) \sqsubseteq \eta_{\varepsilon,UF_\varepsilon Z}$ for each $Z \in \mathbf{C}$. ◀

Note that an effect ε is thunkable if and only if for all X, Y, all morphisms $f : X \longrightarrow U(F_\varepsilon Y)$ are thunkable. This definition is based on the one given by Führmann [27], adapted to order-enriched models.[8]

An example of a side-effect that is thunkable under this definition is undefined behaviour (and the reasoning principle we give below can be used to show that, if side-effects are restricted to undefined behaviour, call-by-value can be replaced with call-by-name). Nondeterministic choice as we present it above (in which we can statically make nondeterministic choices) is not thunkable, but it is if we reverse all of the orders on morphisms. This means we can use the reasoning principle to replace call-by-name with call-by-value, but not the reverse, if we allow nondeterministic choices to be made statically.

Assuming the effect ε is thunkable is enough to show we have a Galois connection in the semantics (this theorem is a semantic version of Theorem 3.3.3).

Theorem 4.4.4 If the effect ε is thunkable, then for every source-language type τ and object $X \in \mathbf{C}$, the pair $(U\phi_{\tau,\varepsilon} \circ -, \psi_{\tau,\varepsilon} \circ -)$ is a Galois connection from $\mathbf{C}(X, U(F_\varepsilon\llbracket\tau\rrbracket^{\mathrm{v}}_{\varepsilon}))$ to $\mathbf{C}(X, U\llbracket\tau\rrbracket^{\mathrm{n}}_{\varepsilon})$. ◀

Recall that in Section 3.4 we state our reasoning principle that relates call-by-value and call-by-name on open terms by composing the call-by-name translation with our maps between evaluation orders, to obtain a computation with the same typing as the call-by-value translation:

$$\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon} \longrightarrow \llparenthesis\Gamma\rrparenthesis^{\mathrm{n}}_{\varepsilon} \xrightarrow{\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}} \llparenthesis\tau\rrparenthesis^{\mathrm{n}}_{\varepsilon} \longrightarrow \langle\varepsilon\rangle\llparenthesis\tau\rrparenthesis^{\mathrm{v}}_{\varepsilon}$$

We defined this formally by giving a substitution $\hat{\Phi}_{\Gamma,\varepsilon}$ from terms in call-by-name contexts $\llparenthesis\Gamma\rrparenthesis^{\mathrm{n}}_{\varepsilon}$ to terms in call-by-value contexts $\llparenthesis\Gamma\rrparenthesis^{\mathrm{v}}_{\varepsilon}$. The composition is then the term $\Psi_{\tau,\varepsilon}(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}])$.

Since in this section we reason using the denotational semantics, we can now say what this means inside models of GCBPV. The interpretation of the substitution $\hat{\Phi}_{\Gamma,\varepsilon}$ is a morphism $\hat{\phi}_{\Gamma,\varepsilon} : \llbracket\Gamma\rrbracket^{\mathrm{v}}_{\varepsilon} \to \llbracket\Gamma\rrbracket^{\mathrm{n}}_{\varepsilon}$, which is defined as follows (recall that $\diamond$ is the empty typing context).

$$\hat{\phi}_{\diamond,\varepsilon} := id : 1 \to 1 \qquad \hat{\phi}_{(\Gamma,x:\tau),\varepsilon} := \hat{\phi}_{\Gamma,\varepsilon} \times (U\phi_{\tau,\varepsilon} \circ \eta_\varepsilon) : \llbracket\Gamma\rrbracket^{\mathrm{v}}_{\varepsilon} \times \llbracket\tau\rrbracket^{\mathrm{v}}_{\varepsilon} \to \llbracket\Gamma\rrbracket^{\mathrm{n}}_{\varepsilon} \times U\llbracket\tau\rrbracket^{\mathrm{n}}_{\varepsilon}$$

The interpretations of the computations we wish to relate are the two sides of the inequality given as the following diagram. Assuming the model is adequate, to show that the two terms

[8] The (directed) definition of thunkable also appears in another form: an effect ε is thunkable if and only if $U \circ F_\varepsilon$ forms a *Kock-Zöberlein* monad [51].

are related by the contextual preorder it therefore suffices to show that this inequality holds.

$$\begin{tikzcd}
\llbracket\Gamma\rrbracket^{\mathrm{v}}_{\varepsilon} \arrow[r, "\hat{\phi}_{\Gamma,\varepsilon}"] \arrow[d, "\llbracket e\rrbracket^{\mathrm{v}}_{\varepsilon}"'] \arrow[dr, phantom, "\sqsubseteq"] & \llbracket\Gamma\rrbracket^{\mathrm{n}}_{\varepsilon} \arrow[d, "\llbracket e\rrbracket^{\mathrm{n}}_{\varepsilon}"] \\
U(F_{\varepsilon}\llbracket\tau\rrbracket^{\mathrm{v}}_{\varepsilon}) & U\llbracket\tau\rrbracket^{\mathrm{n}}_{\varepsilon} \arrow[l, "\psi_{\tau,\varepsilon}"]
\end{tikzcd}$$

Again we can show this is the case directly using the properties of Galois connections (see also Lemma 3.4.1).

Lemma 4.4.5 Suppose that $(U\phi_{\tau',\varepsilon} \circ -, \psi_{\tau',\varepsilon} \circ -)$ is a Galois connection from $\mathbf{C}(X, U(F_{\varepsilon}\llbracket\tau'\rrbracket^{\mathrm{v}}_{\varepsilon}))$ to $\mathbf{C}(X, U\llbracket\tau'\rrbracket^{\mathrm{n}}_{\varepsilon})$ for each τ'. If $\Gamma \vdash e : \tau$ and *ops* $e \subseteq \varepsilon$ then

$$\llbracket e\rrbracket^{\mathrm{v}}_{\varepsilon} \sqsubseteq \psi_{\tau,\varepsilon} \circ \llbracket e\rrbracket^{\mathrm{n}}_{\varepsilon} \circ \hat{\phi}_{\Gamma,\varepsilon} \qquad ◀$$

We use this lemma to show the semantic version of our reasoning principle for relating call-by-value and Levy-style call-by-name (see also Theorem 3.4.2):

Theorem 4.4.6 (Call-by-value and call-by-name) Suppose we given some GCBPV inequational theory containing a Gifford-style effect algebra, and an adequate model of it in which the effect ε is thunkable. If $\Gamma \vdash e : \tau$ and *ops* $e \subseteq \varepsilon$ then

$$\llparenthesis e\rrparenthesis^{\mathrm{v}}_{\varepsilon} \preccurlyeq_{\mathrm{ctx}} \Psi_{\tau,\varepsilon}(\llparenthesis e\rrparenthesis^{\mathrm{n}}_{\varepsilon}[\hat{\Phi}_{\Gamma,\varepsilon}])$$

Proof. By Theorem 4.4.4, the maps between call-by-value and call-by-name computations form Galois connections. Hence we can apply Lemma 4.4.5, which tells us that

$$\llbracket e\rrbracket^{\mathrm{v}}_{\varepsilon} \sqsubseteq \psi_{\tau,\varepsilon} \circ \llbracket e\rrbracket^{\mathrm{n}}_{\varepsilon} \circ \hat{\phi}_{\Gamma,\varepsilon}$$

The result follows from adequacy. □

4.5 Related work

Effect-dependent transformations There is a long line of work on using denotational semantics to verify the correctness of effect-dependent program transformations, for example by Benton and Buchlovsky [5], Benton and Kennedy [8], Benton et al. [10, 9], Birkedal et al. [13]. Here we differ in two major ways. We focus on giving a general framework (somewhat similar to Kammar and Plotkin [42]) that incorporates several side-effects with little effort, and instantiate it for examples. Second, we focus on noninvertible transformations, which are often ignored in previous work. (For example, they are not mentioned at all in [11] but are briefly considered in [6].)

Inequations and order-enriched semantics The inequations and orders we use are induced by considering the possible behaviours of computations. Hoare and He [38] emphasize a similar idea to this in the setting of process calculus.

Previous work on order-enriched semantics (such as Fiore [26]) has concentrated mainly on fixed points and recursive types, using partiality as the order (the least element represents nontermination). Goncharov and Schröder [30] also incorporate other side-effects, but again use partiality for the order on morphisms. They also use a more general notion of order-enrichment, where only certain sets of morphisms are required to be partially ordered, allowing e.g. $\mathbf{C} = \mathbf{Set}$ to be used with a non-trivial order. We do not use this approach, because it is unclear how to extend it to a higher-order language (Goncharov and Schröder consider only first-order).

Relating syntax and semantics To prove adequacy we rely heavily on the fibrational view of logical relations started by Hermida [36]. We also based our presentation somewhat on Katsumata [44], who relates models of a monadic language similar to GMM. We obtain adequacy as a direct consequence of the fundamental lemma. Plotkin and Power [87] and Kammar et al. [40] instead prove termination first, then obtain adequacy as a corollary. We expect our technique for proving adequacy to work even for languages with nontermination.

4.6 Summary

This chapter presents a general denotational semantics for GCBPV, and uses it to prove the validity effect-dependent program transformations. This improves on Chapter 3, in which we reasoned inside the syntax. Denotational models makes the proofs less cumbersome. We include a discussion of various sources of models. In particular, we discuss monadic models to allow us use previous monadic models of side-effects, such as local state [88, 40] and probability [37], as the basis of models of GCBPV. We also include a general logical-relations-based method for relating models of GCBPV, including proofs of adequacy.

We are careful to ensure that we allow noninvertible transformations, by using contextual *preorders* and order-enriched models. This is important because there are various applications that give rise to noninvertible transformations, as our examples show.

Chapter 5

Call-by-need and extended call-by-push-value

We have developed a framework for reasoning about program transformations that supports call-by-value, Moggi-style call-by-name and Levy-style call-by-name. Another common evaluation order is call-by-need (which is sometimes also referred to as “lazy evaluation” when data constructors defer evaluation of arguments until the data structure is traversed). So far we have said nothing about call-by-need, and in fact CBPV (and GCBPV) do not enable us to reason about call-by-need.

An intuitive reason is that call-by-need has “action at a distance” in that reduction of one subterm causes reduction of all other subterms that originated as copies during variable substitution. Indeed call-by-need is often framed using mutable stores (graph reduction [100], or reducing a thunk which is accessed by multiple pointers [53]). CBPV does not allow these to be encoded.

This chapter presents *extended call-by-push-value* (ECBPV), a calculus similar to CBPV, but which can capture call-by-need reduction in addition to call-by-value and call-by-name. Specifically, ECBPV adds an extra primitive M **need** $\underline{x}$. N which runs N, with M being evaluated the first time $\underline{x}$ is used. On subsequent uses of $\underline{x}$, the result of the first run is returned immediately. The term M is evaluated at most once. We give the syntax and type system of ECBPV, together with an equational theory that expresses when terms are considered equal. We then give a compositional translation from a lambda calculus (specifically our source language in Section 2.2) into ECBPV for call-by-need reduction.

Initially we extend *ordinary* CBPV, which does not track the effects of computations in the type system. As we will see, support for call-by-need (and action at a distance more generally) makes tracking effects inside the type system more difficult, so we postpone the discussion of grading until the end of the chapter.

ECBPV can be used to reason about call-by-need and all of the other evaluation orders captured by CBPV. Hence we can consider equivalences between call-by-need and other evaluation orders. For example, if there are no side-effects at all in the source language[1] then call-by-need, call-by-value and call-by-name should be semantically equivalent. If the only effect is nondeterminism, then need and value (but not name) are equivalent. If the only effect is nontermination then need and name (but not value) are equivalent. We show that ECBPV can be used to prove such equivalences by proving the latter. We do this using a technique similar to the one we used in Chapter 3 for call-by-value and call-by-name. The main difference

[1] Without an effect system we have to restrict the side-effects of the entire source language. With an effect system we can restrict individual expressions.

is that we rely on *Kripke logical relations of varying arity* [39], which generalize ordinary logical relations.

This chapter makes the following contributions:

- We describe *extended call-by-push-value*, a version of CBPV containing an extra construct that adds support for call-by-need. We give its syntax, type system, and equational theory (Section 5.1).
- We describe a call-by-need translation from our source language into ECBPV (Section 5.2).
- We show that, if the source language has nontermination as the only side-effect, then call-by-need and Moggi-style call-by-name are equivalent (Section 5.3).
- We refine the type system of ECBPV so that its types also carry effect information (Section 5.4). This allows equivalences between evaluation orders to be exploited, both at ECBPV and at source level, without a whole-language restriction on side-effects.

5.1 Extended call-by-push-value

We describe an extension to call-by-push-value with support for call-by-need. The primary difference between ordinary CBPV and ECBPV is the addition of a primitive that allows *computations* to be added to the environment so that they are evaluated only the first time they are used. Before describing this change, we take a closer look at how CBPV supports call-by-value and call-by-name.

Recall that CBPV stratifies terms into *values*, which do not have side-effects, and *computations*, which might. Evaluation order is irrelevant for values, so we are only concerned with how computations are sequenced. There is exactly one primitive that causes the evaluation of more than one computation, which is $M\ \mathbf{to}\ x.\,N$. This behaves in the same way as it does in GCBPV. The evaluation order is fixed: M is always eagerly evaluated. This construct can be used to implement call-by-value as in Figure 2.15: to apply a function, eagerly evaluate the argument and then evaluate the body of the function.

Thunks let us implement call-by-name in CBPV: arguments to functions are thunked computations. Arguments are used by forcing them, so that the computation is evaluated every time the argument is used. Effectively, we simulate a construct $M\ \mathbf{name}\ \underline{x}.\,N$, which evaluates M each time the variable $\underline{x}$ is used by N, rather than eagerly evaluating. (The variable $\underline{x}$ is underlined here to indicate that it refers to a computation rather than a value: uses of it may have side-effects.)

To support call-by-need, extended call-by-push-value adds another construct $M\ \mathbf{need}\ \underline{x}.\,N$. This runs the computation N, with the computation M being evaluated the first time $\underline{x}$ is used. On subsequent uses of $\underline{x}$, the result of the first run is returned immediately. The computation M is evaluated at most once. This new construct adds the “action at a distance” missing from ordinary CBPV.

We observe that adding general mutable references to call-by-push-value would allow call-by-need to be encoded. However, reasoning about evaluation order would be difficult, and so we do not take this option.

5.1.1 Syntax

The syntax of extended call-by-push-value is as follows. The highlighted parts are new in this chapter. The rest of the syntax is similar to CBPV and GCBPV. The type constructor **F** corresponds to $\langle \varepsilon \rangle$ in GCBPV (without the grading by the effect ε). There are no coercions because of the lack of grading.

$$\begin{array}{rcl@{\qquad}rcl}
A, B & ::= & b & V, W & ::= & c \mid x \\
& & \mid \mathbf{unit} & & & \mid () \\
& & \mid A_1 \times A_2 & & & \mid (V_1, V_2) \mid \mathbf{fst}\, V \mid \mathbf{snd}\, V \\
& & \mid \mathbf{empty} & & & \mid \mathbf{case}_A\ V\ \mathbf{of}\ \{\} \\
& & \mid A_1 + A_2 & & & \mid \mathbf{inl}_{A_2} V \mid \mathbf{inr}_{A_1} V \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} \\
& & \mid \mathbf{U}\, \underline{C} & & & \mid \mathbf{thunk}\, M \\
\underline{C}, \underline{D} & ::= & \underline{\mathbf{unit}} & M, N & ::= & \lambda\{\} \\
& & \mid \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 & & & \mid \lambda\{1.\, M_1, 2.\, M_2\} \mid 1`M \mid 2`M \\
& & \mid A \to \underline{C} & & & \mid \lambda x{:}A.\, M \mid V`M \\
& & \mid \mathbf{F}\, A & & & \mid \mathrm{op}\, V \mid \langle V \rangle \mid M\ \mathbf{to}\ x.\, N \\
& & & & & \mid \mathbf{force}\, V \mid \underline{x} \mid M\ \mathbf{need}\ \underline{x}.\, N
\end{array}$$

We assume two sets of variables: *value variables* $x, y, \ldots$ (which are the same as the variables in GCBPV) and *computation variables* $\underline{x}, \underline{y}, \ldots$. While ordinary CBPV does not include computation variables, they do not of themselves add any expressive power to it. The ability to express call-by-need in ECBPV comes from the **need** construct used to bind them.[2]

The primary new construct is $M\ \mathbf{need}\ \underline{x}.\, N$. This term evaluates N. The first time $\underline{x}$ is evaluated (due to a use of $\underline{x}$ inside N) it behaves the same as the computation M. If M returns a value V then subsequent uses of $\underline{x}$ behave the same as $\langle V \rangle$. Hence only the first use of $\underline{x}$ will evaluate M. If $\underline{x}$ is not used then M is not evaluated at all. The computation variable $\underline{x}$ bound inside the term is primarily used by eagerly sequencing it with other computations. For example,

$$M\ \mathbf{need}\ \underline{x}.\, \underline{x}\ \mathbf{to}\ y.\, \underline{x}\ \mathbf{to}\ z.\, \langle (y, z) \rangle$$

uses $\underline{x}$ twice: once where the result is bound to y, and once where the result is bound to z. Only the first of these uses will evaluate M, so this term has the same semantics as $M\ \mathbf{to}\ x.\, \langle (x, x) \rangle$. The term $M\ \mathbf{need}\ \underline{x}.\, \langle () \rangle$ does not evaluate M at all, and has the same semantics as $\langle () \rangle$.

With the addition of **need** it is not in general possible to statically determine the order in which computations are executed. Uses of computation variables are given statically, but not all of these actually evaluate the corresponding computation dynamically. The set of uses of computation variables that actually cause effects depends on run-time behaviour. This will be important when describing the effect system in Section 5.4.

Typing contexts Γ are ordered lists mapping value variables to value types A, and computation variables to computation types of the form $\mathbf{F}\, A$. The restriction to $\mathbf{F}\, A$ is due to the fact that the only construct that binds computation variables is **need**, and this only sequences computations of returner type. Allowing computation variables to be associated with other forms of computation type in typing contexts is therefore unnecessary. Ground types are value types that do not contain thunks:

$$G \ ::= \ b \mid \mathbf{unit} \mid G_1 \times G_2 \mid \mathbf{empty} \mid G_1 + G_2$$

The syntax is parameterized by a notion of signature identical to the one for GCBPV (Definition 2.7.1), except for the omission of the effects ε.

[2] Computation variables are not strictly required to support call-by-need (since we can use $x : \mathbf{U}\,(\mathbf{F}\, A)$ instead of $\underline{x} : \mathbf{F}\, A$), but they simplify reasoning about evaluation order, and therefore we choose to include them.

Definition 5.1.1 A *ECBPV signature* consists of the following data:

- A set $\mathcal{B}$ of *base types.*
- A family of pairwise disjoint sets $\mathcal{K}_A$ of *constants of type* A, indexed by value types A.
- A set Σ of *operations.*
- For each operation $\mathrm{op} \in \Sigma$, ground types $\mathrm{car}_{\mathrm{op}}$ and $\mathrm{ar}_{\mathrm{op}}$, respectively called the *coarity* and *arity* of op. ◀

(Recall from Definition 2.7.1 that the coarity is the type of the argument of the operation, and the arity is the type of the result, not the other way around.)

The type system of extended call-by-push-value is a minor extension of the type system of ordinary call-by-push-value: we add one typing rule for each of the two new constructs. The rules are given in Figure 5.1. We again write the value typing judgment as $\Gamma \vdash V : A$ and the computation typing judgment as $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$, and follow our usual convention: rules that add a new variable to the typing context implicitly require freshness.

ECBPV admits a substitution lemma similar to the one for GCBPV (Lemma 2.7.2). *Substitutions* σ are given by the following grammar:

$$\sigma \ ::= \ \diamond \mid \sigma, x \mapsto V \mid \sigma, \underline{x} \mapsto M$$

where $\diamond$ is the empty substitution. The typing judgment $\Gamma \vdash \sigma : \Delta$ for substitutions means in the context Γ the terms in σ have the types given in the context Δ. It is defined as follows:

$$\frac{}{\Gamma \vdash \diamond : \diamond} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash V : A}{\Gamma \vdash (\sigma,\ x \mapsto V) : (\Delta, x : A)} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \mathrel{\underline{\vdash}} M : \mathbf{F}A}{\Gamma \vdash (\sigma,\ \underline{x} \mapsto M) : (\Delta, \underline{x} : \mathbf{F}A)}$$

We write $V[\sigma]$ and $M[\sigma]$ for the (capture-avoiding) applications of the substitution σ to the value term V and to the computation term M, and $x \mapsto V$ and $\underline{x} \mapsto M$ for the obvious substitutions that are identities on all other variables. The substitution lemma for ECBPV is:

Lemma 5.1.2 (Substitution) Suppose that $\Gamma \vdash \sigma : \Delta$.

1. (Values) If $\Delta \vdash V : B$ then $\Gamma \vdash V[\sigma] : B$.
2. (Computations) If $\Delta \mathrel{\underline{\vdash}} M : \underline{C}$ then $\Gamma \mathrel{\underline{\vdash}} M[\sigma] : \underline{C}$. ◀

We define the call-by-name construct mentioned above as syntactic sugar for other CBPV primitives:

$$M \ \mathbf{name}\ \underline{x}.\, N \ := \ \mathbf{thunk}\, M \text{ ‘ } \lambda y.\, N[\underline{x} \mapsto \mathbf{force}\, y]$$

where y is not free in N.

5.1.2 Inequational theories for ECBPV

The behaviour of **need** is captured by the core axioms for ECBPV inequational theories, which are listed in Figure 5.2. Each of these axioms is symmetric and holds when both sides of the equation have suitable types. For example, the second axiom of the third group ($M\ \mathbf{need}\ \underline{x}.\, N \equiv N$) holds only if $\underline{x}$ is not free in N.

For the core axioms to capture call-by-need, we might expect computation terms that are not of the form $\langle V \rangle$ to never be duplicated, since they should not be evaluated more than once. There are two exceptions to this rule. Such terms can be duplicated in the axioms that duplicate value terms (such as the β-laws for sum types). In this case, the syntax ensures such terms are thunked.

$$\boxed{\Gamma \vdash V : A}$$

$$\frac{}{\Gamma \vdash c : A}\ \text{if } c \in \mathcal{K}_A \qquad \frac{}{\Gamma \vdash x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

$$\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \mathbf{fst}\, V : A_1} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \mathbf{snd}\, V : A_2} \qquad \frac{\Gamma \vdash V : \mathbf{empty}}{\Gamma \vdash \mathbf{case}_A\ V\ \mathbf{of}\ \{\} : A}$$

$$\frac{\Gamma \vdash V : A_1}{\Gamma \vdash \mathbf{inl}_{A_2} V : A_1 + A_2} \qquad \frac{\Gamma \vdash V : A_2}{\Gamma \vdash \mathbf{inr}_{A_1} V : A_1 + A_2}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash W_1 : B \qquad \Gamma, x_2 : A_2 \vdash W_2 : B}{\Gamma \vdash \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} : B}$$

$$\frac{\Gamma \mathrel{\underline{\vdash}} M : \underline{C}}{\Gamma \vdash \mathbf{thunk}\, M : \mathbf{U}\underline{C}}$$

$$\boxed{\Gamma \mathrel{\underline{\vdash}} M : \underline{C}}$$

$$\frac{}{\Gamma \mathrel{\underline{\vdash}} \lambda\{\} : \underline{\mathbf{unit}}} \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M_1 : \underline{C}_1 \qquad \Gamma \mathrel{\underline{\vdash}} M_2 : \underline{C}_2}{\Gamma \mathrel{\underline{\vdash}} \lambda\{1.\, M_1, 2.\, M_2\} : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2} \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2}{\Gamma \mathrel{\underline{\vdash}} 1`M : \underline{C}_1} \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2}{\Gamma \mathrel{\underline{\vdash}} 2`M : \underline{C}_2}$$

$$\frac{\Gamma, x : A \mathrel{\underline{\vdash}} M : \underline{C}}{\Gamma \mathrel{\underline{\vdash}} \lambda x{:}A.\, M : A \to \underline{C}} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma \mathrel{\underline{\vdash}} M : A \to \underline{C}}{\Gamma \mathrel{\underline{\vdash}} V`M : \underline{C}}$$

$$\frac{\Gamma \vdash V : \mathrm{car}_{\mathrm{op}}}{\Gamma \mathrel{\underline{\vdash}} \mathrm{op}\, V : \mathbf{F}\, \mathrm{ar}_{\mathrm{op}}} \qquad \frac{\Gamma \vdash V : A}{\Gamma \mathrel{\underline{\vdash}} \langle V \rangle : \mathbf{F} A} \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M : \mathbf{F} A \qquad \Gamma, x : A \mathrel{\underline{\vdash}} N : \underline{C}}{\Gamma \mathrel{\underline{\vdash}} M\ \mathbf{to}\ x.\, N : \underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{U}\underline{C}}{\Gamma \mathrel{\underline{\vdash}} \mathbf{force}\, V : \underline{C}}$$

$$\frac{}{\Gamma \mathrel{\underline{\vdash}} \underline{x} : \mathbf{F} A}\ \text{if } (\underline{x} : \mathbf{F} A) \in \Gamma \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M : \mathbf{F} A \qquad \Gamma, \underline{x} : \mathbf{F} A \mathrel{\underline{\vdash}} N : \underline{C}}{\Gamma \mathrel{\underline{\vdash}} M\ \mathbf{need}\ \underline{x}.\, N : \underline{C}}$$

Figure 5.1: Extended call-by-push-value typing rules

$$\left.\begin{array}{rclrcl}
\mathbf{fst}\,(V_1,V_2) & \equiv & V_1 & \mathbf{snd}\,(V_1,V_2) & \equiv & V_2 \\
\begin{array}{l}\mathbf{case}\ \mathbf{inl}_{A_2} V\ \mathbf{of}\\ \quad\{\mathbf{inl}\,x_1.\,W_1\\ \quad,\,\mathbf{inr}\,x_2.\,W_2\}\end{array} & \equiv & W_1[x_1\mapsto V] & \begin{array}{l}\mathbf{case}\ \mathbf{inr}_{A_1} V\ \mathbf{of}\\ \quad\{\mathbf{inl}\,x_1.\,W_1\\ \quad,\,\mathbf{inr}\,x_2.\,W_2\}\end{array} & \equiv & W_2[x_2\mapsto V] \\
1\,\text{`}\,\lambda\{1.\,M_1,2.\,M_2\} & \equiv & M_1 & 2\,\text{`}\,\lambda\{1.\,M_1,2.\,M_2\} & \equiv & M_2 \\
V\,\text{`}\,\lambda x{:}A.\,M & \equiv & M[x\mapsto V] & \langle V\rangle\ \mathbf{to}\ x.\,M & \equiv & M[x\mapsto V] \\
\mathbf{force}\,(\mathbf{thunk}\,M) & \equiv & M & \langle V\rangle\ \mathbf{need}\ \underline{x}.\,M & \equiv & M[\underline{x}\mapsto\langle V\rangle]
\end{array}\right\}\beta\text{-laws}$$

$$\left.\begin{array}{rclrcl}
V & \equiv & () & V & \equiv & (\mathbf{fst}\,V,\mathbf{snd}\,V) \\
V & \equiv & \mathbf{case}_A\ W\ \mathbf{of}\ \{\} & & & \\
V & \equiv & \mathbf{thunk}\,(\mathbf{force}\,V) & V[x\mapsto W] & \equiv & \begin{array}{l}\mathbf{case}\ W\ \mathbf{of}\\ \quad\{\mathbf{inl}\,y_1.\,V[x\mapsto\mathbf{inl}_{A_2}y_1]\\ \quad,\,\mathbf{inr}\,y_2.\,V[x\mapsto\mathbf{inr}_{A_1}y_2]\}\end{array} \\
M & \equiv & \lambda\{\} & M & \equiv & \lambda\{1.\,1\text{`}M,2.\,2\text{`}M\} \\
M & \equiv & \lambda x{:}A.\,x\text{`}M & M & \equiv & M\ \mathbf{to}\ x.\,\langle x\rangle
\end{array}\right\}\eta\text{-laws}$$

$$\left.\begin{array}{rcl}
M\ \mathbf{need}\ \underline{x}.\,\underline{x}\ \mathbf{to}\ y.\,N & \equiv & M\ \mathbf{to}\ y.\,N[\underline{x}\mapsto\langle y\rangle] \\
M\ \mathbf{need}\ \underline{x}.\,N & \equiv & N \\
\lambda\{1.\,M\ \mathbf{to}\ x.\,N_1,\ 2.\,M\ \mathbf{to}\ x.\,N_2\} & \equiv & M\ \mathbf{to}\ x.\,\lambda\{1.\,N_1,\ 2.\,N_2\} \\
\lambda y{:}A.\,M\ \mathbf{to}\ x.\,N & \equiv & M\ \mathbf{to}\ x.\,\lambda y{:}A.\,N \\
\lambda\{1.\,M\ \mathbf{need}\ \underline{x}.\,N_1,\ 2.\,M\ \mathbf{need}\ \underline{x}.\,N_2\} & \equiv & M\ \mathbf{need}\ \underline{x}.\,\lambda\{1.\,N_1,\ 2.\,N_2\} \\
\lambda y{:}A.\,M\ \mathbf{need}\ \underline{x}.\,N & \equiv & M\ \mathbf{need}\ \underline{x}.\,\lambda y{:}A.\,N \\
M_1\ \mathbf{to}\ x.\,M_2\ \mathbf{need}\ \underline{y}.\,M_3 & \equiv & M_2\ \mathbf{need}\ \underline{y}.\,M_1\ \mathbf{to}\ x.\,M_3 \\
(M_1\ \mathbf{to}\ x.\,M_2)\ \mathbf{to}\ y.\,M_3 & \equiv & M_1\ \mathbf{to}\ x.\,M_2\ \mathbf{to}\ y.\,M_3 \\
(M_1\ \mathbf{need}\ \underline{x}.\,M_2)\ \mathbf{to}\ y.\,M_3 & \equiv & M_1\ \mathbf{need}\ \underline{x}.\,M_2\ \mathbf{to}\ y.\,M_3 \\
(M_1\ \mathbf{need}\ \underline{x}.\,M_2)\ \mathbf{need}\ \underline{y}.\,M_3 & \equiv & M_1\ \mathbf{need}\ \underline{x}.\,M_2\ \mathbf{need}\ \underline{y}.\,M_3
\end{array}\right\}\text{Sequencing laws}$$

Figure 5.2: Core axioms of ECBPV inequational theories

The duplication is acceptable because we should allow these terms to be executed once in each separate execution of a computation (and separate executions arise from duplication of thunks). Only duplication within a *single* execution of a computation is problematic. Computations can also be duplicated across both elements of a pair $\lambda\{1.\,M_1, 2.\,M_2\}$. This is also correct, because only one component of a pair can be used within a single computation (without thunking), so the side-effects will still not happen twice. (There is a similar consideration for functions, which can only be applied once.) The other axioms never duplicate **need**-bound computations that might have effects.

We have seen the majority of the core axioms before: they are the same as the ones for GCBPV and CBPV (see Section 2.7.2). Only the axioms involving **need** are new; these are highlighted. The β-law for **need** parallels the usual β-law for **to**: it gives the behaviour of computation terms that return values without having any side-effects. The interesting axioms are those in the third group of Figure 5.2, which we call *sequencing axioms.*

The first sequencing axiom is the crucial one. It states that if a computation will next evaluate $\underline{x}$, where $\underline{x}$ is a computation variable bound to M, then this is the same as evaluating M, and then using the result for subsequent uses of $\underline{x}$. It implies that

$$\begin{aligned} M\ \mathbf{need}\ \underline{x}.\,\underline{x} &\equiv M\ \mathbf{need}\ \underline{x}.\,\underline{x}\ \mathbf{to}\ y.\,\langle y\rangle && (\eta \text{ law for returner types})\\ &\equiv M\ \mathbf{to}\ y.\,\langle y\rangle \\ &\equiv M && (\eta \text{ law for returner types}) \end{aligned}$$

The second sequencing axiom does *garbage collection* [68]: if a computation bound by **need** is not used (because the variable does not appear), then the binding can be dropped. This equation implies, for example, that

$$M_1\ \mathbf{need}\ \underline{x}_1.\, M_2\ \mathbf{need}\ \underline{x}_2.\ \cdots M_n\ \mathbf{need}\ \underline{x}_n.\,\langle()\rangle \ \equiv\ \langle()\rangle$$

The next five sequencing axioms (two from CBPV and three new) state that binding a computation with **to** or **need** commutes with the remaining forms of computation terms. These allow **to** and **need** to be moved to the outside of other constructs *except* thunks. The final three axioms (one from CBPV and two new) capture associativity involving **need** and **to**. Note that associativity between different evaluation orders is not necessarily valid. In particular, we do not have

$$(M_1\ \mathbf{to}\ x.\, M_2)\ \mathbf{need}\ \underline{y}.\, M_3 \quad\equiv\quad M_1\ \mathbf{to}\ x.\,(M_2\ \mathbf{need}\ \underline{x}.\, M_3)$$

(The first term might not evaluate M_1, the second always does.) This is usually the case when evaluation orders are mixed [79].

These final eight axioms allow computation terms to be placed in normal forms where bindings of computations are on the outside. (Compare this with the translation of source-language *answers* in Section 5.2.)

An inequational theory again consists of two judgment forms, one for values and one for computations:

$$\Gamma \vdash V \preccurlyeq W : A \qquad \Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$$

We require inequational theories to be closed under congruence, so define term contexts $C[]$ (value terms with a single hole) and $\underline{C}[]$ (computation terms with a single hole):

$$\begin{aligned} C[] \ ::=\ & \square \mid (C[], V_2) \mid (V_1, C[]) \mid \mathbf{fst}\, C[] \mid \mathbf{snd}\, C[] \mid \mathbf{case}_A\ C[]\ \mathbf{of}\ \{\} \\ & \mid \mathbf{inl}_{A_2} C[] \mid \mathbf{inr}_{A_1} C[] \mid \mathbf{case}\ C[]\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} \\ & \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, C[],\ \mathbf{inr}\, x_2.\, W_2\} \mid \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, C[]\} \\ & \mid \mathbf{thunk}\, \underline{C}[] \\ \underline{C}[] \ ::=\ & \underline{\square} \mid \lambda\{1.\, M_1, 2.\, \underline{C}[]\} \mid \lambda\{1.\, \underline{C}[], 2.\, M_2\} \mid 1\text{‘}\underline{C}[] \mid 2\text{‘}\underline{C}[] \mid \lambda x{:}A.\, \underline{C}[] \mid C[]\text{‘}M \mid V\text{‘}\underline{C}[] \\ & \mid \mathrm{op}\, C[] \mid \langle C[]\rangle \mid \underline{C}[]\ \mathbf{to}\ x.\, N \mid M\ \mathbf{to}\ x.\, \underline{C}[] \mid \mathbf{force}\, C[] \\ & \mid\ \underline{C}[]\ \mathbf{need}\ \underline{x}.\, N \mid M\ \mathbf{need}\ \underline{x}.\, \underline{C}[] \end{aligned}$$

We also require closure under substitution, so define a judgment

$$\Gamma \vdash \sigma \preccurlyeq \sigma' : \Delta$$

on well-typed substitutions componentwise, using both judgments of the inequational theory. The definition of ECBPV inequational theory is similar to the definition for GCBPV (Definition 2.7.4):

Definition 5.1.3 (Inequational theory) An *inequational theory* consists of a ECBPV signature and two judgments

$$\Gamma \vdash V \preccurlyeq W : A \qquad \Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$$

such that:

- Preorder:
  - Values: if $\Gamma \vdash V : A$ then $\Gamma \vdash V \preccurlyeq V : A$, and if $\Gamma \vdash V_1 \preccurlyeq V_2 : A$ and $\Gamma \vdash V_2 \preccurlyeq V_3 : A$ then $\Gamma \vdash V_1 \preccurlyeq V_3 : A$.
  - Computations: if $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ then $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq M : \underline{C}$, and if $\Gamma \mathrel{\underline{\vdash}} M_1 \preccurlyeq M_2 : \underline{C}$ and $\Gamma \mathrel{\underline{\vdash}} M_2 \preccurlyeq M_3 : \underline{C}$ then $\Gamma \mathrel{\underline{\vdash}} M_1 \preccurlyeq M_3 : \underline{C}$.
- Congruence: if $\Gamma \vdash V \preccurlyeq W : A$ then for term contexts with hole $\square$

$$\begin{array}{lcl} \Gamma' \vdash \mathcal{C}[V] : B \wedge \Gamma' \vdash \mathcal{C}[W] : B & \Rightarrow & \Gamma' \vdash \mathcal{C}[V] \preccurlyeq \mathcal{C}[W] : B \\ \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[V] : \underline{D} \wedge \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[W] : \underline{D} & \Rightarrow & \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[V] \preccurlyeq \underline{\mathcal{C}}[W] : \underline{D} \end{array}$$

  If $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$ then for term contexts with hole $\underline{\square}$

$$\begin{array}{lcl} \Gamma' \vdash \mathcal{C}[M] : B \wedge \Gamma' \vdash \mathcal{C}[N] : B & \Rightarrow & \Gamma' \vdash \mathcal{C}[M] \preccurlyeq \mathcal{C}[N] : B \\ \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] : \underline{D} \wedge \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[N] : \underline{D} & \Rightarrow & \Gamma' \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] \preccurlyeq \underline{\mathcal{C}}[N] : \underline{D} \end{array}$$

- Substitution: If $\Gamma \vdash \sigma \preccurlyeq \sigma' : \Delta$ then

$$\begin{array}{lcl} \Delta \vdash V \preccurlyeq W : B & \Rightarrow & \Gamma \vdash V[\sigma] \preccurlyeq W[\sigma'] : B \\ \Delta \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C} & \Rightarrow & \Gamma \mathrel{\underline{\vdash}} M[\sigma] \preccurlyeq N[\sigma'] : \underline{C} \end{array}$$

- Core axioms:
  - Values: if $\Gamma \vdash V : A$ and $\Gamma \vdash W : A$, and $V \equiv W$ is an instance of an axiom in Figure 5.2, then $\Gamma \vdash V \preccurlyeq W : A$ and $\Gamma \vdash W \preccurlyeq V : A$.
  - Computations: if $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ and $\Gamma \mathrel{\underline{\vdash}} N : \underline{C}$, and $M \equiv N$ is an instance of an axiom in Figure 5.2, then $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq N : \underline{C}$ and $\Gamma \mathrel{\underline{\vdash}} N \preccurlyeq M : \underline{C}$. ◀

To specify an inequational theory, one gives a collection of signature axioms and then closes under the core axioms, reflexivity, transitivity and congruence. The signature axioms may involve **to** (e.g. Figure 2.14), but we do not expect any examples to require signature axioms involving **need**, because the behaviour of **need** is completely characterized by the sequencing axioms.

We define the contextual preorder in the usual way (recall that $\diamond$ is the empty typing context).

Definition 5.1.4 (Contextual preorder) The *contextual preorder* consists of two judgment forms.

1. Between value terms: $\Gamma \vdash V \preccurlyeq_{\text{ctx}} W : A$ if $\Gamma \vdash V : A$, $\Gamma \vdash W : A$, and for all ground types G and term contexts $\underline{\mathcal{C}}[]$ with hole $\square$ such that $\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[V] : \mathbf{F}\, G$ and $\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[W] : \mathbf{F}\, G$ we have

$$\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[V] \preccurlyeq \underline{\mathcal{C}}[W] : \mathbf{F}\, G$$

2. Between computation terms: $\Gamma \mathrel{\underline{\vdash}} M \preccurlyeq_{\text{ctx}} N : \underline{C}$ if $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$, $\Gamma \mathrel{\underline{\vdash}} N : \underline{C}$, and for all ground types G and term contexts $\underline{\mathcal{C}}[]$ with hole $\underline{\square}$ such that $\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] : \mathbf{F}\, G$ and $\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[N] : \mathbf{F}\, G$ we have

$$\diamond \mathrel{\underline{\vdash}} \underline{\mathcal{C}}[M] \preccurlyeq \underline{\mathcal{C}}[N] : \mathbf{F}\, G$$

◀

$$\begin{array}{llcl}
\text{Evaluation contexts} & \mathcal{E}[] & ::= & \square \mid \mathrm{op}\,\mathcal{E}[] \mid \mathbf{if}\ \mathcal{E}[]\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathcal{E}[]\ e_2 \\
 & & & \mid (\lambda x{:}\tau.\,\mathcal{E}[x])\,\mathcal{E}'[] \ \mid\ (\lambda x{:}\tau.\,\mathcal{E}[])\,e_2 \\
\text{Values} & v & ::= & \mathbf{true} \mid \mathbf{false} \mid \lambda x{:}\tau.\,e \\
\text{Answers} & a & ::= & v \mid (\lambda x{:}\tau.\,a)\,e
\end{array}$$

$$\mathbf{if\ true\ \ then}\ e_2\ \mathbf{else}\ e_3 \overset{\text{need}}{\rightsquigarrow} e_2 \qquad\qquad \frac{e \overset{\text{need}}{\rightsquigarrow} e'}{\mathcal{E}[e] \overset{\text{need}}{\rightsquigarrow} \mathcal{E}[e']}$$

$$\mathbf{if\ false\ then}\ e_2\ \mathbf{else}\ e_3 \overset{\text{need}}{\rightsquigarrow} e_3$$

$$(\lambda x{:}\tau.\,\mathcal{E}[x])\,v \overset{\text{need}}{\rightsquigarrow} (\lambda x{:}\tau.\,\mathcal{E}[v])\,v \qquad \begin{array}{rcl}
(\lambda x{:}\tau.\,a)\,e_1\,e_2 & \overset{\text{need}}{\rightsquigarrow} & (\lambda x{:}\tau.\,a\,e_2)\,e_1 \\
(\lambda x{:}\tau.\,\mathcal{E}[x])\,((\lambda y{:}\tau'.\,a)\,e) & \overset{\text{need}}{\rightsquigarrow} & (\lambda y{:}\tau'.\,(\lambda x{:}\tau.\,\mathcal{E}[x])\,a)\,e
\end{array}$$

Figure 5.3: Call-by-need operational semantics

5.2 Call-by-need translation

Our goal with ECBPV is to use it to reason about call-by-need evaluation. To do this, we give a call-by-need translation from our source language (Section 2.2) into ECBPV.

Most of the translation is similar to the Moggi-style call-by-name translation in Figure 2.16. The critical difference is how functions are dealt with. We encode call-by-need functions as terms of the form

$$\lambda x'{:}\mathbf{U}\,\mathbf{F}A.\,(\mathbf{force}\,x')\ \mathbf{need}\ \underline{x}.\,M$$

where x' is not free in M. This is an ECBPV function that accepts a thunk as an argument. The thunk is added to the context, and the body of the function is executed. The first time the argument is used (via $\underline{x}$), the computation inside the thunk is evaluated. Subsequent uses do not evaluate the computation again.

Before defining the translation we give a call-by-need operational semantics for our source language, based on Ariola and Felleisen's [3]. The only differences between our source language and Ariola and Felleisen's calculus are the addition of booleans, operations, and a type system. It is likely that we can translate other call-by-need calculi, such as those of Launchbury [53] and Maraist et al. [68]. Call-by-need small-step reductions are written $e \overset{\text{need}}{\rightsquigarrow} e'$; this is defined in Figure 5.3.

The call-by-need semantics needs some auxiliary definitions. An *evaluation context* $\mathcal{E}[]$ is a source-language expression with a single hole $\square$, picked from the grammar given in the figure. (There is only one kind of reduction context and one kind of hole because the syntax of the source language is not stratified into values and computations.) The hole in an evaluation context indicates where reduction is currently taking place: it says which part of the expression is currently *needed*. We write $\mathcal{E}[e]$ for the expression in which the hole is replaced with e. A (source-language) *value* is the result of a computation (the word value should not be confused with the value terms of ECBPV. An *answer* is a value in some *environment*, which maps variables to expressions. Answers can be thought of as *closures*. The environment is encoded in an answer using application and lambda abstraction: the answer $(\lambda x : \tau.\, a)\, e$ means the answer a where the environment maps x to e. Encoding environments in this way makes the translation slightly simpler than if we had used a Launchbury-style [53] call-by-need language with explicit environments. In the latter case, the translation would need to encode

$$\begin{aligned} \llparenthesis \mathbf{unit} \rrparenthesis &:= \mathbf{unit} \\ \llparenthesis \mathbf{bool} \rrparenthesis &:= \mathbf{unit} + \mathbf{unit} \\ \llparenthesis \tau \longrightarrow \tau' \rrparenthesis &:= \mathbf{U}\,(\mathbf{U}\,(\mathbf{F}\,\llparenthesis \tau \rrparenthesis) \to \mathbf{F}\,\llparenthesis \tau' \rrparenthesis) \end{aligned} \qquad \begin{aligned} \llparenthesis \diamond \rrparenthesis &:= \diamond \\ \llparenthesis \Gamma, x : \tau \rrparenthesis &:= \llparenthesis \Gamma \rrparenthesis, \underline{x} : \mathbf{F}\,\llparenthesis \tau \rrparenthesis \end{aligned}$$

$$\frac{}{\llparenthesis \Gamma \vdash x : \tau \rrparenthesis := \underline{x}} \qquad \frac{\llparenthesis \Gamma \vdash e : \mathrm{car}_{\mathrm{op}} \rrparenthesis = M}{\llparenthesis \Gamma \vdash \mathrm{op}\, e : \mathrm{ar}_{\mathrm{op}} \rrparenthesis = M \;\mathbf{to}\; x.\, \mathrm{op}\, x} \qquad \frac{}{\llparenthesis \Gamma \vdash () : \mathbf{unit} \rrparenthesis := \langle () \rangle}$$

$$\frac{}{\llparenthesis \Gamma \vdash \mathbf{true} : \mathbf{bool} \rrparenthesis := \langle \mathbf{inl}_{\mathbf{unit}}\, () \rangle} \qquad \frac{}{\llparenthesis \Gamma \vdash \mathbf{false} : \mathbf{bool} \rrparenthesis := \langle \mathbf{inr}_{\mathbf{unit}}\, () \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash e_1 : \mathbf{bool} \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash e_2 : \tau \rrparenthesis = M_2 \qquad \llparenthesis \Gamma \vdash e_3 : \tau \rrparenthesis = M_3}{\llparenthesis \Gamma \vdash \mathbf{if}\; e_1 \;\mathbf{then}\; e_2 \;\mathbf{else}\; e_3 : \tau \rrparenthesis \;:=\; M_1 \;\mathbf{to}\; x.\, \underline{\mathbf{case}}\; x \;\mathbf{of}\; \{\mathbf{inl}\, \_.\, M_2,\; \mathbf{inr}\, \_.\, M_3\}}$$

$$\frac{\llparenthesis \Gamma, x : \tau \vdash e : \tau' \rrparenthesis = M}{\llparenthesis \Gamma \vdash \lambda x{:}\tau.\, e : \tau \longrightarrow \tau' \rrparenthesis \;:=\; \langle \mathbf{thunk}\, (\lambda x'{:}\mathbf{U}\,(\mathbf{F}\,\llparenthesis \tau \rrparenthesis).\, \mathbf{force}\, x' \;\mathbf{need}\; \underline{x}.\, M) \rangle}$$

$$\frac{\llparenthesis \Gamma \vdash e_1 : \tau \longrightarrow \tau' \rrparenthesis = M_1 \qquad \llparenthesis \Gamma \vdash e_2 : \tau \rrparenthesis = M_2}{\llparenthesis \Gamma \vdash e_1\, e_2 : \tau' \rrparenthesis \;:=\; M_1 \;\mathbf{to}\; f.\, (\mathbf{thunk}\, M_2)\,\text{‘}\,(\mathbf{force}\, f)}$$

Figure 5.4: Call-by-need translation of call-by-need types (top left), contexts (top right), and expressions into ECBPV.

the environments; here they are already encoded inside expressions. Answers are terminal computations: they do not reduce.

The two axioms for reducing **if**-expressions are obvious. The congruence rule only allows reducing the expression that is needed by the computation. The axiom on the bottom left of Figure 5.3 is the most important: it states that if the subexpression currently being evaluated is a variable x, and the environment maps x to a source-language value v, then that use of x can be replaced with v. Note that $\mathcal{E}[v]$ may contain other uses of x; the replacement only occurs when the value is actually needed. This axiom roughly corresponds to the first sequencing axiom in Figure 5.2. The two axioms on the bottom right of Figure 5.3 rearrange the environment into a standard form. Both have a syntactic restriction to answers so that each expression has at most one reduct (this restriction is not needed to ensure that $\overset{\text{need}}{\rightsquigarrow}$ captures call-by-need).

The call-by-need translation from the source language to ECBPV is defined in Figure 5.4. We assume that each operation op in the source-language signature is also included in the ECBPV signature, and that if τ and τ' are respectively the coarity and arity of op in the source-language signature then $\llparenthesis \tau \rrparenthesis^{\text{need}}$ and $\llparenthesis \tau' \rrparenthesis^{\text{need}}$ are the coarity and arity in the ECBPV signature.

The translation of types is the same as the Moggi-style call-by-name translation in Figure 2.16 (ignoring the grading), so types τ are translated into value types $\llparenthesis \tau \rrparenthesis^{\text{need}}$. (We omit the superscript in the figure.) On function types $\tau \longrightarrow \tau'$, we use ECBPV function types that receive thunks of computations as arguments (the argument is not evaluated before the function is applied). The call-by-need translation differs only on typing contexts and expressions (since there is no subeffecting, we do not have to be careful about the distinction between typing derivations and well-typed expressions).

Source-language typing contexts Γ are translated into ECBPV typing contexts $\llparenthesis \Gamma \rrparenthesis^{\text{need}}$, containing computations that return values. The computations in the context are all bound using **need**. An expression $\Gamma \vdash e : \tau$ is translated to a computation $\llparenthesis e \rrparenthesis^{\mathrm{need}}$ that returns $\llparenthesis \tau \rrparenthesis^{\mathrm{need}}$ in the context $\llparenthesis \Gamma \rrparenthesis^{\mathrm{need}}$. The key case is the translation of lambdas. These become computations that immediately return a thunk of a function. The function places the computation given as an argument onto the context using **need**, so that it is evaluated at most once, before executing the body. The remainder of the cases are similar to the Moggi-style call-by-name translation.

Under the call-by-need translation, the expression $(\lambda x : \tau.\, e_1)\, e_2$ is translated into a term that executes the computation $\llparenthesis e_1 \rrparenthesis^{\mathrm{need}}$, and executes $\llparenthesis e_2 \rrparenthesis^{\mathrm{need}}$ only when needed. This is the case because, by the β rules for thunks, functions, and returner types:

$$\llparenthesis (\lambda x : \tau.\, e_1)\, e_2 \rrparenthesis^{\mathrm{need}} \;\equiv\; \llparenthesis e_2 \rrparenthesis^{\mathrm{need}}\ \mathbf{need}\ \underline{x}.\ \llparenthesis e_1 \rrparenthesis^{\mathrm{need}}$$

As a consequence, translations of answers are particularly simple: they have the following form (up to $\equiv$):

$$M_1\ \mathbf{need}\ \underline{x}_1.\, M_2\ \mathbf{need}\ \underline{x}_2.\ \cdots M_n\ \mathbf{need}\ \underline{x}_n.\ \langle V \rangle$$

which intuitively means the value V in the environment mapping each $\underline{x}_i$ to M_i.

The call-by-need translation produces ECBPV computations of the correct type.

Lemma 5.2.1 If $\Gamma \vdash e : \tau$ then $\llparenthesis \Gamma \rrparenthesis^{\mathrm{need}} \underline{\vdash} \llparenthesis e \rrparenthesis^{\mathrm{need}} : \mathbf{F}\, \llparenthesis \tau \rrparenthesis^{\mathrm{need}}$. ◀

We prove that the call-by-need translation is *sound*: if $e \overset{\mathrm{need}}{\rightsquigarrow} e'$ then $\llparenthesis e \rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e' \rrparenthesis^{\mathrm{need}}$. To do this, we first look at translations of evaluation contexts. The following lemma says the translation captures the idea that the hole in an evaluation context corresponds to the term being evaluated.

Lemma 5.2.2 Define, for each evaluation context $\mathcal{E}[]$, the term context $\llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}$ by:

$$\begin{aligned}
\llparenthesis \square \rrparenthesis^{\mathrm{need}} &:= \langle \square \rangle \\
\llparenthesis \mathrm{op}\, \mathcal{E}[] \rrparenthesis^{\mathrm{need}} &:= \llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}\ \mathbf{to}\ x.\, \mathrm{op}\, x \\
\llparenthesis \mathbf{if}\ \mathcal{E}[]\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrparenthesis^{\mathrm{need}} &:= \llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}\ \mathbf{to}\ x.\ \underline{\mathbf{case}}\ x\ \mathbf{of} \\
&\qquad\qquad\quad \{\mathbf{inl}\ \_.\, \mathbf{thunk}\, \llparenthesis e_2 \rrparenthesis^{\mathrm{need}} \\
&\qquad\qquad\quad ,\ \mathbf{inr}\ \_.\, \mathbf{thunk}\, \llparenthesis e_3 \rrparenthesis^{\mathrm{need}}\} \\
\llparenthesis \mathcal{E}[]\ e_2 \rrparenthesis^{\mathrm{need}} &:= \llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}\ \mathbf{to}\ z.\, \mathbf{thunk}\, \llparenthesis e_2 \rrparenthesis^{\mathrm{need}}\, \text{‘}\, \mathbf{force}\, z \\
\llparenthesis (\lambda x : \tau.\, \mathcal{E}[x])\, \mathcal{E}'[] \rrparenthesis^{\mathrm{need}} &:= \llparenthesis \mathcal{E}'[] \rrparenthesis^{\mathrm{need}}\ \mathbf{need}\ \underline{x}.\ \llparenthesis \mathcal{E}[\underline{x}] \rrparenthesis^{\mathrm{need}} \\
\llparenthesis (\lambda x.\, \mathcal{E}[])\, e_2 \rrparenthesis^{\mathrm{need}} &:= \llparenthesis e_2 \rrparenthesis^{\mathrm{need}}\ \mathbf{need}\ \underline{x}.\ \llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}
\end{aligned}$$

For each expression e we have:

$$\llparenthesis \mathcal{E}[e] \rrparenthesis^{\mathrm{need}} \;\equiv\; \llparenthesis e \rrparenthesis^{\mathrm{need}}\ \mathbf{to}\ y.\ \llparenthesis \mathcal{E}[] \rrparenthesis^{\mathrm{need}}[y]$$

(where y is fresh). ◀

This lemma omits the typing of expressions for presentational purposes. Soundness is now easy to show:

Theorem 5.2.3 (Soundness) If e and e' are closed, well-typed source-language expressions then $e \overset{\mathrm{need}}{\rightsquigarrow} e'$ implies $\llparenthesis e \rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e' \rrparenthesis^{\mathrm{need}}$. ◀

5.3 Equivalence between call-by-name and call-by-need

Extended call-by-push-value can be used to prove equivalences between evaluation orders. In this section we prove a classic example: if the only side-effect is nontermination, then call-by-name is equivalent to call-by-need. We do this in two stages.

First, we show that call-by-name is equivalent to call-by-need *within* ECBPV. Specifically, we show that

$$M \,\mathbf{name}\, \underline{x}.\, N \;\cong_{\mathrm{ctx}}\; M \,\mathbf{need}\, \underline{x}.\, N$$

(Recall that M **name** $\underline{x}.\, N$ is syntactic sugar for $\mathbf{thunk}\, M \,\text{`}\, \lambda y.\, N[\underline{x} \mapsto \mathbf{force}\, y]$.)

Second, an important corollary is that the meta-level reduction strategies are equivalent. We show that the call-by-need translation in Section 5.2 and the Moggi-style call-by-name translation in Figure 2.16 (ignoring grading) give contextually equivalent ECBPV terms. We expect the Levy-style translation in Figure 2.17 to work equally well, except that we would have to define a more complicated Galois connection between the two evaluation orders (see Chapter 3).

To model nontermination being the sole source-language effect, we choose the ECBPV signature with no constants or base types, and a single operation:

Operation op	Coarity $\mathrm{car}_{\mathrm{op}}$	Arity $\mathrm{ar}_{\mathrm{op}}$
diverge	**unit**	**empty**

We expect our proofs to work with general fixed-point operators, but for simplicity we do not consider this here. The operation enables us to define a diverging computation $\Omega_{\underline{C}}$ of each computation type $\underline{C}$, using the eliminator of the empty type:

$$\Omega_{\underline{C}} \;:=\; \mathsf{diverge}\,()\ \mathbf{to}\ x.\, \underline{\mathbf{case}}_{\underline{C}}\ x\ \mathbf{of}\ \{\}$$

We use the smallest inequational theory with this signature. This inequational theory is symmetric. We do not need any signature axioms to characterize nontermination; the core axioms are sufficient. For example, associativity of **to** and the η-law for **empty** imply

$$\Omega_{\mathbf{F}A}\ \mathbf{to}\ x.\, M \;\equiv\; \Omega_{\underline{C}}$$

So diverging as part of a larger computation causes the entire computation to diverge.

We first show that

$$M \,\mathbf{name}\, \underline{x}.\, N \;\cong_{\mathrm{ctx}}\; M \,\mathbf{need}\, \underline{x}.\, N$$

As we did in Chapter 3 we use logical relations to get a strong enough inductive hypothesis for the proof to go through. However, unlike the usual case, it does not suffice to relate *closed* terms. To see why, consider a closed term M of the form

$$\Omega_{\mathbf{F}A}\ \mathbf{need}\ \underline{x}.\, N_1\ \mathbf{to}\ y.\, N_2$$

If we relate only closed terms, then we do not learn anything about N_1 itself (since $\underline{x}$ may be free in it). We could attempt to proceed by considering the closed term $\Omega_{\mathbf{F}A}$ **need** $\underline{x}.\, N_1$. For example, if this returns a value V then $\underline{x}$ cannot have been evaluated and M should have the same behaviour as $\Omega_{\mathbf{F}A}$ **need** $\underline{x}.\, N_2[y \mapsto V]$. However, we get stuck when proving the last step. This is only a problem because $\Omega_{\mathbf{F}A}$ is a nonterminating computation: every closed, terminating computation of returner type has the form $\langle V \rangle$ (up to $\equiv$), and when these are bound using **need** we can eliminate the binding using the equation

$$\langle V \rangle\ \mathbf{need}\ \underline{x}.\, N_1\ \mathbf{to}\ y.\, N_2 \;\equiv\; (N_1\ \mathbf{to}\ y.\, N_2)[\underline{x} \mapsto \langle V \rangle]$$

The solution is to relate terms that may have free computation variables (we do not have to consider free value variables). The free computation variables should be thought of as referring to nonterminating computations, because we can remove the bindings of variables that refer to terminating computations. We relate open terms using *Kripke logical relations of varying arity*, which were introduced by Jung and Tiuryn [39] to study lambda definability.

We need a number of definitions first. We define Term_A^Γ as the set of equivalence classes (up to $\equiv$) of terms of value type A in context Γ, and similarly define $\underline{\mathrm{Term}}_{\underline{D}}^\Gamma$ for computation types:

$$\mathrm{Term}_A^\Gamma \coloneqq \{[V]_\equiv \mid \Gamma \vdash V : A\} \qquad \underline{\mathrm{Term}}_{\underline{D}}^\Gamma \coloneqq \{[M]_\equiv \mid \Gamma \mathrel{\underline{\vdash}} M : \underline{D}\}$$

A computation-type context Δ is an ECBPV typing context that contains only computation variables $\underline{x}$. In this section Δ ranges over computation-type contexts only, Γ ranges over arbitrary typing contexts. Variables in computation-type contexts refer to nonterminating computations for the proof of contextual equivalence. A computation-type context Δ' *weakens* another context Δ, written $\Delta' \triangleright \Delta$, whenever Δ is a sublist of Δ'. Formally, this relation is generated by the following rules:

$$\frac{}{\diamond \triangleright \diamond} \qquad\qquad \frac{\Delta' \triangleright \Delta}{\Delta', \underline{x} : \mathbf{F}A \triangleright \Delta, \underline{x} : \mathbf{F}A} \qquad\qquad \frac{\Delta' \triangleright \Delta}{\Delta', \underline{x} : \mathbf{F}A \triangleright \Delta}$$

The type system of ECBPV admits a weakening lemma, so if $\Delta' \triangleright \Delta$ we can weaken any $\Delta \vdash V : A$ to get $\Delta' \vdash V : A$, and similarly for computations.

A *Kripke relation* is a family of binary relations indexed by computation-type contexts that respects weakening of terms:

Definition 5.3.1 (Kripke relation) A *Kripke relation* R over a value type A (respectively a computation type $\underline{D}$) is a family of relations $R^\Delta \subseteq \mathrm{Term}_A^\Delta \times \mathrm{Term}_A^\Delta$ (respectively $R^\Delta \subseteq \underline{\mathrm{Term}}_{\underline{D}}^\Delta \times \underline{\mathrm{Term}}_{\underline{D}}^\Delta$) indexed by computation-type contexts Δ such that whenever $\Delta' \triangleright \Delta$, we have $M \in R^\Delta$ implies $M \in R^{\Delta'}$ (where M is weakened). ◀

We consider binary relations on equivalence classes of terms because we want to relate pairs of terms up to $\equiv$, as we did in Chapter 3 (to prove contextual equivalence).

First we observe that, because computation variables are bound to nonterminating computations, some computations can easily be seen to diverge:

Definition 5.3.2 A computation $\Delta \mathrel{\underline{\vdash}} M : \underline{C}$ is *trivially diverging* if

$$M \;\equiv\; N_1 \;\mathbf{to}\; x.\, N_2$$

for some value type A, computation $N_1 \in \{\underline{x} \mid (\underline{x} : \mathbf{F}A) \in \Delta\} \cup \{\Omega_{\mathbf{F}A}\}$ and computation $\Gamma, x : \mathbf{F}A \mathrel{\underline{\vdash}} N_2 : \underline{C}$. ◀

If M is trivially diverging then we cannot necessarily show $M \equiv \Omega_{\underline{C}}$ because N_1 might be $\underline{x}$. However, we can show this once we bind all of the variables in the computation context Δ to nonterminating computations. We need the Kripke relations we define over computation terms to be *closed under divergence*. (For the rest of this section, we omit the square brackets around equivalence classes.)

Definition 5.3.3 A Kripke relation R over a computation type $\underline{C}$ is *closed under divergence* if it is transitive and each of the following holds:

1. If $\Delta \mathrel{\underline{\vdash}} M : \underline{C}$ and $\Delta \mathrel{\underline{\vdash}} M' : \underline{C}$ are trivially diverging, then $(M, M') \in R^\Delta$.

$$
\begin{aligned}
\mathcal{R}[\![\mathbf{unit}]\!]^{\Delta} &:= \mathrm{Term}_{\mathbf{unit}} \times \mathrm{Term}_{\mathbf{unit}} \\
\mathcal{R}[\![A_1 \times A_2]\!]^{\Delta} &:= \{(V, V') \mid (\mathbf{fst}\, V, \mathbf{fst}\, V') \in \mathcal{R}[\![A_1]\!]^{\Delta} \wedge (\mathbf{snd}\, V, \mathbf{snd}\, V') \in \mathcal{R}[\![A_2]\!]^{\Delta}\} \\
\mathcal{R}[\![\mathbf{empty}]\!]^{\Delta} &:= \emptyset \\
\mathcal{R}[\![A_1 + A_2]\!]^{\Delta} &:= \{(\mathbf{inl}_{A_2} V, \mathbf{inl}_{A_2} V') \mid (V, V') \in \mathcal{R}[\![A_1]\!]^{\Delta}\} \cup \\
& \{(\mathbf{inr}_{A_1} V, \mathbf{inr}_{A_1} V') \mid (V, V') \in \mathcal{R}[\![A_2]\!]^{\Delta}\} \\
\mathcal{R}[\![\mathbf{U}\, \underline{C}]\!]^{\Delta} &:= \{(V, V') \mid (\mathbf{force}\, V, \mathbf{force}\, V') \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}\} \\
\mathcal{R}[\![\mathbf{F}\, A]\!] &:= \text{the smallest closed-under-divergence Kripke relation such that} \\
&\qquad\qquad (V, V') \in \mathcal{R}[\![A]\!]^{\Delta} \implies (\langle V \rangle, \langle V' \rangle) \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta} \\
\mathcal{R}[\![\underline{\mathbf{unit}}]\!]^{\Delta} &:= \underline{\mathrm{Term}}_{\underline{\mathbf{unit}}} \times \underline{\mathrm{Term}}_{\underline{\mathbf{unit}}} \\
\mathcal{R}[\![\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2]\!]^{\Delta} &:= \{(M, M') \mid (1`M, 1`M') \in \mathcal{R}[\![\underline{C}_1]\!]^{\Delta} \wedge (2`M, 2`M') \in \mathcal{R}[\![\underline{C}_2]\!]^{\Delta}\} \\
\mathcal{R}[\![A \to \underline{C}]\!]^{\Delta} &:= \{(M, M') \mid \forall \Delta' \triangleright \Delta,\ (V, V') \in \mathcal{R}[\![A]\!]^{\Delta'}.\, (V`M, V'`M') \in \mathcal{R}[\![\underline{C}]\!]^{\Delta'}\}
\end{aligned}
$$

Figure 5.5: Kripke logical relation for ECBPV with nontermination

2. If $\Delta \mathrel{\underline{\vdash}} M : \mathbf{F}\, A$ and $\Delta \mathrel{\underline{\vdash}} M' : \mathbf{F}\, A$ are trivially diverging, and $(N, N') \in R^{\Delta, \underline{y} : \mathbf{F}\, A}$, then all four of the following pairs are in R^{Δ}:

$$
\begin{aligned}
&(M\ \mathbf{need}\ \underline{y}.\, N,\ M'\ \mathbf{need}\ \underline{y}.\, N') &\qquad& (N[\underline{y} \mapsto M],\ N'[\underline{y} \mapsto M']) \\
&(N[\underline{y} \mapsto M],\ M'\ \mathbf{need}\ \underline{y}.\, N') &\qquad& (M\ \mathbf{need}\ \underline{y}.\, N,\ N'[\underline{y} \mapsto M'])
\end{aligned}
\qquad \blacktriangleleft
$$

This definition works because all of the trivially diverging computations are interchangeable in the semantics. The second part of the definition is the most important. Since we are showing that **need** is contextually equivalent to substitution, we want these to be related; this justifies the two pairs at the bottom of the definition. We have to consider computation variables in the definition (inside trivially diverging computations) only because of our use of Kripke logical relations. For ordinary logical relations, there would be no free variables to consider.

The key part of the proof of contextual equivalence is the definition of the Kripke logical relation $\mathcal{R}[\![-]\!]$, which is a family of relations indexed by value and computation types. It is defined in Figure 5.5 by induction on the structure of the types. In the figure, we again omit square brackets around equivalence classes.

The definition on most of the type formers is standard, and similar to the equations required to hold for GCBPV logical relations (Figure 3.1). For returner types, we want any pair of computations that return related values to be related. We also want the relation to be closed under divergence, in order to show the fundamental lemma (below) for **to** and **need**. We therefore define $\mathcal{R}[\![\mathbf{F}\, A]\!]$ as the smallest such Kripke relation. For function types, we require as usual that related arguments are sent to related results. For this to define a Kripke relation, we have to quantify over all computation-type contexts Δ' that weaken Δ, because of the contravariance of the argument. The computations M and M' are weakened so that they are in the context Δ'.

The relations we define are Kripke relations. Using the sequencing axioms of ECBPV inequational theories, and the β- and η-laws for computation types, we can show that $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence for each computation type $\underline{C}$. These facts are important for the proof of the fundamental lemma.

We define sets $\mathrm{Subst}^{\Delta}_{\Gamma}$ of equivalence classes of substitutions (where $\equiv$ is given componentwise on substitutions), and extend the logical relation by defining $\mathcal{R}[\![\Gamma]\!]^{\Delta} \subseteq \mathrm{Subst}^{\Delta}_{\Gamma} \times \mathrm{Subst}^{\Delta}_{\Gamma}$:

$$\begin{aligned}
\mathrm{Subst}^{\Delta}_{\Gamma} &:= \{[\sigma]_{\equiv} \mid \Delta \vdash \sigma : \Gamma\} \\
\mathcal{R}[\![\diamond]\!]^{\Delta} &:= \{(\diamond, \diamond)\} \\
\mathcal{R}[\![\Gamma, x : A]\!]^{\Delta} &:= \{((\sigma,\ x \mapsto V), (\sigma',\ x \mapsto V')) \mid (\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta} \wedge (V, V') \in \mathcal{R}[\![A]\!]^{\Delta}\} \\
\mathcal{R}[\![\Gamma, \underline{x} : \mathbf{F}\,A]\!]^{\Delta} &:= \{((\sigma,\ \underline{x} \mapsto M), (\sigma',\ \underline{x} \mapsto M')) \mid (\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta} \wedge (M, M') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta}\}
\end{aligned}$$

As usual, the logical relations satisfy a *fundamental lemma.*

Lemma 5.3.4 (Fundamental) Suppose that $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$.

1. (Values) If $\Gamma \vdash V : A$ then $(V[\sigma], V[\sigma']) \in \mathcal{R}[\![A]\!]^{\Delta}$.
2. (Computations) If $\Gamma \vdash_{\mathsf{c}} M : \underline{C}$ then $(M[\sigma], M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$.

Proof. By induction on the structure of the terms. For two of the cases we use lemmas given in the appendix: Lemma B.3.3 for **need** and Lemma B.3.4 for **to**. For computation variables we use the assumption about σ and σ'. For the operation undef, we use the fact that undef () is trivially diverging. The rest of the cases are standard. □

We also have the following two facts about the logical relation. The first roughly is that **name** is related to **need** by the logical relation, and is true because of the additional pairs that are related in the definition of closed-under-divergence (Definition 5.3.3).

Lemma 5.3.5 For all computation terms $\Gamma \vdash_{\mathsf{c}} M : \mathbf{F}\,A$ and $\Gamma, \underline{x} : \mathbf{F}\,A \vdash_{\mathsf{c}} N : \underline{C}$, if $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$ then

$$((N[\underline{x} \mapsto M])[\sigma],\ (M\ \mathbf{need}\ \underline{x}.\, N)[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$$

Proof. Apply the fundamental lemma (Lemma 5.3.4) to M, and then use Lemma B.3.5. □

The second fact is that related terms are contextually equivalent (similar to Lemma 3.1.5).

Lemma 5.3.6

1. For all value terms $\Gamma \vdash V : A$ and $\Gamma \vdash V' : A$, if $(V[\sigma], V'[\sigma']) \in \mathcal{R}[\![A]\!]^{\Delta}$ for all $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$ then
$$\Gamma \vdash V \cong_{\mathrm{ctx}} V' : A$$
2. For all computation terms $\Gamma \vdash_{\mathsf{c}} M : \underline{C}$ and $\Gamma \vdash_{\mathsf{c}} M' : \underline{C}$, if $(M[\sigma], M'[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$ for all $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$ then
$$\Gamma \vdash_{\mathsf{c}} M \cong_{\mathrm{ctx}} M' : \underline{C}$$

Proof. The logical relation is closed under congruence (placing terms in contexts $C[]$ and $\underline{C}[]$) by a similar proof to the fundamental lemma (Lemma 5.3.4). Hence it suffices to show that if $N, N' : \mathbf{F}\,G$ are closed computations, with G a ground type, then $(N, N') \in \mathcal{R}[\![\mathbf{F}\,G]\!]^{\diamond}$ implies $N \equiv N'$. This is proved in the appendix as Corollary B.3.9. □

This gives us enough to prove the internal equivalence between call-by-name and call-by-need:

Theorem 5.3.7 For all computation terms $\Gamma \vdash_{\mathsf{c}} M : \mathbf{F}\,A$ and $\Gamma, \underline{x} : \mathbf{F}\,A \vdash_{\mathsf{c}} N : \underline{C}$, we have

$$\Gamma \vdash_{\mathsf{c}} M\ \mathbf{name}\ \underline{x}.\, N \cong_{\mathrm{ctx}} M\ \mathbf{need}\ \underline{x}.\, N : \underline{C}$$

Proof. Apply Lemma 5.3.5 to show that the two computations are related to each other, and then apply Lemma 5.3.6 to obtain the contextual equivalence. □

We now move on to the meta-level equivalence. Suppose we are given a (possibly open) source-language expression $\Gamma \vdash e : \tau$. Recall that the call-by-need translation uses a context containing computation variables (i.e. $(\!|\Gamma|\!)^{\text{need}}$) and the call-by-name translation uses a context containing value variables, which map to thunks of computations. We have two ECBPV computation terms of type $\mathbf{F}\,(\!|\tau|\!)$ in context $(\!|\Gamma|\!)^{\text{need}}$: one is just $(\!|e|\!)^{\text{need}}$, the other is the call-by-name translation $(\!|e|\!)^{\text{moggi}}$ (defined in Section 2.4) with all of its variables substituted with thunked computations. The theorem then states that these are contextually equivalent.

Theorem 5.3.8 (Equivalence between call-by-name and call-by-need) If e is a source-language expression that satisfies $\underline{x}_1 : \tau_1, \ldots, \underline{x}_n : \tau_n \vdash e : \tau'$ then

$$(\!|e|\!)^{\text{moggi}}[x_1 \mapsto \mathbf{thunk}\,\underline{x}_1, \ldots, x_n \mapsto \mathbf{thunk}\,\underline{x}_n] \quad \cong_{\text{ctx}} \quad (\!|e|\!)^{\text{need}}$$

Proof. By induction on the typing derivation of e. The interesting case is lambda abstraction, where we use the internal equivalence between call-by-name and call-by-need above (Theorem 5.3.7). □

5.4 An effect system for extended call-by-push-value

The equivalence between call-by-name and call-by-need in the previous section is predicated on the only side-effect in the language being nontermination. We discuss how to relax this restriction so that only subterms need to be restricted to nontermination and the language itself have other side-effects.

Call-by-need makes it difficult to statically estimate effects. Computation variables bound using **need** might have effects on their first use, but on subsequent uses do not. Hence to precisely determine the effects of a term, we must track which variables have been previously used.

McDermott and Mycroft [69] show how to achieve this for a call-by-need effect system, and we expect that this technique can be adapted to ECBPV. Here we take a simpler approach. By slightly restricting the effect algebras we consider, we remove the need to track variable usage information, while still ensuring the effect information is not an underestimate (an underestimate would enable incorrect transformations). This can reduce the precision of the effect information obtained, but for our use case (determining equivalences between evaluation orders) this is not an issue, since we primarily care about which side-effects are used (rather than e.g. how many times they are used).

If we do not track usage information and assume any use of a variable may have side-effects, we might misestimate the effect of a call-by-need computation variable evaluated for a second time (whose true effect is 1). To ensure this misestimate is an overestimate, we require that the effect algebra is *pointed*.

Definition 5.4.1 (Pointed preordered monoid) A preordered monoid $(\mathcal{E}, \leq, \cdot, 1)$ is *pointed* if for all $\varepsilon \in \mathcal{E}$ we have $1 \leq \varepsilon$. ◀

Many of our examples of effect algebras are pointed, including Gifford-style effect algebras (Example 2.1.2).

In our example, where we wish to establish whether the effects of an expression are restricted to nontermination, we use the two-element preorder $\{\Omega \leq \top\}$ with join for sequencing and Ω

$$\frac{\Gamma \vdash V : A}{\Gamma \mathrel{\underline{\vdash}} \langle V \rangle : \langle 1 \rangle A} \qquad \frac{\Gamma \mathrel{\underline{\vdash}} M : \langle \varepsilon \rangle A \qquad \Gamma, x : A \mathrel{\underline{\vdash}} N : \underline{C}}{\Gamma \mathrel{\underline{\vdash}} M \,\mathbf{to}\, x.\, N : \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}}$$

$$\frac{\Gamma \mathrel{\underline{\vdash}} M : \langle \varepsilon \rangle A \qquad \Gamma, \underline{x} : \langle \varepsilon \rangle A \mathrel{\underline{\vdash}} N : \underline{C}}{\Gamma \mathrel{\underline{\vdash}} M \,\mathbf{need}\, \underline{x}.\, N : \underline{C}} \qquad \frac{}{\Gamma \mathrel{\underline{\vdash}} \underline{x} : \langle \varepsilon \rangle A}\ \text{if } (\underline{x} : \langle \varepsilon \rangle A) \in \Gamma$$

$$\frac{\Gamma \mathrel{\underline{\vdash}} M : \langle \varepsilon \rangle A}{\Gamma \mathrel{\underline{\vdash}} \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M : \langle \varepsilon' \rangle A}\ \text{if } \varepsilon \leq \varepsilon'$$

Figure 5.6: Effect system modifications to ECBPV

as the unit 1. The effect Ω means side-effects restricted to (at most) nontermination, and $\top$ means unrestricted side-effects. Thus we would enable the equivalence between call-by-name and call-by-need when the effect is Ω, and not when it is $\top$. This effect algebra is pointed.

The effect system is similar to GCBPV. The syntax is a minor modification of the syntax for ECBPV in Section 5.1. We grade the returner types $\mathbf{F}\,A$ so that they have the form $\langle \varepsilon \rangle A$, where $\varepsilon \in \mathcal{E}$. The grammar of types is therefore identical to the one for GCBPV:

$$\underline{C}, \underline{D} \ ::= \ \underline{\mathbf{unit}} \mid \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2 \mid A \to \underline{C} \mid \langle \varepsilon \rangle A$$

Typing contexts still assign returner types to computation variables, but they now also have effects:

$$\Gamma \ ::= \ \diamond \mid \Gamma, x : A \mid \Gamma, \underline{x} : \langle \varepsilon \rangle \underline{C}$$

We need to be able to overapproximate effects, and therefore add explicit subtyping to the grammar of ECBPV terms:

$$\begin{aligned} M, N \ ::= \ & \lambda\{\} \mid \lambda\{1.\, M_1, 2.\, M_2\} \mid 1\text{‘}M \mid 2\text{‘}M \mid \lambda x{:}A.\, M \mid V\text{‘}M \\ & \mid \mathrm{op}\, V \mid \langle V \rangle \mid M \,\mathbf{to}\, x.\, N \mid \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M \mid \mathbf{force}\, V \mid \underline{x} \mid M \,\mathbf{need}\, x.\, N \end{aligned}$$

The syntax of value types and value terms is generated by the same grammar as before (in Section 5.1). The ECBPV effect system is also parameterized by the same notion of signature as for GCBPV (Definition 2.7.1).

We define the action of the preordered monoid of effects on computation types as for GCBPV:

$$\begin{aligned} \langle\!\langle \varepsilon \rangle\!\rangle \underline{\mathbf{unit}} &:= \underline{\mathbf{unit}} & \langle\!\langle \varepsilon \rangle\!\rangle (\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2) &:= \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_1 \mathbin{\underline{\times}} \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}_2 \\ \langle\!\langle \varepsilon \rangle\!\rangle (A \to \underline{C}) &:= A \to \langle\!\langle \varepsilon \rangle\!\rangle \underline{C} & \langle\!\langle \varepsilon \rangle\!\rangle (\langle \varepsilon' \rangle A) &:= \langle \varepsilon \cdot \varepsilon' \rangle A \end{aligned}$$

The typing judgments have exactly the same form as before (except for the new syntax of types). The majority of the typing rules, including all of the rules for values, are also unchanged.

The only rules we change are those for computation variables, returning values, **to** and **need**, which are replaced with the first four rules in Figure 5.6. We also add a subeffecting rule, which is the last rule of the figure. The rules for return, **to** and **coerce** are the same as the GCBPV rules.

We also have to change the inequational theory to add axioms for **coerce** (e.g. transitivity and commutativity with **to**), and also need to add coercions to the axiom

$$M \,\mathbf{need}\, \underline{x}.\, \underline{x} \,\mathbf{to}\, y.\, N \ \equiv \ M \,\mathbf{to}\, y.\, N[\underline{x} \mapsto \langle y \rangle]$$

replacing it with

$$M\ \mathbf{need}\ \underline{x}.\ \underline{x}\ \mathbf{to}\ y.\ N\ \equiv\ M\ \mathbf{to}\ y.\ N[\underline{x} \mapsto \mathbf{coerce}_{1\leq\varepsilon}\ \langle y\rangle]$$

where ε is the effect of M. This works because the effect algebra is pointed, so we have $1 \leq \varepsilon$.

5.4.1 Exploiting effect-dependent equivalences

Our primary goal in adding an effect system to ECBPV is to exploit (local, effect-justified) equivalences between evaluation orders even without a whole-language restriction on effects. We sketch how to do this for our example.

When proving the equivalence between call-by-name and call-by-need in Section 5.3 we assumed that the only operation in the language was diverge. To relax this restriction, we use the effect algebra with preorder $\{\Omega \leq \top\}$ described above, and assign the effect Ω to diverge. We can include other effectful operations (e.g. raise, put), and give them the effect $\top$. The statement of the internal (object-level) equivalence becomes:

If $\Gamma \vdash M : \langle\Omega\rangle A$ and $\Gamma, \underline{x} : \langle\Omega\rangle A \vdash N : \underline{C}$ then

$$\Gamma \vdash M\ \mathbf{name}\ \underline{x}.\ N \cong_{\mathrm{ctx}} M\ \mathbf{need}\ \underline{x}.\ N : \underline{C}$$

The premise restricts the effect of M to Ω so that nontermination is its only possible side-effect, but N is allowed to have *any* effect.

To prove this equivalence, we need a logical relation for the effect system, which means we have to define a Kripke relation $\mathcal{R}[\![\langle\varepsilon\rangle A]\!]$ for each effect ε. For $\mathcal{R}[\![\langle\Omega\rangle A]\!]$ we use the same definition as before (the definition of $\mathcal{R}[\![\mathrm{F}\,A]\!]$). The definition of $\mathcal{R}[\![\langle\top\rangle A]\!]$ depends on the specific other effects included (we could adapt the free lifting in Section 3.1.1).

To state and prove a meta-level equivalence for a source language that includes other side-effects, we need to define a call-by-need effect system for the source language. This would just be the same as the Moggi-style call-by-name effect system in Section 2.4, which is sound for call-by-need for pointed effect algebras. The call-by-need translation of types is then the same as the call-by-name translation of types. Just as for the object-level equivalence, the statement of the meta-level equivalence requires the source-language expression to have the effect Ω. We omit the details here.

5.5 Related work

Reasoning about call-by-need The majority of work on reasoning about call-by-need source languages has concentrated on operational semantics based on environments [53], graphs [100, 96], and answers [4, 3, 17, 68]. However, these do not compare call-by-need with other evaluation orders. The only type-based analysis of a lazy source language we know of apart from McDermott and Mycroft's effect system [69] is Turner et al.'s [97] (and its extension [101]).

Polarized type theories (e.g. Zeilberger [102]) stratify types into several kinds (like values and computations in CBPV) to capture multiple evaluation orders. Downen and Ariola [20, 21] recently described how to capture call-by-need using polarity. They take a different approach to ours, by stratifying the syntax according to evaluation order, rather than whether terms might have effects. This means they have three kinds of type (call-by-value, call-by-need, and call-by-name), resulting in a more complex language than ours. In their language, information about evaluation order can be deduced from types, in ours it cannot (the call-by-need and call-by-name translations of types are identical). They also do not apply their language to reasoning about the differences between evaluation orders, which was the primary motivation for ECBPV. It is not clear whether their language can also be used for this purpose.

Multiple evaluation orders can also be captured in a Moggi-style language by using *joinads* instead of monads [83]. There may be some joinad structure implicit in extended call-by-push-value.

Logical relations Kripke logical relations have previously been applied to the problems of lambda definability [39] and normalization [2, 25]. Previous proofs of contextual equivalence relate only closed terms. We were forced to relate open terms because of the **need** construct.

Reasoning about effects using logical relations often runs into a difficulty in ensuring the relations are closed under sequencing of computations. We are able to work around this due to our specific choice of effects. It is possible that considering other effects would require a technique such as Lindley and Stark's *leapfrog method* [62, 61].

5.6 Summary

We have described extended call-by-push-value, a calculus that can be used for reasoning about several evaluation orders. In particular, ECBPV supports call-by-need via the addition of the construct M **need** $\underline{x}.\, N$. This allows us to prove that call-by-name and call-by-need reduction are equivalent if nontermination is the only effect in the source language, both inside the language itself, and on the meta-level. We proved the latter by giving a translation from our source language into ECBPV that captures call-by-need reduction. We also defined an effect system for ECBPV. The effect system statically bounds the side-effects of terms, allowing us to validate equivalences between evaluation orders without restricting the entire language to particular effects. We end this chapter with a description of some possible future work.

Other equivalences between evaluation orders We have proved one example of an equivalence between evaluation orders using ECBPV, but there are others that we might also expect to hold. For example, we would expect call-by-need and call-by-value to be equivalent if the effects are restricted to nondeterminism, allocating state, and reading from state (but not writing). It should be possible to use ECBPV to prove these by defining suitable logical relations. More generally, it might be possible to characterize when particular equivalences hold in terms of the algebraic properties of the effects we restrict to, and give a reasoning principle similar to the one we derived in Chapter 3.

Denotational semantics In Chapter 4 we emphasized the use of denotational semantics for proving the validity of program transformations. Developing an ECBPV denotational semantics would enable us to reason about ECBPV program transformations in this way. Composing the denotational semantics with the call-by-need translation would also result in a call-by-need denotational semantics for the source language.

Some potential approaches to describing the denotational semantics of ECBPV are Maraist et al.'s [67] translation into an affine calculus, combined with a semantics of linear logic [71], and also continuation-passing-style translations [82]. None of these consider side-effects however.

Chapter 6

Conclusions

This thesis develops a general framework for proving the validity of effect-dependent program transformations. We include an intermediate language in which such transformations can be stated (graded call-by-push-value, defined in Section 2.7), a way to use this intermediate language to reason about a source language (translations into GCBPV in Section 2.7.3), and machinery for proving the instances of contextual preorders (logical relations and order-enriched categorical semantics in Section 3.1 and Chapter 4).

Our framework supports various evaluation orders. The side-effects of programs depend on the evaluation order used. Hence, to discuss effect-dependent transformations for some evaluation order it is desirable to construct an effect system that is bespoke to that evaluation order. It is well-known that we can do this for call-by-value; Chapter 2, we show how to do this for two forms of call-by-name (Moggi-style call-by-name and Levy-style). We also describe GCBPV, which augments Levy's call-by-push-value with an effect system that generalises all of these. We can therefore use the framework to prove the validity of effect-dependent program transformations for source languages with any of these evaluation orders. This extends previous work, which primarily supports only call-by-value.

We then applied the framework to program transformations that change the evaluation order used inside programs. We proved two versions of a novel reasoning principle that relates call-by-value and call-by-name evaluation in the presence of side-effects: one syntactic (Chapter 3) and one semantic (Section 4.4). The reasoning principle arises because of the existence of certain Galois connections between call-by-value and call-by-name computations. We expect that our approach to relating evaluation orders will work more generally: we can relate two evaluation orders by translating into a single intermediate language and then defining similar Galois connections.

Chapter 4 deals with *noninvertible* program transformations. These have mostly been neglected in previous work, which tends to consider only symmetric transformations. We gave several examples, including for undefined behaviour, nondeterminism and concurrency, and showed how to use our framework to validate them. These justify our use of order-enrichment in the denotational semantics of GCBPV. We argue that order-enrichment is crucial for a *general* framework for formal reasoning about program transformations.

Finally, we added call-by-need to our framework. Call-by-need is more difficult to reason about than call-by-value or call-by-name because it involves *action at a distance*. In Chapter 5 we developed ECBPV, which extends CBPV with primitives that allow it to capture call-by-need evaluation. We then discussed how to use ECBPV to reason about call-by-need. In particular, we showed that call-by-name and call-by-need are equivalent when side-effects are limited to nontermination. It turns out that ordinary logical relations do not suffice to do this, and so we used Kripke logical relations of varying arity.

6.1 Future work

There are many possible directions for future work. Some are obvious extensions of the contributions of this thesis, others are open questions that should be solved to continue with the aims of this thesis. We highlight just a few here.

Relating evaluation orders We have given two relationships between evaluation orders: a general reasoning principle for call-by-value and Levy-style call-by-name in Chapter 3 and Section 4.4, and one instance of a relationship between call-by-need and Moggi-style call-by-name in Section 5.3. An obvious question is whether we can develop the latter into a more general reasoning principle that relies only on axiomatic properties of side-effects. It would also be interesting to formulate reasoning principles for other pairs of evaluation orders. In particular, it should be possible to give a reasoning principle that relates the two forms of call-by-name.

Other side-effects We have considered a small number of example side-effects, such as global state, nondeterminism, nontermination and undefined behaviour. There are many other examples of side-effects that it should be possible to apply our framework to, such as local state [88, 40] and probabilistic choice. Incorporating effect *handlers* [86] into our framework is another obvious next step.

Semantics of call-by-need Throughout Chapter 4 we emphasized the (well-known) advantages of denotational semantics for formal reasoning about program transformations, but when adding call-by-need in Chapter 5 we did not consider denotational semantics at all. Denotational semantics for call-by-need (in the presence of arbitrary side-effects) is still an open problem, and it is not clear what a model of ECBPV would look like. Our use of Kripke logical relations of varying arity suggests we might be able to model call-by-need in (**Poset**-)categories of the form $[\mathbf{Ctx}, \mathbf{C}]$, where **Ctx** is the category of typing contexts and weakenings, but we have not pursued this.

How should models be graded? In Chapter 4 we construct a number of graded adjunctions that we use as models of GCBPV. A question we might ask is *which* graded adjunctions are useful as models. For example, we might expect each of the subeffecting morphisms $(\varepsilon \leq \varepsilon') \circledast (-) : \varepsilon \circledast (-) \longrightarrow \varepsilon' \circledast (-)$ to be a full monomorphism (Definition 4.2.15), which corresponds to $\mathbf{coerce}_{\varepsilon \leq \varepsilon'} \, M \preccurlyeq \mathbf{coerce}_{\varepsilon \leq \varepsilon'} \, N \;\Rightarrow\; M \preccurlyeq N$ in the syntax. (This is the case for all of our examples.) There are graded adjunctions that do not have this property, but it is not clear whether any of them are useful for models. Are there any other properties we would expect to hold for all side-effects? Constructing graded adjunctions also takes a lot of work, so it would be useful to find a general technique for doing this. Kammar and McDermott [41] consider such a technique for constructing graded monads for Gifford-style effect algebras.

Intermediate languages for program reasoning We have presented GCBPV as an intermediate language that is useful for formal reasoning about programs. In particular, we chose to base our intermediate language on CBPV partly because we can easily verify the correctness of program transformations in CBPV. However, we do not claim that GCBPV is the best possible intermediate language. More work is needed to settle the question of what the best language is. If we are proving the validity of compiler optimizations then we might want to find an intermediate language that can be used both for formal reasoning and for applying optimizations inside a compiler. There may be a tension between these two applications.

Practical effect systems There are very few instances of effect systems being used in practice, and this thesis does not consider *practical* use of grading. One way to use effect systems as a way to do effect-dependent optimizations inside a compiler would be to have the compiler infer the effects of terms (rather than exposing the effect system to the programmer). This would require effect inference, and to do inference we would probably want to include effect polymorphism. Extending GCBPV (and related machinery, such as the denotational semantics), with effect polymorphism, and then describing how to do effect inference on it, would be interesting future work.

6.2 Final remarks

There are several key aspects of the approach used throughout this thesis that are worth highlighting here.

The first is our use of a common *intermediate* language for stating and proving the validity of program transformations, inspired by compiler design. Intermediate languages are useful because they allow us to prove *general* results about program transformations. When relating call-by-value and call-by-name in Chapter 3 and Section 4.4 we obtained a reasoning principle that is not specific to the choice of side-effects in the language. This was aided partly by our use of inequational theories for specifying the behaviour of side-effects (rather than e.g. operational semantics). The structure of the translations into the intermediate language guided us in doing this. Our results about the relationship between call-by-value and call-by-name, and between call-by-name and call-by-need (Section 5.3) are also about open terms. We can relate open terms because we use intermediate languages that capture all of these evaluation orders.

The intermediate languages approach also allows us to ignore considerations about the design of source languages. We chose to base our intermediate languages on CBPV. Whether CBPV is good for writing programs is irrelevant, as long as there are suitable translations from source languages. On the other hand, the design of CBPV makes it is easy to reason about. Crucially, there are no choices to make about evaluation order in CBPV. We therefore get a separation of concerns: the translations from the source language specify the evaluation order but are independent of the choice of side-effects, and the semantics of the intermediate language specifies the behaviour of the side-effects, but is independent of the evaluation order. This was particularly helpful in Chapter 4, where we barely had to consider evaluation orders.

Finally, the intermediate language can include features that may not be useful in source languages. Call-by-name might be an example of this: even if we do not want to use call-by-name in an effectful *source* language, we might still want to replace call-by-value with call-by-name as an optimization inside the intermediate language.

Another important aspect of our approach is the use of *grading* to specify restrictions on side-effects. This is crucial for considering *effect-dependent* program transformations without restricting the side-effects that can appear in the language itself. It allows us to consider languages with a variety of effects (I/O, mutable state, nontermination, nondeterminism, probabilistic choice, etc.) even though most transformations are only correct for subterms that do not use most of them. Another advantage of grading is that we can use it to obtain *stable* semantics [16], where side-effects can be added without changing the denotations of existing programs.

We also emphasize the use of axiomatic properties of side-effects (such as *thunkable* effects in Chapter 3), which we used to give general reasoning principles for program transformations.

These aspects of our approach, especially our use of intermediate languages, may turn out to be helpful more widely in program semantics.

Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 147–160. https://doi.org/10.1145/292540.292555

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. *Lecture Notes in Computer Science* 953, 182–199. https://doi.org/10.1007/3-540-60164-3_27

[3] Zena M. Ariola and Matthias Felleisen. 1997. The call-by-need lambda calculus. *Journal of functional programming* 7, 3, 265–301.

[4] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 233–246. http://doi.acm.org/10.1145/199448.199507

[5] Nick Benton and Peter Buchlovsky. 2007. Semantics of an Effect Analysis for Exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. ACM, 15–26. https://doi.org/10.1145/1190315.1190320

[6] Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent Transformations for Concurrent Programs. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. ACM, 188–201. https://doi.org/10.1145/2967973.2968602

[7] Nick Benton, John Hughes, and Eugenio Moggi. 2002. Monads and Effects. In *Applied Semantics*. Springer, 42–122. http://dl.acm.org/citation.cfm?id=647424.725798

[8] Nick Benton and Andrew Kennedy. 1999. Monads, Effects and Transformations. *Electronic Notes in Theoretical Computer Science* 26, 3–20. https://doi.org/10.1016/S1571-0661(05)80280-4

[9] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational Semantics for Effect-based Program Transformations: Higher-order Store. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. ACM, 301–312. https://doi.org/10.1145/1599410.1599447

[10] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations: Towards Extensional Semantics for Effect Analyses. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems*. Springer, 114–130. https://doi.org/10.1007/11924661_7

[11] Nick Benton, Andrew Kennedy, Martin Hofmann, and Vivek Nigam. 2016. Counting Successes: Effects and Transformations for Non-deterministic Programs. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer, 56–72. `https://doi.org/10.1007/978-3-319-30936-1_3`

[12] Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM, 129–140. `https://doi.org/10.1145/289423.289435`

[13] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *26th International Workshop/21st Annual Conference of the EACSL (Leibniz International Proceedings in Informatics)*, Patrick Cégielski and Arnaud Durand (Eds.), Vol. 16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 107–121. `https://doi.org/10.4230/LIPIcs.CSL.2012.107`

[14] Anna Bucalo, Carsten Führmann, and Alex Simpson. 2003. An equational notion of lifting monad. *Theoretical Computer Science* 294, 1, 31–60. `https://doi.org/10.1016/S0304-3975(01)00243-2`

[15] Aurelio Carboni, Stephen Lack, and R.F.C. Walters. 1993. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra* 84, 2, 145–158. `https://doi.org/10.1016/0022-4049(93)90035-R`

[16] Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*. Springer, 244–272. `http://dl.acm.org/citation.cfm?id=645868.668496`

[17] Stephen Chang and Matthias Felleisen. 2012. The call-by-need lambda calculus, revisited. In *Proceedings of the 21st European Conference on Programming Languages and Systems*. Springer, 128–147. `http://dx.doi.org/10.1007/978-3-642-28869-2_7`

[18] Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM, 233–243. `https://doi.org/10.1145/351240.351262`

[19] Marco Devesas Campos and Paul Blain Levy. 2018. A Syntactic View of Computational Adequacy. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 71–87.

[20] Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic*. 21:1–21:23. `https://doi.org/10.4230/LIPIcs.CSL.2018.21`

[21] Paul Downen and Zena M. Ariola. 2019. Compiling With Classical Connectives. *arXiv e-prints*, Article 1907.13227.

[22] Anatolij Dvurečenskij and Sylvia Pulmannová. 2000. *New trends in quantum structures*. Mathematics and its Applications, Vol. 516. Kluwer Academic Publishers.

[23] Andrzej Filinski. 1989. *Declarative Continuations and Categorical Duality*. Master's thesis. University of Copenhagen.

[24] Andrzej Filinski. 1996. *Controlling Effects*. Ph.D. Dissertation. Carnegie Mellon University.

[25] Marcelo Fiore. 2002. Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM, 26–37. https://doi.org/10.1145/571157.571161

[26] Marcelo Pablo Fiore. 1996. *Axiomatic Domain Theory in Categories of Partial Maps*. Cambridge University Press.

[27] Carsten Führmann. 1999. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science* 20, 245–292.

[28] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer, 513–530.

[29] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, 476–489. https://doi.org/10.1145/2951913.2951939

[30] Sergey Goncharov and Lutz Schröder. 2013. A Relatively Complete Generic Hoare Logic for Order-Enriched Effects. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 273–282. https://doi.org/10.1109/LICS.2013.33

[31] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 13:1–13:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.13

[32] Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. 2002. Logical Relations for Monadic Types. In *Computer Science Logic*, Julian Bradfield (Ed.). Springer, 553–568.

[33] Jennifer Hackett and Graham Hutton. 2019. Call-by-need is Clairvoyant Call-by-value. *Proceedings of the ACM Programming Languages* 3, Article 114, 23 pages. https://doi.org/10.1145/3341718

[34] John Hatcliff and Olivier Danvy. 1997. Thunks and the λ-calculus. *Journal of Functional Programming* 7, 3, 303–319. https://doi.org/10.1017/S0956796897002748

[35] John Mark Hatcliff. 1995. *The structure of continuation-passing styles*. Ph.D. Dissertation. Kansas State University.

[36] Claudio Hermida. 1993. *Fibrations, Logical Predicates and Indeterminates*. Ph.D. Dissertation. University of Edinburgh.

[37] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Proceedings of the 32nd Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1–12. https://doi.org/10.1109/LICS.2017.8005137

[38] C.A.R. Hoare and He Jifeng. 1998. *Unifying Theories of Programming*. Prentice Hall.

[39] Achim Jung and Jerzy Tiuryn. 1993. A New Characterization of Lambda Definability. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Springer, 245–257. http://dl.acm.org/citation.cfm?id=645891.671429

[40] Ohad Kammar, Paul B. Levy, Sean K. Moss, and Sam Staton. 2017. A monad for full ground reference cells. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Press, 49:1–49:12. http://dl.acm.org/citation.cfm?id=3329995.3330044

[41] Ohad Kammar and Dylan McDermott. 2018. Factorisation Systems for Logical Relations and Monadic Lifting in Type-and-effect System Semantics. *Electronic Notes in Theoretical Computer Science* 341, 239 – 260. https://doi.org/10.1016/j.entcs.2018.11.012 Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV).

[42] Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic Foundations for Effect-dependent Optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 349–360. https://doi.org/10.1145/2103656.2103698

[43] Shin-ya Katsumata. 2005. A Semantic Formulation of ⊤⊤-lifting and Logical Predicates for Computational Metalanguage. In *Proceedings of the 19th International Conference on Computer Science Logic*. Springer, 87–102. https://doi.org/10.1007/11538363_8

[44] Shin-ya Katsumata. 2013. Relating Computational Effects by ⊤⊤-lifting. *Inf. Comput.* 222, 228–246. https://doi.org/10.1016/j.ic.2012.10.014

[45] Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 633–645. https://doi.org/10.1145/2535838.2535846

[46] Shin-ya Katsumata and Tetsuya Sato. 2015. Codensity Liftings of Monads. In *6th Conference on Algebra and Coalgebra in Computer Science*, Lawrence S. Moss and Pawel Sobocinski (Eds.), Vol. 35. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 156–170. https://doi.org/10.4230/LIPIcs.CALCO.2015.156

[47] G.M. Kelly and A.J. Power. 1993. Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. *Journal of Pure and Applied Algebra* 89, 1, 163 – 179. https://doi.org/10.1016/0022-4049(93)90092-8

[48] Max Kelly. 1982. *Basic concepts of enriched category theory*. Cambridge University Press.

[49] Anders Kock. 1971. Bilinearity and cartesian closed monads. *Math. Scand.* 29, 2, 161–174. https://doi.org/10.7146/math.scand.a-11042

[50] Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 113–120.

[51] Anders Kock. 1995. Monads for which structures are adjoint to units. *Journal of Pure and Applied Algebra* 104, 1, 41–59.

[52] Jakov Kučan. 1998. Retraction Approach to CPS Transform. *Higher-Order and Symbolic Computation* 11, 2, 145–175. https://doi.org/10.1023/A:1010012532463

[53] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 144–154. https://doi.org/10.1145/158511.158618

[54] Julia L. Lawall and Olivier Danvy. 1993. Separating Stages in the Continuation-passing Style Transformation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 124–136. https://doi.org/10.1145/158511.158613

[55] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 633–647. https://doi.org/10.1145/3062341.3062343

[56] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer, 228–243. https://doi.org/10.1007/3-540-48959-2_17

[57] Paul Blain Levy. 2001. *Call-by-push-value*. Ph.D. Dissertation. Queen Mary, University of London, UK.

[58] Paul Blain Levy. 2003. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science* 69, 248–271. https://doi.org/10.1016/S1571-0661(04)80568-1 CTCS'02, Category Theory and Computer Science.

[59] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19, 4, 377–414. https://doi.org/10.1007/s10990-006-0480-6

[60] Paul Blain Levy. 2017. Contextual Isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 400–414. https://doi.org/10.1145/3009837.3009898

[61] Sam Lindley. 2005. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. Ph.D. Dissertation. University of Edinburgh, UK.

[62] Sam Lindley and Ian Stark. 2005. Reducibility and ⊤⊤-lifting for Computation Types. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*. Springer, 262–277. https://doi.org/10.1007/11417170_20

[63] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 47–57. https://doi.org/10.1145/73560.73564

[64] Christoph Lüth and Neil Ghani. 1997. Monads and modular term rewriting. In *Category Theory and Computer Science*, Eugenio Moggi and Giuseppe Rosolini (Eds.). Springer, 69–86.

[65] QingMing Ma and John C. Reynolds. 1992. Types, abstraction, and parametric polymorphism, part 2. In *Mathematical Foundations of Programming Semantics*, Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt (Eds.). Springer, 1–40.

[66] Saunders Mac Lane. 1998. *Categories for the working mathematician* (second ed.). Springer.

[67] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-Name, Call-by-Value, Call-by-Need, and the Linear Lambda Calculus, In Proceedings of the Eleventh Annual Mathematical Foundations of Programming Semantics Conference. *Electronic Notes in Theoretical Computer Science*, 370–392.

[68] John Maraist, Martin Odersky, and Philip Wadler. 1998. The call-by-need lambda calculus. *Journal of Functional Programming* 8, 3, 275–317. https://doi.org/10.1017/S0956796898003037

[69] Dylan McDermott and Alan Mycroft. 2018. Call-by-need effects via coeffects. *Open Computer Science* 8, 93–108. https://doi.org/10.1515/comp-2018-0009

[70] Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer, 235–262.

[71] Paul-André Melliès. 2009. Categorical semantics of linear logic. In *Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27, Société Mathématique de France.*

[72] Paul-André Melliès. 2010. Segal Condition Meets Computational Effects. In *25th Annual IEEE Symposium on Logic in Computer Science*. 150–159. https://doi.org/10.1109/LICS.2010.46

[73] A Melton, D A Schmidt, and G E Strecker. 1986. Galois Connections and Computer Science Applications. In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*. Springer, 299–312. http://dl.acm.org/citation.cfm?id=20081.20099

[74] José Meseguer. 1980. Varieties of chain-complete algebras. *Journal of Pure and Applied Algebra* 19, 347–383. https://doi.org/10.1016/0022-4049(80)90106-1

[75] Albert R. Meyer and Mitchell Wand. 1985. Continuation semantics in typed lambda-calculi. In *Logics of Programs*, Rohit Parikh (Ed.). Springer, 219–224.

[76] Stefan Milius, Dirk Pattinson, and Lutz Schröder. 2015. Generic Trace Semantics and Graded Monads. In *6th Conference on Algebra and Coalgebra in Computer Science*, Lawrence S. Moss and Pawel Sobocinski (Eds.), Vol. 35. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 253–269. https://doi.org/10.4230/LIPIcs.CALCO.2015.253

[77] Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, 14–23. http://dl.acm.org/citation.cfm?id=77350.77353

[78] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1, 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[79] Guillaume Munch-Maccagnoni. 2014. Models of a Non-associative Composition. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). Springer, 396–410.

[80] Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2016. Effect Systems Revisited—Control-Flow Algebra and Semantics. In *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. Springer, 1–32. https://doi.org/10.1007/978-3-319-27810-0_1

[81] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances*. Springer, 114–136. http://dl.acm.org/citation.cfm?id=646005.673740

[82] Chris Okasaki, Peter Lee, and David Tarditi. 1994. Call-by-need and continuation-passing style. *LISP and Symbolic Computation* 7, 57–81. https://doi.org/10.1007/BF01019945

[83] Tomas Petricek and Don Syme. 2011. Joinads: A Retargetable Control-flow Construct for Reactive, Parallel and Concurrent Programming. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*. Springer, 205–219. http://dl.acm.org/citation.cfm?id=1946313.1946336

[84] G.D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2, 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[85] Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 69–94. https://doi.org/10.1023/A:1023064908962

[86] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software*. Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[87] Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*. Springer, 1–24. http://dl.acm.org/citation.cfm?id=646793.704708

[88] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*. Springer, 342–356. http://dl.acm.org/citation.cfm?id=646794.704856

[89] John C. Reynolds. 1974. On the Relation Between Direct and Continuation Semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*. Springer, 141–156. http://dl.acm.org/citation.cfm?id=646230.681878

[90] John C. Reynolds. 1997. The Essence of Algol. In *Algol-like Languages*, Peter W. O'Hearn and Robert D. Tennent (Eds.). Birkhäuser Boston, 67–88. https://doi.org/10.1007/978-1-4612-4118-8_4

[91] Amr Sabry. 1998. What is a Purely Functional Language? *Journal of Functional Programming* 8, 1, 1–22. https://doi.org/10.1017/S0956796897002943

[92] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. ACM, 288–298. https://doi.org/10.1145/141471.141563

[93] Amr Sabry and Philip Wadler. 1996. A Reflection on Call-by-value. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ACM, 13–24. `https://doi.org/10.1145/232627.232631`

[94] Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 15–26. `https://doi.org/10.1145/2429069.2429074`

[95] Andrew Tolmach. 1998. Optimizing ML using a hierarchy of monadic types. In *Types in Compilation*, Xavier Leroy and Atsushi Ohori (Eds.). Springer, 97–115.

[96] D. A. Turner. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience* 9, 1, 31–49. `https://doi.org/10.1002/spe.4380090105`

[97] David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once Upon a Type. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. ACM, 1–11. `http://doi.acm.org/10.1145/224164.224168`

[98] Philip Wadler. 1998. The Marriage of Effects and Monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM, 63–74. `https://doi.org/10.1145/289423.289429`

[99] Philip Wadler. 2003. Call-by-value is Dual to Call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM, 189–201. `https://doi.org/10.1145/944705.944723`

[100] C.P. Wadsworth. 1971. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford.

[101] Keith Wansbrough and Simon Peyton Jones. 1999. Once Upon a Polymorphic Type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 15–28. `http://doi.acm.org/10.1145/292540.292545`

[102] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-matching*. Ph.D. Dissertation. Carnegie Mellon University.

Appendix A

Order-enriched category theory

We give a brief introduction to the concepts of order-enriched (specifically **Poset**-enriched) category theory that we use for the denotational semantics of GCBPV in Chapter 4. A more comprehensive introduction to enriched category theory in general is given by Kelly [48] (though note that restricting to **Poset**-enrichment simplifies the definitions significantly). We assume some basic (ordinary) category theory (e.g. Mac Lane [66]).

A **Poset**-category is like an ordinary category, except that morphisms form partially ordered sets and composition of morphisms is monotone.

Definition A.0.1 (Poset-category) A **Poset**-*category* $\mathbf{C}$ consists of a collection of objects and, for each pair of objects X, Y, a poset $\mathbf{C}(X, Y)$ of morphisms, together with:

- For each triple of objects X, Y, Z, a composition function $\circ : \mathbf{C}(Y, Z) \times \mathbf{C}(X, Y) \to \mathbf{C}(X, Z)$. Composition is required to be monotone (in both arguments) and associative.
- For each object X, an identity morphism $id_X \in \mathbf{C}(X, X)$ that is the unit for composition.

◀

We identify each poset (including objects of **Poset** below) with its underlying set, and always use the symbol $\sqsubseteq$ for the order on morphisms in a **Poset**-category. Every **Poset**-category has an underlying ordinary category, by forgetting the order on morphisms. We give several examples, all of which are useful for constructing models of GCBPV.

Example A.0.2 Every locally small category (in particular, the category **Set** of sets and functions) forms a **Poset**-category by using equality for each of the partial orders. In these cases, all of the **Poset**-enriched definitions coincide with the ordinary ones. ◀

Example A.0.3 The prototypical example of a **Poset**-category is **Poset** itself, which has partially ordered sets $(X, \sqsubseteq)$ as objects and monotone functions as morphisms. Morphisms are ordered pointwise (i.e. $f \sqsubseteq g$ if $f\,x \sqsubseteq g\,x$ for all x). When we use **Poset** as a **Poset**-category, this is the order on morphisms we use. ◀

Example A.0.4 We also restrict **Poset** to a smaller category. A poset X is *pointed* if it has a least element $\bot$, and a monotone function $f : X \to Y$ between pointed posets is *strict* if $f\,\bot = \bot$. The **Poset**-category $\mathbf{Poset}_\bot$ is formed by restricting **Poset** to pointed posets and strict monotone functions. ◀

Example A.0.5 A poset $(X, \sqsubseteq)$ is an *ω-complete partial order* if every ω-chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ has a least upper bound $\bigsqcup x$, and a monotone function between two ωcpos is *continuous* if it preserves least upper bounds of ω-chains. The **Poset**-category $\omega\mathbf{Cpo}$ has ωcpos as objects and continuous functions as morphisms. The order on morphisms is pointwise. ◀

As these examples show, each ordinary category may have more than one possible order-enrichment. When using **Poset**-categories as models of GCBPV, the order on morphisms must be chosen based on the side-effects being modelled. The order on morphisms is the only extra structure required in **Poset**-category theory compared to ordinary category theory. The remaining definitions are the same as the ordinary ones, except that wherever there is a function on morphisms it is required to be monotone.

To model various type formers, we require **Poset**-categories $\mathbf{C}$ that come with some given extra structure. A *terminal object* 1 of a **Poset**-category is just a terminal object in the underlying ordinary category. We write $\langle\rangle_X : X \to 1$ for the universal morphisms. Binary *products* are also the same as in the underlying category, except that the pairing operations $\langle -, - \rangle$ on morphisms are required to be monotone (if $f_1 \sqsubseteq f_1' : X \to Y_1$ and $f_2 \sqsubseteq f_2' : X \to Y_2$ then $\langle f_1, f_2 \rangle \sqsubseteq \langle f_1', f_2' \rangle : X \to Y_1 \times Y_2$). For each X, Y, Z, there is a canonical associativity morphism $\mathit{assoc} : (X \times Y) \times Z \to X \times (Y \times Z)$. Similarly, binary *coproducts* (with copairing morphisms $[f_1, f_2] : X_1 + X_2 \to Y$) are the same as in the underlying ordinary category, except that the copairing operations are required to be monotone. *Exponentials* in **Poset**-categories are just exponentials in the underlying category, except that the currying operations $\Lambda : \mathbf{C}(Z \times X, Y) \to \mathbf{C}(Z, X \Rightarrow Y)$ are required to be monotone. This implies that uncurrying $\Lambda^{-1} : \mathbf{C}(Z, X \Rightarrow Y) \to \mathbf{C}(Z \times X, Y)$ is also monotone.

We say that a **Poset**-category $\mathbf{C}$ is *cartesian* if it has chosen finite products (terminal object and binary products). It is *bicartesian* if it is cartesian and has chosen finite coproducts (initial object and binary coproducts). It is *distributive* if it is bicartesian and, for all $X, Y, Z \in \mathbf{C}$, the morphism

$$(X \times Y) + (X \times Z) \xrightarrow{[\langle \pi_1, \mathit{inl} \circ \pi_2 \rangle, \langle \pi_1, \mathit{inr} \circ \pi_2 \rangle]} X \times (Y + Z)$$

has an inverse $\mathit{dist}_{X,Y,Z} : X \times (Y + Z) \to (X \times Y) + (X \times Z)$. A **Poset**-category $\mathbf{C}$ is *(bi)cartesian closed* if it is (bi)cartesian and has chosen exponentials. Every bicartesian closed **Poset**-category is distributive.

Example A.0.6 The **Poset**-categories **Set**, **Poset** and ω**Cpo** are bicartesian closed. For **Set**, the terminal object is the singleton $\{\star\}$, and the binary product $X_1 \times X_2$ is the usual cartesian product. The initial object is the empty set, and the coproduct $X_1 + X_2$ is the disjoint union. For **Poset** and ω**Cpo** each of these is the same as in **Set**, with the obvious orderings. In **Set** exponentials are sets of functions, in **Poset** they are sets of monotone functions, ordered pointwise, and in ω**Cpo** they are sets of continuous functions, ordered pointwise.

The **Poset**-category $\mathbf{Poset}_\perp$ is cartesian; finite products are the same as in **Poset**. (It is also bicartesian, but not distributive or closed.) ◀

For models of GCBPV we also need **Poset**-*adjunctions*. To define these we first consider **Poset**-enrichment of functors. This is property of functors, not some additional structure that must be chosen: each ordinary functor either does or does not **Poset**-enrich.

Definition A.0.7 (Poset-functor) If $\mathbf{C}$ and $\mathbf{D}$ are **Poset**-categories, a **Poset**-*functor* $F : \mathbf{C} \to \mathbf{D}$ is a functor between the underlying ordinary categories, such that each function $F : \mathbf{C}(X, Y) \to \mathbf{D}(FX, FY)$ is monotone. *Natural transformations* between **Poset**-functors are just natural transformations in the ordinary sense. ◀

The identity $\mathit{Id}_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ is a **Poset**-functor for every **Poset**-category $\mathbf{C}$, and the composition $G \circ F$ of two **Poset**-functors is also a **Poset**-functor.

A **Poset**-adjunction is then the same as an ordinary adjunction, except that the two functors are required to **Poset**-enrich.[1]

[1] Throughout we use the name ϖ for counits of adjunctions, reserving ε for effects.

Definition A.0.8 (Poset-adjunction) Suppose that **C** and **D** are **Poset**-categories. A **Poset**-*adjunction* $F \dashv U$ consists of two **Poset**-functors: the *left adjoint* $F : \mathbf{C} \longrightarrow \mathbf{D}$ and the *right adjoint* $U : \mathbf{D} \longrightarrow \mathbf{C}$, together with two natural transformations

$$\eta : Id_{\mathbf{C}} \longrightarrow U \circ F \qquad \varpi : F \circ U \longrightarrow Id_{\mathbf{D}}$$

such that the two triangle laws $U\varpi \circ \eta_U = id_U$ and $\varpi_F \circ F\eta = id_F$ hold. ◀

Example A.0.9 There is a **Poset**-adjunction between posets and pointed posets:

$$\begin{tikzcd}
\mathbf{Poset} \arrow[r, bend left, "(-)_\perp"] \arrow[r, phantom, "\perp"] & \mathbf{Poset}_\perp \arrow[l, bend left, "U"]
\end{tikzcd}$$

The left adjoint $(-)_\perp$ sends each poset X to the pointed poset $X_\perp$ formed by freely adding a least element, and each monotone function f to its strict extension $f_\perp$ (so $f_\perp \perp = \perp$ and $f_\perp x = f\,x$ for $x \neq \perp$). The right adjoint is the forgetful functor, which sends each pointed poset to itself, and each strict monotone function to itself. The unit $\eta_X : X \longrightarrow X_\perp$ is the monotone function that maps $x \in X$ to itself, and the counit $\varpi_X : X_\perp \longrightarrow X$ is the strict monotone function that merges the least elements. ◀

As in the ordinary case, there is an equivalent definition of **Poset**-adjunction in terms of bijections between hom-posets:

Definition A.0.10 (Poset-adjunction (alternative definition)) Suppose that **C** and **D** are **Poset**-categories. A **Poset**-*adjunction* $F \dashv U$ consists of two **Poset**-functors: the *left adjoint* $F : \mathbf{C} \longrightarrow \mathbf{D}$ and the *right adjoint* $U : \mathbf{D} \longrightarrow \mathbf{C}$, together with a family of bijections

$$\theta : \mathbf{C}(FX, Y) \longrightarrow \mathbf{D}(X, UY)$$

natural in $X \in \mathbf{C}$ and $Y \in \mathbf{D}$. ◀

This definition does not explicitly require the functions θ (or their inverses) to be monotone, but this follows from naturality: given $f \sqsubseteq f' : FX \longrightarrow Y$ we have

$$\theta f \;=\; Uf \circ \theta\, id_X \;\sqsubseteq\; Uf' \circ \theta\, id_X \;=\; \theta f'$$

We also need *tensorial strengths* [50] for denotational semantics. The following definition is exactly the same as the ordinary one, except we additionally require the functor to enrich.

Definition A.0.11 (Strong Poset-functor) Suppose that **C** is a cartesian **Poset**-category. A *strong* **Poset**-*functor* (F, str^F) on **C** consists of a **Poset**-functor $F : \mathbf{C} \longrightarrow \mathbf{C}$ and a natural transformation $str^F_{X,Y} : X \times FY \longrightarrow F(X \times Y)$ called the *(tensorial) strength*, such that the following diagrams commute:

$$\begin{tikzcd}
1 \times FX \arrow[r, "str^F_{1,X}"] \arrow[dr, "\pi_2"'] & F(1 \times X) \arrow[d, "F\pi_2"] \\
& FX
\end{tikzcd}
\qquad
\begin{tikzcd}
(X \times Y) \times FZ \arrow[rr, "str^F_{X\times Y,Z}"] \arrow[d, "assoc"'] & & F((X \times Y) \times Z) \arrow[d, "Fassoc"] \\
X \times (Y \times FZ) \arrow[r, "X\times str^F_{Y,Z}"'] & X \times F(Y \times Z) \arrow[r, "str^F_{X,Y\times Z}"'] & F(X \times (Y \times Z))
\end{tikzcd}$$

A *strong natural transformation* $\alpha : (F, str^F) \longrightarrow (G, str^G)$ between strong **Poset**-functors is a natural transformation $\alpha : F \longrightarrow G$ such that

$$\begin{tikzcd}
X \times FY \arrow[r, "str^F_{X,Y}"] \arrow[d, "X\times\alpha_Y"'] & F(X \times Y) \arrow[d, "\alpha_{X\times Y}"] \\
X \times GY \arrow[r, "str^G_{X,Y}"'] & G(X \times Y)
\end{tikzcd}$$

commutes. ◀

Examples include the identity strong **Poset**-functor $(Id_{\mathbf{C}}, id)$ on **C** (which we sometimes write as $Id_{\mathbf{C}}$), and the composition $(G, str^G) \circ (F, str^F) := (G \circ F,\ G\, str^F \circ str^G)$ of strong **Poset**-functors. Strong functors are required in models of programming languages when we interpret open terms. For example, models of the monadic metalanguage [78] use strong monads. We can use the definitions above to give a succinct definition of order-enriched strong monads:

Definition A.0.12 (Strong Poset-monad) Suppose that **C** is a cartesian **Poset**-category. A *strong* **Poset**-*monad* on **C** consists of a strong **Poset**-functor (T, str) and two strong natural transformations

$$\eta : Id_{\mathbf{C}} \longrightarrow (T, str) \qquad \mu : (T, str) \circ (T, str) \longrightarrow (T, str)$$

such that the monad laws hold:

$$\begin{tikzcd}
& T \arrow[dl, bend right, "\eta_T"'] \arrow[d, equal] \arrow[dr, bend left, "T\eta"] & \\
T \circ T \arrow[r, "\mu"'] & T & T \circ T \arrow[l, "\mu"]
\end{tikzcd}
\qquad
\begin{tikzcd}
T \circ T \circ T \arrow[r, "\mu_T"] \arrow[d, "T\mu"'] & T \circ T \arrow[d, "\mu"] \\
T \circ T \arrow[r, "\mu"'] & T
\end{tikzcd}$$

◀

Appendix B

Additional proofs

B.1 Erasing coercions in GCBPV terms

Recall the following lemma about graded call-by-push-value.

Lemma 2.7.5 Suppose that the effect algebra $(\mathcal{E}, \leq, \cdot, 1)$ is a partially ordered monoid with bounded binary joins.

1. Given two value terms V_1, V_2 such that $\Gamma \vdash V_1 : A$ and $\Gamma \vdash V_2 : A$, if $\lfloor V_1 \rfloor = \lfloor V_2 \rfloor$ then $\Gamma \vdash V_1 \equiv V_2 : A$.
2. Given two computation terms M_1, M_2 such that $\Gamma \mathrel{\underline{\vdash}} M_1 : \underline{C}$ and $\Gamma \mathrel{\underline{\vdash}} M_2 : \underline{C}$, if $\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$ then $\Gamma \mathrel{\underline{\vdash}} M_1 \equiv M_2 : \underline{C}$. ◀

We give the proof of this lemma in this section.

B.1.1 Conjectured counterexample

Before giving the proof, we note that Lemma 2.7.5 does not appear to hold when the condition that the effect algebra has bounded binary joins is dropped. Consider the partial order $(\mathcal{E}, \leq)$ with Hasse diagram

$$\begin{tikzcd}
 & b_1 & b_2 & \\
1 & & & c \\
 & a_1 \arrow[uu] \arrow[uur] & a_2 \arrow[uu] \arrow[uul] &
\end{tikzcd}$$

This does not have bounded binary joins: the pair (a_1, a_2) has two upper bounds (b_1 and b_2), but neither is less than the other. We define the multiplication so that 1 is the unit and $\varepsilon \cdot \varepsilon' = c$ for $\varepsilon, \varepsilon' \neq 1$. Using this as an effect algebra, we define a typing context Γ and two computation terms M_1 and M_2 such that $\Gamma \mathrel{\underline{\vdash}} M_i : \langle c \rangle \mathbf{unit}$ for $i \in \{1, 2\}$:

$$\begin{aligned}
\Gamma &:= x : \langle c \rangle (\mathbf{unit} + \mathbf{unit}),\ y_1 : \langle a_1 \rangle \mathbf{unit},\ y_2 : \langle a_1 \rangle \mathbf{unit} \\
M_i &:= y \ \mathbf{to}\ z.\, \mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\, \_.\, \mathbf{coerce}_{a_1 \leq b_i}\, y_1,\ \mathbf{inr}\, \_.\, \mathbf{coerce}_{a_2 \leq b_i}\, y_2\}
\end{aligned}$$

The terms M_i are identical apart from the coercions ($\lfloor M_1 \rfloor = \lfloor M_2 \rfloor$), but we conjecture that $\Gamma \mathrel{\underline{\vdash}} M_1 \not\equiv M_2 : \langle c \rangle \mathbf{unit}$. The situation would be different if a_1 and a_2 had a join $a_1 \vee a_2$: we would be able to reason as follows:

$$\begin{aligned}
M_i &\equiv y\ \mathbf{to}\ z.\, \mathbf{coerce}_{a_1 \vee a_2 \leq b_1}\, \mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\, \_.\, \mathbf{coerce}_{a_1 \leq a_1 \vee a_2}\, y_1,\ \mathbf{inr}\, \_.\, \mathbf{coerce}_{a_2 \leq a_1 \vee a_2}\, y_2\} \\
&\equiv y\ \mathbf{to}\ z.\, \mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\, \_.\, \mathbf{coerce}_{a_1 \leq a_1 \vee a_2}\, y_1,\ \mathbf{inr}\, \_.\, \mathbf{coerce}_{a_2 \leq a_1 \vee a_2}\, y_2\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{to}\ \text{commutes with}\ \mathbf{coerce})
\end{aligned}$$

and hence $M_1 \equiv M_2$. We use similar reasoning in the proof of Lemma 2.7.5.

B.1.2 Proof of Lemma 2.7.5

To prove Lemma 2.7.5, we first define a stronger notion of subtyping for GCBPV in which contravariant subtyping is not allowed. The judgments $A <:' B$ and $\underline{C} <:' \underline{D}$ are given by the rules on the left of Figure 2.12 (which generate the usual notion of subtyping), except that the rule for function types is replaced with

$$\frac{\underline{C} <:' \underline{D}}{(A \to \underline{C}) <:' (A \to \underline{D})}$$

Clearly $A <:' B$ implies $A <: B$, and we have coercions $\mathbf{coerce}_{A<:B}$ from type A to type B if $A <:' B$ holds, as in Figure 2.12.

We have partial binary operations $\vee$ on value and computation types. When it is defined $A \vee B$ is the join of the value types A and B with respect to $<:'$, and similarly for computation types. In the definition, $\varepsilon \vee \varepsilon'$ denotes the join of the effects $\varepsilon, \varepsilon'$ if it exists. The cases are not exhaustive; the join is undefined if no case applies. The left-hand side of a case is undefined if any part of the right-hand side is.

$$b \vee b \ := \ b \qquad \mathbf{unit} \vee \mathbf{unit} \ := \ \mathbf{unit} \qquad (A_1 \times A_2) \ \vee \ (B_1 \times B_2) \ := \ (A_1 \vee B_1) \times (A_2 \vee B_2)$$
$$\mathbf{empty} \vee \mathbf{empty} \ := \ \mathbf{empty} \qquad (A_1 + A_2) \ \vee \ (B_1 + B_2) \ := \ (A_1 \vee B_1) + (A_2 \vee B_2)$$
$$\mathbf{U}\underline{C} \ \vee \ \mathbf{U}\underline{D} \ := \ \mathbf{U}\,(\underline{C} \vee \underline{D})$$

$$\underline{\mathbf{unit}} \vee \underline{\mathbf{unit}} \ := \ \underline{\mathbf{unit}} \qquad (\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2) \ \vee \ (\underline{D}_1 \mathbin{\underline{\times}} \underline{D}_2) \ := \ (\underline{C}_1 \vee \underline{D}_1) \mathbin{\underline{\times}} (\underline{C}_2 \vee \underline{D}_2)$$
$$(A \to \underline{C}) \vee (A \to \underline{D}) \ := \ A \to (\underline{C} {\vee} \underline{D}) \qquad \langle\varepsilon\rangle A \vee \langle\varepsilon'\rangle B \ := \ \langle\varepsilon{\vee}\varepsilon'\rangle(A \vee B)$$

Not allowing contravariant subtyping means we do not have to consider meets. Bounded binary joins lift from the effect algebra to types:

Lemma B.1.1 Suppose that the effect algebra is a partially ordered monoid with bounded binary joins.

1. If there is some B such that $A <:' B$ and $A' <:' B$ then $A \vee A'$ is defined and is the join of A and A'.
2. If there is some $\underline{D}$ such that $\underline{C} <:' \underline{D}$ and $\underline{C}' <:' \underline{D}$ then $\underline{C} \vee \underline{C}'$ is defined and is the join of $\underline{C}$ and $\underline{C}'$.

Proof. The proof is by induction on the structure of B and $\underline{D}$. If $B = b$ then $A = A' = b$ and $b{\vee}b = b$ is clearly the join. Similarly for $B = \mathbf{unit}$, for $B = \mathbf{empty}$ and for $\underline{D} = \underline{\mathbf{unit}}$. If $B = B_1{\times}B_2$ then $A = A_1 \times A_2$ and $A' = A'_1 \times A'_2$ for some $A_1, A'_1 <: B_1$ and $A_2, A'_2 <: B_2$. Then $A_1 \vee A'_1$ and $A_2{\vee}A'_2$ are defined, and are joins. Hence $A{\times}A' = (A_1{\times}A_2) \vee (A'_1{\times}A'_2) \ := \ (A_1{\vee}A'_1){\times}(A_2{\vee}A'_2)$, which is an upper bound. It is the least because if B' is an upper bound then $B' = B'_1 \times B'_2$ for some B'_1, B'_2, such that B'_i is an upper bound of A_i and A'_i. Then because they are joins, $A_i \vee A'_i <: B'_i$. Similar reasoning can be applied for $B = B_1 + B_2$, for $B = \mathbf{U}\,B'$, for $\underline{D} = \underline{D}_1 \mathbin{\underline{\times}} \underline{D}_2$ and for $\underline{D} = A \to \underline{D}'$. The latter uses the fact that subtyping is not allowed on A. Finally, if $\underline{D} = \langle\varepsilon''\rangle B$ then $\underline{C} = \langle\varepsilon\rangle A$ and $\underline{C}' = \langle\varepsilon'\rangle A'$ for some $\varepsilon, \varepsilon' \leq \varepsilon''$ and $A, A' <: B$. Then $\varepsilon{\vee}\varepsilon'$ is defined because we assume bounded binary joins, and so is $A{\vee}A'$. Hence $\langle\varepsilon\rangle A{\vee}\langle\varepsilon'\rangle A' = \langle\varepsilon{\vee}\varepsilon'\rangle(A{\vee}A')$ is defined, and is the join. □

We use this notion of subtyping to give typing judgments $\Gamma \vdash_{\mathrm{e}} V : A$ and $\Gamma \mathrel{\underline{\vdash}_{\mathrm{e}}} M : \underline{C}$ for terms that do not have explicit coercions. These are generated by the same rules as the usual typing judgments (Figure 2.11), except that there is no rule for **coerce**, and the rules for case expressions on sum types and for function application are replaced with:

$$\frac{\Gamma \vdash_{\mathrm{e}} V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_{\mathrm{e}} W_1 : B_1 \qquad \Gamma, x_2 : A_2 \vdash_{\mathrm{e}} W_2 : B_2 \qquad B_1 \vee B_2 \text{ defined}}{\Gamma \vdash_{\mathrm{e}} \textbf{case } V \textbf{ of } \{\textbf{inl}\, x_1 .\, W_1,\ \textbf{inr}\, x_2 .\, W_2\} : B_1 \vee B_2}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} V : A \qquad \Gamma \mathrel{\underline{\vdash}_{\mathrm{e}}} M : B \to \underline{C} \qquad A <:' B}{\Gamma \mathrel{\underline{\vdash}_{\mathrm{e}}} V`M : \underline{C}}$$

Note that these are syntax directed. Given Γ and V, if there is some A such that $\Gamma \vdash_{\mathrm{e}} V : A$ then it is unique, and derivations are unique. Similarly for computations. For each term that is well typed under these judgments, we can add explicit coercions to get terms that are well typed under the original judgments. Given a typing context Γ and value term V such that $\Gamma \vdash_{\mathrm{e}} V : A$ for some A, we have a value term $\lceil V \rceil_\Gamma$ such that $\Gamma \vdash \lceil V \rceil_\Gamma : A$. Similarly, given Γ and M such that $\Gamma \mathrel{\underline{\vdash}_{\mathrm{e}}} M : \underline{C}$ for some $\underline{C}$ we have $\lceil M \rceil_\Gamma$ such that $\Gamma \mathrel{\underline{\vdash}} \lceil M \rceil_\Gamma : \underline{C}$. These are defined by induction on the structure of M and V. Most of the cases just apply $\lceil - \rceil$ to subterms. The interesting cases correspond to the new typing rules:

$$\left\lceil \begin{array}{l} \textbf{case } V \textbf{ of} \\ \quad \{\textbf{inl}\ x_1 .\, W_1 \\ \quad ,\ \textbf{inr}\ x_2 .\, W_2\} \end{array} \right\rceil_\Gamma \coloneqq \begin{array}{l} \textbf{case } \lceil V \rceil_\Gamma \textbf{ of} \\ \quad \{\textbf{inl}\ x_1 .\, \textbf{coerce}_{B_1 <: B_1 \vee B_2} \lceil W_1 \rceil_{\Gamma, x_1 : A_1} \\ \quad ,\ \textbf{inr}\ x_2 .\, \textbf{coerce}_{B_2 <: B_1 \vee B_2} \lceil W_2 \rceil_{\Gamma, x_2 : A_2}\} \end{array} \quad \text{if} \begin{cases} \Gamma \vdash_{\mathrm{e}} V : A_1 + A_2 \\ \Gamma, x_1 : A_1 \vdash_{\mathrm{e}} W_1 : B_1 \\ \Gamma, x_2 : A_2 \vdash_{\mathrm{e}} W_2 : B_2 \end{cases}$$

$$\lceil V`M \rceil_\Gamma \coloneqq (\textbf{coerce}_{A <: B} \lceil V \rceil_\Gamma)\ `\ \lceil M \rceil_\Gamma \qquad \text{if} \begin{cases} \Gamma \vdash_{\mathrm{e}} V : A \\ \Gamma \mathrel{\underline{\vdash}_{\mathrm{e}}} M : B \to \underline{C} \end{cases}$$

These terms clearly have the correct types.

We also extend subtyping to typing contexts. Given two typing contexts $\Gamma' = x_1 : A'_1, \ldots, x_n : A'_n$ and $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ of the same length and with the same variable names, we write $\Gamma' <:' \Gamma$ if $A'_i <:' A_i$ for each i, and also write $\textbf{coerce}_{\Gamma' <: \Gamma}$ for the substitution

$$x_1 \mapsto \textbf{coerce}_{A'_1 <: A_1} x_1, \ \ldots,\ x_n \mapsto \textbf{coerce}_{A'_n <: A_n} x_n$$

If $\Gamma \vdash V : A$ then $\Gamma' \vdash V[\textbf{coerce}_{\Gamma' <: \Gamma}] : A$, and if $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ then $\Gamma' \mathrel{\underline{\vdash}} M[\textbf{coerce}_{\Gamma' <: \Gamma}] : \underline{C}$.

Lemma B.1.2 Suppose that the effect algebra is a partially ordered monoid with bounded binary joins.

1. If $\Gamma \vdash V : A$ and $\Gamma' <:' \Gamma$ then there is some $A' <:' A$ such that $\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : A'$ and
$$\Gamma' \vdash \textbf{coerce}_{A' <: A} \lceil \lfloor V \rfloor \rceil_{\Gamma'} \equiv V[\textbf{coerce}_{\Gamma' <: \Gamma}] : A$$
2. If $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$ and $\Gamma' <:' \Gamma$ then there is some $\underline{C}' <:' \underline{C}$ such that $\Gamma' \mathrel{\underline{\vdash}_{\mathrm{e}}} \lfloor M \rfloor : \underline{C}'$ and
$$\Gamma' \mathrel{\underline{\vdash}} \textbf{coerce}_{\underline{C}' <: \underline{C}} \lceil \lfloor M \rfloor \rceil_{\Gamma'} \equiv M[\textbf{coerce}_{\Gamma' <: \Gamma}] : \underline{C}$$

Proof. By induction the derivations of $\Gamma \vdash V : A$ and $\Gamma \mathrel{\underline{\vdash}} M : \underline{C}$.

- Constants c, and the unit terms $()$ and $\lambda\{\}$ are trivial.
- For variables x, we have $(x : A) \in \Gamma$, and $(x : A') \in \Gamma'$ for some $A' <:' A$. Then we have $\lfloor x \rfloor = x$ and $\textbf{coerce}_{A' <: A} \lceil \lfloor x \rfloor \rceil_{\Gamma'} = \textbf{coerce}_{A' <: A}\, x = x[\textbf{coerce}_{\Gamma' <: \Gamma}]$.

- For pairs $\Gamma \vdash (V_1, V_2) : A_1 \times A_2$ by the inductive hypothesis we have

$$\Gamma' \vdash_{\mathrm{e}} \lfloor V_i \rfloor : A_i' \qquad \Gamma' \vdash \mathbf{coerce}_{A_i' <: A_i} \lceil \lfloor V_i \rfloor \rceil_{\Gamma'} \equiv V_i[\mathbf{coerce}_{\Gamma' <: \Gamma}] : A_i$$

  for some $A_i' <:' A_i$. Hence $A_1' \times A_2' <:' A_1 \times A_2$, $\Gamma' \vdash_{\mathrm{e}} \lfloor (V_1, V_2) \rfloor : A_1' \times A_2'$, and

$$\begin{aligned} & \mathbf{coerce}_{A_1' \times A_2' <: A_1 \times A_2} \lceil \lfloor (V_1, V_2) \rfloor \rceil_{\Gamma'} \\ \equiv\; & (\mathbf{coerce}_{A_1' <: A_1} \lceil \lfloor V_1 \rfloor \rceil_{\Gamma'}, \mathbf{coerce}_{A_2' <: A_2} \lceil \lfloor V_2 \rfloor \rceil_{\Gamma'}) && (\beta \text{ laws for products}) \\ \equiv\; & (V_1[\mathbf{coerce}_{\Gamma' <: \Gamma}], V_2[\mathbf{coerce}_{\Gamma' <: \Gamma}]) \\ =\; & (V_1, V_2)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

  The cases for **inl**, **inr**, **thunk**, pairs of computation terms, and $\langle - \rangle$ are similar.

- For $\Gamma \vdash \mathbf{fst}\, V : A_1$ we have $\Gamma \vdash V : A_1 \times A_2$. By the inductive hypothesis there is some subtype of $A_1 \times A_2$, which necessarily has the form $A_1' \times A_2'$ for $A_i' <:' A_i$, such that

$$\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : A_1' \times A_2' \qquad \Gamma' \vdash \mathbf{coerce}_{A_1' \times A_2' <: A_1 \times A_2} \lceil \lfloor V \rfloor \rceil_{\Gamma'} \equiv V[\mathbf{coerce}_{\Gamma' <: \Gamma}] : A_1 \times A_2$$

  We have $\Gamma' \vdash_{\mathrm{e}} \lfloor \mathbf{fst}\, V \rfloor : A_1'$ and

$$\begin{aligned} \mathbf{coerce}_{A_1' <: A_1} \lceil \lfloor \mathbf{fst}\, V \rfloor \rceil_{\Gamma'} &\equiv \mathbf{fst}\,(\mathbf{coerce}_{A_1' \times A_2' <: A_1 \times A_2} \lceil \lfloor V \rfloor \rceil_{\Gamma'}) && (\beta \text{ for products}) \\ &\equiv \mathbf{fst}\,(V[\mathbf{coerce}_{\Gamma' <: \Gamma}]) \\ &= (\mathbf{fst}\, V)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

  The cases for **snd**, projections for products of computations and **force** are similar.

- For $\Gamma \vdash \mathbf{case}_A\ V\ \mathbf{of}\ \{\} : A$ we have $\Gamma \vdash V : \mathbf{empty}$. By the inductive hypothesis there is some subtype of **empty**, which can only be **empty** itself, such that $\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : \mathbf{empty}$. We have $A <:' A$, $\Gamma' \vdash_{\mathrm{e}} \lfloor \mathbf{case}_A\ V\ \mathbf{of}\ \{\} \rfloor : A$, and

$$\begin{aligned} \mathbf{coerce}_{A <: A} \lceil \lfloor \mathbf{case}_A\ V\ \mathbf{of}\ \{\} \rfloor \rceil_{\Gamma'} &\equiv \mathbf{case}_A\ V[\mathbf{coerce}_{\Gamma' <: \Gamma}]\ \mathbf{of}\ \{\} && (\eta \text{ law for } \mathbf{empty}) \\ &= (\mathbf{case}_A\ V\ \mathbf{of}\ \{\})[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

- For $\Gamma \,\underline{\vdash}\, \lambda x\!:\!A.\, M : A \to \underline{C}$ we have $\Gamma, x : A \,\underline{\vdash}\, M : \underline{C}$. Applying the inductive hypothesis to $\Gamma', x : A <:' \Gamma, x : A$, there is some $\underline{C}' <:' \underline{C}$ such that

$$\Gamma', x : A \,\underline{\vdash}_{\mathrm{e}}\, \lfloor M \rfloor : \underline{C}' \qquad \Gamma', x : A \,\underline{\vdash}\, \mathbf{coerce}_{\underline{C}' <: \underline{C}} \lceil \lfloor M \rfloor \rceil_{\Gamma', x:A} \equiv M[\mathbf{coerce}_{\Gamma', x:A <: \Gamma, x:A}] : \underline{C}$$

  We have $(A \to \underline{C}') <:' (A \to \underline{C}')$, $\Gamma' \,\underline{\vdash}_{\mathrm{e}}\, \lfloor \lambda x\!:\!A.\, M \rfloor : A \to \underline{C}'$, and

$$\begin{aligned} & \mathbf{coerce}_{(A \to \underline{C}') <: (A \to \underline{C})} \lceil \lfloor \lambda x\!:\!A.\, M \rfloor \rceil_{\Gamma'} \\ \equiv\; & \lambda y\!:\!A.\, \mathbf{coerce}_{\underline{C}' <: \underline{C}}\, (y \text{ ‘ } \lambda x\!:\!A.\, \lceil \lfloor M \rfloor \rceil_{\Gamma', x:A}) && (\mathbf{coerce}_{A <: A}\, W \equiv W) \\ \equiv\; & \lambda x\!:\!A.\, \mathbf{coerce}_{\underline{C}' <: \underline{C}} \lceil \lfloor M \rfloor \rceil_{\Gamma, x:A} && (\beta \text{ law for functions}) \\ \equiv\; & \lambda x\!:\!A.\, M[\mathbf{coerce}_{\Gamma', x:A <: \Gamma, x:A}] \\ \equiv\; & (\lambda x\!:\!A.\, M)[\mathbf{coerce}_{\Gamma' <: \Gamma}] && (\mathbf{coerce}_{A <: A}\, W \equiv W) \end{aligned}$$

- For $\Gamma \,\underline{\vdash}\, \mathrm{op}\, V : \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}$ we have $\Gamma \vdash V : \mathrm{car}_{\mathrm{op}}$. Since $\mathrm{car}_{\mathrm{op}}$ is a ground type, its only subtype is $\mathrm{car}_{\mathrm{op}}$ itself. Hence by the inductive hypothesis we have

$$\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : \mathrm{car}_{\mathrm{op}} \qquad \Gamma' \vdash \mathbf{coerce}_{\mathrm{car}_{\mathrm{op}} <: \mathrm{car}_{\mathrm{op}}} \lceil \lfloor V \rfloor \rceil_{\Gamma'} \equiv V[\mathbf{coerce}_{\Gamma' <: \Gamma}] : \mathrm{car}_{\mathrm{op}}$$

We have $\langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}} <:' \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}$, $\Gamma' \underline{\vdash}_{\mathrm{e}} \mathrm{op}\, V : \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}$, and

$$\begin{aligned} & \mathbf{coerce}_{\langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}} <: \langle \mathrm{eff}_{\mathrm{op}} \rangle \mathrm{ar}_{\mathrm{op}}} \lceil \lfloor \mathrm{op}\, V \rfloor \rceil_{\Gamma'} \\ \equiv\ & \mathrm{op}\, \lceil \lfloor V \rfloor \rceil_{\Gamma'} && (\mathbf{coerce}_{\underline{C} <: \underline{C}}\, N \equiv N) \\ \equiv\ & \mathrm{op}\, (\mathbf{coerce}_{\mathrm{car}_{\mathrm{op}} <: \mathrm{car}_{\mathrm{op}}} \lceil \lfloor V \rfloor \rceil_{\Gamma'}) && (\mathbf{coerce}_{A <: A}\, W \equiv W) \\ \equiv\ & \mathrm{op}\, (V[\mathbf{coerce}_{\Gamma' <: \Gamma}]) \\ =\ & (\mathrm{op}\, V)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

- For $\Gamma \underline{\vdash} M\ \mathbf{to}\ x.\, N : \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}$ we have $\Gamma \underline{\vdash} M : \langle \varepsilon \rangle A$ and $\Gamma, x : A \underline{\vdash} N : \underline{C}$. There is some subtype of $\langle \varepsilon \rangle A$, which necessarily has the form $\langle \varepsilon' \rangle A'$ for $\varepsilon' \leq \varepsilon$ and $A' <:' A$ such that

  $$\Gamma \underline{\vdash}_{\mathrm{e}} \lfloor M \rfloor : \langle \varepsilon' \rangle A' \qquad \Gamma' \underline{\vdash}\ \mathbf{coerce}_{\langle \varepsilon' \rangle A' <: \langle \varepsilon \rangle A} \lceil \lfloor M \rfloor \rceil_{\Gamma'} \equiv M[\mathbf{coerce}_{\Gamma' <: \Gamma}] : \langle \varepsilon \rangle A$$

  Now $\Gamma', x : A' <:' \Gamma, x : A$, so there is some $\underline{C}' <: \underline{C}$ such that

  $$\Gamma', x : A' \underline{\vdash}_{\mathrm{e}} \lfloor N \rfloor : \underline{C}' \quad \Gamma', x : A' \underline{\vdash}\ \mathbf{coerce}_{\underline{C}' <: \underline{C}} \lceil \lfloor N \rfloor \rceil_{\Gamma', x : A'} \equiv N[\mathbf{coerce}_{\Gamma', x : A' <: \Gamma, x : A}] : \underline{C}$$

  We have $\langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}' <:' \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}$, $\Gamma' \underline{\vdash}_{\mathrm{e}} \lfloor M\ \mathbf{to}\ x.\, N \rfloor : \langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}'$, and

  $$\begin{aligned} & \mathbf{coerce}_{\langle\!\langle \varepsilon' \rangle\!\rangle \underline{C}' <: \langle\!\langle \varepsilon \rangle\!\rangle \underline{C}} \lceil \lfloor M\ \mathbf{to}\ x.\, N \rfloor \rceil_{\Gamma'} \\ \equiv\ & (\mathbf{coerce}_{\varepsilon' \leq \varepsilon} \lceil \lfloor M \rfloor \rceil_{\Gamma'})\ \mathbf{to}\ x.\, \mathbf{coerce}_{\underline{C}' <: \underline{C}} \lceil \lfloor N \rfloor \rceil_{\Gamma', x : A'} \\ \equiv\ & (\mathbf{coerce}_{\varepsilon' \leq \varepsilon} \lceil \lfloor M \rfloor \rceil_{\Gamma'})\ \mathbf{to}\ x.\, N[\mathbf{coerce}_{\Gamma', x : A' <: \Gamma, x : A}] \\ \equiv\ & (\mathbf{coerce}_{\langle \varepsilon' \rangle A' <: \langle \varepsilon \rangle A} \lceil \lfloor M \rfloor \rceil_{\Gamma'})\ \mathbf{to}\ x.\, N[\mathbf{coerce}_{\Gamma' <: \Gamma}] \\ \equiv\ & M[\mathbf{coerce}_{\Gamma' <: \Gamma}]\ \mathbf{to}\ x.\, N[\mathbf{coerce}_{\Gamma' <: \Gamma}] \\ =\ & (M\ \mathbf{to}\ x.\, N)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

- For $\Gamma \underline{\vdash}\ \mathbf{coerce}_{\varepsilon' \leq \varepsilon}\, M : \langle \varepsilon \rangle A$ we have $\Gamma \underline{\vdash} M : \langle \varepsilon' \rangle A$ and $\varepsilon' \leq \varepsilon$. By the inductive hypothesis there is some subtype of $\langle \varepsilon' \rangle A$, which necessarily has the form $\langle \varepsilon'' \rangle A'$ for some $\varepsilon'' \leq \varepsilon'$ and $A' <:' A$, such that

  $$\Gamma' \underline{\vdash}_{\mathrm{e}} \lfloor M \rfloor : \langle \varepsilon'' \rangle A' \quad \Gamma' \underline{\vdash}\ \mathbf{coerce}_{\langle \varepsilon'' \rangle A' <: \langle \varepsilon' \rangle A} \lceil \lfloor M \rfloor \rceil_{\Gamma'} \equiv M[\mathbf{coerce}_{\Gamma' <: \Gamma}] : \langle \varepsilon' \rangle A$$

  We have $\langle \varepsilon'' \rangle A' <:' \langle \varepsilon \rangle A$, and $\Gamma' \underline{\vdash}_{\mathrm{e}} \lfloor \mathbf{coerce}_{\varepsilon' \leq \varepsilon}\, M \rfloor : \langle \varepsilon'' \rangle A'$ because $\lfloor \mathbf{coerce}_{\varepsilon' \leq \varepsilon}\, M \rfloor = \lfloor M \rfloor$. The required equation also holds:

  $$\begin{aligned} & \mathbf{coerce}_{\langle \varepsilon'' \rangle A' <: \langle \varepsilon \rangle A} \lceil \lfloor \mathbf{coerce}_{\varepsilon' \leq \varepsilon}\, M \rfloor \rceil_{\Gamma'} \\ \equiv\ & \mathbf{coerce}_{\langle \varepsilon' \rangle A <: \langle \varepsilon \rangle A}\, (\mathbf{coerce}_{\langle \varepsilon'' \rangle A' <: \langle \varepsilon' \rangle A} \lceil \lfloor M \rfloor \rceil_{\Gamma'}) \\ & \qquad (\mathbf{coerce}_{\underline{D}'' <: \underline{D}}\, N \equiv \mathbf{coerce}_{\underline{D}' <: \underline{D}}\, (\mathbf{coerce}_{\underline{D}'' <: \underline{D}'}\, N)) \\ \equiv\ & \mathbf{coerce}_{\langle \varepsilon' \rangle A <: \langle \varepsilon \rangle A}\, (M[\mathbf{coerce}_{\Gamma' <: \Gamma}]) \\ \equiv\ & (\mathbf{coerce}_{\varepsilon' \leq \varepsilon}\, M)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \end{aligned}$$

- For $\Gamma \vdash \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} : B$ we have $\Gamma \vdash V : A_1 + A_2$. By the inductive hypothesis there is a subtype of $A_1 + A_2$, which necessarily has the form $A_1' + A_2'$ for $A_i' <:' A_i$, such that

  $$\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : A_1' + A_2' \qquad \Gamma' \vdash \mathbf{coerce}_{A_1' + A_2' <: A_1 + A_2} \lceil \lfloor V \rfloor \rceil_{\Gamma'} \equiv V[\mathbf{coerce}_{\Gamma' <: \Gamma}] : A_1 + A_2$$

  For $i \in \{1, 2\}$ we have $\Gamma, x_i : A_i \vdash W_i : B$ and $\Gamma', x_i : A_i' <:' \Gamma, x_i : A_i$, so there exists $B_i' <:' B$ such that

  $$\Gamma', x_i : A_i' \vdash_{\mathrm{e}} \lfloor W_i \rfloor : B_i'$$
  $$\Gamma', x_i : A_i' \vdash \mathbf{coerce}_{B_i' <: B} \lceil \lfloor W_i \rfloor \rceil_{\Gamma', x_i : A_i'} \equiv W_i[\mathbf{coerce}_{\Gamma', x_i : A_i' <: \Gamma, x_i : A_i}] : B$$

Since B'_1 and B'_2 have an upper bound, by Lemma B.1.1 (1) their join $B'_1 \vee B'_2 <: B$ is defined. We therefore have $\Gamma' \vdash_{\mathrm{e}} \lfloor \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} \rfloor : B'_1 \vee B'_2$ and

$$
\begin{aligned}
&\mathbf{coerce}_{B'_1 \vee B'_2 <: B} \lceil \lfloor \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\} \rfloor \rceil_{\Gamma'} \\
&\mathbf{case}\ \lceil \lfloor V \rfloor \rceil_{\Gamma'}\ \mathbf{of} \\
\equiv\ &\quad \{\mathbf{inl}\ x_1.\, \mathbf{coerce}_{B'_1 \vee B'_2 <: B}\, (\mathbf{coerce}_{B'_1 <: B'_1 \vee B'_2}\, \lceil \lfloor W_1 \rfloor \rceil_{\Gamma', x_1 : A'_1}) \qquad (\beta \text{ and } \eta \text{ laws for sums}) \\
&\quad ,\, \mathbf{inr}\, x_2.\, \mathbf{coerce}_{B'_2 \vee B'_2 <: B}\, (\mathbf{coerce}_{B'_2 <: B'_2 \vee B'_2}\, \lceil \lfloor W_2 \rfloor \rceil_{\Gamma', x_2 : A'_2})\} \\
\equiv\ &\mathbf{case}\ \lceil \lfloor V \rfloor \rceil_{\Gamma'}\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, \mathbf{coerce}_{B'_1 <: B}\, \lceil \lfloor W_1 \rfloor \rceil_{\Gamma', x_1 : A'_1},\ \mathbf{inr}\, x_2.\, \mathbf{coerce}_{B'_2 <: B}\, \lceil \lfloor W_2 \rfloor \rceil_{\Gamma', x_2 : A'_2}\} \\
&\hfill (\mathbf{coerce}_{A'' <: A}\, V \equiv \mathbf{coerce}_{A' <: A}\, (\mathbf{coerce}_{A'' <: A'}\, V)) \\
\equiv\ &\mathbf{case}\ \lceil \lfloor V \rfloor \rceil_{\Gamma'}\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1[\mathbf{coerce}_{\Gamma', x_1 : A'_1 <: \Gamma, x_1 : A_1}],\ \mathbf{inr}\, x_2.\, W_2[\mathbf{coerce}_{\Gamma', x_2 : A'_2 <: \Gamma, x_2 : A_2}]\} \\
\equiv\ &\mathbf{case}\ \mathbf{coerce}_{A'_1 + A'_2 <: A_1 + A_2}\, \lceil \lfloor V \rfloor \rceil_{\Gamma'}\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1[\mathbf{coerce}_{\Gamma' <: \Gamma}],\ \mathbf{inr}\, x_2.\, W_2[\mathbf{coerce}_{\Gamma' <: \Gamma}]\} \\
&\hfill (\beta \text{ and } \eta \text{ laws for sums}) \\
\equiv\ &(\mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inl}\, x_1.\, W_1,\ \mathbf{inr}\, x_2.\, W_2\})[\mathbf{coerce}_{\Gamma' <: \Gamma}]
\end{aligned}
$$

- For $\Gamma \underline{\vdash} V\text{`}M : \underline{C}$ we have $\Gamma \vdash V : A$ and $\Gamma \underline{\vdash} M : A \to \underline{C}$. By the inductive hypothesis, there exist $A' <:' A$ and a subtype of $A \to \underline{C}$ that necessarily has the form $A \to \underline{C}'$ for $\underline{C}' <: \underline{C}$ (recall that we do not allow subtyping on arguments), such that

$$
\begin{array}{ll}
\Gamma' \vdash_{\mathrm{e}} \lfloor V \rfloor : A' & \Gamma' \vdash \mathbf{coerce}_{A' <: A}\, \lceil \lfloor V \rfloor \rceil_{\Gamma'} \equiv V[\mathbf{coerce}_{\Gamma' <: \Gamma}] : A \\
\Gamma' \underline{\vdash}_{\mathrm{e}} \lfloor M \rfloor : A \to \underline{C}' & \Gamma' \underline{\vdash}\, \mathbf{coerce}_{A \to \underline{C}' <: A \to \underline{C}}\, \lceil \lfloor M \rfloor \rceil_{\Gamma'} \equiv M[\mathbf{coerce}_{\Gamma' <: \Gamma}] : A \to \underline{C}
\end{array}
$$

  We therefore have $\Gamma \underline{\vdash}_{\mathrm{e}} \lfloor V\text{`}M \rfloor : \underline{C}'$ and

$$
\begin{aligned}
\mathbf{coerce}_{\underline{C}' <: \underline{C}}\, \lceil \lfloor V\text{`}M \rfloor \rceil_{\Gamma'} &\equiv \mathbf{coerce}_{\underline{C}' <: \underline{C}}\, ((V[\mathbf{coerce}_{\Gamma' <: \Gamma}])\, \text{`}\, \lceil \lfloor M \rfloor \rceil_{\Gamma'}) \\
&\equiv (V[\mathbf{coerce}_{\Gamma' <: \Gamma}])\, \text{`}\, \mathbf{coerce}_{A \to \underline{C}' <: A \to \underline{C}}\, \lceil \lfloor M \rfloor \rceil_{\Gamma'} \\
&\hfill (\beta \text{ law for function types}) \\
&\equiv (V\text{`}M)[\mathbf{coerce}_{\Gamma' <: \Gamma}] \qquad \square
\end{aligned}
$$

We obtain Lemma 2.7.5 as a corollary.

Proof (of Lemma 2.7.5). We give the proof for the first part of the lemma (values). The proof of the second part (computations) is similar.

Suppose that $\Gamma \vdash V_1 : A$ and $\Gamma \vdash V_2 : A$. We have $\Gamma <:' \Gamma$, so by Lemma B.1.2 (1) there are value types $A'_i <:' A$ such that $\Gamma \vdash_{\mathrm{e}} \lfloor V_i \rfloor : A'_i$ and

$$\Gamma \vdash \mathbf{coerce}_{A'_i <: A}\, \lceil \lfloor V_i \rfloor \rceil_{\Gamma} \equiv V_i[\mathbf{coerce}_{\Gamma <: \Gamma}] : A$$

for $i \in \{1, 2\}$. If $\lfloor V_1 \rfloor = \lfloor V_2 \rfloor$ then $A'_1 = A'_2$, and we have:

$$V_1 \equiv V_1[\mathbf{coerce}_{\Gamma <: \Gamma}] \equiv \mathbf{coerce}_{A'_1 <: A}\, \lceil \lfloor V_1 \rfloor \rceil_{\Gamma} = \mathbf{coerce}_{A'_2 <: A}\, \lceil \lfloor V_2 \rfloor \rceil_{\Gamma} \equiv V_2[\mathbf{coerce}_{\Gamma <: \Gamma}] \equiv V_2$$

where we use the fact that the inequational theory is closed under substitution. □

B.2 Logical relations and the free lifting

This section contains proofs related to the free lifting defined in Section 3.1.1.

Lemma 3.1.6 If $(c, c) \in \mathcal{R}_{\mathrm{free}}[\![A]\!]$ for each constant $c \in \mathcal{K}_A$, then $\mathcal{R}_{\mathrm{free}}[\![-]\!]$ is a logical relation.

Proof. We show that the free lifting meets each requirement in the definition of logical relation.

- The equations in Figure 3.1 hold by definition.
- For closure under operations we note that

  $$\begin{array}{l} \mathrm{op}\,V \;\equiv \mathrm{op}\,V\;\; \mathbf{to}\; x.\,\langle x\rangle \\ \mathrm{op}\,V' \equiv \mathrm{op}\,V'\; \mathbf{to}\; x.\,\langle x\rangle \end{array} \qquad (\langle x\rangle, \langle x\rangle) : \mathcal{R}_{\mathrm{free}}[\![x : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}_{\mathrm{free}}[\![\langle 1\rangle \mathrm{ar}_{\mathrm{op}}]\!]$$

  and apply the final case in the definition of $\mathcal{R}_{\mathrm{free}}[\![\langle 1\rangle \mathrm{ar}_{\mathrm{op}}]\!]$.
- Closure under pure computations is one of the cases in the definition of $\mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$.
- For closure under **to** suppose that $(M, M') \in \mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$ and $(N, N') : \mathcal{R}_{\mathrm{free}}[\![x : A]\!] \dot{\to} \mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon'\rangle B]\!]$. We show that

  $$(M\;\mathbf{to}\;x.\,N, M'\;\mathbf{to}\;x.\,N') \in \mathcal{R}[\![\langle \varepsilon\cdot\varepsilon'\rangle B]\!]$$

  by induction on $(M, M') \in \mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$. For the $(\langle V\rangle, \langle V'\rangle)$ case use the assumption about (N, N') and

  $$\langle V\rangle\;\mathbf{to}\;x.\,N \equiv N[x \mapsto V] \qquad \langle V'\rangle\;\mathbf{to}\;x.\,N' \equiv N'[x \mapsto V']$$

  For $(\mathbf{coerce}_{\varepsilon''\le\varepsilon}\,M, \mathbf{coerce}_{\varepsilon''\le\varepsilon}\,M')$ use the inductive hypothesis and

  $$\begin{array}{l} (\mathbf{coerce}_{\varepsilon''\le\varepsilon}\,M\;)\;\mathbf{to}\;x.\,N\;\; \equiv \mathbf{coerce}_{\varepsilon''\cdot\varepsilon'\le\varepsilon\cdot\varepsilon'}\,(M\;\;\mathbf{to}\;x.\,N\;) \\ (\mathbf{coerce}_{\varepsilon''\le\varepsilon}\,M')\;\mathbf{to}\;x.\,N' \equiv \mathbf{coerce}_{\varepsilon''\cdot\varepsilon'\le\varepsilon\cdot\varepsilon'}\,(M'\;\mathbf{to}\;x.\,N') \end{array}$$

  For $(\mathrm{op}\,V\;\mathbf{to}\;y.\,M,\; \mathrm{op}\,V'\;\mathbf{to}\;y.\,M')$, note that the inductive hypothesis implies

  $$(M\;\mathbf{to}\;x.\,N,\; M'\;\mathbf{to}\;x.\,N') : \mathcal{R}_{\mathrm{free}}[\![y : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\cdot\varepsilon'\rangle B]\!]$$

  and then use

  $$\begin{array}{l} (\mathrm{op}\,V\;\;\mathbf{to}\;y.\,M\;)\;\mathbf{to}\;x.\,N\;\; \equiv \mathrm{op}\,V\;\;\mathbf{to}\;y.\,(M\;\;\mathbf{to}\;x.\,N\;) \\ (\mathrm{op}\,V'\;\mathbf{to}\;y.\,M')\;\mathbf{to}\;x.\,N' \equiv \mathrm{op}\,V'\;\mathbf{to}\;y.\,(M'\;\mathbf{to}\;x.\,N') \end{array}$$

- Closure under **coerce** is one of the cases in the definition of $\mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$.
- Each constant is related to itself by assumption. □

Lemma 3.1.7 Suppose that $\mathcal{R}[\![-]\!]$ is a logical relation and $\mathcal{R}_{\mathrm{free}}[\![b]\!] = \mathcal{R}[\![b]\!]$ for all base types b. Then $\mathcal{R}_{\mathrm{free}}[\![A]\!] \subseteq \mathcal{R}[\![A]\!]$ implies $\mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!] \subseteq \mathcal{R}[\![\langle \varepsilon\rangle A]\!]$.

Proof. By induction on the definition of $\mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$. Each case uses the closure properties of the logical relation $\mathcal{R}[\![-]\!]$. For the $(\langle V\rangle, \langle V'\rangle)$ case use $\mathcal{R}_{\mathrm{free}}[\![A]\!] \subseteq \mathcal{R}[\![A]\!]$ and closure under pure computations. For $(\mathbf{coerce}_{\varepsilon\le\varepsilon'}\,M, \mathbf{coerce}_{\varepsilon\le\varepsilon'}\,M')$ use the inductive hypothesis and closure under **coerce**. The $(\mathrm{op}\,V\;\mathbf{to}\;x.\,M,\; \mathrm{op}\,V'\;\mathbf{to}\;x.\,M')$ case is the most difficult. Suppose that $(V, V') \in \mathcal{R}_{\mathrm{free}}[\![\mathrm{car}_{\mathrm{op}}]\!]$ and $(M, M') : \mathcal{R}_{\mathrm{free}}[\![x : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}_{\mathrm{free}}[\![\langle \varepsilon\rangle A]\!]$. By the assumption on base types, we have $\mathcal{R}_{\mathrm{free}}[\![G]\!] = \mathcal{R}[\![G]\!]$ for all ground types G, in particular for the coarity and arity of op. Hence the inductive hypothesis implies $(V, V') \in \mathcal{R}[\![\mathrm{car}_{\mathrm{op}}]\!]$ and $(M, M') : \mathcal{R}[\![x : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}[\![\langle \varepsilon\rangle A]\!]$, and the result follows by closure of $\mathcal{R}[\![-]\!]$ under operations and under **to**. □

Lemma 3.1.8 For each value type A and effect $\varepsilon \subseteq \Sigma$,

$$\mathcal{R}'_{\text{free}}[\![\langle \varepsilon \rangle A]\!] = \mathcal{R}_{\text{free}}[\![\langle \varepsilon \rangle A]\!]$$

Proof. ($\subseteq$): For a fixed effect ε, we have $\mathcal{R}'_{\text{free}}[\![\langle \varepsilon \rangle A]\!] \subseteq \mathcal{R}_{\text{free}}[\![\langle \varepsilon \rangle A]\!]$ by a trivial induction.

($\supseteq$): First, a trivial induction shows that if $\varepsilon \subseteq \varepsilon'$ then

$$(M, M') \in \mathcal{R}'_{\text{free}}[\![\langle \varepsilon \rangle A]\!] \quad \Rightarrow \quad (\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M, \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M') \in \mathcal{R}'_{\text{free}}[\![\langle \varepsilon' \rangle A]\!]$$

We then show by another induction that for all ε and $(M, M') \in \mathcal{R}_{\text{free}}[\![\langle \varepsilon \rangle A]\!]$ we have $(M, M') \in \mathcal{R}'_{\text{free}}[\![\langle \varepsilon \rangle A]\!]$. For the $(\langle V \rangle, \langle V' \rangle) \in \mathcal{R}_{\text{free}}[\![\langle 1 \rangle A]\!]$ case use

$$\langle V \rangle \equiv \mathbf{coerce}_{\emptyset \leq \emptyset}\, \langle V \rangle \qquad \langle V' \rangle \equiv \mathbf{coerce}_{\emptyset \leq \emptyset}\, \langle V' \rangle$$

For $(\mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M, \mathbf{coerce}_{\varepsilon \leq \varepsilon'}\, M) \in \mathcal{R}_{\text{free}}[\![\langle \varepsilon' \rangle A]\!]$ use the above implication. Finally, for

$$((\mathrm{op}\, V\ \mathbf{to}\ x.\, M), (\mathrm{op}\, V'\ \mathbf{to}\ x.\, M')) \in \mathcal{R}_{\text{free}}[\![\langle \{\mathrm{op}\} \cup \varepsilon \rangle A]\!]$$

we assume that

$$(M, M') : \mathcal{R}_{\text{free}}[\![x : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}_{\text{free}}[\![\langle \varepsilon \rangle A]\!] \qquad (V, V') \in \mathcal{R}_{\text{free}}[\![\mathrm{car}_{\mathrm{op}}]\!]$$

By the inductive hypothesis, the above implication, and the fact that $\mathcal{R}_{\text{free}}$ and $\mathcal{R}'_{\text{free}}$ coincide on ground types, we have

$$(\mathbf{coerce}_{\varepsilon \leq \{\mathrm{op}\} \cup \varepsilon}\, M, \mathbf{coerce}_{\varepsilon \leq \{\mathrm{op}\} \cup \varepsilon}\, M') : \mathcal{R}_{\text{free}}[\![x : \mathrm{ar}_{\mathrm{op}}]\!] \dot{\to} \mathcal{R}'_{\text{free}}[\![\langle \{\mathrm{op}\} \cup \varepsilon \rangle A]\!]$$

so by the definition of $\mathcal{R}'_{\text{free}}[\![\langle \{\mathrm{op}\} \cup \varepsilon \rangle A]\!]$,

$$((\mathrm{op}\, V\ \mathbf{to}\ x.\, \mathbf{coerce}_{\varepsilon \leq \{\mathrm{op}\} \cup \varepsilon}\, M), (\mathrm{op}\, V'\ \mathbf{to}\ x.\, \mathbf{coerce}_{\varepsilon \leq \{\mathrm{op}\} \cup \varepsilon}\, M')) \in \mathcal{R}'_{\text{free}}[\![\langle \{\mathrm{op}\} \cup \varepsilon \rangle A]\!]$$

The result follows because

$$\mathrm{op}\, V\ \mathbf{to}\ x.\, \mathbf{coerce}_{\varepsilon \leq \{\mathrm{op}\} \cup \varepsilon'}\, M \ \equiv\ \mathrm{op}\, V\ \mathbf{to}\ x.\, M$$

and similarly for $\mathrm{op}\, V'\ \mathbf{to}\ x.\, M'$. □

B.3 Call-by-name and call-by-need

This section contains lemmas that are used to relate call-by-name and call-by-need evaluation Section 5.3.

First, we give two lemmas that are useful in later proofs in this section.

Lemma B.3.1 The relations $\mathcal{R}[\![A]\!]^{\Delta}$ and $\mathcal{R}[\![\underline{C}]\!]^{\Delta}$ are partial equivalence relations (symmetric and transitive).

Proof. By induction on A and $\underline{C}$. Most of the cases are trivial. For returner types, transitivity comes from the definition of closed under divergence. The proof that $(M, M') \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta}$ implies $(M', M) \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta}$ is a simple induction on $(M, M') \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta}$. □

Lemma B.3.2 The Kripke relation $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence for every computation type $\underline{C}$.

Proof. By induction on $\underline{C}$. We already have transitivity (by Lemma B.3.1), so in each case we only consider the other requirements.

- For returner types this holds by definition.
- For $\underline{C} = \underline{\mathbf{unit}}$, this is trivial because $\mathcal{R}[\![\underline{\mathbf{unit}}]\!]^{\Delta}$ is the total relation.
- For $\underline{C} = \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2$ we check each of the requirements individually.
  - If $\Delta \mathrel{\underline{\vdash}} M : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2$ and $\Delta \mathrel{\underline{\vdash}} M' : \underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2$ are trivially diverging then we have

    $$M \;\equiv\; P_1 \;\mathbf{to}\; x.\, P_2 \qquad M' \;\equiv\; P_1' \;\mathbf{to}\; x'.\, P_2'$$

    for some

    $$P_1 \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B) \in \Delta\} \cup \{\Omega_{\mathbf{F}B}\} \qquad P_1' \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B') \in \Delta\} \cup \{\Omega_{\mathbf{F}B'}\}$$

    and P_2, P_2'. For each $i \in \{1, 2\}$ we have $i\text{‘}M \equiv P_1 \;\mathbf{to}\; x.\, i\text{‘}P_2$, and similarly for M'. Hence $i\text{‘}M$ and $i\text{‘}M'$ are both trivially diverging, so are related. This implies that $(M, M') \in \mathcal{R}[\![\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2]\!]^{\Delta}$.
  - For the second case of the definition of closed under divergence (Definition 5.3.3) we give the proof for the pair

    $$(M \;\mathbf{need}\; \underline{y}.\, N,\;\; N'[\underline{y} \mapsto M'])$$

    where M and M' are trivially diverging and $(N, N') \in \mathcal{R}[\![\underline{C}_1 \mathbin{\underline{\times}} \underline{C}_2]\!]^{\Delta, \underline{y} : \mathbf{F}\, A}$ (the other three pairs are similar). For each $i \in \{1, 2\}$, we have

    $$i\text{‘}(M \;\mathbf{need}\; \underline{y}.\, N) \;\equiv\; M \;\mathbf{need}\; \underline{y}.\, i\text{‘}N \qquad i\text{‘}(N'[\underline{y} \mapsto M']) \;=\; (i\text{‘}N')[\underline{y} \mapsto M']$$

    So it suffices to show that $(M \;\mathbf{need}\; \underline{y}.\, i\text{‘}N,\; (i\text{‘}N')[\underline{y} \mapsto M']) \in \mathcal{R}[\![\underline{C}_i]\!]^{\Delta}$. This follows from the inductive hypothesis.
- For $\underline{C} = A \to \underline{D}$ we again check each requirement individually.
  - If $\Delta \mathrel{\underline{\vdash}} M : A \to \underline{D}$ and $\Delta \mathrel{\underline{\vdash}} M' : A \to \underline{D}$ are trivially diverging then we have

    $$M \;\equiv\; P_1 \;\mathbf{to}\; x.\, P_2 \qquad M' \;\equiv\; P_1' \;\mathbf{to}\; x'.\, P_2'$$

    for some

    $$P_1 \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B) \in \Delta\} \cup \{\Omega_{\mathbf{F}B}\} \qquad P_1' \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B') \in \Delta\} \cup \{\Omega_{\mathbf{F}B'}\}$$

    and P_2, P_2'. For each $\Delta' \triangleright \Delta$ and $(V, V') \in \mathcal{R}[\![A]\!]^{\Delta'}$ we have $V\text{‘}M \equiv P_1 \;\mathbf{to}\; x.\, V\text{‘}P_2$, and similarly for M'. Hence $V\text{‘}M$ and $V'\text{‘}M'$ are both trivially diverging, so are related. This implies that $(M, M') \in \mathcal{R}[\![A \to \underline{D}]\!]^{\Delta}$.
  - For the second case of the definition of closed under divergence (Definition 5.3.3) we give the proof for the pair

    $$(M \;\mathbf{need}\; \underline{y}.\, N,\;\; N'[\underline{y} \mapsto M'])$$

    where M and M' are trivially diverging and $(N, N') \in \mathcal{R}[\![A \to \underline{D}]\!]^{\Delta, \underline{y} : \mathbf{F}\, A}$ (the other three pairs are similar). For each $\Delta' \triangleright \Delta$ and $(V, V') \in \mathcal{R}[\![A]\!]^{\Delta}$, we have

    $$V\text{‘}(M \;\mathbf{need}\; \underline{y}.\, N) \;\equiv\; M \;\mathbf{need}\; \underline{y}.\, V\text{‘}N \qquad V'\text{‘}(N'[\underline{y} \mapsto M']) \;=\; (V'\text{‘}N')[\underline{y} \mapsto M']$$

    So it suffices to show that $(M \;\mathbf{need}\; \underline{y}.\, V\text{‘}N,\; (V'\text{‘}N')[\underline{y} \mapsto M']) \in \mathcal{R}[\![\underline{D}]\!]^{\Delta'}$. This follows from the inductive hypothesis, using the fact that $\mathcal{R}[\![\underline{D}]\!]$ is a Kripke relation to weaken N and N'. □

We next turn to the proof of the fundamental lemma. As usual, this is by induction on the structure of the terms. Most of the cases are completely standard, so in this section we only give the cases for **need** and **to**, which involve the definition of the logical relation on returner types. These are the next two lemmas.

Lemma B.3.3 If $\Gamma, \underline{x} : \mathbf{F}\,A \vdash M : \underline{C}$ satisfies

$$(M[\sigma], M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$$

for all Δ and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma, \underline{x} : \mathbf{F}\,A]\!]^{\Delta}$, then for all $(N, N') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta}$ and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$, we have

$$(N \textbf{ need } \underline{x}.\, M[\sigma],\ N' \textbf{ need } \underline{x}.\, M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$$

Proof. By induction on the derivation of $(N, N') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta}$.

- For transitivity, use the inductive hypothesis and transitivity of $\mathcal{R}[\![\underline{C}]\!]^{\Delta}$ (Lemma B.3.1).
- If $N \equiv \langle V \rangle$ and $N' \equiv \langle V' \rangle$ with $(V, V') \in \mathcal{R}[\![A]\!]^{\Delta}$ then the pair we are considering is the same up to $\equiv$ as
$$(M[\sigma, \underline{x} \mapsto \langle V \rangle],\ M[\sigma', \underline{x} \mapsto \langle V' \rangle])$$
The result therefore follows from the assumption about M.
- If N and N' are trivially diverging, the result follows from the fact that $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence (Lemma B.3.2).
- For the second case of the definition of closed under divergence (Definition 5.3.3), the pair (N, N') can have one of four forms. We give the proof for one of them (the other three are similar). Suppose that
$$N \ \equiv\ N_1 \textbf{ need } \underline{y}.\, N_2 \qquad N' \ \equiv\ N_2'[\underline{y} \mapsto N_1']$$
where N_1 and N_1' are trivially diverging and $(N_2, N_2') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta, \underline{y}:\mathbf{F}\,B}$. We have
$$\begin{aligned} N \ \textbf{ need } \underline{x}.\, M[\sigma\ ] \ &\equiv\ N_1 \textbf{ need } \underline{y}.\, N_2 \textbf{ need } \underline{x}.\, M[\sigma] \\ N' \textbf{ need } \underline{x}.\, M[\sigma'] \ &\equiv\ (N_2' \textbf{ need } \underline{x}.\, M[\sigma'])[\underline{y} \mapsto N_1'] \end{aligned}$$
Since $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence (Lemma B.3.2), it therefore suffices to show that
$$(N_2 \textbf{ need } \underline{x}.\, M[\sigma]\ , N_2' \textbf{ need } \underline{x}.\, M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta, \underline{y}:\mathbf{F}\,B}$$
This follows from the inductive hypothesis, using the fact that Kripke relations respect weakening (σ and σ' are weakened). □

Lemma B.3.4 If $\Gamma, x : A \vdash M : \underline{C}$ satisfies

$$(M[\sigma], M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$$

for all Δ and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma, x : A]\!]^{\Delta}$, then for all $(N, N') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta}$ and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$ we have

$$(N \textbf{ to } x.\, M[\sigma],\ N' \textbf{ to } x.\, M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta}$$

Proof. By induction on the derivation of $(N, N') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta}$.

- For transitivity, use the inductive hypothesis and transitivity of $\mathcal{R}[\![\underline{C}]\!]^{\Delta}$ (Lemma B.3.1).

- If $N \equiv \langle V\rangle$ and $N' \equiv \langle V'\rangle$ with $(V, V') \in \mathcal{R}[\![A]\!]^{\Delta}$ then the pair we are considering is the same up to $\equiv$ as
$$(M[\sigma, x \mapsto \langle V\rangle],\ M[\sigma', x \mapsto \langle V'\rangle])$$
The result therefore follows from the assumption about M.
- If N and N' are trivially diverging then for some
$$P_1 \in \{\underline{y} \mid (\underline{y} : \mathbf{F}B) \in \Delta\} \cup \{\Omega_{\mathbf{F}B}\} \qquad P'_1 \in \{\underline{y} \mid (\underline{y} : \mathbf{F}B') \in \Delta\} \cup \{\Omega_{\mathbf{F}B'}\}$$
and P_2, P'_2 we have
$$N \ \equiv\ P_1 \ \mathbf{to}\ z.\, P_2 \qquad N' \ \equiv\ P'_1 \ \mathbf{to}\ z'.\, P'_2$$
Hence
$$N \ \mathbf{to}\ x.\, M[\sigma] \ \equiv\ P_1 \ \mathbf{to}\ z.\, P_2 \ \mathbf{to}\ x.\, M[\sigma] \qquad N' \ \mathbf{to}\ x.\, M[\sigma'] \ \equiv\ P'_1 \ \mathbf{to}\ z'.\, P'_2 \ \mathbf{to}\ x.\, M[\sigma']$$
so the two computations we wish to relate are both trivially diverging, and we can use the fact that $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence (Lemma B.3.2).
- For the second case of the definition of closed under divergence (Definition 5.3.3), the pair (N, N') can have one of four forms. We give the proof for one of them (the other three are similar). Suppose that
$$N \ \equiv\ N_1 \ \mathbf{need}\ \underline{y}.\, N_2 \qquad N' \ \equiv\ N'_2[\underline{y} \mapsto N'_1]$$
where N_1 and N'_1 are trivially diverging and $(N_2, N'_2) \in \mathcal{R}[\![\mathbf{F}A]\!]^{\Delta, \underline{y}:\mathbf{F}B}$. We have
$$\begin{aligned} N \ \ \mathbf{to}\ x.\, M[\sigma\ ] \ &\equiv\ N_1 \ \mathbf{need}\ \underline{y}.\, N_2 \ \mathbf{to}\ x.\, M[\sigma] \\ N' \ \mathbf{to}\ \underline{x}.\, M[\sigma'] \ &\equiv\ (N'_2 \ \mathbf{to}\ x.\, M[\sigma'])[\underline{y} \mapsto N'_1] \end{aligned}$$
Since $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence (Lemma B.3.2), it therefore suffices to show that
$$(N_2 \ \mathbf{to}\ x.\, M[\sigma]\ , N'_2 \ \mathbf{to}\ x.\, M[\sigma']) \in \mathcal{R}[\![\underline{C}]\!]^{\Delta, \underline{y}:\mathbf{F}B}$$
This follows from the inductive hypothesis, using the fact that Kripke relations respect weakening (σ and σ' are weakened). □

At this point, we know that the fundamental lemma holds. We next show that call-by-name and call-by-need are related by the logical relation:

Lemma B.3.5 If $\Gamma, \underline{x} : \mathbf{F}A \vdash M : \underline{C}$ then for all $(N, N') \in \mathcal{R}[\![\mathbf{F}A]\!]^{\Delta}$ and $(\sigma, \sigma') \in \mathcal{R}[\![\Gamma]\!]^{\Delta}$, each of the following pairs is in $\mathcal{R}[\![\underline{C}]\!]^{\Delta}$:
$$\begin{array}{rl} (N \ \mathbf{need}\ \underline{x}.\, M[\sigma],\ N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']) & (M[\sigma][\underline{x} \mapsto N],\ M[\sigma'][\underline{x} \mapsto N']) \\ (M[\sigma][\underline{x} \mapsto N],\ N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']) & (N \ \mathbf{need}\ \underline{x}.\, M[\sigma],\ M[\sigma'][\underline{x} \mapsto N']) \end{array}$$

Proof. For the two pairs on the top row, we apply Lemma B.3.3 and the fundamental lemma (Lemma 5.3.4). The remaining two require more effort. We give only the proof for
$$(M[\sigma][\underline{x} \mapsto N],\ N' \ \mathbf{need}\ \underline{x}.\, M[\sigma'])$$
The other is similar. The proof is by induction on the derivation of $(N, N') \in \mathcal{R}[\![\mathbf{F}A]\!]^{\Delta}$.

- For transitivity, use the inductive hypothesis and transitivity of $\mathcal{R}[\![\underline{C}]\!]^{\Delta}$ (Lemma B.3.1).

- If $N \equiv \langle V \rangle$ and $N' \equiv \langle V' \rangle$ with $(V, V') \in \mathcal{R}[\![A]\!]^{\Delta}$ then the pair we are considering is the same up to $\equiv$ as
$$(M[\sigma, \underline{x} \mapsto \langle V \rangle],\ M[\sigma', \underline{x} \mapsto \langle V' \rangle])$$
The result therefore follows from the fundamental lemma (Lemma 5.3.4).
- If N and N' are trivially diverging, the result follows from the fact that $\mathcal{R}[\![\underline{C}]\!]$ is closed under divergence (Lemma B.3.2).
- For the second case of the definition of closed under divergence (Definition 5.3.3), suppose that $\Delta \underline{\vdash} N_1 : \mathbf{F}\,B$ and $\Delta \underline{\vdash} N_1' : \mathbf{F}\,B$ are trivially diverging, and that $(N_2, N_2') \in \mathcal{R}[\![\mathbf{F}\,A]\!]^{\Delta, \underline{y} : \mathbf{F}\,B}$. We consider each of the four forms of pair (N, N') in turn. In each case, we use the fact that the logical relation is a partial equivalence relation (Lemma B.3.1), and that σ is related to itself (which follows from the fundamental lemma). We also weaken terms, and therefore use the fact that we have Kripke relations.
  - If $(N, N') = (N_1 \ \mathbf{need}\ \underline{y}.\, N_2,\ N_1' \ \mathbf{need}\ \underline{y}.\, N_2')$ then
$$\begin{aligned}
M[\sigma][\underline{x} \mapsto N] &\equiv M[\sigma, \underline{x} \mapsto (N_1 \ \mathbf{need}\ \underline{y}.\, N_2)] \\
&\mathrel{\mathcal{R}} M[\sigma, \underline{x} \mapsto N_2[\underline{y} \mapsto N_1]] && \text{(5.3.4, closure under divergence)} \\
&\equiv M[\sigma, \underline{x} \mapsto N_2][\underline{y} \mapsto N_1] \\
&\mathrel{\mathcal{R}} N_1' \ \mathbf{need}\ \underline{y}.\, N_2' \ \mathbf{need}\ \underline{x}.\, M[\sigma'] && \text{(IH, closure under divergence)} \\
&\equiv N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']
\end{aligned}$$
  - If $(N, N') = (N_2[\underline{y} \mapsto N_1],\ N_2'[\underline{y} \mapsto N_1'])$ then
$$\begin{aligned}
M[\sigma][\underline{x} \mapsto N] &\equiv M[\sigma, \underline{x} \mapsto N_2][\underline{y} \mapsto N_1] \\
&\mathrel{\mathcal{R}} (N_2' \ \mathbf{need}\ \underline{x}.\, M[\sigma'])[\underline{y} \mapsto N_1'] && \text{(IH, closure under divergence)} \\
&\equiv N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']
\end{aligned}$$
  - If $(N, N') = (N_2[\underline{y} \mapsto N_1],\ N_1' \ \mathbf{need}\ \underline{y}.\, N_2')$ then
$$\begin{aligned}
M[\sigma][\underline{x} \mapsto N] &\equiv M[\sigma, \underline{x} \mapsto N_2][\underline{y} \mapsto N_1] \\
&\mathrel{\mathcal{R}} N_1' \ \mathbf{need}\ \underline{y}.\, N_2' \ \mathbf{need}\ \underline{x}.\, M[\sigma'] && \text{(IH, closure under divergence)} \\
&\equiv N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']
\end{aligned}$$
  - If $(N, N') = (N_1 \ \mathbf{need}\ \underline{y}.\, N_2,\ N_2'[\underline{y} \mapsto N_1'])$ then
$$\begin{aligned}
M[\sigma][\underline{x} \mapsto N] &\equiv M[\sigma, \underline{x} \mapsto (N_1 \ \mathbf{need}\ \underline{y}.\, N_2)] \\
&\mathrel{\mathcal{R}} M[\sigma, \underline{x} \mapsto N_2[\underline{y} \mapsto N_1]] && \text{(5.3.4, closure under divergence)} \\
&\equiv M[\sigma, \underline{x} \mapsto N_2][\underline{y} \mapsto N_1] \\
&\mathrel{\mathcal{R}} (N_2' \ \mathbf{need}\ \underline{x}.M[\sigma'])[\underline{y} \mapsto N_1'] && \text{(IH, closure under divergence)} \\
&\equiv N' \ \mathbf{need}\ \underline{x}.\, M[\sigma']
\end{aligned}$$
□

Finally, we show that we can use the logical relation to prove contextual equivalences. First we note that closed, trivially diverging computations actually diverge:

Lemma B.3.6 If $\diamond \underline{\vdash} M : \underline{C}$ is trivially diverging then $M \equiv \Omega_{\underline{C}}$.

Proof. There are no free computation variables, so $M \equiv \Omega_{\mathbf{F}\,A}$ **to** $x.\, N$ for some N. The result then follows from the η-law for the empty type. □

Next, we note that on ground types the logical relation matches the equational theory:

Lemma B.3.7 If G is a ground type and $(V, V') \in \mathcal{R}[\![G]\!]^{\Delta}$, then $V \equiv W \equiv V'$ for some closed value W.

Proof. By induction on G.

- If $G = \mathbf{unit}$ then $V \equiv () \equiv V'$ by the η-law for **unit**.
- If $G = G_1 \times G_2$ then

$$V \;\equiv\; (\mathbf{fst}\, V_1, \mathbf{snd}\, V_2) \;\equiv\; (W_1, W_2) \;\equiv\; (\mathbf{fst}\, V_1', \mathbf{snd}\, V_2') \;\equiv\; V'$$

  where W_1 and W_2 come from applying the inductive hypothesis.
- If $G = \mathbf{empty}$ then we have a contradiction ($\mathcal{R}[\![\mathbf{empty}]\!]^{\Delta}$ is empty).
- If $G = G_1 + G_2$ then there are two cases to consider. If $V \equiv \mathbf{inl}_{G_2} V_1$ and $V' \equiv \mathbf{inl}_{G_2} V_1'$ with $(V_1, V_1') \in \mathcal{R}[\![G_1]\!]^{\Delta}$, then

$$V \;\equiv\; \mathbf{inl}_{G_2} V_1 \;\equiv\; \mathbf{inl}_{G_2} W \;\equiv\; \mathbf{inl}_{G_2} V_1 \;\equiv\; V'$$

  where W comes from applying the inductive hypothesis. The **inr** case is similar. □

We also show that closed returners of ground type either return a result or diverge. To do this, we also need to consider computations with free computation variables. The next lemma is where it is important that computation variables are only bound inside the definition of the logical relation to trivially diverging computations.

Lemma B.3.8 If G is a ground type and $(M, M') \in \mathcal{R}[\![\mathbf{F}\, G]\!]^{\Delta}$ then one of the following holds:

1. $M \equiv \langle V \rangle \equiv M'$ for some closed value V.
2. M and M' are both trivially diverging.

Proof. By definition, $\mathcal{R}[\![\mathbf{F}\, A]\!]$ is the smallest closed-under-divergence relation such that

$$(V, V') \in \mathcal{R}[\![A]\!]^{\Delta} \;\Rightarrow\; (\langle V \rangle, \langle V' \rangle) \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta} \tag{B.1}$$

so we can proceed by induction on the derivation of $(M, M') \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta}$. If this holds because of Equation (B.1) then the result follows immediately from Lemma B.3.7. Transitivity is trivial (using transitivity of $\equiv$). Otherwise, we consider each of the cases in the definition of closed under divergence (Definition 5.3.3).

- In the first case of the definition, both computations are required to be trivially diverging.
- For the second case we have two trivially diverging computations, which necessarily have the form

$$M \equiv P_1 \;\mathbf{to}\; x.\, P_2 \qquad M' \equiv P_1' \;\mathbf{to}\; x'.\, P_2'$$

  for some

$$P_1 \in \{\underline{z} \mid (\underline{z} : \mathbf{F} B) \in \Delta\} \cup \{\Omega_{\mathbf{F} B}\} \qquad P_1' \in \{\underline{z} \mid (\underline{z} : \mathbf{F} B') \in \Delta\} \cup \{\Omega_{\mathbf{F} B'}\}$$

  and also have some $(N_2, N_2') \in \mathcal{R}[\![\mathbf{F}\, A]\!]^{\Delta, \underline{y}:\mathbf{F} A}$. By the inductive hypothesis, there are two cases to consider:

- We have $N_2 \equiv \langle V \rangle \equiv N_2'$ for some closed V. Then for each of the pairs in the second case of Definition 5.3.3 we have

$$M \equiv \langle V \rangle \equiv M'$$

where we use the garbage collection rule and the fact that V is closed to remove the **need** bindings.

- Both N_2 and N_2' are trivially diverging, which implies that

$$N_2 \equiv Q_1 \;\mathbf{to}\; z.\, Q_2 \qquad N_2' \equiv Q_1' \;\mathbf{to}\; z'.\, Q_2'$$

for some

$$Q_1 \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B_1) \in \Delta\} \cup \{\Omega_{\mathbf{F}\, B_1}\} \qquad Q_1' \in \{\underline{z} \mid (\underline{z} : \mathbf{F}\, B_1') \in \Delta\} \cup \{\Omega_{\mathbf{F}\, B_1'}\}$$

We show that M is trivially diverging (M' is trivially diverging by an identical proof). By looking at the four pairs in the second case of Definition 5.3.3, we know that M has one of two forms. The first is

$$M \;\equiv\; (P_1 \;\mathbf{to}\; x.\, P_2) \;\mathbf{need}\; \underline{y}.\, Q_1 \;\mathbf{to}\; z.\, Q_2$$

In this case, if $Q_1 \neq \underline{y}$ then

$$M \;\equiv\; Q_1 \;\mathbf{to}\; z.\, (P_1 \;\mathbf{to}\; x.\, P_2) \;\mathbf{need}\; \underline{y}.\, Q_2$$

and if $Q_1 = \underline{y}$ then

$$M \;\equiv\; P_1 \;\mathbf{to}\; x.\, P_2 \;\mathbf{to}\; z.\, Q_2[\underline{y} \mapsto \langle z \rangle]$$

both of which are trivially diverging. The second form of M is

$$M \;\equiv\; (Q_1 \;\mathbf{to}\; z.\, Q_2)[\underline{y} \mapsto P_1 \;\mathbf{to}\; x.\, P_2]$$

In this case, if $Q_1 \neq \underline{y}$ then

$$M \;\equiv\; Q_1 \;\mathbf{to}\; z.\, (Q_2[\underline{y} \mapsto (P_1 \;\mathbf{to}\; x.\, P_2)])$$

and if $Q_1 = \underline{y}$ then

$$M \;\equiv\; P_1 \;\mathbf{to}\; x.\, P_2 \;\mathbf{to}\; z.\, Q_2[\underline{y} \mapsto (P_1 \;\mathbf{to}\; x.\, P_2)]$$

both of which are again trivially diverging. □

Finally, we can put the last few lemmas together to show that the logical relation matches the equational theory on closed returners of ground type.

Corollary B.3.9 If G is a ground type and $(M, M') \in \mathcal{R}[\![\mathbf{F}\, G]\!]^{\diamond}$ then $M \equiv M'$.

Proof. Applying Lemma B.3.8 gives us two cases to consider:

- If we have $M \equiv \langle V \rangle \equiv M'$, then we are done.
- If both M and M' are both trivially diverging, then we apply Lemma B.3.6. □