Controlling Effects

Andrzej Filinski

May 1996

CMU–CS–96–119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:

Robert Harper, Co-Chair
John Reynolds, Co-Chair
Stephen Brookes
Gordon Plotkin, University of Edinburgh

This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", under Contract F19628-95-C-0050. The research was also partially sponsored by the National Science Foundation under Grant No. CCR-94-09997.

Keywords: monads, continuations, computational effects, monadic reflection, logical relations, Scheme, ML.

Abstract

Many computational effects, such as exceptions, state, or nondeterminism, can be conveniently specified in terms of *monads*. We investigate a technique for uniformly adding arbitrary such effects to ML-like languages, without requiring any structural changes to the programs themselves. Instead, we use *monadic reflection*, a new language construct for explicitly converting back and forth between representations of effects as *behavior* and as *data*.

Using monadic reflection to characterize concisely all effects expressible with a given monad, we can give a precise meaning to the notion of *simulating* one effect by another, more general one. We isolate a simple condition allowing such a simulation, and in particular show that any monadic effect can be simulated by a continuation monad. In other words, under relatively mild assumptions on the base language (allowing formation of a suitably large answer type), control becomes a *universal effect*.

Concluding the development, we show that this universal effect can itself be explicitly implemented in terms of only standard first-class continuations (call/cc) and a piece of global state. This means that we can specify an effect such as nondeterminism abstractly, in terms of result lists, then directly obtain from this description a nondeterministic-choice operator performing imperatively-implemented backtracking. We include a full realization of the general construction in Standard ML of New Jersey, and give several programming examples.

Acknowledgments

I wish to thank my advisors, Robert Harper and John Reynolds, for their guidance in refining my informal ideas into mathematically quantifiable form, and for suffering through too many early drafts. Without their insistence on rigor and concreteness, I might have finished sooner, but probably with an ugly hole or two in the formal development. Any flaws that may remain in this final version are, of course, my own. Thanks are also due to my other committee members, Stephen Brookes and Gordon Plotkin, who each contributed useful insights.

My thinking and intuition was deeply influenced by a number of graduate courses I attended at CMU, especially those taught by Stephen Brookes, Robert Harper, Peter Lee, Frank Pfenning, and John Reynolds; I found that each of them gave me something that I could put to good use in some aspect of this thesis.

Several people provided very helpful comments on a previous version of this work, including notably Olivier Danvy, Matthias Felleisen, Julia Lawall, Greg Morrisett, Amr Sabry, and Phil Wadler. Many others, too numerous to mention, provided bits and pieces of advice or insights that helped me get one detail or another just right.

For their prompt and effective help with various administrative hassles, I want to thank Sharon Burks, Catherine Copetas, Monika Lekuse, Linda Melville, and Karen Olack; they saved me countless hours of fighting with practical problems when I could least afford the time.

Finally, I am grateful to a number of friends who offered encouragement and support throughout this endeavor, including Lars Birkedal, Prasad Chalasani, Rowan Davies, Kevin Lynch, Karoline Malmkjær, Greg Morrisett, Mark Wheeler, Chi Wong, and most especially Olivier Danvy and Chen Lee. And, of course, my greatest thanks go to my parents for their unwavering faith in me.

Andrzej Filinski

Edinburgh, Scotland
May 1996

Contents

Chapter 1

Introduction

In this chapter, we cover some general background and motivation for the work described in this document. After a brief discussion of two popular conceptual models for programming with computational effects, we introduce a new approach that in many ways combines the best aspects of both. We give an informal overview of this approach and sketch the concrete results obtained in the following chapters.

1.1 Effects in functional languages

An important topic in the field of programming language semantics is the study of *computational effects*. Informally, an effect is any deviation from the intuitive characterization of a program fragment as representing a simple function from inputs to outputs. Examples are numerous, including such familiar concepts as partiality, exceptions, state, computational complexity, I/O, nondeterminism, and concurrency.

The treatment of effects is particularly interesting in the context of modern functional programming languages, such as Standard ML or Haskell. Such languages have relatively simple and tractable mathematical descriptions, amenable to a formal analysis. In fact, their basic model of computation is precisely the definition and evaluation of functions, as opposed to sequential execution of program instructions.

But even though most aspects of functional programming can indeed be usefully captured with this simple declarative model, the natural formulation of many non-trivial programming tasks still tends to involve occasional uses of "imperative" concepts – whether for convenience in expressing an algorithm, or for interaction with the outside world.

The challenge to the semanticist is thus to admit the possibility of effects, while retaining as many as possible of the appealing properties of functional programming. This problem, of course, is not new; but somewhat surprisingly, two distinct schools of thought have evolved on how best to proceed, exemplified by the treatment of effects in "purely functional" languages, such as Miranda or Haskell, versus "algorithmic functional" ones, such as Scheme or ML.

The remainder of this section briefly presents and contrasts these two approaches, then introduces the basic thesis underlying this dissertation: that we can integrate key ideas from each framework to obtain a model of specifying and using effects that combines the best of both worlds.

1.1.1 "Effects as data"

When the basic ideas of denotational semantics were originally proposed, a significant challenge was to demonstrate that a very abstract mathematical model of computation based on "pure" functions could adequately model some apparently very non-functional but (at least at that time) important constructs in existing programming languages. Perhaps the most striking example of this was the use of *continuation functions* for modeling unstructured control constructs (gotos) [SW74, Rey93]; simpler techniques sufficed for concepts such as state or exceptions.

There is a close similarity between the *metalanguage* of denotational semantics (a concise notation for specifying continuous functions between domains) and actual functional programming languages. This has the fortunate consequence that often an appropriately expressed denotational definition can be directly executed to get an interpreter for the defined language [Rey72]. And in fact, many of the techniques pioneered in denotational semantics were quickly adopted for functional programs that were not in any reasonable sense language processors.

Specifically, the denotational representation of almost every computational effect leads to a characteristic pattern or *style* in functional programs using that effect. For example, a global store can be modeled functionally by passing an additional store argument to every function, together with returning from each function the possibly updated store; the resulting specification is commonly said to be expressed in *state-passing style*.

Similarly, exceptions can be modeled by tagging every function return value as either "normal" or "exceptional"; the caller of a function must then explicitly check for and propagate exceptions (*exception-passing style*). And, perhaps best-known, *continuation-passing style (CPS)* passes to every function an explicit representation of the remainder of the computation, to be invoked on the result of that function [Rey72, Fis72, Plo75].

While these techniques for modeling computational effects all share a similar feel, they do differ substantially in the details. It was therefore a remarkable observation by Moggi that they could each be seen as a particular instance of a generic schema, parameterized by a *monad*, a simple concept from category theory [Mog89]. This meant that much of the theory of computational effects could be derived abstractly, without reference to any specific notion of effect.

Again, it did not take long for this idea to migrate from mathematical semantics to mainstream functional programming. Work by Wadler and others [Wad90, Wad92b, PW93] established *monadic style* as a practical technique for structuring purely functional programs in a way that could reasonably conveniently express both program-internal effects (exceptions, state, etc.) and external ones (foreign function interfaces and monadic I/O).

The benefits of a denotational specification of an effect are substantial: we get a concise yet very precise characterization of how a program fragment can behave. For example, in a language with exceptions and non-termination as the only effects, the meaning of an integer-returning computation could be a meta-language value of type $\mathbf{Z}_\perp \oplus X$ where X is some fixed domain of exception names.

This immediately tells us that evaluation of an expression can have only three possible outcomes: it either returns a number, raises an exception, or diverges. Thus, for example, a simple case analysis suffices to formally show that evaluating an expression twice is equivalent to evaluating it only once and duplicating the result – even if we interpose an arbitrary other computation between the two evaluations.

Analogous considerations apply to a purely functional program in exception-passing style: we can use standard reasoning principles for sum types to deduce properties of programs with exception-effects, again validating program transformations such as common-subexpression elimination.

On the other hand, the purely-functional approach is not without problems, which become particularly evident at larger scales. One such disadvantage is that programming with effects turns into an all-or-nothing choice: to add even the most innocuous effect, such as debugging output or a "gensym" facility for generating unique names, we may have to rewrite substantial parts of the program in effect-passing style.

Monads alleviate this inconvenience somewhat, by allowing the program to be structured uniformly, independently of what effects will eventually be present. Still, we need to explicitly re-express the underlying functional program in monadic style – sometimes after the fact, duplicating effort, and sometimes preemptively, in anticipation of possibly having to add effects in the future.

Of course, since the conversion into monadic style is easily mechanizable, we can always express programs in a more concise notation, and have them automatically expanded into monadic style – either explicitly as a source transformation, or implicitly by an interpreter. But such an approach is not without problems either: by interposing a translation phase for expanding monadic effects into their denotations, we are effectively defining an entirely new programming language.

And for writing any non-trivial programs in this new language, we will want all the conveniences commonly provided by a language environment: pattern-matching function definitions, a static type system (giving meaningful error messages), a module system, a standard library, etc. Thus, the practical effort involved may be much larger than what might be expected from only looking at the core translation equations.

A related, but logically distinct, problem is that monadic-style definitions impose a substantial overhead on execution, whether implemented interpretively or compiled. Even if effects are rare (which is one of the tenets of functional programming), the infrastructure required to support an occasional imperative construct imposes a uniform burden on the entire evaluation process.

For example, for exception-passing style, the specification demands that after every subcomputation that may raise an exception, we have to check for this possibility and either proceed normally, or propagate the exception to the rest of the computation. A simple realization of exception-passing could thus spend a large fraction of its time checking for conditions that only occur very rarely. A more sophisticated implementation, especially a translation-based one, may be able to eliminate some of those checks, but in general calls to "unknown" functions (passed as parameters or separately compiled) have to be explicitly guarded by a check for exceptional returns.

Any further improvements seem to require pushing the effect down into the language implementation itself, still providing a purely functional interface to the relevant operations, but implementing them more efficiently than what could be expressed explicitly in the language. In particular, several proposals have been made for direct implementations of state in languages like Haskell [PW93, LPJ95].

Unfortunately, such a "black-box" approach negates one of the main advantages of the denotational approach: the ability to represent effect meanings explicitly as ordinary, functional constructs that can be reasoned about directly within the language. And even more significantly, these more efficient implementations are only available for a select few effects; programmer-defined, application-specific monads cannot take advantage of any non-functional implementation techniques.

1.1.2 "Effects as behavior"

As noted in the previous section, there are some compelling advantages to the denotational approach to effects, but also some significant practical problems. Whether the former outweigh the latter is still a controversial issue, especially because there is an alternative way of treating effects in programs. This approach is often referred to by the rather loaded term of "impure" functional programming; we will generally use the more neutral names *behavioral* or *operational*.

The basis of this approach is that a program expressed in terms of function definitions and applications can still be given a very natural algorithmic reading. Specifically, the fundamental principle of applicative-order reduction (namely, reducing the argument part of a β-redex before performing the substitution) can be seen as specifying a particular sequencing of evaluations. And this sequencing can serve as a robust skeleton for organizing general computational effects.

Although this idea can be traced back to early Lisp [$\mathrm{M}^{+}62$], perhaps the prototypical functional language based on such an approach is Scheme [CR91]. The three key semantic differences distinguishing Scheme from a "purely functional" language are its call-by-value evaluation strategy, the presence of explicitly mutable state, and a feature known as *first-class continuations.*

These three characteristics make Scheme a very versatile language, but potentially significantly complicate reasoning about programs. The problem is not that call-by-value, state, or continuations are *inherently* particularly hard to reason about. After all, they have simple denotational counterparts, and any Scheme program can be relatively easily expanded into a "purely functional" one by a continuation-passing transform.

In fact, we do not need to explicitly translate at all; direct-style equational theories such as the computational λ-calculus are only slightly more complicated than $\beta\eta$-conversion [Mog89]. (The situation is complicated somewhat by dynamic creation of mutable cells, but those too can be dealt with [FH92].) It would thus seem that ease of reasoning about impure functional programs should be "within a constant factor" of that about pure programs.

But there is a more subtle reason why general Scheme programs can be much harder to analyze formally than effect-free ones: because the set of computational effects is effectively fixed at two low-level but very powerful operations, the natural programming style consists of encoding higher-level effect abstractions imperatively in terms of the available effects, rather than through explicit effect-passing.

For a simple example, consider parameterization. Suppose that at a few places in a program, perhaps spread across several separately compiled modules, we need access to a parameter supplied as part of the initial expression to be evaluated. The two basic approaches for achieving this are to pass the parameter around everywhere it might eventually be needed, or to store its value in a global cell and access it only where actually used.

If the program is sufficiently large, and the accesses to the parameter sufficiently infrequent, the second approach becomes the natural choice (indeed, probably the correct choice from a software-engineering perspective). But by encoding parameterization in terms of state, we have effectively failed to represent the important fact that the parameter is immutable.

That is, suppose the program were otherwise purely functional. Then in explicitly-parameterized style, we could easily argue that the phrase $f\,3\,p + f\,3\,p$ could always be safely replaced by $2 \times f\,3\,p$, even when f was an unknown function. But once we admit general state, the optimization is no longer automatically valid: we need to be able to inspect f, making sure that it does not change the global value of p, before we can eliminate the common subexpressions.

In other words, where the transformation was a simple equational property in a purely functional setting, it is at best only provable in a language with effects when f and all the functions it calls are known. The reason is that the program does not explicitly embody the specification that the global state can never be modified after its initialization.

Of course, in a purely functional language, the state-based solution would not be available at all. Or, more accurately, given that all effects in a pure language must be written out in full, parameterization would naturally be expressed as such, rather than through (functional) state-passing. That is, if we have to be explicit about effects anyway, we may as well be precise.

With control effects, the problem is of course compounded. Again, an imperative realization of effects such as exceptions, nondeterminism, or concurrency may well be practically preferable to its more declarative counterpart in explicit effect-passing style, but the price we pay is in loss of simple reasoning principles.

In fact, the problem is not only in analyzing programs *using* the effects: it is often challenging even to show formally that the implementation of the effect itself is correct. For example, in Scheme we can encode (the control aspects of) an ML-like exception facility in a few lines of code using `call/cc` and a "current handler" cell. But a proof that such an implementation actually agrees with the explicit exception-passing used in the formal definition of SML [MTH90] is by no means a trivial task.

Similar considerations apply to analyzing control-based implementations of backtracking [Hay87, HDM93] or concurrency [Wan80]: while the code may be short, elegant, and intuitively plausible, formally relating it to a more abstract denotational specification, such as success lists [Wad85] or resumptions [Sch86, Mog90], is often a serious undertaking.

One could thus say that it is in this sense that "purely functional" programs offer a pragmatic reasoning advantage over “impure” ones: by penalizing all effects equally, they do not discourage the use of precise, custom-tailored effect representations. Impure languages, on the other hand, leave us an unpleasant choice, trading off precision against verbosity: if the desired effect is not already explicitly available, we must decide whether to encode it in terms of some more general standard effect, or to rewrite the program in explicit effect-passing style.

1.1.3 A unified view

As we have seen, each of the approaches has its advantages and disadvantages. The question thus naturally arises, whether there might be a way to somehow combine the best features of both. In particular, would it be possible to set things up so that we could *think* (both formally and informally) in terms of precise, functional denotations, but *work* (both when writing and executing programs) with the concise, operational behaviors?

The main goal of this thesis is to answer this question affirmatively. We will see how it is indeed possible to take a purely functional *denotational specification* of any monadic effect and obtain from it a directly executable *operational implementation* using `call/cc` and state. In fact, we will be able to define functions for converting back and forth between denotational and behavioral views of the same effect with no loss of information.

The fundamental idea is to distinguish carefully between *transparent* and *opaque* representations of a computational effect. The transparent representation is the explicit, denotational one: a computation that may raise an exception is represented as an effect-free computation of a sum-typed result; a computation with state-effects is represented as a pure function from old state to result and new state; a nondeterministic computation is represented as a deterministic computation of a list of results; and so forth.

On the other hand, the opaque representation is effectively an abstract data type with two operations: we can construct a trivial computation out of a value, and we can sequence two computations, where one may depend on the outcome of the other. How these operations are realized depends on the particular notion of effects, of course. But when writing the bulk of a typical program, the opaque representation is all we need. For example, a program written in monadic style would mostly use abstract *unit* and *bind* operations for structuring, regardless of what they actually expanded to.

Only when we actually wish to perform an effect, such as raising an exception, accessing the store, or making a nondeterministic choice, do we need additional operations. For such explicit effect-manipulations, we introduce two additional operations, converting between transparent and opaque representations of an effect.

That is, given an explicit representation of the effect, such as a value representing a raised exception, a function modifying the state, or a list of possibilities, we can obtain from it the corresponding opaque representation, which can then be further combined with other opaque computations in the usual way.

Conversely, and equally importantly, from an opaque representation, we can recover its transparent counterpart. For example, to handle an exception, we explicitly examine the sum-based representation of a computation and perform the appropriate action in each case. Or to determine whether a nondeterministic subcomputation has at least one successful outcome, we check if its transparent, list-based representation is non-empty.

Although this distinction may not at first appear particularly profound or useful, we now sketch three crucial observations that together summarize the main contributions of the thesis (with each one roughly corresponding to a chapter):

1. Even though opaque computations are a priori simply another abstract datatype, the two operations of value-inclusion and sequencing are exactly what forms the effect-backbone of an "imperative functional" language such as Scheme or ML. In such a language, any subcomputation may have an effect. A *value* such as a constant or a lambda-abstraction is therefore a special case, which must be implicitly coerced into a general computation. This corresponds to the first operation on our ADT of opaque computations.

   Similarly, effects in compound computations are implicitly sequenced by the call-by-value evaluation order. For example, in an application $E_1 \, E_2$, first E_1 is evaluated to a value, then E_2, and finally the application is performed. Again, this corresponds to an explicit sequencing of opaque computations, where subsequent computations may depend on the values produced by earlier ones.

   With this view, then, the conversions between transparent and opaque representations of computations provide an *effect-introspective* capability in the language, exposing the underlying notion of effects when and only when it matters. That is, the two operators convert between computations as data and as behavior within a single setting, integrating the views of effects as either *being* or *happening*. The key requirement is that the two conversion operations must be (two-sided) inverses, so that no information is lost when switching between the two view.

   That is, by relaxing the relationship between transparent and opaque representations from their being *identical* to merely *isomorphic*, we have already gained something important: a model for programming in a convenient, concise ML-like language, with an intuitive imperative reading, yet at the same permitting equational reasoning about our programs as if they were written purely functionally, with explicit effect-passing. But we can actually go further:

2. Since the ultimate goal of reasoning about programs is to characterize their observable behavior, we actually have some freedom in choosing the opaque representation of effects, as long as we can guarantee that it properly tracks the transparent representation in all complete programs. In other words, we only need to ensure that the two representations are *observationally* isomorphic, whether or not they actually are *denotationally* so.

   More explicitly, in addition to the *canonical* opaque representation, which simply encapsulates the *specification monad* of the transparent representation, there may also be a *variant* opaque representation, based on a different *implementation monad*. Then, as long as we choose the implementation monad such that it successfully mimics its specification counterpart in all program contexts, we can still reason about programs as if opaque effects were directly represented by the specification monad.

This is of course a well-known property of abstract data types in general. In our case, however, the ADT operations of value inclusion and sequencing are implicitly invoked at every single subcomputation (whether it actually performs any effects or not) in addition to the explicit conversions between transparent and opaque representations. Thus, efficiency of the implementation becomes a significant concern.

We will consider several examples of such effect-simulations, and give a general characterization of the relationships the two monads must satisfy in order for one to act as the opaque representation of the other. But perhaps the most remarkable and useful such instance is that, under suitable assumptions, *any monad can be simulated by a continuation monad.* In particular, this means that no matter how apparently complex the transparent specification may be, it can be implemented uniformly by continuation-passing.

This further adds to the attractiveness of programming with monadic effects: we can still reason about our programs as if their operational behavior were realized by explicit effect-passing according to a (potentially computationally costly) declarative specification. Yet the actual implementation only needs to incur the relatively low (and fixed) cost of continuation-passing. And we can do better still:

3. Although we nominally have a way to simulate arbitrary monadic effects with continuation monads, we are still some way off from a full implementation of our hypothetical ML-like language with behavior-data duality for user-definable effects. We have shown that continuations are in a sense a *universal* effect, but we still need to actually exploit this property in practice.

   A key third step is therefore to note that the *variant* opaque representation of an effect is also the *canonical* opaque representation of the effect induced by the implementation monad. That is, we can define an ML-like language with a notion of native effects that directly corresponds to the continuation monads we use for implementing other monadic effects. Any language in the style of (1) above can then be directly embedded into this one language of *control effects.*

   Moreover, we can show the perhaps equally surprising result that our universal control-effect language can *itself* be embedded in a language with only Scheme-style first-class continuations and mutable global variables. This could be said to validate the informal claim in the Scheme *Rationale* for call/cc that most useful control abstractions can be implemented explicitly, without changing or extending the language itself [CR91].

   With this correspondence, we have effectively bridged the gap between the denotational and the operational view of effects: we can reason safely in terms of the former, but work in a practical, familiar programming language in terms of the latter. The general construction takes a non-trivial amount of work to develop and prove correct, but we only need to perform it once and for all, not once for every new effect we want to implement.

The presentation in this document is oriented towards call-by-value languages, which can take full advantage of point (1) above. Still, there is in principle no reason why (2) and

(3) could not also be exploited in a purely functional language; efficient implementations of effects are as important for Haskell-like languages as they are for Scheme-like ones. We will not develop the details of such an application, however.

1.2 Monads and monadic reflection

In this section, we give somewhat simplified introduction to monadic effects. In particular, we will assume that the monad under investigation represents the *only* computational effect in the language. The formal development in the next chapter considers a more general notion of computation, where a monad serves to introduce a new effect on top of potentially already existing ones. Although the basic idea is the same, the details become substantially more involved. For the moment, let us therefore ignore the possibility of effects other than the one being introduced.

Monads originate in category theory; like many such concepts, they have several equivalent definitions. For our purposes, the following variant (usually known as the *Kleisli triple* formulation) seems most convenient:

Definition 1.1 (preliminary) *A* monad $\mathbf{T}$ *in a functional language consists of the following:*

- *A type constructor* T-.
- *For any type* α*, a function* $\eta_\alpha : \alpha \to T\alpha$ *(the* unit *function at* α*).*
- *For any function* $f : \alpha_1 \to T\alpha_2$*, a function* $f^* : T\alpha_1 \to T\alpha_2$ *(the* extension *of* f*).*

These components must further satisfy the three monad laws*:*

$$f^* \circ \eta_\alpha = f \qquad \eta_\alpha^* = \mathsf{id}_{T\alpha} \qquad (f^* \circ g)^* = f^* \circ g^*$$

Remark 1.2 In category theory, the a monad is conventionally defined in terms of a functor T and natural transformations $\eta : \mathsf{Id} \to T$ and $\mu : T^2 \to T$ satisfying certain equalities [ML71, VI.1]. (In the context of functional programming, the corresponding operations are usually referred to as *map*, *unit*, and *join* [Wad92a].)

It is easy to see, however, that the two formulations are equivalent: every Kleisli triple $(T, \eta, \text{-}^*)$ determines a monad (T, η, μ) by

$$Tf = (\eta \circ f)^* \qquad \text{and} \qquad \mu_\alpha = \mathsf{id}_{T_\alpha}^* \,.$$

Conversely, every monad determines a Kleisli triple by

$$f^* = \mu_\beta \circ T(f)\,,$$

and moreover these assignments are inverses. In the following, we will therefore use the terms “Kleisli triple” and “monad” synonymously.

A simple syntactic variation on Kleisli triples, popularized by Wadler [Wad92b], uses a binary infix operator to denote application of an extended function, writing t `‘bind‘` f or $t \star f$ for our $f^* t$. This “continuation last” notation is usually preferable for writing actual functional programs in monadic style, but the formulation in the definition is more convenient for our purposes.

It should also be mentioned that our monads are properly called *strong monads* in category theory [Mog89], essentially because the f being extended need not be closed. (The monad laws must then also hold for open terms; the formal definition in the next chapter will reflect this.) We will use the "functional programming" rather than the "categorical" terminology throughout this document. ■

Monads provide a uniform framework for reasoning about *computational effects* (such as state, exceptions, or I/O) in applicative programming languages [Mog89, Mog91]. Informally, $\eta\, a$ represents a "pure" (i.e., effect-free) computation yielding a, while $f^* t$ represents the computation consisting of t's effects followed by an application of f to the result (if any) computed by t. A concrete instance may help clarify this:

Example 1.3 For any fixed type χ, the *monad of χ-carrying exceptions* is given by

$$T\alpha = \alpha + \chi, \qquad \eta = \lambda a.\,\mathsf{inl}\,a, \qquad f^* = \lambda t.\,\mathbf{case}\ t\ \mathbf{of}\ \mathsf{inl}\,a \Rightarrow f\,a \parallel \mathsf{inr}\,e \Rightarrow \mathsf{inr}\,e$$

Here, a computation of type α is either a value a of type α (the left summand), denoting a successful computation of a, or a value e of type χ (right summand), representing a specific failure. The unit and extension operations capture the expected operational behavior of exceptions; in particular, if evaluation of a function argument t raises an exception e, that exception is simply propagated without ever applying f. It is easy to check that these definitions do in fact satisfy the equations in Definition 2.15. ■

The use of monads for structuring purely functional *programs* – as opposed to language semantics – is by now quite commonplace [Wad92b, PW93]. Of course, those same structuring techniques can usually also be used with Scheme-like languages (only rarely do monadic-style programs rely on lazy evaluation in a fundamental way), but the benefits seem less clear: often a "mostly pure" program with a few isolated effects (e.g., a `gensym` or occasional output) is both more efficient *and* easier to understand at a glance than an equivalent "completely pure" program expressed in monadic style throughout.

There is a more interesting way, however, of explicitly using monads as a structuring tool for programs in "impure" functional languages, one that takes full advantage of an eager evaluation strategy instead of trying to ignore it. The study of this alternative is the main focus of this thesis. Specifically, our development is based on a simple functional language based on "Moggi's principle":

> Computations *of type α correspond to* values *of type $T\alpha$.*

As also noted by Moggi, this abstract correspondence principle can be embodied into a concrete language construct which we will call *monadic reflection* (by analogy to the more general notion of computational reflection [Smi82, WF88]). Specifically, we take:

Definition 1.4 (preliminary) *A* reflection *of a monad* **T** *in a language is given by two operators*

$$\frac{\Gamma \vdash V : T\alpha}{\Gamma \vdash \mu(V) : \alpha} \qquad \textit{and} \qquad \frac{\Gamma \vdash E : \alpha}{\Gamma \vdash [E] : T\alpha}$$

satisfying that for any expression $E : \alpha$ *(possibly with computational effects) and any* value $V : T\alpha$,

$$[E] \textit{ is a value}, \qquad \mu([E]) = E, \qquad \textit{and} \qquad [\mu(V)] = V$$

Although the presence of these two operators arises naturally from the monadic framework, little is generally said about their computational interpretation, let alone their usefulness in actual functional programming. As it turns out, however, monadic reflection provides exactly what we need to program with monadic effects without having to rewrite the code in monadic style.

In operational terms, for any value $V : T\alpha$, $\mu(V)$ *reflects* V as an "effectful" computation of type α: we can construct an explicit representation of the effect, then *perform* or *execute* it by passing it to $\mu(\text{-})$. Conversely, given a general computation $E : \alpha$, $[E]$ *reifies* it as the corresponding effect-free value of type $T\alpha$, which can then be further inspected and analyzed like any other inert piece of data.

Although it is possible to write programs using the reflection and reification operators directly, an actual programming language would typically define a collection of more convenient operations in terms of $\mu(\text{-})$ and $[\text{-}]$:

Example 1.5 Consider again the exception monad from Example 1.3. We can express the usual exception-raising construct directly as

$$\textbf{raise}\ E \overset{\text{def}}{=} \textbf{let}\ e = E\ \textbf{in}\ \mu(\mathsf{inr}\ e)$$

where E is an expression – typically just a value – of type χ. That is, we explicitly construct a right-tagged value in the explicit representation of computations, then pass it to $\mu(\text{-})$ to perform the effect.

Conversely, $[E]$ reifies a possibly exception-raising α-expression E into a value of type $\alpha + \chi$, so we can define an exception-handling construct like this:

$$\textbf{try}\ E_1\ \textbf{handle}\ e \Rightarrow E_2 \overset{\text{def}}{=} \textbf{case}\ [E_1]\ \textbf{of}\ \mathsf{inl}\ a \Rightarrow a \ [\!]\ \mathsf{inr}\ e \Rightarrow E_2$$

That is, if E_1 returns normally, E_2 is ignored, but if E_1 raises an exception, the handler E_2 is invoked with e bound to the exception data; a general pattern-matching **handle** construct as found in SML can easily be expressed in terms of this one. ■

Example 1.6 For any type σ, the *σ-state monad* is defined by:

$$T\alpha = \sigma \to \alpha \times \sigma, \qquad \eta = \lambda a.\,\lambda s.\,\langle a, s\rangle, \qquad f^* = \lambda t.\,\lambda s.\,\textbf{let}\ \langle a, s'\rangle = t\,s\ \textbf{in}\ f\,a\,s'$$

Here, a computation t is represented as a (pure) function accepting a current state s and returning a value a and a new state s'; an effect-free computation passes the state along without modifying or reading it; and the extension of f first evaluates t in the current state s and then $f\,a$ in the state s' resulting from evaluation of t.

Using reflection, we can define operators for updating and reading the state:

$$\begin{aligned} \mathsf{state} := E &\overset{\text{def}}{=} \textbf{let } v = E \textbf{ in } \mu(\lambda s.\, \langle \langle\rangle, v\rangle) \\ !\mathsf{state} &\overset{\text{def}}{=} \mu(\lambda s.\, \langle s, s\rangle) \end{aligned}$$

That is, $\mathsf{state} := E$ is the effect represented by a function replacing the state with the value of E (and returning $\langle\rangle$ as the result of the operation), while $!\mathsf{state}$ denotes the effect of reading the current state without modifying it. Neither of these definitions explicitly uses the reification operator. That one is only used implicitly at the top level: if E is a program with state effects, then

$$\textbf{run } E \overset{\text{def}}{=} \textbf{let } \langle a, s'\rangle = [E]\, s_0 \textbf{ in } a$$

is the result of evaluating E starting with an initial state s_0 and discarding the final state. A simple refinement is of course to permit the state to persist across a sequence of top-level evaluations, as, in the interactive read-eval-print loops of ML or Scheme.

More generally, if we take σ to be a whole *store* (a finite map from locations to values), we can define, for any mutable variable x,

$$\begin{aligned} x := E &\overset{\text{def}}{=} \textbf{let'}\ v \Leftarrow E \textbf{ in } \mu(\lambda s.\, \langle \langle\rangle, s\{\ell_x \leftarrow v\}\rangle) \\ !x &\overset{\text{def}}{=} \mu(\lambda s.\, \langle s\,\ell_x, s\rangle) \end{aligned}$$

where ℓ_x is the location corresponding to the cell x.

Note that the state-accessing operations export only a subset of the functionality of the state monad. To express general reflection/reification in the store case, we need access to a “first-class store” mechanism. This can actually be implemented reasonably efficiently using version trees, without requiring the whole store to be copied [JD88, Mor93], but it does impose some overhead. ■

The latter example illustrates that it may not always be feasible or desirable to export the full reflection/reification pair for a monad in a real programming language. Nevertheless, it will be important for analysis purposes to consider the fully general formulation of an effect in terms of $\mu(\text{-})$ and $[\text{-}]$, with any restrictions on accessible functionality viewed as purely pragmatic considerations.

This is not to trivialize such concerns, only to emphasize that they are an orthogonal issue. Reflection and reification expose *exactly* the range of effects expressible in the corresponding state-passing formulation – much as a traditional denotational semantics of a language with a store does not formally enforce that the store is used in a single-threaded way.

The exception and state monads by no means exhaust the interesting possibilities. Some other examples of simple monadic effects are listed in Table 1.1; we will encounter many of these in more detail later. And although this collection may still seem limited, we have not even considered all the combinations that encode multiple effects. For example, $T\alpha = \sigma \to (\alpha + \chi) \times \sigma$ represents computations with both exceptions and state.

(How to combine specifications of individual effects into composites is actually a non-trivial problem. Although we will not develop the details in full generality, the

Common name	Functor, $T\alpha$	Unit, η_α	Extension, f^*
Identity	α	$\lambda a.\, a$	$\lambda t.\, f\, t$
Partiality	$\alpha_\perp$	$\lambda a.\, \mathsf{up}\, a$	$\lambda t.\, \textbf{case}\ t\ \textbf{of}\ \mathsf{up}\, a \Rightarrow f\, a \mathrel{[\!]} \perp \Rightarrow \perp$
Exception	$\alpha + \chi$	$\lambda a.\, \mathsf{inl}\, a$	$\lambda t.\, \textbf{case}\ t\ \textbf{of}\ \mathsf{inl}\, a \Rightarrow f\, a \mathrel{[\!]} \mathsf{inr}\, e \Rightarrow \mathsf{inr}\, e$
State	$\sigma \to \alpha \times \sigma$	$\lambda a.\, \lambda s.\, \langle a, s \rangle$	$\lambda t.\, \lambda s.\, \textbf{let}\ \langle a, s' \rangle = t\, s\ \textbf{in}\ f\, a\, s'$
Environment	$\epsilon \to \alpha$	$\lambda a.\, \lambda e.\, a$	$\lambda t.\, \lambda e.\, f\, (t\, e)\, e$
Complexity	$\alpha \times \mathbf{N}$	$\lambda a.\, \langle a, 0 \rangle$	$\lambda \langle a, n \rangle.\, \textbf{let}\ \langle b, n' \rangle = f\, a\ \textbf{in}\ \langle b, n + n' \rangle$
List-nondeterm.	α^*	$\lambda a.\, [a]$	$\lambda [a_1, \ldots, a_n].\, f\, a_1 +\!\!+ \cdots +\!\!+ f\, a_n$
Set-nondeterm.	$\mathcal{P}_{\mathit{fin}}\alpha$	$\lambda a.\, \{a\}$	$\lambda \{a_1, \ldots, a_n\}.\, f\, a_1 \cup \cdots \cup f\, a_n$
Continuation	$(\alpha \to o) \to o$	$\lambda a.\, \lambda k.\, k\, a$	$\lambda t.\, \lambda k.\, t\, (\lambda a.\, f\, a\, k)$

Table 1.1: Some simple monads

incremental approach used in the next chapter to layer a new effect on top of an existing one illustrates the basic principle: we must refine the definition of a monad to explicitly account for the original effects in the new specification.)

1.3 Overview of the thesis

1. This *Introduction* presents some background material about computational effects and informally introduces the notion of *monadic reflection* as the bridge between the denotational and operational view of monadic effects.

2. In *Programming with Monadic Effects*, we first specify a simple functional *base language* with some notion of *ambient effects*, such as partiality. We then formally define monads in this setting and show how a monad **T** induces an extension of the base language with a new *focus effect.* We specify the semantics of this effect-enriched language by a simple but somewhat impractical *(definitional) monadic translation* back the original language, for which we already have a semantics. Our task in the remaining chapters will then be to devise and prove correct an alternative implementation of the extended language.

3. *Relating Effects* contains the main technical contribution of the thesis. We consider a *specification monad* **T** and an *implementation monad* **U**, and investigate when **U** can be said to simulate **T**. We first show how, given some data connecting **T** and **U**, we can define a *variant translation* from the **T**-extended language to the base language, using **U**-effects to perform **T**-effects (but without any correctness guarantees yet).

   After some technical preliminaries, setting up the proof context, we then introduce the concept of a *monad relation* between **T** and **U**, and show that given such a relation, the definitional and the variant translation agree on complete **T**-programs. In many cases we can obtain the required monad relation directly from existence of a *monad morphism* from **T** to **U**, but the general continuation-simulation of **T** with a continuation monad involves an additional twist to capture the parametricity properties of the “final answer” type.

We conclude the chapter by showing that the correspondence between monads and continuations also allows us to define the monadic reflection operators for **T** directly in terms of those for the continuation monad **U**. Thus, it suffices to provide an implementation of the language with continuations as the notion of focus effect.

4. Completing the construction, *Implementing Continuation-Effects* shows how the effects corresponding to a continuation monad can be embedded into a Scheme-like language. The proof can be broken into three distinct steps. First, we show that the monadic effects for continuations can be expressed in terms of a control abstraction called *composable continuations*, which can be further decomposed into three even simpler control operators.

   Second, we show that the distinction between ambient and focus effects introduced by the definitional monadic translation does not actually affect evaluation, thus leaving us to implement a language with a single level of effects. And third, we show that this language can be implemented by embedding in language with first-class continuations and state. We conclude by showing a concrete implementation of the construction and a few programming examples.

5. Finally, the *Conclusion* summarizes the results and outlines some promising directions for further work.

Chapter 2

Programming with Monadic Effects

In this chapter, we introduce a simple functional programming language that will serve as a concrete framework for the results and proofs throughout the thesis. We also formally define the notion of *monad* in this setting, and show how a monad allows us to systematically define a extended language with a new notion of effects.

2.1 The base language

2.1.1 Terminology

A *language* consists of a *syntax* L and a *semantics* $\mathcal{L}$. The syntax defines the sets of well-formed types and of well-typed terms of a given type by means of a *language signature*, i.e., a set of type constructors and (typed) term constructors from which language phrases are built up inductively.

The semantics assigns some notion of meaning to the terms. As a practical minimum, we expect a semantics to provide a notion of *program evaluation*, i.e., a partial function $\mathit{Eval}_{\mathcal{L}}$ from a suitable subset of L-terms (e.g., closed terms of base type) to some set of observable results, say natural numbers. An evaluation semantics induces a notion of *observational equivalence* on terms, where two terms are considered equivalent if they can be substituted for each other in any program context without changing the observable outcome of the program. It is easy to see that this relation on terms is in fact a congruence wrt. all term constructors of the language.

A *denotational semantics* provides more, namely a *model*. That is, for every type, a set of *meanings* of terms of that type, and to every term constructor, a meaning of the constructed term expressed as a function of the meanings of the subterms. In particular, this provides a notion of equality, denotational equivalence, where two terms are equal iff they denote the same element of the model; because of the compositionality requirement, denotational equivalence is likewise a congruence. Two terms may be observationally equivalent without being denotationally so, but it is usually simpler to reason about denotational equivalence.

We obtain an evaluation semantics from a denotational semantics by defining a function from the meanings of closed terms to observable results. For example, the denotation of a program could be an element of the flat domain $\mathbf{N}_{\perp}$ of lifted natural numbers; the induced evaluation semantics is then given simply by the evident partial function mapping every lifted natural number to itself, and undefined on the $\perp$-element of the domain. Two different semantics for a syntax (say, direct and continuation) may determine the same evaluation semantics (and hence the same notion of observational equivalence), but induce different notions of denotational equivalence.

We say that a language $(L, \mathcal{L})$ is a member of a *language class* (e.g., the class of lambda-calculi) if its signature contains some specified set of type constructors (product, function space, etc.) and term constructors (abstraction, application, etc.), and the meanings of these types and terms in $\mathcal{L}$ satisfy some equational constraints (congruences, β-conversion, etc.). Often we can prove a result for an entire class of languages by showing that it holds generically in any model of the equations.

2.1.2 The base syntax

We now present a concrete language, in which we will be doing most of the formal development. Its syntax and semantics are very similar to PCF [Plo77] (even more so to PCF with lifted types [Mit96], except that the effect structure is made more explicit. We call it *Effect-PCF*. We present the syntax and informal operational interpretation in this section, with a precise denotational semantics in the next.

Effect-PCF is somewhat more verbose than a typical practical programming language, because all computation sequencing is made explicit in the syntax. For example, in an application, both the function and the argument must be explicitly evaluated if they are not already values.

Although we could have worked in an ML-like CBV language directly, the general treatment of monadic effects becomes awkward when the sequencing is left implicit. The present formulation allows us to cleanly separate out the handling of effects from the "purely functional" structure (exponentials, products, etc.).

Moreover, there is a simple, effect-independent elaboration of a standard, ML-like syntax into Effect-PCF, so we can view the implicit sequencing of computations in call-by-value languages as merely convenient shorthand for the corresponding Effect-PCF terms. We will return to this elaboration in Section 2.1.6.

The base signature is displayed in Figure 2.1.

Type structure

The most notable characteristic of the syntax is the division of the types into two classes: *value types* (α), and a subset called *(generalized) computation types* (β). The operational significance of this division is to make the possibility of effects explicit in the types, separating *trivial* or *manifestly effect-free* from *serious* computations.

We make value types properly include computation types by taking one possibility for a value to be an unevaluated computation, represented by a value of type $^{\circ}\alpha$. For example, a closed expression of type ι will always be equivalent to a numeral; a general expression expected to yield a natural number, but which may diverge (or have some other effect) has type $^{\circ}\iota$.

Types:

$$\begin{array}{rcl} \alpha & ::= & \mathrm{a} \mid \iota \mid 1 \mid \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2 \mid \alpha_1 + \alpha_2 \mid \beta \\ \beta & ::= & {}^{\circ}\alpha \mid \alpha \to \beta \mid 1 \mid \beta_1 \overset{\mathbf{c}}{\times} \beta_2 \end{array}$$

Terms:

$$\begin{array}{rcl} M & ::= & x \mid \mathsf{z} \mid \mathsf{s}\,M \mid \mathsf{ifz}\,(M, M_1, x.\,M_2) \mid \langle\rangle \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\,M \mid \mathsf{snd}\,M \\ & & \mid \mathsf{inl}\,M \mid \mathsf{inr}\,M \mid \mathsf{case}\,(M, x_1.M_1, x_2.M_2) \mid \lambda x^{\alpha}.\,M \mid M_1\,M_2 \\ & & \mid {}^{\circ}M \mid \mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 \mid \mathsf{fix}_{\beta}\,M \end{array}$$

Typing: Γ is a type assignment $x_1{:}\,\alpha_1, \ldots, x_n{:}\,\alpha_n$ (with all x_i distinct).

$$\dfrac{(x{:}\,\alpha) \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \dfrac{}{\Gamma \vdash \mathsf{z} : \iota} \qquad \dfrac{\Gamma \vdash M : \iota}{\Gamma \vdash \mathsf{s}\,M : \iota}$$

$$\dfrac{\Gamma \vdash M : \iota \qquad \Gamma \vdash M_z : \alpha \qquad \Gamma, x{:}\,\iota \vdash M_s : \alpha}{\Gamma \vdash \mathsf{ifz}\,(M, M_1, x.\,M_2) : \alpha} \qquad \dfrac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\dfrac{\Gamma \vdash M_1 : \alpha_1 \qquad \Gamma \vdash M_2 : \alpha_2}{\Gamma \vdash \langle M_1, M_2 \rangle : \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2} \qquad \dfrac{\Gamma \vdash M : \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2}{\Gamma \vdash \mathsf{fst}\,M : \alpha_1} \qquad \dfrac{\Gamma \vdash M : \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2}{\Gamma \vdash \mathsf{snd}\,M : \alpha_2}$$

$$\dfrac{\Gamma \vdash M : \alpha_1}{\Gamma \vdash \mathsf{inl}\,M : \alpha_1 + \alpha_2} \qquad \dfrac{\Gamma \vdash M : \alpha_2}{\Gamma \vdash \mathsf{inr}\,M : \alpha_1 + \alpha_2}$$

$$\dfrac{\Gamma \vdash M : \alpha_1 + \alpha_2 \qquad \Gamma, x_1{:}\,\alpha_1 \vdash M_1 : \alpha \qquad \Gamma, x_2{:}\,\alpha_2 \vdash M_2 : \alpha}{\Gamma \vdash \mathsf{case}\,(M, x_1.M_1, x_2.M_2) : \alpha}$$

$$\dfrac{\Gamma, x{:}\,\alpha \vdash M : \beta}{\Gamma \vdash \lambda x^{\alpha}.\,M : \alpha \to \beta} \qquad \dfrac{\Gamma \vdash M_1 : \alpha \to \beta \qquad \Gamma \vdash M_2 : \alpha}{\Gamma \vdash M_1\,M_2 : \beta}$$

$$\dfrac{\Gamma \vdash M_1 : \beta_1 \qquad \Gamma \vdash M_2 : \beta_2}{\Gamma \vdash \langle M_1, M_2 \rangle : \beta_1 \overset{\mathbf{c}}{\times} \beta_2} \qquad \dfrac{\Gamma \vdash M : \beta_1 \overset{\mathbf{c}}{\times} \beta_2}{\Gamma \vdash \mathsf{fst}\,M : \beta_1} \qquad \dfrac{\Gamma \vdash M : \beta_1 \overset{\mathbf{c}}{\times} \beta_2}{\Gamma \vdash \mathsf{snd}\,M : \beta_2}$$

$$\dfrac{\Gamma \vdash M : \alpha}{\Gamma \vdash {}^{\circ}M : {}^{\circ}\alpha} \qquad \dfrac{\Gamma \vdash M_1 : {}^{\circ}\alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : {}^{\circ}\alpha_2}{\Gamma \vdash \mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : {}^{\circ}\alpha_2} \qquad \dfrac{\Gamma \vdash M : \beta \to \beta}{\Gamma \vdash \mathsf{fix}_{\beta}\,M : \beta}$$

Figure 2.1: Base signature, L_0

A computation of type $\alpha \rightarrow \beta$ is often more conveniently thought of as β-computation *parameterized* by an α-value, rather than as a function from values to computations. In particular, when β is itself an arrow type, no actual evaluation occurs until all the parameters are present.

Similarly, a computation of type $\beta_1 \overset{\mathrm{c}}{\times} \beta_2$ can be viewed as a single computation implicitly parameterized by the choice of which component to evaluate, not as two unrelated computations. (In the case where β_1 and β_2 are the same type β, this is reflected in the isomorphism $\beta \overset{\mathrm{c}}{\times} \beta \cong (1 + 1) \to \beta$.)

There are actually two product-type constructors, one for value types and one for computation-types only. Although their typing and equational properties are identical (and in the standard semantics, they are even interpreted by the same cpo constructor), the two variants are logically distinct. Still, we generally omit the superscripts when it is clear which one is meant – i.e., when one of the factors is a value type, or the context requires a computation-type.

(Analogous consideration apply to the unit type, of course, but there we can simply assume that the two types are identical without unduly constraining the semantics.)

The set of value-types also includes a countable set of *type variables.* (There are no type variables for the computation-types.) When $\Delta = \{\mathrm{a}_1, \ldots, \mathrm{a}_n\}$ is a finite set of type variables, we write

$$\vdash_\Delta \alpha \ \mathsf{type} \qquad \text{and} \qquad \vdash_\Delta \beta \ \mathsf{ctype}$$

if all type variables occurring free in α and β are in Δ; in this case we say that α or β is a *type over* Δ. Clearly when $\Delta' \supseteq \Delta$ then also $\vdash_{\Delta'} \alpha \ \mathsf{type}$ and $\vdash_{\Delta'} \beta \ \mathsf{ctype}$. We say that α and β are *(type-)closed* when Δ is empty. A type over Δ determines a *type family* consisting of all types obtained by substituting closed types for type variables in Δ.

Note that there are no constructs within the language for explicitly binding type variables. (There will be, however, in an extension of L_0 with recursively-defined types in Section 3.2.1.)

The canonical model of the language is given by the category of "bottomless" cpos (predomains) and continuous functions, with computation-types interpreted by pointed cpos and the $^\circ$-operator on types corresponding to lifting. See Section 2.1.3 for details.

Terms

The term structure and associated typing rules are again mostly conventional. Similarly to the parameterization of types by type variables, we write

$$\Gamma \vdash_\Delta M : \alpha$$

if all type variables occurring free in Γ, M, and α are listed in Δ. M is then said to be a *term over* Δ; again it is called type-closed when Δ is empty, and the set of type-closed instances of a term over Δ forms a *term family.*

Complementing the $^\circ$-operator on types, there is a term operator $^\circ M$, which constructs an effect-free computation returning M, and $\mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2$, which constructs a computation consisting of evaluating M_1 and M_2 in sequence, with x in the second evaluation bound to the result of the first one. A computation is treated as a first-class object, and is not actually performed until its value is explicitly requested, either directly by a top-level program evaluation, or through evaluation of an enclosing **let**.

We generally omit the type tags in terms when they are clear from the context. Also, we will occasionally use pattern-matching syntax in let- and lambda-bindings, with the usual expansions, e.g.,

$$(\mathbf{let}^{\circ}\ \langle x_1, x_2\rangle \Leftarrow M\ \mathbf{in}\ M') \stackrel{\text{def}}{=} (\mathbf{let}^{\circ}\ x \Leftarrow M\ \mathbf{in}\ M'\{\mathsf{fst}\, x/x_1, \mathsf{snd}\, x/x_2\}) \quad {\scriptstyle (x \notin FV(M'))}$$

(Note that the projections are considered to be trivial by the typing rules, so the result of the substitution is still well-typed.)

The constructs associated with type ι allow us to program with natural numbers using a zero-constant, a successor function, and a combined zero-test/predecessor operation. Given general recursion, we can construct the standard arithmetic operations out of those primitives. For example, we can define addition as:

$$\mathsf{plus} : \iota \times \iota \to {}^{\circ}\iota = \lambda\langle n_1, n_2\rangle.\, \mathsf{fix}_{\iota \to {}^{\circ}\iota}\, (\lambda f.\, \lambda n.\, \mathsf{ifz}\, (n, {}^{\circ}n_2, n'.\, \mathbf{let}^{\circ}\ r \Leftarrow f\, n'\ \mathbf{in}\ {}^{\circ}(\mathsf{s}\, r)))\, n_1$$

Note, however, that because of the use of fix, the result of an addition is an ι-*computation*, even though the addition function happens to be total. (We could of course extend the language with additional primitives for arithmetic or a primitive-recursion construct, which could then be given pure value-types.)

Having fixed points at all computation-types β also allows us to express mutual recursion easily, as in:

$$\begin{aligned} &\mathsf{even} : \iota \to {}^{\circ}2 \\ &\quad = \mathsf{fst}\, (\mathsf{fix}_{(\iota \to {}^{\circ}2) \times (\iota \to {}^{\circ}2)}\, (\lambda\langle e, o\rangle.\, \langle \lambda n.\, \mathsf{ifz}\, (n, {}^{\circ}(\mathsf{inl}\, \langle\rangle), n'.\, o n'), \lambda n.\, \mathsf{ifz}\, (n, {}^{\circ}(\mathsf{inr}\, \langle\rangle), n'.\, e n')\rangle)) \end{aligned}$$

where $2 \stackrel{\text{def}}{=} 1 + 1$ is the type of Boolean values.

On the other hand, we cannot write down a term corresponding to a fixed point of the pure successor function; indeed, the type $\iota \to \iota$ is not even expressible in the language. (We *can* write $\mathsf{fix}_{{}^{\circ}\iota}\, (\lambda l^{{}^{\circ}\iota}.\, \mathbf{let}^{\circ}\ n \Leftarrow l\ \mathbf{in}\ {}^{\circ}(\mathsf{s}\, n)) : {}^{\circ}\iota$. Not surprisingly, this denotes a diverging computation in the intended interpretation of fix.)

Finally, we occasionally use the standard abbreviations:

$$\frac{}{\Gamma \vdash \mathsf{id}_{\beta} : \beta \to \beta} \qquad \frac{\Gamma \vdash M_1 : \alpha \to \beta_1 \qquad \Gamma \vdash M_2 : \beta_1 \to \beta_2}{\Gamma \vdash M_2 \circ M_1 : \alpha \to \beta_2}$$

$$\frac{}{\Gamma \vdash \bot_{\beta} : \beta} \qquad \frac{}{\Gamma \vdash \underline{n} : \iota}{\scriptstyle (n \in \mathbf{N})}$$

with $\mathsf{id}_{\beta} \stackrel{\text{def}}{=} \lambda x^{\beta}.\, x$, $M_2 \circ M_1 \stackrel{\text{def}}{=} \lambda x^{\alpha}.\, M_2\, (M_1\, x)$, $\bot_{\beta} \stackrel{\text{def}}{=} \mathsf{fix}_{\beta}\, \mathsf{id}_{\beta}$, and $\underline{n} \stackrel{\text{def}}{=} \mathsf{s}^{n}\, \mathsf{z}$.

2.1.3 A denotational semantics

A *program* in our base language is a closed term of type ${}^{\circ}\iota$; if in the semantics that term is equivalent to ${}^{\circ}\underline{n}$ for some n, the program denotes a successful computation with result n; otherwise, a diverging computation. A denotational semantics gives the meaning of a complete program by induction on its syntactic structure.

Preliminaries We give a simple model of the base language in the setting of *bottomless cpos* (also called *pre-domains*), i.e., complete partial orders not necessarily having a least element. For completeness, we review the associated terminology and constructions.

Definition 2.1 *A* cpo *A is a set equipped with an ω-complete partial order $\sqsubseteq$, i.e., such that every countable chain $a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ of elements in A has a least upper bound, written $\bigsqcup_i a_i$. A monotone (= order-preserving) function from A to A' is called* continuous *if for every chain $(a_i)_{i\in\omega}$ in A, $f(\bigsqcup_i a_i) = \bigsqcup_i f(a_i)$.*

A cpo B is called pointed *if it has a least element $\bot_B$, i.e., if $\bot_B \sqsubseteq b$ for every b in B. A function between pointed cpos B and B' is called* strict *if $f(\bot_B) = \bot_{B'}$. We use the name* domain *synonymously with pointed cpo.*

There are a number of standard cpos and cpo constructions:

- **base types**. Any set, such as the natural numbers $\mathbf{N}$, can be organized as a cpo by equipping it with the *discrete ordering*, $n \sqsubseteq_{\mathbf{N}} n'$ iff $n = n'$.
- **unit type**. The one-element set $1 = \{*\}$ is trivially a cpo. It is even degenerately pointed, with $\bot_1 = *$.
- **products**. $A_1 \times A_2$ is the *cartesian product* of cpos, ordered componentwise (i.e., $(a_1, a_2) \sqsubseteq_{A_1\times A_2} (a_1', a_2')$ iff $a_1 \sqsubseteq_{A_1} a_1'$ and $a_2 \sqsubseteq_{A_2} a_2'$). If B_1 and B_2 are pointed then so is $B_1 \times B_2$, with $\bot_{B_1\times B_2} = (\bot_{B_1}, \bot_{B_2})$.
- **sums**. $A_1 + A_2$ is the *disjoint union* of cpos (note: *not* the "separated sum" from standard domain-theoretic notation),

  $$A_1 + A_2 = \{(1, a_1) \mid a_1 \in A_1\} \cup \{(2, a_2) \mid a_2 \in A_2\}$$

  ordered inject-wise, i.e., $(i, a) \sqsubseteq_{A_1+A_2} (i', a')$ iff $i = i'$ and $a \sqsubseteq_{A_i} a'$. Such a cpo is in general not pointed, even if the summands are.
- **function space**. $A_1 \to A_2$ is the cpo of continuous functions from A_1 to A_2, with $f \sqsubseteq_{A_1\to A_2} f'$ if $\forall a \in A_1.\, f(a) \sqsubseteq_{A_2} f'(a)$. To minimize confusion with abstraction and application in the language, we write $\underline{\lambda} x.\, \varphi(x)$ and $f(a)$ for abstraction and application in the cpo model. $A \to B$ is pointed when B is, with $\bot_{A\to B} = \underline{\lambda} x.\, \bot_B$.
- **lifting**. For any cpo A, we define the *lifted* cpo,

  $$A_\bot = \{\{a\} \mid a \in A\} \cup \{\emptyset\}$$

  ordered such that $\{a\} \sqsubseteq_{A_\bot} \{a'\}$ if $a \sqsubseteq_A a'$, and $\emptyset \sqsubseteq \{a\}$ for any a. We write $\mathsf{up}(a)$ for $\{a\}$ and $\bot$ for $\emptyset$. Naturally, $A_\bot$ is pointed.

  The *strict extension* of a function $f : A \to A'_\bot$ is the function $f^\dagger : A_\bot \to A'_\bot$ given by $f^\dagger(\mathsf{up}(a)) = f(a)$ and $f^\dagger(\bot) = \bot$. More generally, for any pointed B and $f : A \to B$, $f^\ddagger : A_\bot \to B$ maps $\mathsf{up}(a)$ to $f(a)$ and $\bot_{A_\bot}$ to $\bot_B$.
- **reflexive types**. Finally, the CPO model allows us to construct solutions (up to isomorphism) to recursive type equations. While we will not need this immediately, it will become important in Section 3.2.1.

Given a finite set I, we write $\prod_{i \in I} A_i$ for the I-indexed product of cpos, i.e., the set of functions $\rho : I \to \bigcup_{i \in I} A_i$ such that for each $i \in I$, $\rho(i) \in A_i$. Such a function is usually called an *environment*. Environments can be naturally ordered pointwise, with $\rho \sqsubseteq \rho'$ iff $\forall i \in I.\, \rho(i) \sqsubseteq_{A_i} \rho'(i)$.

Finally, we write $\bullet$ for the empty environment; $\rho[i' \mapsto a] \in \prod_{i \in I \cup \{i'\}} A_i$ (where i' may or may not already be a member of I) for the function mapping i' to $a \in A_{i'}$ and every other $i \in I$ to $\rho(i)$; and $\rho \setminus i$ for the function ρ restricted to $I \setminus \{i\}$.

Base effects To give a semantics to our language in the predomain model, we first need to choose a notion of *base* or *ambient* effects, to be denoted by the computation-type constructor. The canonical example of such an effect is partiality, but the structure of the later proofs is largely independent of the exact choice; we only need to show that it satisfies a few simple relational properties.

We treat the case of partiality formally, and sketch how the setup generalizes to other ambient effects where appropriate. We do not develop the semantics of ambient effects in detail, however; where possible, it is more convenient to treat ambient effects more uniformly, using the monadic translations to be introduced a little later.

A valid reason for considering more complicated base effects, however, is to model language features that cannot be eliminated by a source-to-source transform. For example, "true" non-determinism (as opposed to a finitary variant, which can be defined by a backtracking transformation), can be modelled by a powerdomain or similar construct. Similarly, any notion of I/O operations or other extra-linguistic effects must somehow be accounted for in the semantics rather than at source level. We will not treat any of those formally, however.

Definition 2.2 *An* ambient-effect monad *for the cpo semantics is given by the following data:*

- *A cpo constructor $\mathcal{T}$, such that for any cpo A, $\mathcal{T}A$ is a* pointed *cpo.*
- *A family of continuous functions $\xi_A : A \to \mathcal{T}A$.*
- *An assignment to any continuous function $f : A \to \mathcal{T}A'$, a* strict *continuous function $f^\diamond : \mathcal{T}A \to \mathcal{T}A'$. This assignment must itself be continuous, i.e., satisfy the equality $(\bigsqcup_i f_i)^\diamond = \bigsqcup_i f_i^\diamond$.*

Further, the components must satisfy the three monad laws*:*

$$f^\diamond \circ \xi_A = f \qquad \xi_A^\diamond = \mathsf{id}_{\mathcal{T}A} \qquad f^\diamond \circ g^\diamond = (f^\diamond \circ g)^\diamond$$

Definition 2.3 *The* partiality semantics *is given by taking $\mathcal{T}A = A_\perp$ (pointed, as required), $\xi(a) = \mathsf{up}(a)$, and $f^\diamond = f^\dagger$ (strict by definition). It is easy to check that the monad laws hold for this triple.*

(Incidentally, the requirement that $f^\diamond$ be strict ensures that $\xi_A^\dagger : A_\perp \to \mathcal{T}A$ is a *monad morphism* from the partiality monad to $\mathcal{T}$. We will phrase this in more general terms in Definitions 2.11, 2.15, and 3.1, and in Example 3.2.)

Semantics of types To every well-formed $\vdash_\Delta \alpha$ type, we assign a cpo, and to every $\vdash_\Delta \beta$ ctype, a pointed cpo:

$$
\begin{array}{rcl}
\mathcal{L}\llbracket \mathrm{a} \rrbracket & = & \theta \mathrm{a} \\
\mathcal{L}\llbracket \iota \rrbracket & = & \mathbf{N} \\
\mathcal{L}\llbracket 1 \rrbracket & = & 1 \\
\mathcal{L}\llbracket \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2 \rrbracket & = & \mathcal{L}\llbracket \alpha_1 \rrbracket \times \mathcal{L}\llbracket \alpha_2 \rrbracket \\
\mathcal{L}\llbracket \alpha_1 + \alpha_2 \rrbracket & = & \mathcal{L}\llbracket \alpha_1 \rrbracket + \mathcal{L}\llbracket \alpha_2 \rrbracket \\
\mathcal{L}\llbracket \alpha \to \beta \rrbracket & = & \mathcal{L}\llbracket \alpha \rrbracket \to \mathcal{L}\llbracket \beta \rrbracket \\
\mathcal{L}\llbracket \beta_1 \overset{\mathbf{c}}{\times} \beta_2 \rrbracket & = & \mathcal{L}\llbracket \beta_1 \rrbracket \times \mathcal{L}\llbracket \beta_2 \rrbracket \\
\mathcal{L}\llbracket {}^\circ\alpha \rrbracket & = & \mathcal{T}(\mathcal{L}\llbracket \alpha \rrbracket)
\end{array}
$$

Semantics of terms To every well-typed $\Gamma \vdash_\Delta M : \alpha$, we assign an element $\mathcal{L}\llbracket M \rrbracket^\theta \in (\prod_{(x_i:\alpha_i)\in\Gamma} \mathcal{L}\llbracket \alpha_i \rrbracket^\theta) \to \mathcal{L}\llbracket \alpha \rrbracket^\theta$:

$$
\begin{array}{rcl}
\mathcal{L}\llbracket x \rrbracket(\rho) & = & \rho(x) \\
\mathcal{L}\llbracket \mathsf{z} \rrbracket(\rho) & = & 0 \\
\mathcal{L}\llbracket \mathsf{s}\, M \rrbracket(\rho) & = & \mathcal{L}\llbracket M \rrbracket(\rho) + 1 \\
\mathcal{L}\llbracket \mathsf{ifz}\,(M, M_z, x.\, M_s) \rrbracket(\rho) & = & \left\{ \begin{array}{ll} \mathcal{L}\llbracket M_z \rrbracket(\rho) & \text{when } \mathcal{L}\llbracket M \rrbracket(\rho) = 0 \\ \mathcal{L}\llbracket M_s \rrbracket(\rho[x \mapsto n]) & \text{when } \mathcal{L}\llbracket M \rrbracket(\rho) = n+1 \end{array} \right. \\
\mathcal{L}\llbracket \langle\rangle \rrbracket(\rho) & = & * \\
\mathcal{L}\llbracket \langle M_1, M_2 \rangle \rrbracket(\rho) & = & (\mathcal{L}\llbracket M_1 \rrbracket(\rho), \mathcal{L}\llbracket M_2 \rrbracket(\rho)) \\
\mathcal{L}\llbracket \mathsf{fst}\, M \rrbracket(\rho) & = & a_1 \text{ when } \mathcal{L}\llbracket M \rrbracket(\rho) = (a_1, a_2) \\
\mathcal{L}\llbracket \mathsf{snd}\, M \rrbracket(\rho) & = & a_2 \text{ when } \mathcal{L}\llbracket M \rrbracket(\rho) = (a_1, a_2) \\
\mathcal{L}\llbracket \mathsf{inl}\, M \rrbracket(\rho) & = & (1, \mathcal{L}\llbracket M \rrbracket(\rho)) \\
\mathcal{L}\llbracket \mathsf{inr}\, M \rrbracket(\rho) & = & (2, \mathcal{L}\llbracket M \rrbracket(\rho)) \\
\mathcal{L}\llbracket \mathsf{case}\,(M, x_1.M_1, x_2.M_2) \rrbracket(\rho) & = & \left\{ \begin{array}{ll} \mathcal{L}\llbracket M_1 \rrbracket(\rho[x_1 \mapsto a_1]) & \text{when } \mathcal{L}\llbracket M \rrbracket(\rho) = (1, a_1) \\ \mathcal{L}\llbracket M_2 \rrbracket(\rho[x_2 \mapsto a_2]) & \text{when } \mathcal{L}\llbracket M \rrbracket(\rho) = (2, a_2) \end{array} \right. \\
\mathcal{L}\llbracket \lambda x.\, M \rrbracket(\rho) & = & \underline{\lambda} a.\, \mathcal{L}\llbracket M \rrbracket(\rho[x \mapsto a]) \\
\mathcal{L}\llbracket M_1\, M_2 \rrbracket(\rho) & = & \mathcal{L}\llbracket M_1 \rrbracket(\rho)(\mathcal{L}\llbracket M_2 \rrbracket(\rho)) \\
\mathcal{L}\llbracket {}^\circ M \rrbracket(\rho) & = & \xi(\mathcal{L}\llbracket M \rrbracket(\rho)) \\
\mathcal{L}\llbracket \mathbf{let}^\circ\ x \Leftarrow M_1 \ \mathbf{in}\ M_2 \rrbracket(\rho) & = & (\underline{\lambda} a.\, \mathcal{L}\llbracket M_2 \rrbracket(\rho[x \mapsto a]))^\diamond(\mathcal{L}\llbracket M_1 \rrbracket(\rho)) \\
\mathcal{L}\llbracket \mathsf{fix}_\beta\, M \rrbracket(\rho) & = & \bigsqcup_i (\mathcal{L}\llbracket M \rrbracket(\rho))^i(\bot_{\mathcal{L}\llbracket \beta \rrbracket})
\end{array}
$$

(where for any $i \geq 0$, f^i is the i-th iterate of f, i.e., $f^0(a) = a$ and $f^{i+1}(a) = f(f^i(a))$).

Figure 2.2: Denotational semantics $\mathcal{L}^\theta_{(\mathcal{T},\xi,-^\diamond)}$ of the base language

---

Other examples can be easily adapted from the source-level monads to be presented in Section 2.2.3; for example, we obtain a notion of ambient state by taking $\mathcal{T}A = \mathbf{N} \to (A \times \mathbf{N})_\perp$, cf. Example 2.18. However, such L_0-definable ambient effects are more conveniently dealt with at the syntactic level, through an explicit monadic translation.

Although the only explicitly accessible effect in our base language is divergence (via fix), it is still useful to consider more general effect-structures in the semantics. For example, a continuation semantics may well be of interest even for a language that does not contain explicit control operators.

We can now give a denotational semantics of the base language (parameterized by the choice of effect structure) in Figure 2.2. Let Δ be a finite set of type variables, and θ be a mapping of type variables in Δ to cpos. The semantics then assigns to every Δ-type $\vdash_\Delta \alpha$, a cpo $\mathcal{L}[\![\alpha]\!]^\theta$ (pointed if α is computational), and to every Δ-term $\Gamma \vdash_\Delta M : \alpha$, a continuous function $\mathcal{L}[\![M]\!]^\theta$ from $\mathcal{L}[\![\Gamma]\!]^\theta = \Pi_{(x_i:\alpha_i)\in\Gamma}\mathcal{L}[\![\alpha_i]\!]^\theta$ to $\mathcal{L}[\![\alpha]\!]^\theta$. (We usually omit θ when it is clear from context. In particular, since there are no language constructs for binding type variables, θ stays constant throughout the semantic equations, and is omitted throughout the figure to reduce clutter.)

Although the denotational semantics thus assigns meanings to types and terms over arbitrary Δs (interpreting type variables by arbitrary cpos), for most purposes we will not use this generality; syntactic substitutions of closed types for type variables suffice. The only uses of the θ-parameterized semantics are in showing that a syntactic monad may be used to express a semantic one in Proposition 2.20, and when introducing recursively-defined types in Section 3.2.1.

2.1.4 Generalized let

Definition 2.4 *For any computation-type β of L_0, we define a derived term constructor $\mathbf{let}^\circ_\beta$, the* generalized let *with typing rule*

$$\frac{\Gamma \vdash M_1 : {}^\circ\alpha \qquad \Gamma, x{:}\,\alpha \vdash M_2 : \beta}{\Gamma \vdash \mathbf{let}^\circ_\beta\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 : \beta}$$

by induction on the structure of β:

$$\begin{aligned}
\mathbf{let}^\circ_{{}^\circ\alpha}\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 &= \mathbf{let}^\circ\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 \\
\mathbf{let}^\circ_1\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 &= \langle\rangle \\
\mathbf{let}^\circ_{\beta_1\times\beta_2}\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 &= \langle \mathbf{let}^\circ_{\beta_1}\, x \Leftarrow M_1 \;\mathbf{in}\; \mathsf{fst}\, M_2, \mathbf{let}^\circ_{\beta_2}\, x \Leftarrow M_1 \;\mathbf{in}\; \mathsf{snd}\, M_2 \rangle \\
\mathbf{let}^\circ_{\alpha\to\beta}\, x \Leftarrow M_1 \;\mathbf{in}\; M_2 &= \lambda a^\alpha.\, \mathbf{let}^\circ_\beta\, x \Leftarrow M_1 \;\mathbf{in}\; M_2\, a
\end{aligned}$$

In the predomain semantics (for any $\mathcal{T}$), two particular consequences of this definition are:

$$\mathcal{L}[\![\mathbf{let}^\circ_\beta\, x \Leftarrow M_1 \;\mathbf{in}\; M_2]\!](\rho) = \begin{cases} \mathcal{L}[\![M_2]\!](\rho[x \mapsto a]) & \text{when } \mathcal{L}[\![M_1]\!](\rho) = \xi(a) \\ \perp_{\mathcal{L}[\![\beta]\!]} & \text{when } \mathcal{L}[\![M_1]\!](\rho) = \perp \end{cases}$$

In the case of the partiality semantics, these are in fact the *only* two possibilities for $\mathcal{L}[\![M_1]\!](\rho)$. We thus obtain a natural generalization of the existing "strict extension"

let, in that M_2 can now be of a type interpreted by any pointed cpo, not necessarily a directly lifted one. Note, however, that we still need to restrict M_1 to be of type ${}^{\circ}\alpha$, not a general computation type: while *strictness* makes sense for functions $B_1 \to B_2$ between arbitrary pointed cpos, the *strict extension* operation can only extend a function $A \to B$ to $A_\bot \to B$, not "strictify" an arbitrary function $B_1 \to B_2$.

It is important to note the difference between, say, a computation returning a pair of integers:

$$\mathbf{let}^{\circ}_{{}^{\circ}(\iota\times\iota)}\; x \Leftarrow M\ \mathbf{in}\ {}^{\circ}\langle x, \underline{5}\rangle$$

(where M is evaluated exactly once, yielding a pair of numbers), and the construction of a pair of computations:

$$\mathbf{let}^{\circ}_{{}^{\circ}\iota\times{}^{\circ}\iota}\; x \Leftarrow M\ \mathbf{in}\ \langle {}^{\circ}x, {}^{\circ}\underline{5}\rangle$$

(where M is evaluated when *either* component of the result is requested). Similarly, we distinguish between the diverging computation of a function:

$$\mathbf{let}^{\circ}_{{}^{\circ}(\iota\to{}^{\circ}\iota)}\; y \Leftarrow \bot_{{}^{\circ}\iota}\ \mathbf{in}\ {}^{\circ}(\lambda x.\,{}^{\circ}\underline{3}) : {}^{\circ}(\iota\to{}^{\circ}\iota)$$

and a successful computation yielding a function which diverges for all inputs:

$${}^{\circ}(\mathbf{let}^{\circ}_{\iota\to{}^{\circ}\iota}\; y \Leftarrow \bot_{{}^{\circ}\iota}\ \mathbf{in}\ (\lambda x.\,{}^{\circ}\underline{3})) : {}^{\circ}(\iota\to{}^{\circ}\iota)$$

2.1.5 Equational properties

When the semantics $\mathcal{L}$ is fixed and clear from context, it is often preferable to reason about programs at the level of terms, rather than explicitly about their denotations in the semantics. More generally, we can often isolate a set of reasoning principles that hold for a large variety of interpretations, then check that our specific semantics $\mathcal{L}$ verifies those principles.

As mentioned before, for our purposes, it will suffice to consider equational properties of type-closed terms (i.e., with no free type variables), although the following should extend naturally to type and term families over a nonempty set of type variables.

Definition 2.5 *A* signature *L consists of a set of type and term constructors. An* interpretation *$\mathcal{L}$ of L assigns to every type $\vdash_\emptyset \alpha$* `type` *of L, a set* $\mathrm{Val}(\alpha)$*; and to every L-term $\Gamma \vdash_\emptyset M : \alpha$ and finite function ρ with $\forall (x_i\colon \alpha_i) \in \Gamma.\, \rho(x_i) \in \mathrm{Val}(\alpha_i)$, an element* $\mathrm{Int}(M)^\rho \in \mathrm{Val}(\alpha)$.

An equational theory *$\mathcal{E}$ for L is a set of typed equalities between (type-closed) L-terms, $\Gamma \vdash M = M' : \alpha$. A* model *of an equational theory is an interpretation that satisfies all the equations of the theory, i.e., whenever $\Gamma \vdash M = M' : \alpha$ is provable and ρ is a Γ-environment, then* $\mathrm{Int}(M)^\rho = \mathrm{Int}(M')^\rho$ *as elements of* $\mathrm{Val}(\alpha)$.

It is clear that the predomain semantics (for any notion of ambient effects $\mathcal{T}$) determines an interpretation of L_0, by taking $\mathrm{Val}(\alpha)$ as the set underlying $\mathcal{L}[\![\alpha]\!]$ and $\mathrm{Int}(M)^\rho$ as $\mathcal{L}[\![M]\!](\rho)$, forgetting continuity of $\mathcal{L}[\![M]\!]$ (as a function from environments to values). We present an equational theory $\mathcal{E}_0$ for L_0 and simultaneously argue that the predomain interpretation is a model of that theory. For particular classes of $\mathcal{T}$s, additional equations may be axiomatizable, for example that ambient effects are commutative or idempotent; we do not consider such extensions, however.

In most cases, the axioms listed below can be immediately verified by referring to the semantics; we often omit the details where they can be easily filled in. Also, since equality judgments are always about type-closed terms, we omit the implicit $\vdash_{\emptyset}$ in all typing assumptions in the rules.

Lemma 2.6 *In the semantics $\mathcal{L}$ of Figure 2.2, for any terms M and M', the following* weakening *and* substitution *principles hold:*

$$\mathcal{L}[\![M]\!](\rho) = \mathcal{L}[\![M]\!](\rho \setminus x) \quad \textit{if } x \notin FV(M)$$

$$\mathcal{L}[\![M\{M'/x\}]\!](\rho) = \mathcal{L}[\![M]\!](\rho[x \mapsto \mathcal{L}[\![M']\!](\rho)])$$

Proof. Routine, by induction on M. ■

Given this lemma, the verification of the following equations is straightforward.

Congruences, substitutions By the denotational assumption, our notion of equivalence is inherently a congruence wrt. all the term constructors of the language. We also have general principles of *closure under weakening and substitution*:

$$\frac{\Gamma \vdash M = M' : \alpha}{\Gamma, x : \alpha_1 \vdash M = M' : \alpha} \qquad \frac{\Gamma, x : \alpha_1 \vdash M = M' : \alpha \qquad \Gamma \vdash M_1 = M_1' : \alpha_1}{\Gamma \vdash M\{M_1/x\} = M'\{M_1'/x\} : \alpha}$$

which follow directly from Lemma 2.6:

$$\mathcal{L}[\![M]\!](\rho) = \mathcal{L}[\![M]\!](\rho \setminus x) = \mathcal{L}[\![M']\!](\rho \setminus x) = \mathcal{L}[\![M']\!](\rho)$$

$$\begin{aligned}&\mathcal{L}[\![M\{M_1/x\}]\!](\rho) = \mathcal{L}[\![M]\!](\rho[x \mapsto \mathcal{L}[\![M_1]\!](\rho)]) = \mathcal{L}[\![M']\!](\rho[x \mapsto \mathcal{L}[\![M_1']\!](\rho)]) \\ &\quad = \mathcal{L}[\![M'\{M_1'/x\}]\!](\rho)\end{aligned}$$

Natural numbers

$$\frac{\Gamma \vdash M_z : \alpha \qquad \Gamma, x{:}\,\iota \vdash M_s : \alpha}{\Gamma \vdash \mathsf{ifz}\,(\mathsf{z}, M_z, x.\, M_s) = M_z : \alpha} \qquad \frac{\Gamma \vdash M : \iota \qquad \Gamma \vdash M_z : \alpha \qquad \Gamma, x{:}\,\iota \vdash M_s : \alpha}{\Gamma \vdash \mathsf{ifz}\,(\mathsf{s}\,M, M_z, x.\, M_s) = M_s\{M/x\} : \alpha}$$

$$\frac{\Gamma \vdash M : \iota \qquad \Gamma, x{:}\,\iota \vdash M' : \alpha}{\Gamma \vdash \mathsf{ifz}\,(M, M'\{\mathsf{z}/x\}, x'.\, M'\{\mathsf{s}\,x'/x\}) = M'\{M/x\} : \alpha}$$

Unit type

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash M = \langle\rangle : 1}$$

(1 is a terminal object). Note that this equation means that there is only one *value* of type 1; there may well be different computations of that value, i.e., terms of type ${}^{\circ}1$. Already in the partiality case there are two such closed terms: termination (${}^{\circ}\langle\rangle$) and divergence ($\bot_{{}^{\circ}1}$).

Products

$$\frac{\Gamma \vdash M_1 : \alpha_1 \qquad \Gamma \vdash M_2 : \alpha_2}{\Gamma \vdash \mathsf{fst}\,\langle M_1, M_2\rangle = M_1 : \alpha_1}(+\mathit{symm}) \qquad \frac{\Gamma \vdash M : \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2}{\Gamma \vdash \langle \mathsf{fst}\,M, \mathsf{snd}\,M\rangle = M : \alpha_1 \overset{\mathbf{v}}{\times} \alpha_2}$$

$$\frac{\Gamma \vdash M_1 : \beta_1 \qquad \Gamma \vdash M_2 : \beta_2}{\Gamma \vdash \mathsf{fst}\,\langle M_1, M_2\rangle = M_1 : \beta_1}(+\mathit{symm}) \qquad \frac{\Gamma \vdash M : \beta_1 \overset{\mathbf{c}}{\times} \beta_2}{\Gamma \vdash \langle \mathsf{fst}\,M, \mathsf{snd}\,M\rangle = M : \beta_1 \overset{\mathbf{c}}{\times} \beta_2}$$

(Both are products in the categorical sense.) Although in the predomain semantics the two notions of products are interpreted by the same object, we do not actually require this in general.

Sums

$$\frac{\Gamma \vdash M : \alpha_1 \qquad \Gamma, x_1{:}\,\alpha_1 \vdash M_1 : \alpha \qquad \Gamma, x_2{:}\,\alpha_2 \vdash M_2 : \alpha}{\Gamma \vdash \mathsf{case}\,(\mathsf{inl}\,M, x_1.M_1, x_2.M_2) = M_1\{M/x_1\} : \alpha}(+\mathit{symm})$$

$$\frac{\Gamma \vdash M : \alpha_1 + \alpha_2 \qquad \Gamma, x{:}\,\alpha_1 + \alpha_2 \vdash M' : \alpha}{\Gamma \vdash \mathsf{case}\,(M, x_1.M'\{\mathsf{inl}\,x_1/x\}, x_2.M'\{\mathsf{inr}\,x_2/x\}) = M'\{M/x\} : \alpha}$$

(Sums are coproducts in the categorical sense.) Verification of the first law is immediate, given Lemma 2.6. For the second, we rely on the fact that $\mathcal{L}[\![M]\!](\rho)$ must be a value of the form (i, a); the equation would not be sound if $+$ were interpreted by, e.g., a separated sum and M denoted a diverging computation. A useful consequence of the above equations is that for any h (not necessarily denoting a strict function),

$$\begin{aligned}
&h\,(\mathsf{case}\,(M, x_1.M_1, x_2.M_2)) = h\,(\mathsf{case}\,(x, x_1.M_1, x_2.M_2))\{M/x\}\\
&\quad = \mathsf{case}\,(M, x_1.h\,(\mathsf{case}\,(\mathsf{inl}\,x_1, x_1.M_1, x_2.M_2)), x_2.h\,(\mathsf{case}\,(\mathsf{inr}\,x_2, x_1.M_1, x_2.M_2)))\\
&\quad = \mathsf{case}\,(M, x_1.h\,M_1, x_2.h\,M_2)
\end{aligned}$$

Function space

$$\frac{\Gamma, x{:}\,\alpha \vdash M_1 : \beta \qquad \Gamma \vdash M_2 : \alpha}{\Gamma \vdash (\lambda x^{\alpha}.\,M_1)\,M_2 = M_2\{M_1/x\} : \beta} \qquad \frac{\Gamma \vdash M : \alpha \to \beta}{\Gamma \vdash (\lambda x^{\alpha}.\,M\,x) = M : \alpha \to \beta}$$

(Categorically, $\alpha \to \beta$ is an exponentiation of α and β, but we do not require existence of exponentiations with arbitrary codomain types, so the category of types and terms is not quite a ccc.)

Computations

$$\frac{\Gamma \vdash M_1 : \alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : {}^{\circ}\alpha_2}{\Gamma \vdash (\mathbf{let}^{\circ}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ M_2) = M_2\{M_1/x\} : {}^{\circ}\alpha_2} \qquad \frac{\Gamma \vdash M : {}^{\circ}\alpha}{\Gamma \vdash (\mathbf{let}^{\circ}\ x \Leftarrow M\ \mathbf{in}\ {}^{\circ}x) = M : {}^{\circ}\alpha}$$

$$\frac{\Gamma \vdash M_1 : {}^{\circ}\alpha_1 \qquad \Gamma, x_1{:}\,\alpha_1 \vdash M_2 : {}^{\circ}\alpha_2 \qquad \Gamma, x_2{:}\,\alpha_2 \vdash M_3 : {}^{\circ}\alpha_3}{\Gamma \vdash (\mathbf{let}^{\circ}\ x_2 \Leftarrow (\mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3) = (\mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^{\circ}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3) : {}^{\circ}\alpha_3}$$

Each of these corresponds directly to one of the monad equations governing the ambient-effect monad $(\mathcal{T}, \xi, {}^{\diamond})$. For example, for the first one we verify:

$$\begin{aligned}
&\mathcal{L}[\![\mathbf{let}^{\circ}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ M_2]\!](\rho) = (\underline{\lambda} a.\,\mathcal{L}[\![M_2]\!](\rho[x \mapsto a]))^{\diamond}(\xi(\mathcal{L}[\![M_1]\!](\rho)))\\
&\quad = (\underline{\lambda} a.\,\mathcal{L}[\![M_2]\!](\rho[x \mapsto a]))(\mathcal{L}[\![M_1]\!](\rho)) = \mathcal{L}[\![M_2]\!](\rho[x \mapsto \mathcal{L}[\![M_1]\!](\rho)])
\end{aligned}$$

Fixed points

$$\frac{\Gamma \vdash M : \beta \rightarrow \beta}{\Gamma \vdash \mathsf{fix}_\beta\, M = M\,(\mathsf{fix}_\beta\, M) : \beta}$$

This suffices for evaluation, but for more general formal reasoning we will need additional properties; the details of this are covered in the next chapter. (Actually, none of the results in the thesis depend on the fixed-point equation being in $\mathcal{E}_0$, so in principle we could safely omit it without affecting correctness.)

We have thus established:

Proposition 2.7 *For any ambient-effect monad, the predomain semantics $\mathcal{L}_{(\mathcal{T},\xi,\diamond)}$ is a model of $\mathcal{E}_0$, the equational theory generated by the inference rules listed above.*

2.1.6 Encodings of implicitly-sequenced languages

In actual programming languages there is often no explicit syntactic or typing distinction between values and general terms. Rather, the grammar of types and terms is of the form:

$$\begin{array}{rcl}
\sigma & ::= & \iota \mid 1 \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \\
E & ::= & x \mid \mathsf{z} \mid \mathsf{s}\, M \mid \langle\rangle \mid \langle E_1, E_2\rangle \mid \mathsf{fst}\, E \mid \mathsf{snd}\, E \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \\
 & & \mid \mathsf{case}\,(E, x_1.E_1, x_2.E_2) \mid \lambda x^\sigma.\, E \mid E_1\, E_2 \mid \mathsf{fix}\, E
\end{array}$$

with trivial and serious computations a priori occupying the same type.

Our explicitly-sequenced syntax simplifies formal manipulation of programs, but it is somewhat inconvenient for actual programming. (However, in common practice, it is relatively uncommon to see, e.g., applications of the form $E_1\, E_2$ where one or both of E_1 and E_2 themselves have effects; in particular, evaluation of E_1 only very rarely has effects). Nevertheless, typical programs in ML-like languages do have some sequencing left implicit, and it would be too burdensome to force them to always be explicitly sequenced.

Fortunately, we can treat the more compact general syntax as merely shorthand for explicitly-sequenced terms, with programs first being *desugared* or *elaborated* into sequenced form, and only then given an operational or denotational semantics. We present two such elaborations, leading to either a call-by-value (CBV) or a call-by-name (CBN) interpretation of the implicitly-sequenced language.

CBV Translation on types. If σ is a type of the CBV language, $(\!(\sigma)\!)^{\mathbf{v}}$ is a type of L_0.

$$\begin{array}{rcl}
(\!(\iota)\!)^{\mathbf{v}} & = & \iota \\
(\!(1)\!)^{\mathbf{v}} & = & 1 \\
(\!(\sigma_1 \times \sigma_2)\!)^{\mathbf{v}} & = & (\!(\sigma_1)\!)^{\mathbf{v}} \overset{\mathbf{v}}{\times} (\!(\sigma_2)\!)^{\mathbf{v}} \\
(\!(\sigma_1 + \sigma_2)\!)^{\mathbf{v}} & = & (\!(\sigma_1)\!)^{\mathbf{v}} + (\!(\sigma_2)\!)^{\mathbf{v}} \\
(\!(\sigma_1 \rightarrow \sigma_2)\!)^{\mathbf{v}} & = & (\!(\sigma_1)\!)^{\mathbf{v}} \rightarrow {}^{\circ}(\!(\sigma_2)\!)^{\mathbf{v}}
\end{array}$$

Translation on terms. If $\Gamma \vdash E : \sigma$ in the source language, then $\llparenthesis \Gamma \rrparenthesis^{\mathsf{v}} \vdash \llparenthesis E \rrparenthesis^{\mathsf{v}} : {}^\circ\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}$ in L_0:

$$\begin{array}{rcl}
\llparenthesis x \rrparenthesis^{\mathsf{v}} & = & {}^\circ x \\
\llparenthesis \mathsf{z} \rrparenthesis^{\mathsf{v}} & = & {}^\circ \mathsf{z} \\
\llparenthesis \mathsf{s}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{s}\, x) \\
\llparenthesis \mathsf{ifz}\,(E, E_z, x.\, E_s) \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, n \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ \mathsf{ifz}\,(n, \llparenthesis E_z \rrparenthesis^{\mathsf{v}}, x.\, \llparenthesis E_s \rrparenthesis^{\mathsf{v}}) \\
\llparenthesis \langle\rangle \rrparenthesis^{\mathsf{v}} & = & {}^\circ\langle\rangle \\
\llparenthesis \langle E_1, E_2\rangle \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x_1 \Leftarrow \llparenthesis E_1 \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ \mathbf{let}^\circ\, x_2 \Leftarrow \llparenthesis E_2 \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ\langle x_1, x_2\rangle \\
\llparenthesis \mathsf{fst}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{fst}\, x) \\
\llparenthesis \mathsf{snd}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{snd}\, x) \\
\llparenthesis \mathsf{inl}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{inl}\, x) \\
\llparenthesis \mathsf{inr}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{inr}\, x) \\
\llparenthesis \mathsf{case}\,(E, x_1.E_1, x_2.E_2) \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, x \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ \mathsf{case}\,(x, x_1.\llparenthesis E_1 \rrparenthesis^{\mathsf{v}}, x_2.\llparenthesis E_2 \rrparenthesis^{\mathsf{v}}) \\
\llparenthesis \lambda x^{\sigma}.\, E \rrparenthesis^{\mathsf{v}} & = & {}^\circ(\lambda x^{\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}}.\, \llparenthesis E \rrparenthesis^{\mathsf{v}}) \\
\llparenthesis E_1\, E_2 \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, f \Leftarrow \llparenthesis E_1 \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ \mathbf{let}^\circ\, a \Leftarrow \llparenthesis E_2 \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ f\, a \\
\llparenthesis \mathsf{fix}_{\sigma}\, E \rrparenthesis^{\mathsf{v}} & = & \mathbf{let}^\circ\, F \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ {}^\circ(\mathsf{fix}_{\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}}\, (\lambda f.\, \mathbf{let}^\circ_{\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}}\, f' \Leftarrow F\, f \ \mathbf{in}\ f'))
\end{array}$$

where for the CBV fix, σ must be a functional type, so that $\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}$ is a computation-type. (We can actually also allow it to be a product of computation-types, if in the context of mutually-recursive definitions we interpret $\times$ as the computation-product $\mathring{\times}$.) The explicit $\mathbf{let}_{\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}}$ is necessary because the type of F is $\llparenthesis \sigma \to \sigma \rrparenthesis^{\mathsf{v}} = \llparenthesis \sigma \rrparenthesis^{\mathsf{v}} \to {}^\circ\llparenthesis \sigma \rrparenthesis^{\mathsf{v}}$.

When E is syntactically a value V, we have $\llparenthesis V \rrparenthesis^{\mathsf{v}} = {}^\circ M$ for some M. Thus, for example, we get validity of beta-value reduction because

$$\begin{array}{l}
\llparenthesis (\lambda x.\, E)\, V \rrparenthesis^{\mathsf{v}} = \mathbf{let}^\circ\, f \Leftarrow \llparenthesis \lambda x.\, E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ \mathbf{let}^\circ\, a \Leftarrow \llparenthesis V \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ f\, a \\
\quad = \mathbf{let}^\circ\, f \Leftarrow {}^\circ(\lambda x.\, \llparenthesis E \rrparenthesis^{\mathsf{v}}) \ \mathbf{in}\ \mathbf{let}^\circ\, a \Leftarrow {}^\circ M \ \mathbf{in}\ f\, a = (\lambda x.\, \llparenthesis E \rrparenthesis^{\mathsf{v}})\, M = \llparenthesis E \rrparenthesis^{\mathsf{v}}\{M/x\} \\
\quad = \llparenthesis E\{V/x\} \rrparenthesis^{\mathsf{v}}
\end{array}$$

Similarly, in general we have

$$\llparenthesis \lambda x.\, E\, x \rrparenthesis^{\mathsf{v}} = {}^\circ(\lambda x.\, \mathbf{let}^\circ\, f \Leftarrow \llparenthesis E \rrparenthesis^{\mathsf{v}} \ \mathbf{in}\ f\, x) =^{?} \llparenthesis E \rrparenthesis^{\mathsf{v}}$$

but when $\llparenthesis E \rrparenthesis^{\mathsf{v}} = {}^\circ M$ for some M, the equality does hold.

CBN A CBN interpretation gives a language essentially identical to PCF with product and sum types. The type translation is now:

$$\begin{array}{rcl}
\llparenthesis \iota \rrparenthesis^{\mathbf{n}} & = & {}^\circ\iota \\
\llparenthesis 1 \rrparenthesis^{\mathbf{n}} & = & 1 \\
\llparenthesis \sigma_1 \times \sigma_2 \rrparenthesis^{\mathbf{n}} & = & \llparenthesis \sigma_1 \rrparenthesis^{\mathbf{n}} \mathring{\times} \llparenthesis \sigma_2 \rrparenthesis^{\mathbf{n}} \\
\llparenthesis \sigma_1 + \sigma_2 \rrparenthesis^{\mathbf{n}} & = & {}^\circ(\llparenthesis \sigma_1 \rrparenthesis^{\mathbf{n}} + \llparenthesis \sigma_2 \rrparenthesis^{\mathbf{n}}) \\
\llparenthesis \sigma_1 \to \sigma_2 \rrparenthesis^{\mathbf{n}} & = & \llparenthesis \sigma_1 \rrparenthesis^{\mathbf{n}} \to \llparenthesis \sigma_2 \rrparenthesis^{\mathbf{n}}
\end{array}$$

Since the interpretation of every type is computational, we can form exponentials between any pair of types exponentials, and thus the source language forms a ccc. When the elaboration is composed with our predomain semantics, every type is interpreted by a (proper) domain; in fact, for a partiality semantics, this gives exactly the standard domain-theoretic model of PCF.

Translation on terms. If $\Gamma \vdash E : \sigma$ in the source language, then $\llparenthesis\Gamma\rrparenthesis^{\mathbf{n}} \vdash \llparenthesis E\rrparenthesis^{\mathbf{n}} : \llparenthesis\sigma\rrparenthesis^{\mathbf{n}}$ in L_0.

$$\begin{aligned}
\llparenthesis x\rrparenthesis^{\mathbf{n}} &= x \\
\llparenthesis \mathsf{z}\rrparenthesis^{\mathbf{n}} &= {}^{\circ}\mathsf{z} \\
\llparenthesis \mathsf{s}\,E\rrparenthesis^{\mathbf{n}} &= \mathbf{let}^{\circ}\; x \Leftarrow \llparenthesis E\rrparenthesis^{\mathbf{n}} \;\mathbf{in}\; {}^{\circ}(\mathsf{s}\,x) \\
\llparenthesis \mathsf{ifz}\,(E, E_{\mathrm{z}}, x.\,E_{\mathrm{s}})\rrparenthesis^{\mathbf{n}} &= \mathbf{let}^{\circ}_{\llparenthesis\sigma\rrparenthesis^{\mathbf{n}}}\; n \Leftarrow \llparenthesis E\rrparenthesis^{\mathbf{n}} \;\mathbf{in}\; \mathsf{ifz}\,(n, \llparenthesis E_{\mathrm{z}}\rrparenthesis^{\mathbf{n}}, x.\,\llparenthesis E_{\mathrm{s}}\rrparenthesis^{\mathbf{n}}\{{}^{\circ}x/x\}) \\
\llparenthesis \langle\rangle\rrparenthesis^{\mathbf{n}} &= \langle\rangle \\
\llparenthesis \langle E_1, E_2\rangle\rrparenthesis^{\mathbf{n}} &= \langle \llparenthesis E_1\rrparenthesis^{\mathbf{n}}, \llparenthesis E_2\rrparenthesis^{\mathbf{n}}\rangle \\
\llparenthesis \mathsf{fst}\; E\rrparenthesis^{\mathbf{n}} &= \mathsf{fst}\; \llparenthesis E\rrparenthesis^{\mathbf{n}} \\
\llparenthesis \mathsf{snd}\; E\rrparenthesis^{\mathbf{n}} &= \mathsf{snd}\; \llparenthesis E\rrparenthesis^{\mathbf{n}} \\
\llparenthesis \mathsf{inl}\; E\rrparenthesis^{\mathbf{n}} &= {}^{\circ}(\mathsf{inl}\; \llparenthesis E\rrparenthesis^{\mathbf{n}}) \\
\llparenthesis \mathsf{inr}\; E\rrparenthesis^{\mathbf{n}} &= {}^{\circ}(\mathsf{inr}\; \llparenthesis E\rrparenthesis^{\mathbf{n}}) \\
\llparenthesis \mathsf{case}\,(E, x_1.E_1, x_2.E_2)\rrparenthesis^{\mathbf{n}} &= \mathbf{let}^{\circ}_{\llparenthesis\sigma\rrparenthesis^{\mathbf{n}}}\; x \Leftarrow \llparenthesis E\rrparenthesis^{\mathbf{n}} \;\mathbf{in}\; \mathsf{case}\,(x, x_1.\llparenthesis E_1\rrparenthesis^{\mathbf{n}}, x_2.\llparenthesis E_2\rrparenthesis^{\mathbf{n}}) \\
\llparenthesis \lambda x^{\sigma}.\,E\rrparenthesis^{\mathbf{n}} &= \lambda x^{\llparenthesis\sigma\rrparenthesis^{\mathbf{n}}}.\,\llparenthesis E\rrparenthesis^{\mathbf{n}} \\
\llparenthesis E_1\,E_2\rrparenthesis^{\mathbf{n}} &= \llparenthesis E_1\rrparenthesis^{\mathbf{n}}\,\llparenthesis E_2\rrparenthesis^{\mathbf{n}} \\
\llparenthesis \mathsf{fix}_{\sigma}\; E\rrparenthesis^{\mathbf{n}} &= \mathsf{fix}_{\llparenthesis\sigma\rrparenthesis^{\mathbf{n}}}\; \llparenthesis E\rrparenthesis^{\mathbf{n}}
\end{aligned}$$

where we now have fixed points at all source types, including ι. Note also that numbers are still represented by a flat domain (as opposed to the *lazy natural numbers*, which also include partially-defined values, such as $\mathsf{s}\,\bot$).

2.2 Monads in a computational setting

In this section, we present a formal definition of monads, suitable for a language that already has a notion of ambient effects. This definition is phrased in terms of few basic concepts, which we need to introduce first.

2.2.1 A framework for effects

The ultimate goal of the line of research presented here is a framework for computational effects which makes it possible to describe effects in a modular way. Specifically, we want the ability to add effects *incrementally*: the resulting language is specified by a sequence of definitional translations, each one of which "translates away" one level of effects. For example, we can have a language with exceptions and state, specified as a composition of an exception-passing and a state-passing transform.

For now, however, we only consider the two-level case, with a notion of *ambient* effects (possibly already a combination of several primitive ones), specified by the "semantic" monad $\mathcal{T}$ used in the denotational semantics); and a *focus* effect, specified by a "syntactic" monad $\mathbf{T}$.

In order to define the notion of a monad that interacts in a suitable way with ambient effects, we need some amount of structure in the language. The following provides what we will need:

Definition 2.8 *We say that a language* $(L, \mathcal{L})$ *is a* computational lambda-language (cll) *if it has a class of* computation types β*, forming a (not necessarily proper) subset of all types* α*, and with the following properties:*

- *There are computations at any type, and the set of computation-types is closed under finite products and function spaces (with arbitrary domain):*

  $$\frac{\alpha\ \mathsf{type}}{{}^\circ\alpha\ \mathsf{ctype}} \qquad \frac{}{1\ \mathsf{ctype}} \qquad \frac{\beta_1\ \mathsf{ctype} \qquad \beta_2\ \mathsf{ctype}}{\beta_1 \overset{c}{\times} \beta_2\ \mathsf{ctype}} \qquad \frac{\alpha\ \mathsf{type} \qquad \beta\ \mathsf{ctype}}{\alpha \to \beta\ \mathsf{ctype}}$$

  We write $\vdash_\Delta \alpha$ type *and* $\vdash_\Delta \beta$ ctype *for types over a set of type variables* Δ*, but do not require* $\mathcal{L}$ *to assign any meaning to such types when* Δ *is nonempty.*

- *The syntax* L *includes at least the following terms and term constructors:*

  $$\frac{(x{:}\,\alpha) \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \frac{\Gamma \vdash M : \alpha}{\Gamma \vdash {}^\circ M : {}^\circ\alpha} \qquad \frac{\Gamma \vdash M_1 : {}^\circ\alpha \qquad \Gamma, x{:}\,\alpha \vdash M_2 : \beta}{\Gamma \vdash \mathbf{let}^\circ_\beta\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : \beta}$$

  (that is, we have variables, computation-inclusions, and a generalized let*), together with the term constructors for products and function spaces. Again, we write* $\Gamma \vdash_\Delta M : \alpha$ *for a term over* Δ*, not necessarily given a meaning by* $\mathcal{L}$*.*

- *In the semantics* $\mathcal{L}$*, the following equations hold (between type-closed terms):*

  $$\frac{\Gamma \vdash M_1 : \alpha \qquad \Gamma, x{:}\,\alpha \vdash M_2 : \beta}{\Gamma \vdash (\mathbf{let}^\circ_\beta\ x \Leftarrow {}^\circ M_1\ \mathbf{in}\ M_2) = M_2\{M_1/x\} : \beta} \qquad \frac{\Gamma \vdash M : {}^\circ\alpha}{\Gamma \vdash (\mathbf{let}^\circ_{{}^\circ\alpha}\ x \Leftarrow M\ \mathbf{in}\ {}^\circ x) = M : {}^\circ\alpha}$$

  $$\frac{\Gamma \vdash M_1 : {}^\circ\alpha_1 \qquad \Gamma, x_1{:}\,\alpha_1 \vdash M_2 : {}^\circ\alpha_2 \qquad \Gamma, x_2{:}\,\alpha_2 \vdash M_3 : \beta}{\begin{array}{l}\Gamma \vdash (\mathbf{let}^\circ_\beta\ x_2 \Leftarrow (\mathbf{let}^\circ_{{}^\circ\alpha_2}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3) \\ \qquad = (\mathbf{let}^\circ_\beta\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^\circ_\beta\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3) : \beta\end{array}}$$

  together with the congruence and substitution rules, as well as the axioms for unit, products, and functions (as listed in Section 2.1.5). And finally, the generalized let must satisfy (not necessarily directly by definition) the equations in Definition 2.4.

For example, in any cll, for every type α, there exists a (computation-)type $\alpha \to \alpha \to {}^\circ\alpha \times {}^\circ\alpha$. Note that this is a slightly stronger requirement than Moggi's T-exponentials [Mog89], which only guaranteed existence of all function spaces of the form $\alpha_1 \to {}^\circ\alpha_2$.

A weaker notion would be to take the computation types to be *exactly* the set of types of the form ${}^\circ\alpha$. However, requiring computation-types to be closed under products and (especially) function spaces will allow us to give a uniform treatment of definable computational effects.

Proposition 2.9 *Our base language (with any model $\mathcal{L}$ satisfying the equations in Section 2.1.5) can be organized as a cll by defining* $\mathbf{let}_\beta$ *inductively as in Definition 2.4.*

Proof. We only need to verify that the equational properties of the generalized let hold for the definition. The proof is a simple induction on β. We show two sample cases; the others are very similar.

$$\begin{aligned}
&\mathbf{let}^{\circ}_{\beta_1\times\beta_2}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ M_2 = \langle \mathbf{let}^{\circ}_{\beta_1}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ \mathsf{fst}\, M_2, \mathbf{let}^{\circ}_{\beta_2}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ \mathsf{snd}\, M_2\rangle \\
&\quad \overset{\text{ih}}{=} \langle \mathsf{fst}\, M_2\{M_1/x\}, \mathsf{snd}\, M_2\{M_1/x\}\rangle = \langle \mathsf{fst}\, M_2, \mathsf{snd}\, M_2\rangle\{M_1/x\} = M_2\{M_1/x\}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{let}^{\circ}_{\alpha\to\beta}\ x_2 \Leftarrow (\mathbf{let}^{\circ}_{{}^{\circ}\alpha_2}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3 \\
&\quad = \lambda a.\, \mathbf{let}^{\circ}_{\beta}\ x_2 \Leftarrow (\mathbf{let}^{\circ}_{{}^{\circ}\alpha_2}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3\, a \\
&\quad \overset{\text{ih}}{=} \lambda a.\, \mathbf{let}^{\circ}_{\beta}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^{\circ}_{\beta}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3\, a \\
&\quad = \lambda a.\, \mathbf{let}^{\circ}_{\beta}\ x_1 \Leftarrow M_1\ \mathbf{in}\ (\lambda a.\, \mathbf{let}^{\circ}_{\beta}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3\, a)\, a \\
&\quad = \lambda a.\, \mathbf{let}^{\circ}_{\beta}\ x_1 \Leftarrow M_1\ \mathbf{in}\ (\mathbf{let}^{\circ}_{\alpha\to\beta}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3)\, a \\
&\quad = \mathbf{let}^{\circ}_{\alpha\to\beta}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^{\circ}_{\alpha\to\beta}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3
\end{aligned}$$

■

Other ways of constructing computation-types may be possible, depending on the actual set of types available. For example, in a language with explicit polymorphism, it seems natural to take $\beta ::= \cdots \mid \forall \mathrm{a}.\, \beta$, with the generalized let extended accordingly.

For lack of a better name, we say that a computational λ-language is *effect-free* if α and ${}^{\circ}\alpha$ are actually the same type (with ${}^{\circ}M = M$); in this case, the cll requirements degenerate to those of a ccc. But effect-freeness should not be confused with existence of a type $\alpha \to \alpha'$ (with associated abstraction and application operations) for all α' – we can have the latter without the former.

(We do not actually work with any concrete effect-free languages; the concept is mainly used to show that various definitions and results reduce to their more familiar counterparts in the existing work on monads for computational effects.)

Remark 2.10 The essence of a generalized let at a computation-type β can be expressed simply as existence of the function

$$\zeta_\beta : {}^{\circ}\beta \to \beta \overset{\text{def}}{=} \lambda m.\, \mathbf{let}^{\circ}_{\beta}\ x \Leftarrow m\ \mathbf{in}\ x$$

satisfying the equations

$$x{:}\, \beta \vdash \zeta_\beta\, ({}^{\circ}x) = x : \beta$$

and

$$m{:}\, {}^{\circ}({}^{\circ}\beta) \vdash \zeta_\beta\, (\mathbf{let}^{\circ}\ x \Leftarrow m\ \mathbf{in}\ x) = \zeta_\beta\, (\mathbf{let}^{\circ}\ x \Leftarrow m\ \mathbf{in}\ {}^{\circ}(\zeta_\beta\, x)) : \beta$$

(In category-theoretic terms, this says that ζ_β is the structure map of an algebra (β, ζ) for the monad underlying ${}^{\circ}$ [ML71, VI.2].) Specifically, given such a function, we can define a generalized-let operator by

$$\mathbf{let}^{\circ}_{\beta}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 = \zeta_\beta\, (\mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ {}^{\circ}M_2)$$

However, the generalized-let formulation is more convenient to work with, its equational properties being a natural generalization of the existing **let**, as formalized in Definition 2.8. Remember also that our generalized let (or, equivalently, ζ) is characterized uniquely by Definition 2.4. ■

Among other applications, the generalized let-operation can be used to define a simple "effect-theoretic" generalization of strictness, which in turn plays a key role in the definition of layerable monads.

2.2.2 Rigidity

Definition 2.11 *We say that a term* $\Gamma \vdash M : \beta \to \beta'$ *in a cll is a* rigid *function between computation-types* β *and* β' *if*

$$\Gamma, m{:}\,{}^\circ\beta \vdash M\,(\mathbf{let}^\circ_\beta\; x \Leftarrow m \;\mathbf{in}\; x) = (\mathbf{let}^\circ_{\beta'}\; x \Leftarrow m \;\mathbf{in}\; M\,x) : \beta'$$

We write this as $\Gamma \vdash M : \beta \xrightarrow{\mathbf{r}} \beta'$*.*

Rigidity is a purely equational property; as such, we distinguish between *provable* rigidity (i.e., when the above equation is derivable in an equational theory) and *semantic* rigidity (when the equation holds in a model); the former implies the latter.

As an immediate consequence of the definition, we get:

Lemma 2.12 *An application of a rigid function can be "moved through" an arbitrary let-binding:*

$$\frac{\Gamma \vdash M : \beta \xrightarrow{\mathbf{r}} \beta' \qquad \Gamma \vdash M_1 : {}^\circ\alpha \qquad \Gamma, x{:}\,\alpha \vdash M_2 : \beta}{\Gamma \vdash M\,(\mathbf{let}^\circ_\beta\; x \Leftarrow M_1 \;\mathbf{in}\; M_2) = (\mathbf{let}^\circ_{\beta'}\; x \Leftarrow M_1 \;\mathbf{in}\; M\,M_2) : \beta'}$$

(i.e., the above is derivable in $\mathcal{E}_0$ *and hence true in any model.)*

Proof. Simple verification:

$$\begin{aligned}
M\,(\mathbf{let}^\circ_\beta\; x \Leftarrow M_1 \;\mathbf{in}\; M_2) &= M\,(\mathbf{let}^\circ_\beta\; x \Leftarrow M_1 \;\mathbf{in}\; \mathbf{let}^\circ_\beta\; y \Leftarrow {}^\circ M_2 \;\mathbf{in}\; y)\\
&= M\,(\mathbf{let}^\circ_\beta\; y \Leftarrow (\mathbf{let}^\circ_{{}^\circ\beta}\; x \Leftarrow M_1 \;\mathbf{in}\; {}^\circ M_2) \;\mathbf{in}\; y)\\
&= (M\,(\mathbf{let}^\circ_\beta\; y \Leftarrow m \;\mathbf{in}\; y))\{(\mathbf{let}^\circ_{{}^\circ\beta}\; x \Leftarrow M_1 \;\mathbf{in}\; {}^\circ M_2)/m\}\\
&=^\dagger (\mathbf{let}^\circ_{\beta'}\; y \Leftarrow m \;\mathbf{in}\; M\,y)\{(\mathbf{let}^\circ_{{}^\circ\beta}\; x \Leftarrow M_1 \;\mathbf{in}\; {}^\circ M_2)/m\}\\
&= \mathbf{let}^\circ_{\beta'}\; y \Leftarrow (\mathbf{let}^\circ_{{}^\circ\beta}\; x \Leftarrow M_1 \;\mathbf{in}\; {}^\circ M_2) \;\mathbf{in}\; M\,y\\
&= \mathbf{let}^\circ_{\beta'}\; x \Leftarrow M_1 \;\mathbf{in}\; \mathbf{let}^\circ_{\beta'}\; y \Leftarrow {}^\circ M_2 \;\mathbf{in}\; M\,y = \mathbf{let}^\circ_{\beta'}\; x \Leftarrow M_1 \;\mathbf{in}\; M\,M_2
\end{aligned}$$

where † marks the application of rigidity of M. ■

In particular, for any $M' : {}^\circ\alpha$ and *rigid* $M : {}^\circ\alpha \to {}^\circ\alpha'$,

$$M\,M' = M\,(\mathbf{let}^\circ\; x \Leftarrow M' \;\mathbf{in}\; {}^\circ x) = \mathbf{let}^\circ\; x \Leftarrow M' \;\mathbf{in}\; M\,({}^\circ x)$$

Operationally, this says that an argument to a rigid function can be evaluated before the call and the result coerced into a trivial computation, instead of the nominal CBN evaluation for parameters of °-type. This is usually a property associated with strictness. And indeed we have

Proposition 2.13 *In the predomain model (for any* $\mathcal{T}$*) of our base language, a rigid function is necessarily strict. In the particular case of the partiality semantics, the converse also holds, i.e., any strict function is rigid.*

Proof. First, it is easy to check the following equational reasoning principle:

$$\frac{\Gamma, x{:}\,\alpha \vdash M : \beta}{\Gamma \vdash (\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow \bot_{{}^\circ\alpha}\; \mathbf{in}\; M) = \bot_{\beta} : \beta}$$

(because for any f, $f^{\diamond}$ is strict, and $\mathcal{L}[\![\bot_\beta]\!](\rho) = \bot_{\mathcal{L}[\![\beta]\!]}$). Now, let $h : \beta \to \beta'$ be a rigid function. Then

$$h\,\bot_\beta = h\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow \bot_{{}^\circ 1}\; \mathbf{in}\; \bot_\beta) = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow \bot_{{}^\circ 1}\; \mathbf{in}\; h\,\bot_\beta = \bot_{\beta'}$$

Conversely, let h be strict and let $m : {}^\circ\beta$. When $\mathcal{T}A = A_\bot$, there are only two possibilities for m:

- $m = \bot_{{}^\circ\beta}$. Then
$$\begin{aligned} &h\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow m\; \mathbf{in}\; x) = h\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow \bot_{{}^\circ\beta}\; \mathbf{in}\; x) = h\,\bot_\beta = \bot_{\beta'} \\ &\quad = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow \bot_{{}^\circ\beta}\; \mathbf{in}\; h\,x = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow m\; \mathbf{in}\; h\,x \end{aligned}$$
- $m = {}^\circ b$ for some $b : \beta$:
$$\begin{aligned} &h\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow m\; \mathbf{in}\; x) = h\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow {}^\circ b\; \mathbf{in}\; x) = h\,b = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow {}^\circ b\; \mathbf{in}\; h\,x \\ &\quad = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow m\; \mathbf{in}\; h\,x \end{aligned}$$

■

In general, a function is rigid if it uses its argument exactly once, and before any other serious computation. But in the particular case of partiality, a function like $h = \lambda x.\, \bot$ ($= \lambda x.\, \mathbf{let}^{\circ}\; y \Leftarrow x\; \mathbf{in}\; \bot$ in the model) also qualifies as rigid, even though it does not explicitly reference its argument.

It is easy to check that identity and composition of two rigid functions are rigid; so are fst, snd, $\lambda f.\, f\,a$ for any a, and $\lambda x.\, \mathbf{let}^{\circ}\; a \Leftarrow x\; \mathbf{in}\; f\,a$ for any f. Likewise, if f_1 and f_2 are rigid, so is $\lambda x.\, \langle f_1\,x, f_2\,x\rangle$, and if $f\,a$ is rigid for every a, so is $\lambda x.\, \lambda a.\, f\,a\,x$. These are well-known properties of strictness, but also hold for general rigidity.

In the effect-free case (${}^\circ\alpha = \alpha$), every function is trivially rigid.

Remark 2.14 We can give an alternative, equivalent characterization of rigidity. There is a natural functorial action of ${}^\circ$, mapping a function $f : \beta \to \beta'$ to

$$f^{\sharp} : {}^\circ\beta \to {}^\circ\beta' = \lambda m.\, \mathbf{let}^{\circ}\; x \Leftarrow m\; \mathbf{in}\; {}^\circ(f\,x)$$

Further, recall from remark 2.10 that for any β, we can define

$$\zeta_\beta : {}^\circ\beta \to \beta = \lambda m.\, \mathbf{let}^{\circ}_{\beta}\; x \Leftarrow m\; \mathbf{in}\; x\,.$$

Then a function $f : \beta \to \beta'$ is rigid iff $\zeta_{\beta'} \circ f^{\sharp} = f \circ \zeta_\beta$ (i.e., if f is a morphism of the corresponding ${}^\circ$-algebras), because

$$\begin{aligned} &(\zeta_{\beta'} \circ f^{\sharp})\,m = \zeta_{\beta'}\,(\mathbf{let}^{\circ}\; x \Leftarrow m\; \mathbf{in}\; {}^\circ(f\,x)) = \mathbf{let}^{\circ}_{\beta'}\; y \Leftarrow (\mathbf{let}^{\circ}\; x \Leftarrow m\; \mathbf{in}\; {}^\circ(f\,x))\; \mathbf{in}\; y \\ &\quad = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow m\; \mathbf{in}\; \mathbf{let}^{\circ}_{\beta'}\; y \Leftarrow {}^\circ(f\,x)\; \mathbf{in}\; y = \mathbf{let}^{\circ}_{\beta'}\; x \Leftarrow m\; \mathbf{in}\; f\,x \end{aligned}$$

while

$$(f \circ \zeta_\beta)\,m = f\,(\mathbf{let}^{\circ}_{\beta}\; x \Leftarrow m\; \mathbf{in}\; x)\,.$$

■

2.2.3 Definable monads

The notion of a monad in a language $(L, \mathcal{L})$ consists of both a syntactic and a semantic aspect. Syntactically, we exhibit a type constructor T- and term families η and -* in L. Semantically, we establish that certain equational properties hold among these terms in $\mathcal{L}$ (but not necessarily in any particular equational theory for L). The separation is important – we will eventually have to consider interpretations of the η and * in a semantics where they do not necessarily satisfy the monad laws.

When $\vdash_\Delta \alpha$ type is a type over Δ in L and θ is a substitution of (closed) L-types for variables in Δ, $\alpha\{\theta\}$ is itself a (closed) type of L. In particular, a type constructor $F\mathrm{a} = \alpha$ (where a may occur in α) can be identified with a type schema $\vdash_{\{\mathrm{a}\}} \alpha$ type. Analogously, given a term $\Gamma \vdash M : \alpha$, θ determines a (type-closed) L-term $\Gamma\{\theta\} \vdash M\{\theta\} : \alpha\{\theta\}$.

We can now give a formal definition of a monad (in the Kleisli-triple formulation):

Definition 2.15 *Let L be a signature of a cll. A* monad-triple $\mathbf{T}$ *in L consists of the following items:*

- *A computation-type constructor, $\vdash_{\{\mathrm{a}\}} T\mathrm{a}$ ctype. We write $T\alpha$ for $T\mathrm{a}\{\alpha/\mathrm{a}\}$.*
- *A term family of* unit functions*, given as instances of a term $\vdash_{\{\mathrm{a}\}} \eta_\mathrm{a} : \mathrm{a} \to T\mathrm{a}$. We write η_α for $\eta_\mathrm{a}\{\alpha/\mathrm{a}\}$.*
- *A term family of* extension operators*, $f{:}\, \mathrm{a}_1 \to T\mathrm{a}_2 \vdash_{\{\mathrm{a}_1,\mathrm{a}_2\}} f^* : T\mathrm{a}_1 \to T\mathrm{a}_2$ (strictly speaking, type-indexed as above, but we always omit the type indices).*

Such a triple is an actual monad *in the cll $(L, \mathcal{L})$ if in $\mathcal{L}$ the following equations hold at all closed type instances:*

0. $f{:}\, \alpha_1 \to T\alpha_2 \vdash f^* : T\alpha_1 \xrightarrow{\mathsf{r}} T\alpha_2.$

1. $f{:}\, \alpha_1 \to T\alpha_2 \vdash f^* \circ \eta_{\alpha_1} = f : \alpha_1 \to T\alpha_2.$

2. $\vdash \eta^*_\alpha = \mathsf{id}_{T\alpha} : T\alpha \to T\alpha.$

3. $f{:}\, \alpha_1 \to T\alpha_2, g{:}\, \alpha_2 \to T\alpha_3 \vdash f^* \circ g^* = (f^* \circ g)^* : T\alpha_1 \to T\alpha_3.$

(Note that (0) is an equational condition like the others, because of its expansion in Definition 2.11. Conditions (0–3) also cover equations between non-variable terms, such as $M_1^*\,(\eta\, M_2) = M_1\, M_2$, because of closure under substitution of terms for variables.)

Actually the above definition is more akin to that of a *monad constructor* than of a simple monad; the necessary information for composition is implicit in the representation of the monad in the computational language. Nevertheless, we will refer to it as a monad over $\mathcal{L}$, since that is where the monad laws are required to hold – as opposed to being provable in some equational theory for L. Of course, showing the monad laws in the equational theory is sufficient to establish them for a model of that theory.

Note that condition (0) only makes sense because both $T\alpha_1$ and $T\alpha_2$ are required to be computation types. When $\mathcal{L}$ is effect-free, the rigidity requirement is vacuous, and the definition reduces to that of an ordinary monad.

Although it is important for our concrete language that rigidity implies strictness, the rigidity requirement for f^* is not merely present for domain-theoretic reasons; it is crucial for composing effects in general, and would be present even in a purely set-theoretic formulation of composable monads in a setting without general recursion. In practice, natural monad extensions always seem to be rigid anyway.

Example 2.16 (Identity) Perhaps the simplest possible monad, definable in any cll, is given by:

$$T\alpha = {}^\circ\alpha, \qquad \eta = \lambda a.\, {}^\circ a, \qquad f^* = \lambda t.\, \mathbf{let}^\circ\ a \Leftarrow t\ \mathbf{in}\ f\,a$$

The verification of the monad laws is straightforward. The identity monad is actually a degenerate case of many others; for example, we obtain it by specializing the exception monad below to $\chi = 0$ (a type with no values, hence no possibility of raising an exception) or the state monad to $\sigma = 1$ (a type with one value, hence an information-free state). ■

Example 2.17 (Exceptions) Let χ be some fixed type of *exception names* (exn in SML). We then obtain a monad by:

$$T\alpha = {}^\circ(\alpha + \chi), \qquad \eta = \lambda a.\, {}^\circ(\mathsf{inl}\, a), \qquad f^* = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e))$$

For completeness, we show the complete verification, since it is slightly more involved than for an exception monad over an effect-free language:

$$\begin{aligned}
& f^*\,(\mathbf{let}^\circ_{T\alpha_1}\ x \Leftarrow m\ \mathbf{in}\ x) = \mathbf{let}^\circ\ v \Leftarrow (\mathbf{let}^\circ_{{}^\circ(\alpha_1+\chi)}\ x \Leftarrow m\ \mathbf{in}\ x)\ \mathbf{in}\ \mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e)) \\
& \quad = \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^\circ\ v \Leftarrow x\ \mathbf{in}\ \mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e)) = \mathbf{let}^\circ_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f^*\,x
\end{aligned}$$

$$\begin{aligned}
& f^* \circ \eta = \lambda a.\, f^*\,(\eta\, a) = \lambda a.\, \mathbf{let}^\circ\ v \Leftarrow {}^\circ(\mathsf{inl}\, a)\ \mathbf{in}\ \mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e)) \\
& \quad = \lambda a.\, \mathsf{case}\,(\mathsf{inl}\, a, a.f\,a, e.{}^\circ(\mathsf{inr}\, e)) = \lambda a.\, f\,a = f
\end{aligned}$$

$$\eta^* = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.{}^\circ(\mathsf{inl}\, a), e.{}^\circ(\mathsf{inr}\, e)) = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ {}^\circ v = \lambda t.\, t = \mathsf{id}$$

$$\begin{aligned}
& g^* \circ f^* = \lambda t.\, g^*\,(f^*\,t) = \lambda t.\, g^*\,(\mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e))) \\
& \quad = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ g^*\,(\mathsf{case}\,(v, a.f\,a, e.{}^\circ(\mathsf{inr}\, e))) \\
& \quad = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.g^*\,(f\,a), e.g^*\,({}^\circ(\mathsf{inr}\, e))) \\
& \quad = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.g^*\,(f\,a), e.\mathbf{let}^\circ\ w \Leftarrow {}^\circ(\mathsf{inr}\, e)\ \mathbf{in}\ \mathsf{case}\,(w, b.g\,b, e.{}^\circ(\mathsf{inr}\, e))) \\
& \quad = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.g^*\,(f\,a), e.\mathsf{case}\,(\mathsf{inr}\, e, b.g\,b, e.{}^\circ(\mathsf{inr}\, e))) \\
& \quad = \lambda t.\, \mathbf{let}^\circ\ v \Leftarrow t\ \mathbf{in}\ \mathsf{case}\,(v, a.g^*\,(f\,a), e.{}^\circ(\mathsf{inr}\, e)) = (\lambda a.\, g^*\,(f\,a))^* = (g^* \circ f)^*
\end{aligned}$$

(Note that the type and term constructors are in the image of the CBN translation; thus exceptions also form a monad in a language like Haskell, where the language-level sum type is actually a "separated sum" in domain terminology.) ■

Example 2.18 (State) Let σ be any type. Then the *σ-state* monad is defined by:

$$T\alpha = \sigma \to {}^\circ(\alpha \times \sigma), \qquad \eta = \lambda a.\, \lambda s.\, {}^\circ\langle a, s\rangle, \qquad f^* = \lambda t.\, \lambda s.\, \mathbf{let}^\circ\ \langle a, s'\rangle \Leftarrow t\,s\ \mathbf{in}\ f\,a\,s'$$

Again, the verification is fairly simple:

$$\begin{aligned}
& f^*\,(\mathbf{let}^\circ_{T\alpha_1}\ x \Leftarrow m\ \mathbf{in}\ x) = \lambda s.\, \mathbf{let}^\circ\ \langle a, s'\rangle \Leftarrow (\mathbf{let}^\circ_{\sigma\to{}^\circ(\alpha_1\times\sigma)}\ x \Leftarrow m\ \mathbf{in}\ x)\,s\ \mathbf{in}\ f\,a\,s' \\
& \quad = \lambda s.\, \mathbf{let}^\circ\ \langle a, s'\rangle \Leftarrow (\mathbf{let}^\circ_{{}^\circ(\alpha_1\times\sigma)}\ x \Leftarrow m\ \mathbf{in}\ x\,s)\ \mathbf{in}\ f\,a\,s' \\
& \quad = \lambda s.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^\circ\ \langle a, s'\rangle \Leftarrow x\,s\ \mathbf{in}\ f\,a\,s' = \lambda s.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ f^*\,x\,s \\
& \quad = \mathbf{let}^\circ_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f^*\,x
\end{aligned}$$

$$\begin{aligned}
f \circ \eta &= \lambda a.\, f^*\,(\eta\, a) = \lambda a.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s'\rangle \Leftarrow [\lambda s.\, {}^{\circ}\langle a, s\rangle]\, s\ \mathbf{in}\ f\, a\, s' \\
&= \lambda a.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s'\rangle \Leftarrow {}^{\circ}\langle a, s\rangle\ \mathbf{in}\ f\, a\, s' = \lambda a.\, \lambda s.\, f\,(\mathsf{fst}\ \langle a, s\rangle)\,(\mathsf{snd}\ \langle a, s\rangle) = \lambda a.\, \lambda s.\, f\, a\, s \\
&= \lambda a.\, f\, a = f
\end{aligned}$$

$$\begin{aligned}
\eta^* &= \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s'\rangle \Leftarrow t\, s\ \mathbf{in}\ [\lambda a.\, \lambda s.\, {}^{\circ}\langle a, s\rangle]\, a\, s' = \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s'\rangle \Leftarrow t\, s\ \mathbf{in}\ {}^{\circ}\langle a, s'\rangle \\
&= \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ p \Leftarrow t\, s\ \mathbf{in}\ {}^{\circ}p = \lambda t.\, \lambda s.\, t\, s = \lambda t.\, t = \mathsf{id}
\end{aligned}$$

$$\begin{aligned}
g^* \circ f^* &= \lambda t.\, g^*\,(f^*\, t) = \lambda t.\, g^*\,(\lambda s.\, \mathbf{let}^{\circ}\ \langle a, s_1\rangle \Leftarrow t\, s\ \mathbf{in}\ f\, a\, s_1) \\
&= \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle b, s_2\rangle \Leftarrow (\mathbf{let}^{\circ}\ \langle a, s_1\rangle \Leftarrow t\, s\ \mathbf{in}\ f\, a\, s_1)\ \mathbf{in}\ g\, b\, s_2 \\
&= \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s_1\rangle \Leftarrow t\, s\ \mathbf{in}\ \mathbf{let}^{\circ}\ \langle b, s_2\rangle \Leftarrow f\, a\, s_1\ \mathbf{in}\ g\, b\, s_2 \\
&= \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s_1\rangle \Leftarrow t\, s\ \mathbf{in}\ g^*\,(f\, a)\, s_1 = \lambda t.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s_1\rangle \Leftarrow t\, s\ \mathbf{in}\ [\lambda a. g^*\,(f\, a)]\, a\, s_1 \\
&= (\lambda a. g^*\,(f\, a))^* = (g^* \circ f)^*
\end{aligned}$$

■

Although most practically useful monads over $(L_0, \mathcal{L})$ are actually monads in any model of the equational theory $\mathcal{E}_0$, there are two important reasons to only require the monad laws to hold with respect to specific interpretations. First, since the monad components may be defined using fix, it can be arbitrarily hard to show that a given monad-triple is actually a monad; certainly $\mathcal{E}_0$ alone will not always be sufficient. We only need $\mathcal{E}_0$ to validate a few equational properties that will be used frequently in the proofs later; the results do not rely on the monad laws for particular monads being provable in $\mathcal{E}_0$.

The second, and more fundamental, reason is that certain very useful notions of computation do not actually form monads in the presence of arbitrary ambient effects. Perhaps the best known such example [KW93] is the *list* monad $T\alpha = {}^{\circ}(\alpha\ \mathsf{list})$, used to model nondeterminism. It turns out to only be a monad if the ambient effects are *commutative*, i.e., if the equation

$$\mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^{\circ}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M = \mathbf{let}^{\circ}\ x_2 \Leftarrow M_2\ \mathbf{in}\ \mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M$$

(where neither x_i occurs free in an M_j) holds in $\mathcal{L}$. Partiality satisfies the above equation, but many other possible notions of ambient effects, such as state or continuations, do not. Other examples of "fragile" monads require the ambient effects to also be *idempotent*, a property shared by few effects other than partiality.

Thus, distinguishing between satisfaction of the monad laws in the equational theory and in a specific model (such as the partiality semantics) makes our results applicable to list-like monads as well as the "robust" ones (such as exceptions or state), that satisfy the monad laws for any notion of ambient effect.

Let us finally note that given a semantics that also assigns a meaning to type-open types and terms, a stronger definition of monad is possible:

Definition 2.19 *When* $\mathbf{T}$ *is a monad in* $(L, \mathcal{L})$ *where* $\mathcal{L}$ *is the predomain interpretation for any* $\mathcal{T}$*,* $\mathbf{T}$ *is said to be* uniform *if its equations also hold for type-open terms. That is, for each of the four monad laws* $\Gamma \vdash M_1 = M_2 : \alpha$ *in Definition 2.15, if we allow the types and terms to contain type variables from* Δ*,* θ *assigns a cpo to each* $\mathsf{a} \in \Delta$*, and* $\rho x_i \in \mathcal{L}[\![\alpha_i]\!]^{\theta}$ *for each* $(x_i\colon \alpha_i) \in \Gamma$ *then* $\mathcal{L}[\![M_1]\!]^{\theta}(\rho) = \mathcal{L}[\![M_2]\!]^{\theta}(\rho)$ *as elements of* $\mathcal{L}[\![\alpha]\!]^{\theta}$*.*

We can then state the simple consequence:

Proposition 2.20 *Let* $\mathbf{T}$ *be a uniform monad in the predomain semantics (for some notion of ambient effect). Then the following determines a* new *ambient-effect monad in the sense of Definition 2.2:*

$$\mathcal{T}A = \mathcal{L}[\![T\mathrm{a}]\!]^{\mathrm{a}\mapsto A} \qquad \xi_A = \mathcal{L}[\![\eta_{\mathrm{a}}]\!]^{\mathrm{a}\mapsto A}(\bullet) \qquad f^{\diamond} = \mathcal{L}[\![x^*]\!]^{\mathrm{a}_1\mapsto A_1,\mathrm{a}_2\mapsto A_2}(\bullet[x\mapsto f])$$

Proof. We first note that because computation-types were interpreted as pointed cpos by $\mathcal{L}$, $\mathcal{T}A$ is pointed as required. Similarly, because arrow-types are interpreted as continuous-function spaces, ξ_A and $f^{\diamond}$ are continuous; and because $\mathcal{L}[\![M]\!]$ is a continuous function from environments to values, so is the mapping $f \mapsto f^{\diamond}$. Finally, by Proposition 2.13, we get strictness of $f^{\diamond}$ from rigidity of x^*. The verification of the monad laws is also straightforward given uniformity of $\mathbf{T}$. ■

However, usually there is no need to modify the semantic characterization of ambient effects explicitly; we can define a language with a new notion of ambient effects via iterated monadic translation, in which case it is sufficient for the monad laws to hold only for type-closed instances.

2.3 Extending the language with effects

2.3.1 The monadic translation

We now show how a monad in a language allows us to define a new language with a richer set of computational effects.

Definition 2.21 *Let* $\mathbf{T} = (T, \eta, \text{-}^*)$ *be a monad-triple over a cll signature* L. *Then the signature* L^T *consists of* L *extended with a new computation-type constructor,*

$$\frac{\vdash_\Delta \alpha\ \mathsf{type}}{\vdash_\Delta \mathsf{'}\alpha\ \mathsf{ctype}}$$

and new term constructors:

$$\frac{\Gamma \vdash M_1 : {}^\circ\alpha_1 \quad \Gamma, x{:}\,\alpha_1 \vdash M_2 : \mathsf{'}\alpha_2}{\Gamma \vdash \mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : \mathsf{'}\alpha_2}$$

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash \mathsf{'}M : \mathsf{'}\alpha} \qquad \frac{\Gamma \vdash M_1 : \mathsf{'}\alpha_1 \quad \Gamma, x{:}\,\alpha \vdash M_2 : \mathsf{'}\alpha_2}{\Gamma \vdash \mathbf{let}\mathsf{'}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : \mathsf{'}\alpha_2}$$

$$\frac{\Gamma \vdash M : \mathsf{'}\alpha}{\Gamma \vdash \mathsf{[}M\mathsf{]} : T\alpha} \qquad \frac{\Gamma \vdash M : T\alpha}{\Gamma \vdash \mu(M) : \mathsf{'}\alpha}$$

(Note that we overload the syntactic construct $\mathbf{let}^\circ$ *to represent two distinct term constructors: the existing one, where* $M_2 : {}^\circ\alpha_2$ *and the new one defined above, where* $M_2 : \mathsf{'}\alpha_2$. *It will always be clear from context which one is meant.)*

There are now two basic notions of computation: the original ${}^{\circ}\alpha$ (e.g., partiality) and ${}^{\prime}\alpha$ which also includes T-effects (e.g., raising exceptions). As before, the set of computation-types is closed under products and function spaces.

Because we have extended the signature (rather than merely the unstructured set of types), every type constructor of L is still a type constructor of L^T. In particular, for any L^T-type α, $T\alpha$ is a well-formed L^T-type. Generalized let (still for binding results of ${}^{\circ}$-computations) is also definable at all computation-types, with the new clause for $\mathbf{let}^{\circ}_{{}^{\prime}\alpha}$ using the mixed-level let.

Reflection ($\mu(M)$) and reification ($\mathtt{[}M\mathtt{]}$) establish a correspondence between *opaque* and *transparent* representation of computations: opaque computations may only be constructed and sequenced using ${}^{\prime}$ and $\mathbf{let}^{\prime}$, while transparent ones may be manipulated using the full range of operations available on the type $T\alpha$, such as injections, case analysis, etc. See Example 2.26 below.

Together with the extension, we define a *canonical* or *definitional* translation of the extended signature back into the original one.

Since we will be dealing with several source-to-source translations, let us introduce the following shorthand:

Definition 2.22 (Translation convention) *When specifying a translation* $\llbracket\text{-}\rrbracket$ *from a signature* L *to* L' *that share a lot of operations, we generally omit clauses of the form*

$$\llbracket\varphi(X_1,\ldots,X_n)\rrbracket = \varphi(\llbracket X_1\rrbracket,\ldots,\llbracket X_n\rrbracket)$$

where the construct φ *in* L *is translated to the same-named construct in* L'. *(We do occasionally include selected clauses of this form for emphasis or clarity; but no formal distinction should be attached to whether a clause is included or not.)*

Definition 2.23 *The* monadic translation $\llbracket\text{-}\rrbracket_T$ *maps types and terms of* L^T *to their* L*-counterparts, such that:*

- *For any* $\vdash_\Delta \alpha$ type *in* L^T, $\vdash_\Delta \llbracket\alpha\rrbracket_T$ type *in* L.
- *For any* $\vdash_\Delta \beta$ ctype *in* L^T, $\vdash_\Delta \llbracket\beta\rrbracket_T$ ctype *in* L.
- *For any* $\Gamma \vdash_\Delta M : \alpha$ *in* L^T, $\llbracket\Gamma\rrbracket_T \vdash_\Delta \llbracket M\rrbracket_T : \llbracket\alpha\rrbracket_T$ *in* L.

The translation on types merely replaces ${}^{\prime}\alpha$ *with its definitional expansion:*

$$\llbracket{}^{\prime}\alpha\rrbracket_T \;=\; T\llbracket\alpha\rrbracket_T$$

Other type constructors are left intact, as are type variables (i.e., $\llbracket\mathrm{a}\rrbracket_T = \mathrm{a}$*). Similarly, the term translation expands away the new term constructors:*

$$\begin{aligned}
\llbracket{}^{\prime}M\rrbracket_T &= \eta\,\llbracket M\rrbracket_T\\
\llbracket\mathbf{let}^{\circ}\; x \Leftarrow M_1 \;\mathbf{in}\; M_2\,(:{}^{\prime}\alpha)\rrbracket_T &= \mathbf{let}^{\circ}_{T\llbracket\alpha\rrbracket_T}\; x \Leftarrow \llbracket M_1\rrbracket_T \;\mathbf{in}\; \llbracket M_2\rrbracket_T\\
\llbracket\mathbf{let}^{\prime}\; x \Leftarrow M_1 \;\mathbf{in}\; M_2\rrbracket_T &= (\lambda x.\,\llbracket M_2\rrbracket_T)^{*}\,\llbracket M_1\rrbracket_T\\
\llbracket\mu(M)\rrbracket_T &= \llbracket M\rrbracket_T\\
\llbracket\mathtt{[}M\mathtt{]}\rrbracket_T &= \llbracket M\rrbracket_T
\end{aligned}$$

with variables and other term constructors of L *translated into themselves (but with any type-annotations expanded according to the type translation).*

Although the monadic translation is simply a definitional extension, rather than a full syntactic transformation, we adopt the translation formulation to get an explicit syntactic handle on the expansion. In particular, when we later consider alternative ways of translating away the new constructs of L^T, it will be convenient to have an concise notation for referring to the different expansions.

Note also that because type and term variables are translated into themselves, the translations are *compositional* in the sense that

$$\llbracket \alpha\{\alpha'/\mathrm{a}\} \rrbracket_T = \llbracket \alpha \rrbracket_T \{\llbracket \alpha' \rrbracket_T/\mathrm{a}\} \quad \text{and} \quad \llbracket M\{M'/x\} \rrbracket_T = \llbracket M \rrbracket_T \{\llbracket M' \rrbracket_T/x\}\,.$$

In particular,

$$\llbracket T\alpha \rrbracket_T = \llbracket T\mathrm{a}\{\alpha/\mathrm{a}\} \rrbracket_T = \llbracket T\mathrm{a} \rrbracket_T \{\llbracket \alpha \rrbracket_T/\mathrm{a}\} = T\mathrm{a}\{\llbracket \alpha \rrbracket_T/\mathrm{a}\} = T\llbracket \alpha \rrbracket_T\,,$$

and likewise for the term translations of the monad components: $\llbracket \eta_\alpha \rrbracket_T = \eta_{\llbracket \alpha \rrbracket_T}$ and $\llbracket f^* \rrbracket_T = f^*$ (with the implicit type-tags on -^* appropriately translated).

The translation of the "mixed let" may need a little explanation. Consider the case where **T** is the state monad, and the base effect is partiality. Then if in the extended language, $\Gamma \vdash M_1 : {}^{\circ}\alpha_1$ (i.e., evaluation of M_1 may diverge, but has no state effects), $\llbracket M_1 \rrbracket_T : {}^{\circ}\llbracket \alpha_1 \rrbracket_T$ does not take a state argument, nor does it return a new state. On the other hand, when $\Gamma, x{:}\,\alpha \vdash M_2 : {}'\alpha_2$ (i.e., M_2 may both diverge and access the store), $\llbracket M_2 \rrbracket_T : \sigma \to {}^{\circ}(\llbracket \alpha_2 \rrbracket_T \times \sigma)$, so the translation of M_2 should be passed the current state, and the new state it returns is the state returned by the whole **let**-expression. The appropriate state-passing translation is therefore

$$\llbracket \mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 \rrbracket_T = \lambda s.\, \mathbf{let}^{\circ}\ x \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ \llbracket M_2 \rrbracket_T\, s$$

which is precisely what the generalized **let** expands to.

More generally, it is easy to check the following derived rule, where β is an L^T-computation type (i.e., may contain $'$):

$$\llbracket \mathbf{let}^{\circ}_{\beta}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 \rrbracket_T = \mathbf{let}^{\circ}_{\llbracket \beta \rrbracket_T}\ x \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ \llbracket M_2 \rrbracket_T$$

In the definitional translation, the opaque and transparent T-computations are represented by the same underlying L-type; consequently, the term translations for reflection and reification are trivial. Later, when we consider a different representation of effects, the two operators will have more interesting definitions.

This syntactic translation also determines a semantics:

Definition 2.24 *Given a semantics $\mathcal{L}$ for our base language L, we obtain a semantics $\mathcal{L}^T$ of the extended language L^T by taking*

$$\mathcal{L}^T\llbracket \alpha \rrbracket^\theta = \mathcal{L}\llbracket \llbracket \alpha \rrbracket_T \rrbracket^\theta \quad \textit{and} \quad \mathcal{L}^T\llbracket M \rrbracket^\theta = \mathcal{L}\llbracket \llbracket M \rrbracket_T \rrbracket^\theta$$

In fact, this semantics extends the standard monadic semantics for the new ambient-effect monad induced by **T**:

Proposition 2.25 *Let $\mathcal{L}$ be a predomain semantics of L_0 with some underlying ambient-effect monad, $\mathbf{T}$ a uniform monad in that semantics (Definition 2.19), and let $|\text{-}| : L_0 \to L_0^T$ be the syntactic transformation replacing every ${}^\circ$ in types and terms with $\text{\textbf{'}}$.*

Then $\mathcal{L}^T\llbracket|\text{-}|\rrbracket = \mathcal{L}_{\mathcal{T}}\llbracket\text{-}\rrbracket$ (for types and terms) where $\mathcal{L}_{\mathcal{T}}$ is the monadic semantics of L_0 for the ambient-effect monad $\mathcal{T}$ given by interpreting in $\mathcal{L}$ the components of $\mathbf{T}$, as shown in Proposition 2.20.

Proof. Induction on the structure of the types and terms. Most cases are immediate; for computations, we get:

$$\begin{aligned}
&\mathcal{L}^T\llbracket|{}^\circ\alpha|\rrbracket^\theta = \mathcal{L}^T\llbracket\text{\textbf{'}}|\alpha|\rrbracket^\theta = \mathcal{L}\llbracket\llbracket\text{\textbf{'}}|\alpha|\rrbracket_T\rrbracket^\theta = \mathcal{L}\llbracket T\llbracket|\alpha|\rrbracket_T\rrbracket^\theta = \mathcal{L}\llbracket(T\mathrm{a})\{\llbracket|\alpha|\rrbracket_T/\mathrm{a}\}\rrbracket^\theta\\
&\quad= \mathcal{L}\llbracket T\mathrm{a}\rrbracket^{\theta[\mathrm{a}\mapsto\mathcal{L}\llbracket\llbracket|\alpha|\rrbracket_T\rrbracket^\theta]} = \mathcal{L}\llbracket T\mathrm{a}\rrbracket^{\mathrm{a}\mapsto\mathcal{L}\llbracket\llbracket|\alpha|\rrbracket_T\rrbracket^\theta} \overset{\mathrm{ih}}{=} \mathcal{L}\llbracket T\mathrm{a}\rrbracket^{\mathrm{a}\mapsto\mathcal{L}_{\mathcal{T}}\llbracket\alpha\rrbracket^\theta} = \mathcal{T}(\mathcal{L}_{\mathcal{T}}\llbracket\alpha\rrbracket^\theta)\\
&\quad= \mathcal{L}_{\mathcal{T}}\llbracket{}^\circ\alpha\rrbracket^\theta
\end{aligned}$$

$$\begin{aligned}
&\mathcal{L}^T\llbracket|{}^\circ M|\rrbracket^\theta(\rho) = \mathcal{L}^T\llbracket\text{\textbf{'}}|M|\rrbracket^\theta(\rho) = \mathcal{L}\llbracket\llbracket\text{\textbf{'}}|M|\rrbracket_T\rrbracket^\theta(\rho) = \mathcal{L}\llbracket\eta_{\llbracket|\alpha|\rrbracket_T}\,\llbracket|M|\rrbracket_T\rrbracket^\theta(\rho)\\
&\quad= \mathcal{L}\llbracket\eta_{\llbracket|\alpha|\rrbracket_T}\rrbracket^\theta(\rho)(\mathcal{L}\llbracket\llbracket|M|\rrbracket_T\rrbracket^\theta(\rho)) = \mathcal{L}\llbracket\eta_{\mathrm{a}}\{\llbracket|\alpha|\rrbracket_T/\mathrm{a}\}\rrbracket^\theta(\rho)(\mathcal{L}\llbracket\llbracket|M|\rrbracket_T\rrbracket^\theta(\rho))\\
&\quad= \mathcal{L}\llbracket\eta_{\mathrm{a}}\rrbracket^{\theta[\mathrm{a}\mapsto\mathcal{L}\llbracket\llbracket|\alpha|\rrbracket_T\rrbracket^\theta]}(\rho)(\mathcal{L}\llbracket\llbracket|M|\rrbracket_T\rrbracket^\theta(\rho)) = \mathcal{L}\llbracket\eta_{\mathrm{a}}\rrbracket^{\mathrm{a}\mapsto\mathcal{L}\llbracket\llbracket|\alpha|\rrbracket_T\rrbracket^\theta}(\bullet)(\mathcal{L}\llbracket\llbracket|M|\rrbracket_T\rrbracket^\theta(\rho))\\
&\quad\overset{\mathrm{ih}}{=} \mathcal{L}\llbracket\eta_{\mathrm{a}}\rrbracket^{\mathrm{a}\mapsto\mathcal{L}_{\mathcal{T}}\llbracket\alpha\rrbracket^\theta}(\bullet)(\mathcal{L}_{\mathcal{T}}\llbracket M\rrbracket^\theta(\rho)) = \xi_{\mathcal{L}_{\mathcal{T}}\llbracket\alpha\rrbracket^\theta}(\mathcal{L}_{\mathcal{T}}\llbracket M\rrbracket^\theta(\rho)) = \mathcal{L}_{\mathcal{T}}\llbracket{}^\circ M\rrbracket^\theta(\rho)
\end{aligned}$$

The case for $\mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2$ is similar. ■

Similarly, given an evaluation semantics for $\mathcal{L}$ (i.e., a computable partial function $Eval_{\mathcal{L}}$ from closed L-terms of type ${}^\circ\iota$ to natural numbers), we get an evaluation semantics for $\mathcal{L}^T$ by taking $Eval_{\mathcal{L}^T}(M) = Eval_{\mathcal{L}}(\llbracket M\rrbracket_T)$. (We can do this directly, regardless of T, because the T-translation of a term of type ${}^\circ\iota$ is itself a term of type ${}^\circ\iota$.)

It is worth remarking that when T is the "identity" monad ($T\alpha = {}^\circ\alpha$, $\eta = \lambda x.{}^\circ x$, $f^*t = \mathbf{let}^\circ\ a \Leftarrow t\ \mathbf{in}\ f\,a$), the translation effectively replaces all occurrences of $\text{\textbf{'}}$ in the source term with ${}^\circ$:

$$\llbracket\text{\textbf{'}}M\rrbracket_T = (\lambda x.{}^\circ x)\,\llbracket M\rrbracket_T = {}^\circ\llbracket M\rrbracket_T = \llbracket{}^\circ M\rrbracket_T$$

$$\begin{aligned}
&\llbracket\mathbf{let}\text{\textbf{'}}\ x \Leftarrow M_1\ \mathbf{in}\ M_2\rrbracket_T = (\lambda x.\llbracket M_2\rrbracket_T)^*\,\llbracket M_1\rrbracket_T = \mathbf{let}^\circ\ a \Leftarrow \llbracket M_1\rrbracket_T\ \mathbf{in}\ (\lambda x.\llbracket M_2\rrbracket_T)\,a\\
&\quad= \mathbf{let}^\circ\ x \Leftarrow \llbracket M_1\rrbracket_T\ \mathbf{in}\ \llbracket M_2\rrbracket_T = \llbracket\mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2\rrbracket_T
\end{aligned}$$

so when we later exhibit a relation between the T-translation and a monadic translation for a continuation monad, we will get a relation between direct and continuation-passing style in the presence of arbitrary (sufficiently well-behaved) ambient effects by simply taking $\mathbf{T}$ to be the identity monad.

Defining T-specific operators The reflection and reification primitives allow us to define the meanings of effectful terms as abbreviations within the extended language, instead of through additional clauses in the translation equations.

Example 2.26 When T is the exception monad, we can define the usual ML-like exception primitives

$$\frac{\Gamma \vdash M : \text{\textbf{'}}\chi}{\Gamma \vdash \mathbf{raise}\ M : \text{\textbf{'}}\alpha} \qquad \frac{\Gamma \vdash M_1 : \text{\textbf{'}}\alpha \qquad \Gamma, x{:}\,\chi \vdash M_2 : \text{\textbf{'}}\alpha}{\Gamma \vdash \mathbf{try}\ M_1\ \mathbf{handle}\ x \Rightarrow M_2 : \text{\textbf{'}}\alpha}$$

as follows:

$$\begin{aligned}
\textbf{raise}\ M &\stackrel{\text{def}}{=} \textbf{let}\text{'}\ e \Leftarrow M\ \textbf{in}\ \mu({}^{\circ}(\mathsf{inr}\, e)) \\
\textbf{try}\ M_1\ \textbf{handle}\ x \Rightarrow M_2 &\stackrel{\text{def}}{=} \textbf{let}^{\circ}\ t \Leftarrow [M_1]\ \textbf{in}\ \mathsf{case}\,(t, a.\text{'}a, x.M_2)
\end{aligned}$$

That is, to raise an exception, we explicitly construct its sum-representation as a value in the right inject, then "activate" it by reflecting it into the process of computation. Conversely, to handle a potential exception in a computation M_1, we first reify M_1 and then inspect it, taking the appropriate action for either of the two possibilities (normal or exceptional value).

And in fact, expanding the definitions using the monadic translation gives the expected results:

$$\begin{aligned}
&\llbracket \textbf{raise}\ M \rrbracket_T = \llbracket \textbf{let}\text{'}\ e \Leftarrow M\ \textbf{in}\ \mu({}^{\circ}(\mathsf{inr}\, e)) \rrbracket_T = (\lambda e.\, \llbracket \mu({}^{\circ}(\mathsf{inr}\, e)) \rrbracket_T)^* \, \llbracket M \rrbracket_T \\
&\quad = \textbf{let}^{\circ}\ t \Leftarrow \llbracket M \rrbracket_T\ \textbf{in}\ \mathsf{case}\,(t, e.\llbracket {}^{\circ}(\mathsf{inr}\, e) \rrbracket_T, e.{}^{\circ}(\mathsf{inr}\, e)) \\
&\quad = \textbf{let}^{\circ}\ t \Leftarrow \llbracket M \rrbracket_T\ \textbf{in}\ \mathsf{case}\,(t, e.{}^{\circ}(\mathsf{inr}\, e), e.{}^{\circ}(\mathsf{inr}\, e))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \textbf{try}\ M_1\ \textbf{handle}\ x \Rightarrow M_2 \rrbracket_T = \cdots = \llbracket \textbf{let}^{\circ}\ t \Leftarrow [M_1]\ \textbf{in}\ \mathsf{case}\,(t, a.\text{'}a, x.M_2) \rrbracket_T \\
&\quad = \textbf{let}^{\circ}\ t \Leftarrow \llbracket [M_1] \rrbracket_T\ \textbf{in}\ \mathsf{case}\,(t, a.\llbracket \text{'}a \rrbracket_T, x.M_2) \\
&\quad = \textbf{let}^{\circ}\ t \Leftarrow \llbracket M_1 \rrbracket_T\ \textbf{in}\ \mathsf{case}\,(t, a.{}^{\circ}(\mathsf{inl}\, a), x.\llbracket M_2 \rrbracket_T)
\end{aligned}$$

■

2.3.2 Induced equational theory

The translation induces a natural equational theory on terms of the extended language:

Definition 2.27 *Given an equational theory $\mathcal{E}$ (including the cll axioms) for L and a monad-triple $\mathbf{T}$ in L, the equational theory $\mathcal{E}^T$ for L^T consists of $\mathcal{E}$ extended with the following rules (where we write $\boxed{0}$ for $^{\circ}$ and $\boxed{1}$ for '):*

$$\frac{\Gamma \vdash M_1 : \alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : \boxed{j}\alpha_2}{\Gamma \vdash (\textbf{let}^{\boxed{i}}\ x \Leftarrow \boxed{i}M_1\ \textbf{in}\ M_2) = M_2\{M_1/x\} : \boxed{j}\alpha_2}\,(i \le j)$$

$$\frac{\Gamma \vdash M : \boxed{i}\alpha}{\Gamma \vdash (\textbf{let}^{\boxed{i}}\ x \Leftarrow M\ \textbf{in}\ \boxed{i}x) = M : \boxed{i}\alpha}$$

$$\frac{\Gamma \vdash M_1 : \boxed{i}\alpha_1 \qquad \Gamma, x_1{:}\,\alpha_1 \vdash M_2 : \boxed{j}\alpha_2 \qquad \Gamma, x_2{:}\,\alpha_2 \vdash M_3 : \boxed{k}\alpha_3}{\begin{array}{l}\Gamma \vdash (\textbf{let}^{\boxed{j}}\ x_2 \Leftarrow (\textbf{let}^{\boxed{i}}\ x_1 \Leftarrow M_1\ \textbf{in}\ M_2)\ \textbf{in}\ M_3) \\ \qquad = (\textbf{let}^{\boxed{i}}\ x_1 \Leftarrow M_1\ \textbf{in}\ \textbf{let}^{\boxed{j}}\ x_2 \Leftarrow M_2\ \textbf{in}\ M_3) : \boxed{k}\alpha_3\end{array}}\,(i \le j \le k)$$

$$\frac{\Gamma \vdash M : \text{'}\alpha}{\Gamma \vdash \mu([M]) = M : \text{'}\alpha} \qquad \frac{\Gamma \vdash M : T\alpha}{\Gamma \vdash [\mu(M)] = M : T\alpha}$$

$$\frac{\Gamma \vdash M_1 : {}^{\circ}\alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : \text{'}\alpha_2}{\Gamma \vdash [\textbf{let}^{\circ}\ x \Leftarrow M_1\ \textbf{in}\ M_2] = (\textbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow M_1\ \textbf{in}\ [M_2]) : T\alpha_2}$$

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash [\text{'}M] = \eta\, M : T\alpha} \qquad \frac{\Gamma \vdash M_1 : \text{'}\alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : \text{'}\alpha_2}{\Gamma \vdash [\textbf{let}\text{'}\ x \Leftarrow M_1\ \textbf{in}\ M_2] = (\lambda x.\, [M_2])^* \, [M_1] : T\alpha_2}$$

(The instance $i = j = k = 0$ in the first three rules is already part of $\mathcal{E}$.)

Proposition 2.28 *The equational theory is sound for the monadic translation, in the sense that if $M = M'$ is provable in $\mathcal{E}^T$, then $\llbracket M \rrbracket_T = \llbracket M' \rrbracket_T$ is provable in $\mathcal{E}$ extended with the monad laws for $\mathbf{T}$ (which may or may not already be provable in $\mathcal{E}$). This again implies that $\mathcal{L}\llbracket \llbracket M \rrbracket_T \rrbracket = \mathcal{L}\llbracket \llbracket M' \rrbracket_T \rrbracket$ in any model $\mathcal{L}$ of $\mathcal{E}$ in which $\mathbf{T}$ is a monad.*

Proof. Simple equational reasoning, using Proposition 2.9 (the derivable equational properties for the generalized let). For example,

$$\begin{aligned}
&\llbracket \mathbf{let}^{\circ}\ x \Leftarrow {}^{\circ}M_1\ \mathbf{in}\ M_2 : \text{'}\alpha_2 \rrbracket_T = \mathbf{let}^{\circ}_{T\llbracket \alpha_2 \rrbracket_T}\ x \Leftarrow {}^{\circ}\llbracket M_1 \rrbracket_T\ \mathbf{in}\ \llbracket M_2 \rrbracket_T = \llbracket M_2 \rrbracket_T \{\llbracket M_1 \rrbracket_T / x\} \\
&\quad = \llbracket M_2\{M_1/x\} \rrbracket_T
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathbf{let'}\ x \Leftarrow \text{'}M_1\ \mathbf{in}\ M_2 \rrbracket_T = (\lambda x.\, \llbracket M_2 \rrbracket_T)^* (\eta\, \llbracket M_1 \rrbracket_T) = (\lambda x.\, \llbracket M_2 \rrbracket_T)\, \llbracket M_1 \rrbracket_T \\
&\quad = \llbracket M_2 \rrbracket_T \{\llbracket M_1 \rrbracket_T / x\} = \llbracket M_2\{M_1/x\} \rrbracket_T
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathbf{let}^{\circ}\ x_2 \Leftarrow (\mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3 : \text{'}\alpha_3 \rrbracket_T \\
&\quad = \mathbf{let}^{\circ}_{T\llbracket \alpha_3 \rrbracket_T}\ x_2 \Leftarrow (\mathbf{let}^{\circ}\ x_1 \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ \llbracket M_2 \rrbracket_T)\ \mathbf{in}\ \llbracket M_3 \rrbracket_T \\
&\quad = \mathbf{let}^{\circ}_{T\llbracket \alpha_3 \rrbracket_T}\ x_1 \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ \mathbf{let}^{\circ}_{T\llbracket \alpha_3 \rrbracket_T}\ x_2 \Leftarrow \llbracket M_2 \rrbracket_T\ \mathbf{in}\ \llbracket M_3 \rrbracket_T \\
&\quad = \llbracket \mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let}^{\circ}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3 \rrbracket_T
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathbf{let'}\ x_2 \Leftarrow (\mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ M_2)\ \mathbf{in}\ M_3 \rrbracket_T \\
&\quad = (\lambda x_2.\, \llbracket M_3 \rrbracket_T)^* (\mathbf{let}^{\circ}_{T\llbracket \alpha_2 \rrbracket_T}\ x_1 \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ \llbracket M_2 \rrbracket_T) \\
&\quad = \mathbf{let}^{\circ}_{T\llbracket \alpha_3 \rrbracket_T}\ x_1 \Leftarrow \llbracket M_1 \rrbracket_T\ \mathbf{in}\ (\lambda x_2.\, \llbracket M_3 \rrbracket_T)^*\, \llbracket M_2 \rrbracket_T \\
&\quad = \llbracket \mathbf{let}^{\circ}\ x_1 \Leftarrow M_1\ \mathbf{in}\ \mathbf{let'}\ x_2 \Leftarrow M_2\ \mathbf{in}\ M_3 \rrbracket_T
\end{aligned}$$

$$\llbracket \texttt{[}\text{'}M\texttt{]} \rrbracket_T = \llbracket \text{'}M \rrbracket_T = \eta\, \llbracket M \rrbracket_T = \llbracket \eta\, M \rrbracket_T$$

■

The first three rules of Definition 2.27 say that let-elimination and let-flattening are valid even for mixed levels, as long as the types match. That is, there is a single notion of computation-sequencing shared by all effects; the level-tags merely keep track of which kinds of effects can happen where.

The next two express that reflection and reification are exact inverses. For example, in the exception case, there is a one-to-one correspondence between “dynamic”, effectful computations of type $\text{'}\alpha$, that may raise exceptions, and “static”, exception-free values of type ${}^{\circ}(\alpha + \chi)$.

The remaining three equations show that `[-]` acts as a “shallow” version of the monadic translation. According to the first one, terms with ambient effects only are unaffected by reification for the focus effect, and may hence move across a `[-]`-barrier freely. For example, if the ambient effect is state and focus effect is exceptions, a computation that cannot raise an exception can be moved out of a try-handle. (The equation is necessarily satisfied when the base language is effect-free, because in that case $\mathbf{let}^{\circ}$ is simply a substitution of M_1 for x in M_2.)

The final two rules make it explicit how T-computations are realized in terms of the monad operations, enabling us to reason “locally” about propagation of effects entirely at the extended-language level.

Moreover, for all the rules it is the case that when the LHS is type-correct, then so is the RHS. Thus, we can always use the equations from left to right without worrying about type preservation.

As a simple consequence of the proposition, we get:

Corollary 2.29 *Let* $|\text{-}| : L_0 \to L_0^T$ *be the translation replacing every occurrence of* $^\circ$ *in types and terms by* $\texttt{'}$. *Let* $\mathcal{L}$ *be a model of* $\mathcal{E}_0$. *Then the interpretation* $\mathcal{L}_0^{|T|}$ *of* L_0, *given by* $\mathcal{L}_0^{|T|}[\![\text{-}]\!] = \mathcal{L}_0^T[\![|\text{-}|]\!]$ *is also a model of* $\mathcal{E}_0$ *(and hence in particular itself a computational lambda-language).*

Proof. The equations for numbers, products, sums, functions, and fixed points follow immediately from the translation. For computations, take $i = j = k = 1$ in the first three rules of $\mathcal{E}^T$ and use Proposition 2.28. ■

Note also that $\mathcal{E}^T$ implicitly asserts that $(T, \eta, \text{-}^*)$ form a monad in L^T, because it is easy to see that the equalities

$$\begin{aligned} \eta &= \lambda a.\, \texttt{[}\texttt{'}a\texttt{]} \\ f^* &= \lambda t.\, \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(t)\ \mathbf{in}\ \mu(f\,a)\texttt{]} \end{aligned}$$

are derivable, and the monad laws for η and f^* then follow from the equations, e.g.,

$$\begin{aligned} & f^*\,(\eta\,a) = (\lambda t.\, \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(t)\ \mathbf{in}\ \mu(f\,a)\texttt{]})\,\texttt{[}\texttt{'}a\texttt{]} = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(\texttt{[}\texttt{'}a\texttt{]})\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \texttt{'}a\ \mathbf{in}\ \mu(f\,a)\texttt{]} = \texttt{[}\mu(f\,a)\texttt{]} = f\,a \end{aligned}$$

and for rigidity of f^*:

$$\begin{aligned} & f^*\,(\mathbf{let}^{\circ}_{T\alpha_1}\ x \Leftarrow M\ \mathbf{in}\ x) = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(\mathbf{let}^{\circ}_{T\alpha_1}\ x \Leftarrow M\ \mathbf{in}\ x)\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(\mathbf{let}^{\circ}_{T\alpha_1}\ x \Leftarrow M\ \mathbf{in}\ \texttt{[}\mu(x)\texttt{]})\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(\texttt{[}\mathbf{let}^{\circ}\ x \Leftarrow M\ \mathbf{in}\ \mu(x)\texttt{]})\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow (\mathbf{let}^{\circ}\ x \Leftarrow M\ \mathbf{in}\ \mu(x))\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \texttt{[}\mathbf{let}^{\circ}\ x \Leftarrow M\ \mathbf{in}\ \mathbf{let}\texttt{'}\ a \Leftarrow \mu(x)\ \mathbf{in}\ \mu(f\,a)\texttt{]} \\ & \quad = \mathbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow M\ \mathbf{in}\ \texttt{[}\mathbf{let}\texttt{'}\ a \Leftarrow \mu(x)\ \mathbf{in}\ \mu(f\,a)\texttt{]} = \mathbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow M\ \mathbf{in}\ f^*\,x \end{aligned}$$

Example 2.30 We can use the extended-language equations to verify the following η-like rule for exceptions:

$$\begin{aligned} & (\mathbf{try}\ M\ \mathbf{handle}\ x \Rightarrow \mathbf{raise}\ \texttt{'}x) \\ & \quad = \mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \mathsf{case}\,(t, a.\texttt{'}a, x.\mathbf{let}\texttt{'}\ e \Leftarrow \texttt{'}x\ \mathbf{in}\ \mu(^{\circ}(\mathsf{inr}\,e))) \\ & \quad = \mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \mathsf{case}\,(t, a.\mu(\texttt{[}\texttt{'}a\texttt{]}), x.\mu(^{\circ}(\mathsf{inr}\,x))) \\ & \quad = \mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \mathsf{case}\,(t, a.\mu(^{\circ}(\mathsf{inl}\,a)), x.\mu(^{\circ}(\mathsf{inr}\,x))) = \mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \mu(^{\circ}t) \\ & \quad = \mu(\texttt{[}\mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \mu(^{\circ}t)\texttt{]}) = \mu(\mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ \texttt{[}\mu(^{\circ}t)\texttt{]}) = \mu(\mathbf{let}^{\circ}\ t \Leftarrow \texttt{[}M\texttt{]}\ \mathbf{in}\ ^{\circ}t) \\ & \quad = \mu(\texttt{[}M\texttt{]}) = M \end{aligned}$$

This identity is crucial for pattern-matching exception handlers, where an exception is implicitly re-raised if it does not match any of the clauses in a handler; we want to ensure that such a handler has no effect on the result of the program. ■

2.4 Related work

There has already been much work on combining monadic effects, e.g., [Mog90, KW93, CM93, Ste94, LHJ95, Esp95], of varying degrees of generality and formality. None of these approaches, however, were particularly concerned about *nonstandard implementations* of the newly-specified effects; effectively, they all interpret programs using the modular specification directly, often at a significant cost in execution time.

It seems likely that the framework outlined here for the two-level case generalizes to multiple, explicitly-specified effects, each with a full reflection and reification operator. However, the primary constraint was not only to define a workable notion of layered effect, but also to ensure that it could be simulated in a strong sense by continuation-passing, and further by escapes and state, as detailed in the next two chapters. Consequently, any broader modularity aspects of the approach have not been properly developed.

Chapter 3

Relating Effects

It is part of continuation folklore that continuations provide a very general notion of effects, in that many others (such as partiality, exceptions, or state) can be expressed as a continuation semantics with a suitable answer type. In the presence of higher-order functions, however, proving correctness of a continuation-based simulation is decidedly non-trivial [Rey74a, Sto81, MW85], even for a "purely functional" language with partiality as the only notion of computational effect.

In this chapter, we will consider the relationship between a direct and a continuation semantics for arbitrary monadic effects. In fact, the continuation semantics can itself be conveniently cast in the monadic mold, making the result a particular instance of simulating one monadic effect with another. However, the continuation-passing case is especially complicated, and a significant part of the proof consists of establishing the general framework and necessary lemmas for this case.

Very broadly, the general idea is as follows: assume we have two monads $\mathbf{T}$ and $\mathbf{U}$ over a base language, where $\mathbf{U}$ is in a suitable sense "more general" than $\mathbf{T}$. We can then give two different translations from L^T to L: the original monadic translation for $\mathbf{T}$ and a *variant* translation using U-representations of T-effects. Moreover, we can exhibit a type-indexed family of relations $\trianglelefteq_\alpha$ with the property that the two translations of an L^T-term of type α are related by $\trianglelefteq_\alpha$, and such that the relation at base types is the identity. Thus, the two translations induce the same evaluation semantics.

3.1 Simulating monadic effects

In this section, we present a general principle for relating effects, introduce the variant translation, and argue informally for its correctness. The actual simulation proof, however, will be postponed until Section 3.3.

3.1.1 Monad morphisms

A natural way of relating two monads consists of exhibiting a function mapping one to the other:

Definition 3.1 *Let* $\mathbf{T} = (T, \eta, \text{-}^*)$ *and* $\mathbf{U} = (U, \varepsilon, \text{-}^+)$ *be monads over a cll* $(L, \mathcal{L})$*. A* (definable) monad morphism *from* $\mathbf{T}$ *to* $\mathbf{U}$ *is a type-indexed family of L-terms,*

$$\vdash_{\{\mathrm{a}\}} \mathsf{i}_{\mathrm{a}} : T\mathrm{a} \to U\mathrm{a}\,,$$

respecting the monad structure, i.e., such that the following holds in $\mathcal{L}$ *for all closed L-types:*

0. $\vdash \mathsf{i}_\alpha : T\alpha \overset{\mathbf{r}}{\to} U\alpha$.

1. $\vdash \mathsf{i}_\alpha \circ \eta_\alpha = \varepsilon_\alpha : \alpha \to U\alpha$.

2. $f : \alpha_1 \to T\alpha_2 \vdash \mathsf{i}_{\alpha_2} \circ f^* = (\mathsf{i}_{\alpha_2} \circ f)^+ \circ \mathsf{i}_{\alpha_1} : T\alpha_1 \to U\alpha_2$.

We can think of i as converting T-representations of effects to U-representations. Condition (0) is a technical constraint, ensuring essentially that the conversion of focus effects respects any underlying ambient effects (for example, a nonterminating T-computation must be represented by a nonterminating U-computation). More explicitly, (1) says that a trivial T-computation is mapped into a trivial U-computation. Condition (2) may look somewhat arbitrary at first, but note that it can be written in the form of a conditional equality emphasizing the parallel to (1):

$$\mathsf{i}_{\alpha_2} \circ f = g \Rightarrow \mathsf{i}_{\alpha_2} \circ f^* = g^+ \circ \mathsf{i}_{\alpha_1}$$

(where $f : \alpha_1 \to T\alpha_2$ and $g : \alpha_1 \to U\alpha_2$). It expresses the requirement that if g is the U-counterpart of an α_1-parameterized T-computation f, then T-extending f and converting its output is equivalent to applying the U-extended g to the conversion of the input.

Example 3.2 For any monad $\mathbf{U}$, the function family

$$\mathsf{h}_\alpha : {}^{\circ}\alpha \to U\alpha = \lambda t^{{}^{\circ}\alpha}.\,\mathbf{let}^{\circ}_{U\alpha}\; a \Leftarrow t \;\mathbf{in}\; \varepsilon\, a$$

is a monad morphism from the identity monad $\mathbf{I}$ (Example 2.16) to $\mathbf{U}$:

$$\begin{aligned}\mathsf{h}\,(\mathbf{let}^{\circ}\; x \Leftarrow m \;\mathbf{in}\; x) &= \mathbf{let}^{\circ}_{U\alpha}\; a \Leftarrow (\mathbf{let}^{\circ}\; x \Leftarrow m \;\mathbf{in}\; x) \;\mathbf{in}\; \varepsilon\, a \\ &= \mathbf{let}^{\circ}_{U\alpha}\; x \Leftarrow m \;\mathbf{in}\; \mathbf{let}^{\circ}_{U\alpha}\; a \Leftarrow x \;\mathbf{in}\; \varepsilon\, a = \mathbf{let}^{\circ}_{U\alpha}\; x \Leftarrow m \;\mathbf{in}\; h\, x\end{aligned}$$

$$\mathsf{h}\,(\eta\, a) = h\,({}^{\circ}a) = \mathbf{let}^{\circ}_{U\alpha}\; a \Leftarrow {}^{\circ}a \;\mathbf{in}\; \varepsilon\, a = \varepsilon\, a$$

$$\begin{aligned}\mathsf{h}\,(f^*\, t) &= \mathsf{h}\,(\mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; f\, a_1) = \mathbf{let}^{\circ}\; a_2 \Leftarrow (\mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; f\, a_1) \;\mathbf{in}\; \varepsilon\, a_2 \\ &= \mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; \mathbf{let}^{\circ}\; a_2 \Leftarrow f\, a_1 \;\mathbf{in}\; \varepsilon\, a_2 = \mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; h\,(f\, a_1) \\ &= \mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; (\mathsf{h} \circ f)^+\,(\varepsilon\, a_1) = (\mathsf{h} \circ f)^+\,(\mathbf{let}^{\circ}\; a_1 \Leftarrow t \;\mathbf{in}\; \varepsilon\, a_1) = (\mathsf{h} \circ f)^+\,(\mathsf{h}\, t)\end{aligned}$$

In fact, it is the only such morphism. This is immediate when the base language is effect-free (condition (1) with $\eta = \mathsf{id}$), but it also holds in general: Suppose $\mathsf{h}' : {}^{\circ}\alpha \to U\alpha$ is another monad morphism from $\mathbf{I}$ to $\mathbf{U}$. Then

$$\begin{aligned}\mathsf{h}'\, t &= \mathsf{h}'\,(\mathbf{let}^{\circ}\; x \Leftarrow t \;\mathbf{in}\; {}^{\circ}x) = \mathbf{let}^{\circ}_{U\alpha}\; x \Leftarrow t \;\mathbf{in}\; \mathsf{h}'\,({}^{\circ}x) = \mathbf{let}^{\circ}_{U\alpha}\; x \Leftarrow t \;\mathbf{in}\; \mathsf{h}'\,(\eta\, x) \\ &= \mathbf{let}^{\circ}_{U\alpha}\; x \Leftarrow t \;\mathbf{in}\; \varepsilon\, x = \mathsf{h}\, t\end{aligned}$$

This captures the intuitive notion that *any* effect can simulate the absence of effects, which we would probably consider a minimal requirement for any notion of effect simulation. ■

A more interesting example is provided by the following:

Example 3.3 For simplicity, assume that our base language includes a term constructor $+ : \iota \times \iota \to \iota$, satisfying equations $\underline{0} + M = M + \underline{0} = M$ and $M_1 + (M_2 + M_3) = (M_1 + M_2) + M_3$ in the model, i.e., such that $(\iota, \underline{0}, +)$ forms a monoid. (We could of course have defined an addition operator in the existing language using recursion, but that would necessarily give it the type $\iota \times \iota \to {}^{\circ}\iota$, cluttering up the terms with explicit sequencing of the additions. Still, it is easy to check that everything does work out correctly even for a defined +.)

Then the following determines a monad, usually called the *complexity* monad:

$$\begin{aligned} T\alpha &= {}^{\circ}(\alpha \times \iota) \\ \eta &= \lambda a.\, {}^{\circ}\langle a, \underline{0} \rangle \\ f^* &= \lambda t.\, \mathbf{let}^{\circ}\ \langle a_1, n_1 \rangle \Leftarrow t\ \mathbf{in}\ \mathbf{let}^{\circ}\ \langle a_2, n_2 \rangle \Leftarrow f\, a_1\ \mathbf{in}\ {}^{\circ}\langle a_2, n_1 + n_2 \rangle \end{aligned}$$

Here, a computation of type α is represented by a base-computation yielding a value of type α together with some notion of the *cost* involved in computing it, such as the number of floating-point operations performed (perhaps using an encoding of floating-point numbers in terms of ι), or the amount of I/O (assuming our ambient effects include some notion of communication with the outside world). A trivial computation, $\eta\, a$, resulting from viewing an already given value as a computation, has zero cost; the cost of evaluating $f^* t$ is the sum of the cost n_1 of computing the value a_1 of t and the cost n_2 of evaluating f at a_1.

The complexity monad works by summing the complexities of each subcomputation. But if most subcomputations do not invoke the operation being counted, this is potentially wasteful, since we will be adding zeros most of the time. Even more important, complexity is a fairly “ad hoc” monad, so that we will most likely have to perform an actual translation to get a language with the corresponding monadic effects.

There is an alternative way to keep track of complexity, however: maintain a running total, which is updated only by the cost-incurring operations themselves, and passively transmitted everywhere else. We achieve this using the ι-state monad (Example 2.18):

$$U\alpha = \iota \to {}^{\circ}(\alpha \times \iota), \qquad \varepsilon = \lambda a.\, \lambda s.\, {}^{\circ}\langle a, s \rangle, \qquad f^{+} = \lambda u.\, \lambda s.\, \mathbf{let}^{\circ}\ \langle a, s' \rangle \Leftarrow u\, s\ \mathbf{in}\ f\, a\, s'$$

We can then represent a computation of a with complexity n as a function adding n to the current total, in addition to returning a. And in fact,

$$\mathsf{i}_{\alpha} = \lambda t^{T\alpha}.\, \lambda s^{\iota}.\, \mathbf{let}^{\circ}\ \langle a, n \rangle \Leftarrow t\ \mathbf{in}\ {}^{\circ}\langle a, s + n \rangle$$

is a monad morphism from $\mathbf{T}$ to $\mathbf{U}$. ■

We now formally define a very important class of monads:

Definition 3.4 *Let* o *be any computation-type. Then the* continuation monad with answer type o, $\mathbf{K}_o = (K_o, \varepsilon, {}^+)$ *is given by:*

$$K_o\alpha = (\alpha \to o) \to o, \qquad \varepsilon = \lambda a.\,\lambda k.\,k\,a, \qquad f^+ = \lambda u.\,\lambda k.\,u\,(\lambda a.\,f\,a\,k)$$

It is easy to check that this actually determines a monad. For rigidity of f^+, we have:

$$\begin{aligned} &f^+\,(\mathbf{let}^\circ_{U\alpha}\;x \Leftarrow m\;\mathbf{in}\;x) = \lambda k.\,(\mathbf{let}^\circ_{U\alpha}\;x \Leftarrow m\;\mathbf{in}\;x)\,(\lambda a.\,f\,a\,k)\\ &\quad = \lambda k.\,\mathbf{let}^\circ_o\;x \Leftarrow m\;\mathbf{in}\;x\,(\lambda a.\,f\,a\,k) = \lambda k.\,\mathbf{let}^\circ_o\;x \Leftarrow m\;\mathbf{in}\;f^+\,x\,k\\ &\quad = \mathbf{let}^\circ_{U\alpha}\;x \Leftarrow m\;\mathbf{in}\;f^+\,x \end{aligned}$$

Satisfaction of the other three equations is completely straightforward.

The importance of continuation monads stems from the following property:

Lemma 3.5 *Let* $\mathbf{T} = (T, \eta, \text{-}^*)$ *be a monad in a computational* λ*-language, and let* γ *be an arbitrary type (not necessarily computational). Take* $\mathbf{U}$ *as* $\mathbf{K}_{T\gamma}$*, the continuation monad with answer type* $T\gamma$*. Then the family of functions*

$$\mathsf{i}_\alpha : T\alpha \to U\alpha = \lambda t^{T\alpha}.\,\lambda k^{\alpha\to T\gamma}.\,k^*\,t$$

forms a monad morphism from $\mathbf{T}$ *to* $\mathbf{U}$.

Proof. Straightforward verification:

$$\begin{aligned} &\mathsf{i}\,(\mathbf{let}^\circ_{T\alpha}\;t \Leftarrow m\;\mathbf{in}\;t) = \lambda k.k^*\,(\mathbf{let}^\circ_{T\alpha}\;t \Leftarrow m\;\mathbf{in}\;t) =^\dagger \lambda k.\mathbf{let}^\circ_{T\gamma}\;t \Leftarrow m\;\mathbf{in}\;k^*\,t\\ &\quad = \lambda k.\mathbf{let}^\circ_{T\gamma}\;t \Leftarrow m\;\mathbf{in}\;\mathsf{i}\,t\,k = \mathbf{let}^\circ_{U\alpha}\;t \Leftarrow m\;\mathbf{in}\;\mathsf{i}\,t \end{aligned}$$

$$\mathsf{i}\,(\eta\,a) = \lambda k.\,k^*\,(\eta\,a) =^\dagger \lambda k.\,k\,a = \varepsilon\,a$$

$$\begin{aligned} &\mathsf{i}\,(f^*\,t) = \lambda k.\,k^*\,(f^*\,t) =^\dagger \lambda k.\,(\lambda a.\,k^*\,(f\,a))^*\,t = \lambda k.\mathsf{i}\,t\,(\lambda a.\,k^*\,(f\,a)) = \lambda k.\mathsf{i}\,t\,(\lambda a.\mathsf{i}\,(f\,a)\,k)\\ &\quad = \lambda k.\mathsf{i}\,t\,(\lambda a.[\lambda a.\mathsf{i}\,(f\,a)]\,a\,k) = (\lambda a.\mathsf{i}\,(f\,a))^+\,(\mathsf{i}\,t) = (\mathsf{i} \circ f)^+\,(\mathsf{i}\,t) \end{aligned}$$

(where the equations marked with † signify application of the monad laws of $\mathbf{T}$ from Definition 2.15). ■

When $\mathbf{T}$ is the identity monad, this (necessarily) degenerates to an instance of Example 3.2. More interestingly,

Example 3.6 For exceptions, $T\alpha = {}^\circ(\alpha + \chi)$, the monad morphism from $\mathbf{T}$ to $\mathbf{K}_{T\gamma}$ specializes to:

$$\mathsf{i}_\alpha \;=\; \lambda t^{{}^\circ(\alpha+\chi)}.\,\lambda k^{\alpha\to{}^\circ(\gamma+\chi)}.\,\mathbf{let}^\circ\;v \Leftarrow t\;\mathbf{in}\;\mathsf{case}\,(v, a.k\,a, e.{}^\circ(\mathsf{inr}\;e))$$

Recall that the T-representation of a successful computation of type α is an included value a in the left inject of $\alpha + \chi$. The corresponding continuation-passing computation should immediately apply its continuation to a. And in fact, we have

$$\begin{aligned} &\mathsf{i}\,({}^\circ(\mathsf{inl}\;a)) = \lambda k.\,\mathbf{let}^\circ\;v \Leftarrow {}^\circ(\mathsf{inl}\;a)\;\mathbf{in}\;\mathsf{case}\,(v, a.k\,a, e.{}^\circ(\mathsf{inr}\;e))\\ &\quad = \lambda k.\,\mathsf{case}\,(\mathsf{inl}\;a, a.k\,a, e.{}^\circ(\mathsf{inr}\;e)) = \lambda k.\,k\,a \end{aligned}$$

Similarly, a computation that terminates with a raised exception e is represented by a value in the right inject; the continuation-passing analog simply discards the current continuation and returns the exceptional value as the result:

$$\begin{aligned} &\mathsf{i}\,({}^\circ(\mathsf{inr}\,e)) = \lambda k.\,\mathbf{let}^\circ\ v \Leftarrow {}^\circ(\mathsf{inr}\,e)\ \mathbf{in}\ \mathsf{case}\,(v, a.k\,a, e.{}^\circ(\mathsf{inr}\,e)) \\ &\quad = \lambda k.\,\mathsf{case}\,(\mathsf{inr}\,e, a.k\,a, e.{}^\circ(\mathsf{inr}\,e)) = \lambda k.\,{}^\circ(\mathsf{inr}\,e) \end{aligned}$$

And finally, a nonterminating computation is represented by a non-terminating computation (for any continuation):

$$\mathsf{i}\,\bot = \lambda k.\,\mathbf{let}^\circ\ v \Leftarrow \bot\ \mathbf{in}\ \mathsf{case}\,(v, a.k\,a, e.{}^\circ(\mathsf{inr}\,e)) = \lambda k.\,\bot$$

■

This monad morphism from an arbitrary monad $\mathbf{T}$ to a continuation monad $\mathbf{K}_{T\gamma}$ will form the core of our simulation result. However, the fact that the continuation-based representation is in a sense *parametric* in the choice of γ cannot be captured equationally in our setting. (It might be possible in a language with F_2-polymorphism [Gir72, Rey74b].) For the formal proof in Section 3.3, we will therefore need a stronger, relational characterization of i to accurately express this property.

3.1.2 The variant translation

In this section, we show how to actually exploit the existence of a monad morphism (with some further properties) to simulate one kind of effects with another. Specifically, we will show how to interpret our T-enriched effect language in terms of U-effects. The exposition is slightly simplified in that we consider only a single semantics for the base language — the actual proof in Section 3.3 distinguishes between a specification and the implementation semantics, mostly to make get a result of sufficient strength to support Chapter 4. However, the formal definitions we give are general enough for both cases.

As motivated in the previous section, monad morphisms give us a simple way of relating two notions of effects. Nevertheless, a monad morphism by itself does not guarantee that U-effects simulate T-effects in any *useful* sense. For example, for any $\mathbf{T}$ there is a (unique) monad morphism from $\mathbf{T}$ to the *degenerate monad*, ($U\alpha = 1, \varepsilon = \lambda a.\,\langle\rangle, f^+ = \mathsf{id}_1$). To get a proper simulation, we also need a way to recover the T-representation of an effect from its U-representation:

Definition 3.7 *Let* i *be a monad morphism from* $\mathbf{T}$ *to* $\mathbf{U}$. *A* monad retraction *at type* α *is a left inverse of* i_α, *i.e., a term* $\mathsf{j}_\alpha : U\alpha \to T\alpha$ *such that* $\mathsf{j}_\alpha \circ \mathsf{i}_\alpha = \mathsf{id}_{T\alpha}$. *We say that such a retraction is* schematic *if all the* j_α *are themselves members of a term family, i.e., if* $\mathsf{j}_\alpha = \mathsf{j}_{\mathrm{a}}\{\alpha/\mathrm{a}\}$.

We usually expect at least j_ι to exist; this gives us a way of extracting meanings of complete programs. In many cases, however, it is easy to find a suitable inverse at all types:

Example 3.8 For the complexity-state simulation from Example 3.3, where in particular the monad morphism was given by

$$\mathsf{i}_\alpha = \lambda t^{{}^\circ(\alpha\times\iota)}.\,\lambda s^\iota.\,\mathbf{let}^\circ\ \langle a, n\rangle \Leftarrow t\ \mathbf{in}\ {}^\circ\langle a, s+n\rangle\,,$$

taking

$$\mathsf{j}_\alpha = \lambda u^{\iota \to^\circ (\alpha \times \iota)}.\, u\,\underline{0}$$

determines a monad retraction at all types. That is, to actually extract the complexity of a computation from its state-passing representation, we simply initialize the state to zero, perform the computation, and read off the complexity as the final state.

It is easy to see that this j is a schematic left inverse of i. It is not, however, a *monad morphism* from $\mathbf{U}$ to $\mathbf{T}$: it does not in any meaningful sense simulate arbitrary state-passing computations using complexity-effects. ■

Given terms typed like the monad morphisms and retractions, we can give a *different* translation of our effect-enriched language L^T, back into L, using a U-based representation of T-effects instead of the T-representation from the definitional translation. To define the translation itself, of course, we do not need to assume any equational properties of the terms involved:

Definition 3.9 *In L, let $\mathbf{T}$ and $\mathbf{U}$ be monad-triples, and let i be a family of terms such that for any L^T-type α, $\mathsf{i}_{\llbracket\alpha\rrbracket_U} : T\llbracket\alpha\rrbracket_U \to U\llbracket\alpha\rrbracket_U$. Further, let $\aleph$ be a set (finite or infinite) of L^T-types and for every α in $\aleph$, a term $\mathsf{j}_{\llbracket\alpha\rrbracket_U} : U\llbracket\alpha\rrbracket_U \to T\llbracket\alpha\rrbracket_U$.*

Now, let $L^{T[\aleph]}$ be L^T but with reification restricted to $\aleph$-types, i.e., with $\mathtt{[}\text{-}\mathtt{]} : \text{'}\alpha \to T\alpha$ only for α in $\aleph$. We then define the variant *or* implementation *translation from $L^{T[\aleph]}$ to L as follows. For types, we take*

$$\llbracket \text{'}\alpha \rrbracket'_T = U\llbracket\alpha\rrbracket'_T$$

(so for the type translation we have $\llbracket\alpha\rrbracket'_T = \llbracket\alpha\rrbracket_U$), and for terms,

$$\begin{aligned}
\llbracket \text{'}M \rrbracket'_T &= \varepsilon\,\llbracket M\rrbracket'_T \\
\llbracket \mathbf{let}\text{'}\; x \Leftarrow M_1 \;\mathbf{in}\; M_2 \rrbracket'_T &= (\lambda x.\, \llbracket M_2\rrbracket'_T)^+\, \llbracket M_1\rrbracket'_T \\
\llbracket \mathbf{let}^\circ\; x \Leftarrow M_1 \;\mathbf{in}\; M_2 : \text{'}\alpha \rrbracket'_T &= \mathbf{let}^\circ_{U\llbracket\alpha\rrbracket'_T}\; x \Leftarrow \llbracket M_1\rrbracket'_T \;\mathbf{in}\; \llbracket M_2\rrbracket'_T \\
\llbracket \mu^T(M) \rrbracket'_T &= \mathsf{i}_{\llbracket\alpha\rrbracket'_T}\, \llbracket M\rrbracket'_T \\
\llbracket \mathtt{[}M\mathtt{]}^T \rrbracket'_T &= \mathsf{j}_{\llbracket\alpha\rrbracket'_T}\, \llbracket M\rrbracket'_T
\end{aligned}$$

(We write $\mu^T(\text{-})$ and $\mathtt{[}\text{-}\mathtt{]}^T$ to emphasize that these are reflection and reification operators for T, not U.) Like the definitional translation (Definition 2.23), $\llbracket\text{-}\rrbracket'_T$ is easily seen to preserve types, i.e., if $\Gamma \vdash_\Delta M : \alpha$ in L^T then $\llbracket\Gamma\rrbracket'_T \vdash_\Delta \llbracket M\rrbracket'_T : \llbracket\alpha\rrbracket'_T$ in L.

Of course, when $\mathbf{U} = \mathbf{T}$, with $\mathsf{i}_\alpha = \mathsf{j}_\alpha = \mathsf{id}_{T\alpha}$ (trivially a monad morphism with a schematic retraction), we get exactly the original $\llbracket\text{-}\rrbracket_T$-translation as a special case. In general, however, we now have $\llbracket T\alpha\rrbracket'_T = T\llbracket\alpha\rrbracket'_T \neq U\llbracket\alpha\rrbracket'_T = \llbracket\text{'}\alpha\rrbracket'_T$: the transparent and opaque representations of a computation with T-effects are different. This is why for reflection we need to *internalize* a T-representation of an effect into a U-representation that fits with the rest of the U-passing translation. Conversely, for reification, we *externalize* the U-representation into the definitional T-representation of the effect.

Although the definitional and the variant translation of a type are in general different, they do agree on base types, so in particular the results of transforming complete programs (closed terms of type $^\circ\iota$) are directly comparable. And in fact, will show in Proposition 3.29 that the two translations of a closed L_0^T-term of type $^\circ\iota$ are indeed equal in our partiality semantics (and appropriately related for other notions of ambient effects).

In the monad-continuation case of a monad morphism (Lemma 3.5), it is not obvious how to define j_α in general. We will see in Section 3.3.4 how to achieve this. For the purposes of this section, however, let us simply restrict ourselves to performing T-reification at a *single L-type* γ (as opposed to at arbitrary L^T-types, as the standard T-translation allows us to). That is, we take $\aleph = \{\gamma\}$.

If we then let $\mathbf{U}$ be the continuation monad with answer type $T\gamma$, we can directly take $\mathsf{j}_\gamma : U\gamma \to T\gamma = \lambda u.\, u\,\eta_\gamma$, which gives us

$$\mathsf{j}_\gamma\,(\mathsf{i}_\gamma\,t) = (\lambda u.\,u\,\eta_\gamma)\,(\lambda k.\,k^*\,t) = (\lambda k.\,k^*\,t)\,\eta_\gamma = \eta_\gamma^*\,t = t$$

i.e., that j_γ is a monad retraction at γ.

Example 3.10 Let $T\alpha = {}^\circ(\alpha + \chi)$ be the exception monad, with the continuation-based representation $U\alpha = K_{T\gamma}\alpha = (\alpha \to {}^\circ(\gamma+\chi)) \to {}^\circ(\gamma+\chi)$ from Example 3.6. In this case, the translation equations specialize to:

$$\begin{array}{rcl}
\llbracket \text{'}M \rrbracket'_T &=& \lambda k.\,k\,\llbracket M \rrbracket'_T \\
\llbracket \mathbf{let'}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 \rrbracket'_T &=& \lambda k.\,\llbracket M_1 \rrbracket'_T\,(\lambda x.\,\llbracket M_2 \rrbracket'_T\,k) \\
\llbracket \mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : \text{'}\alpha \rrbracket'_T &=& \lambda k.\,\mathbf{let}^\circ_{{}^\circ(\gamma+\chi)}\ x \Leftarrow \llbracket M_1 \rrbracket'_T\ \mathbf{in}\ \llbracket M_2 \rrbracket'_T\,k \\
\llbracket \mu^T(M) \rrbracket'_T &=& \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket M \rrbracket'_T\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, e.{}^\circ(\mathsf{inr}\,e)) \\
\llbracket [M]^T \rrbracket'_T &=& \llbracket M \rrbracket'_T\,(\lambda a.\,{}^\circ(\mathsf{inl}\,a))
\end{array}$$

(where, for the third equation, we have used Definition 2.4 to expand out the generalized let in Definition 3.9). The continuation-passing analogs of **raise** and **handle**, as given by the expansions in Example 2.26 then work out to:

$$\begin{aligned}
&\llbracket \mathbf{raise}\ M \rrbracket'_T = \llbracket \mathbf{let'}\ e \Leftarrow M\ \mathbf{in}\ \mu^T({}^\circ(\mathsf{inr}\,e)) \rrbracket'_T = \lambda k.\,\llbracket M \rrbracket'_T\,(\lambda e.\,\llbracket \mu^T({}^\circ(\mathsf{inr}\,e)) \rrbracket'_T\,k) \\
&\quad = \lambda k.\,\llbracket M \rrbracket'_T\,(\lambda e.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket {}^\circ(\mathsf{inr}\,e) \rrbracket'_T\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, e.{}^\circ(\mathsf{inr}\,e))) \\
&\quad = \lambda k.\,\llbracket M \rrbracket'_T\,(\lambda e.\,\mathbf{let}^\circ\ t \Leftarrow {}^\circ(\mathsf{inr}\,e)\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, e.{}^\circ(\mathsf{inr}\,e))) \\
&\quad = \lambda k.\,\llbracket M \rrbracket'_T\,(\lambda e.\,\mathsf{case}\,(\mathsf{inr}\,e, a.k\,a, e.{}^\circ(\mathsf{inr}\,e))) = \lambda k.\,\llbracket M \rrbracket'_T\,(\lambda e.\,{}^\circ(\mathsf{inr}\,e))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathbf{try}\ M_1\ \mathbf{handle}\ x \Rightarrow M_2 \rrbracket'_T = \llbracket \mathbf{let}^\circ\ t \Leftarrow [M_1]^T\ \mathbf{in}\ \mathsf{case}\,(t, a.\text{'}a, x.M_2) \rrbracket'_T \\
&\quad = \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket [M_1]^T \rrbracket'_T\ \mathbf{in}\ \mathsf{case}\,(t, a.\llbracket \text{'}a \rrbracket'_T, x.\llbracket M_2 \rrbracket'_T)\,k \\
&\quad = \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket M_1 \rrbracket'_T\,(\lambda a.\,{}^\circ(\mathsf{inl}\,a))\ \mathbf{in}\ \mathsf{case}\,(t, a.(\lambda k.\,k\,a)\,k, x.\llbracket M_2 \rrbracket'_T\,k) \\
&\quad = \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket M_1 \rrbracket'_T\,(\lambda a.\,{}^\circ(\mathsf{inl}\,a))\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, x.\llbracket M_2 \rrbracket'_T\,k)
\end{aligned}$$

(where **handle** can only be used with expressions of type $\text{'}\gamma$). This again should match the operational intuition that to raise an exception determined by M, we simply return name directly as an answer (tagged as a right inject, so that an enclosing **handle** can tell the difference). Conversely, to handle a potential exception in M_1, we invoke it with the left injection as the continuation. If M_1 returns normally i.e., by returning $\mathsf{inl}\,a$, we pass a to the continuation of the **handle**. On the other hand, if M_1 raises an exception e, i.e., returns $\mathsf{inr}\,e$, we instead evaluate M_2 with x bound to e, again in the control context of the **handle**. ■

Comparison Suppose the restrictions on reification were not an issue, for example if we were content to only allow uses of **handle** at a single base type (not an entirely unreasonable restriction; we still have **raise** at all types). Then given the fairly simple correspondence between "direct" and "continuation-passing" definitions of exceptions, one might reasonably ask why we formalize the T-translation at all – why not simply take the continuation-based $\llbracket\text{-}\rrbracket'_T$ as the "official" definition of exceptions? Then we could view exceptions as simply syntactic sugar for the corresponding continuation effects.

The problem is that the CPS translation does not satisfy the desirable equational reasoning principles that pure exception-passing does. For example, consider again the reasoning principle

$$(\mathbf{try}\ M\ \mathbf{handle}\ x \Rightarrow \mathbf{raise}\ \text{'}x) = M$$

We saw in Example 2.30 that the T-translation verifies this law; indeed, it is provable in $\mathcal{E}_0^T$. But with the continuation-based semantics we get:

$$\begin{aligned}
\llbracket \mathbf{try}\ M\ \mathbf{handle}\ x \Rightarrow \mathbf{raise}\ \text{'}x \rrbracket'_T &= \llbracket \mathbf{let}^\circ\ t \Leftarrow \mathtt{[}M\mathtt{]}\ \mathbf{in}\ \mathsf{case}\,(t, a.\text{'}a, x.\mu(^\circ(\mathsf{inr}\,x)))\rrbracket'_T \\
&= \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket \mathtt{[}M\mathtt{]} \rrbracket'_T\ \mathbf{in}\ \mathsf{case}\,(t, a.\llbracket \text{'}a\rrbracket'_T\,k, x.\llbracket \mu(^\circ(\mathsf{inr}\,x))\rrbracket'_T\,k) \\
&= \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \mathsf{j}\,\llbracket M \rrbracket'_T\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, x.\mathsf{i}\,(^\circ(\mathsf{inr}\,x))\,k) \\
&= \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket M \rrbracket'_T\,\eta\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, x.k^*\,(^\circ(\mathsf{inr}\,x))) \\
&= \lambda k.\,\mathbf{let}^\circ\ t \Leftarrow \llbracket M \rrbracket'_T\,(\lambda a.\,^\circ(\mathsf{inl}\,a))\ \mathbf{in}\ \mathsf{case}\,(t, a.k\,a, x.^\circ(\mathsf{inr}\,x)) \overset{?}{=} \llbracket M \rrbracket'_T
\end{aligned}$$

It is easy to check that this does in fact hold when $\llbracket M\rrbracket'_T$ is of the form $\lambda k.\,k\,a$ for some a, corresponding to an effect-free computation of a. Similarly, the equation is satisfied when $\llbracket M\rrbracket'_T = \lambda k.\,^\circ(\mathsf{inr}\,e)$ for some e, corresponding to a computation raising the exception e. Even when $\llbracket M\rrbracket'_T = \lambda k.\,\bot$, representing a non-terminating computation, the terms have equal denotations. But there is no simple guarantee that $\llbracket M\rrbracket'_T$ is in fact in one of those forms, especially when M may call an "unknown" function.

For example, consider the case $\gamma = \iota$. Then one element of the type $\llbracket \text{'}\alpha\rrbracket'_T = (\llbracket\alpha\rrbracket'_T \to {}^\circ(\iota + \chi)) \to {}^\circ(\iota + \chi)$, is $\lambda k.\,^\circ(\mathsf{inl}\,\underline{42})$, which we could call an *exotic* T-computation: it represents neither a normal value, nor a raised exception, nor divergence. And in fact, if $\llbracket M_0\rrbracket'_T = \lambda k.\,^\circ(\mathsf{inl}\,\underline{42})$, our desired reasoning principle fails because we get

$$\llbracket \mathbf{try}\ M_0\ \mathbf{handle}\ x \Rightarrow \mathbf{raise}\ \text{'}x\rrbracket'_T = \lambda k.\,k\,\underline{42} \neq \lambda k.\,^\circ(\mathsf{inl}\,\underline{42}) = \llbracket M_0\rrbracket'_T$$

The presence of such computations means that we cannot derive the identity directly in the U-model – we need a much more elaborate argument, involving at least an induction over all syntactic terms in the language, and further complicated by the presence of higher-order functions.

An analogous situation holds for the complexity-state simulation from Example 3.3. It is easy to see that if the ambient effects are commutative, then so are the '-effects defined by the complexity monad. General state passing, on the other hand, is not commutative, so again we lose a useful equational property by specifying complexity-effects directly in terms of state-passing.

That does not mean, however, that using $\llbracket\text{-}\rrbracket'_T$ inherently presents a problem for formal reasoning. Recall that we will show independently that the $\llbracket\text{-}\rrbracket_T$-translation and the $\llbracket\text{-}\rrbracket'_T$-translation do agree on complete L_0^T-programs. Since the equations induced by the T-translation are (by definition) valid for observational equivalence in L_0^T, and the evaluation semantics induced by the two translations is the same, we can thus reason about effects in terms of their (relatively) declarative T-specification, rather than their derived U-implementation.

Remark 3.11 In the particular case of exceptions, we could actually construct an ad-hoc continuation semantics where the translation of a term $M : \alpha$ takes both a *normal* continuation (of type $[\![\alpha]\!] \to o$) and an *exceptional* one (of type $\chi \to o$), invoking whichever is appropriate. Such a translation does verify the handle/re-raise equation above, and it does not have a problem with the choice of answer type.

However, such a scheme requires *all* translation equations to be modified to pass the extra continuation along, so we cannot use a standard cps transform for the bulk of the language. And even more importantly, this two-continuations trick does not generalize, because it relies on the isomorphism $((\alpha + \chi) \to o) \to o \cong ((\alpha \to o) \times (\chi \to o)) \to o$, which does not have a counterpart for other monadic effects. ■

3.2 The proof setting

This section establishes the general framework for the simulation proof in the next section. Much of the material is relatively standard, and has consequently been relegated to an appendix.

3.2.1 The implementation language

The base signature L_0, and the derived L_0^T need to be tightly constrained because we will rely on induction over L_0^T-types and -terms in the proof. The target language for the variant translation, on the other hand, need not be restricted to simple types. And in fact, to obtain the simulation result for continuations in full generality, we will need more of the structure of our predomain model to be denotable in the implementation language. Accordingly, we now define the required extensions for expressing (1) a weak notion of infinitary sums and (2) recursively-defined types.

Embedding-types

To simulate T-reification using continuations, we will need to embed several different types into a single type of answers. A suitable construct for expressing this is given by the following:

Definition 3.12 *The signature L_0^Σ extends L_0 with a new type constructor Σ:*

$$\frac{\forall i \in I.\ \vdash_\Delta \aleph(i)\ \mathsf{type}}{\vdash_\Delta \Sigma_i \aleph(i)\ \mathsf{type}}$$

where $(\aleph(i))_{i\in I}$ is any countable family of L_0^Σ-types (possibly with repetitions); we usually abbreviate $\Sigma_i\aleph(i)$ as $\Sigma\aleph$. The associated term constructors are:

$$\frac{\Gamma \vdash M : \aleph(i)}{\Gamma \vdash \mathsf{in}_i M : \Sigma\aleph}{(i\in I)} \qquad \textit{and} \qquad \frac{\Gamma \vdash M : \Sigma\aleph}{\Gamma \vdash \mathsf{outd}_i M : \aleph(i) + 1}{(i\in I)}$$

for injecting *into and* projecting *from the embedding-type. Correspondingly,* $\mathcal{E}_0^{\Sigma}$ *extends* $\mathcal{E}_0$ *with the equations*

$$\frac{\Gamma \vdash M : \aleph(i)}{\Gamma \vdash \mathsf{outd}_i\,(\mathsf{in}_i\,M) = \mathsf{inl}\,M : \aleph(i) + 1}{}^{(i \in I)}$$

and

$$\frac{\Gamma \vdash M : \aleph(i)}{\Gamma \vdash \mathsf{outd}_{i'}\,(\mathsf{in}_i\,M) = \mathsf{inr}\,\langle\rangle : \aleph(i') + 1}{}^{(i,i' \in I;\, i' \neq i)}\,.$$

From outd_i*, we can define a derived term constructor,*

$$\frac{\Gamma \vdash M : \Sigma\aleph}{\Gamma \vdash \mathsf{out}_i\,M : {}^{\circ}\aleph(i)}{}^{(i \in I)}$$

by $\mathsf{out}_i\,M \stackrel{\text{def}}{=} \mathsf{case}\,(\mathsf{outd}_i\,M, a.{}^{\circ}a, u.\bot_{{}^{\circ}\aleph(i)})$*. Then we easily get the following derived inference rule in* $\mathcal{E}_0^{\Sigma}$*:*

$$\frac{\Gamma \vdash M : \aleph(i)}{\Gamma \vdash \mathsf{out}_i\,(\mathsf{in}_i\,M) = {}^{\circ}M : {}^{\circ}\aleph(i)}{}^{(i \in I)}.$$

In this chapter, in_i and out_i with the above equation will suffice (in particular, we will *not* use that $\mathsf{out}_{i'}\,(\mathsf{in}_i\,M) = \bot$ when $i \neq i'$), but in Chapter 4, an explicit outd_i, not tied to any particular notion of ambient effects, will be more convenient.

It is important that even for infinite index sets, embedding types do not introduce any circularity: each summand $\aleph(i)$ must already be a well-defined type before we can form $\Sigma\aleph$.

When the index set is finite, $\{i_0, \ldots, i_{n-1}\}$, we can simply take

$$\Sigma\aleph = \aleph(i_0) + (\cdots + (\aleph(i_{n-1}) + 1)\cdots)$$

(the terminating 1 merely ensures a uniform encoding for all summands) with the corresponding operations:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{in}_{i_0}\,M & = & \mathsf{inl}\,M & \mathsf{outd}_{i_0}\,M & = & \mathsf{case}\,(M, a_0.\,\mathsf{inl}\,a, s.\,\mathsf{inr}\,\langle\rangle) \\
\mathsf{in}_{i_{k+1}}\,M & = & \mathsf{inr}\,(\mathsf{in}_{i_k}\,M) & \mathsf{outd}_{i_{k+1}}\,M & = & \mathsf{case}\,(M, a.\,\mathsf{inr}\,\langle\rangle, s.\mathsf{outd}_{i_k}\,s)
\end{array}$$

which are easily seen to satisfy the required equations.

In the general case, we obtain a model by a straightforward extension of the predomain semantics to I-indexed coproducts:

$$\begin{array}{rcl}
\mathcal{L}[\![\Sigma_i\aleph(i)]\!]^{\theta} & = & \{(i,a) \mid i \in I, a \in \mathcal{L}[\![\aleph(i)]\!]^{\theta}\} \\
\mathcal{L}[\![\mathsf{in}_i\,M]\!]^{\theta}(\rho) & = & (i, \mathcal{L}[\![M]\!]^{\theta}(\rho)) \\
\mathcal{L}[\![\mathsf{outd}_i\,M]\!]^{\theta}(\rho) & = & \begin{cases} (1,a) & \text{when } \mathcal{L}[\![M]\!]^{\theta}(\rho) = (i,a) \\ (2,*) & \text{when } \mathcal{L}[\![M]\!]^{\theta}(\rho) = (i',a'), i' \neq i \end{cases}
\end{array}$$

It is immediate to check that this interpretation validates the equations.

Recursive type specifications

Independently of the embedding-types, to express the continuation-based variant translation of reification at types containing ', we will need a recursively-defined answer type. Accordingly, we take:

Definition 3.13 *For a signature L, L^μ extends L with a new type constructor $\mu \mathrm{a}.\, \alpha$ with well-formedness rule*

$$\frac{\vdash_{\{\mathrm{a}\}} \alpha \; \mathsf{type}}{\vdash_{\emptyset} \mu \mathrm{a}.\, \alpha \; \mathsf{type}},$$

and new term constructors $\mathsf{roll}_{\mathrm{a}.\alpha}$ *and* $\mathsf{unroll}_{\mathrm{a}.\alpha}$ *with typings*

$$\frac{\Gamma \vdash M : \alpha\{(\mu \mathrm{a}.\, \alpha)/\mathrm{a}\}}{\Gamma \vdash \mathsf{roll}_{\mathrm{a}.\alpha} M : \mu \mathrm{a}.\, \alpha} \quad \textit{and} \quad \frac{\Gamma \vdash M : \mu \mathrm{a}.\, \alpha}{\Gamma \vdash \mathsf{unroll}_{\mathrm{a}.\alpha} M : \alpha\{(\mu \mathrm{a}.\, \alpha)/\mathrm{a}\}}.$$

Likewise, $\mathcal{E}^\mu$ extends $\mathcal{E}$ with the isomorphism equations

$$\frac{\Gamma \vdash M : \alpha\{(\mu \mathrm{a}.\, \alpha)/\mathrm{a}\}}{\Gamma \vdash \mathsf{unroll}_{\mathrm{a}.\alpha}(\mathsf{roll}_{\mathrm{a}.\alpha} M) = M : \alpha\{(\mu \mathrm{a}.\, \alpha)/\mathrm{a}\}} \quad \textit{and} \quad \frac{\Gamma \vdash M : \mu \mathrm{a}.\, \alpha}{\Gamma \vdash \mathsf{roll}_{\mathrm{a}.\alpha}(\mathsf{unroll}_{\mathrm{a}.\alpha} M) = M : \mu \mathrm{a}.\, \alpha}.$$

(For simplicity we do not allow parameterized recursive types, although it would probably do no harm to include them.)

Unlike the case for domains, not every predomain equation expressed in terms of the standard cpo constructors has a solution. (For example, consider the equation $V \cong V \to 0$, where 0 is the empty set organized as a cpo; both assuming V empty and non-empty lead to a contradiction.) But equations arising from interpretations of L_0^Σ-types (which notably require codomains of arrow types to be computational, thus ruling out the above counterexample) do have solutions, essentially because we can extend the interpretation of a parameterized type to a functor in a suitable category. We will need the following result:

Theorem 3.14 *Let $\vdash_{\{\mathrm{a}\}} \alpha$ be a parameterized type of L_0^Σ. Then there exists a cpo A with an isomorphism $i : \mathcal{L}[\![\alpha]\!]^{a \mapsto A} \xrightarrow{\sim} A$.*

Proof. See Corollary A.8 in the appendix (ignoring for now the additional minimal-invariant property of i). ■

Then with the interpretation of $\mu \mathrm{a}.\, \alpha$ as the A in the theorem, $\mathsf{roll}_{\mathrm{a}.\alpha}$ as i, and $\mathsf{unroll}_{\mathrm{a}.\alpha}$ as i^{-1}, our predomain semantics (for any $\mathcal{T}$) becomes a model of $\mathcal{E}_0^{\Sigma,\mu}$.

3.2.2 Admissible relations

Much as an equational theory allows us to reason about equivalence of terms axiomatically, rather than about equality of their denotations in a specific interpretation, we can reason about more general relations between terms at the syntactic level. That is, we first establish a set of generic relational reasoning principles, validated by a wide range of interpretations. If we then confine our reasoning about programs to those principles, the results will necessarily hold in each particular relational interpretation.

Our denotational semantics associates to every open term a (continuous) function from the meanings assigned to the free variables to the meaning of the resulting term. When the semantics $\mathcal{L}$ is fixed, we use the term constructors of L directly to denote this semantic function. For example, for any element a of $\mathrm{Val}_{\mathcal{L}}(\alpha)$ (not necessarily denotable by a closed L-term), we write $\mathsf{inl}\, a$ for the element $(1, a)$ of $\mathrm{Val}_{\mathcal{L}}(\alpha + \alpha')$.

Further, when $\sigma = (a_1/x_1, \ldots, a_n/x_n)$ assigns to every variable $x_i\colon \alpha_i$ in Γ a value $a_i \in \mathrm{Val}(\alpha_i) = \mathcal{L}[\![\alpha_i]\!]$, we write $M\{\sigma\}$ for the value $\mathcal{L}[\![M]\!](\bullet[x_1 \mapsto a_1, \ldots, x_1 \mapsto \alpha_1])$. (To improve readability, we will usually write $M\{\sigma\}$ as M^σ; the two notations are equivalent.)

Unlike the equational case, we can talk about relations between terms of two *different* languages. That is, given $(L, \mathcal{L})$ and $(L', \mathcal{L}')$ we say that R is a relation between types α of L and α' of L' if it is a relation between the sets $\mathrm{Val}_{\mathcal{L}}(\alpha)$ and $\mathrm{Val}_{\mathcal{L}'}(\alpha')$. When the languages are fixed, we write simply $\mathrm{Rel}(\alpha, \alpha')$ for the set of all such relations. (Actually, we will only be interested in the set of all *admissible* relations; see Definition 3.16 below. But sometimes it is useful to classify a relation wrt. types before we have established that it is admissible.)

The motivation for considering different languages is that when implementing a monadic effect, we may need different resources than when specifying it. In particular, L' may contain constructs not in L, with $\mathcal{L}'$ providing an interpretation for those. Moreover, $\mathcal{L}$ and $\mathcal{L}'$ may arise from different choices of the base-effect monad $\mathcal{T}$.

This means that we can specify a monad $\mathbf{T}$ over $(L, \mathcal{L})$ (say, with only the constructs of L_0 and with partiality as the only ambient effect), and show how to implement L^T using a T'-translation into $(L', \mathcal{L}')$ (say, L_0 extended with recursive types, and a continuation semantics for ambient effects) – even if $\mathbf{T}$ does not satisfy the monad laws in $\mathcal{L}'$.

For a relation $R \in \mathrm{Rel}(\alpha, \alpha')$, we often write $\forall a\ R\ a'.\, P(a, a')$ as shorthand for $\forall a \in \mathrm{Val}_{\mathcal{L}}(\alpha), a' \in \mathrm{Val}_{\mathcal{L}'}(\alpha').\, a\ R\ a' \Rightarrow P(a, a')$. Similarly, $\exists a\ R\ a'.\, P(a, a')$ abbreviates $\exists a \in \mathrm{Val}_{\mathcal{L}}(\alpha), a' \in \mathrm{Val}_{\mathcal{L}'}(\alpha').\, a\ R\ a' \wedge P(a, a')$.

We can now isolate the subset of relations we will be working with. First, we define a relational analog of (pointed) cpos:

Definition 3.15 *A binary relation R between (the sets underlying) cpos A and A' is called* chain-complete *if for any pair of chains $a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ in A and $a'_1 \sqsubseteq a'_2 \sqsubseteq \cdots$ in A' with $a_i\ R\ a'_i$ for each i, it also holds that $(\bigsqcup_i a_i)\ R\ (\bigsqcup_i a'_i)$. A relation between pointed cpos B and B' is called* pointed *if $\bot_B\ R\ \bot_{B'}$.*

We can then define a suitable notion of relations for our predomain semantics:

Definition 3.16 *Let there be given languages $(L, \mathcal{L})$ and $(L', \mathcal{L}')$, where L and L' are extensions of L_0, and $\mathcal{L}$ and $\mathcal{L}'$ are the corresponding extensions of the predomain semantics from Section 2.1.3 (with possibly different ambient-effect monads).*

We say that a relation between types α of L and α' of L' is admissible *if it is interpreted as a chain-complete relation between* $\mathrm{Val}_{\mathcal{L}}(\alpha)$ *and* $\mathrm{Val}_{\mathcal{L}'}(\alpha')$*; we write* $\mathrm{ARel}(\alpha, \alpha')$ *for the set of such relations. Similarly, a* computation-admissible *relation between types β and*

β' is one whose interpretation is chain-complete and pointed; we use $\mathrm{CARel}(\beta, \beta')$ *for the corresponding set.*

We will refer to such a pair of languages with type-indexed sets of (computation-) admissible relations as a relational correspondence. *(This is for conciseness only; the correspondence is already fully determined by the languages themselves and the above definitions of* $\mathrm{ARel}(\alpha, \alpha')$ *and* $\mathrm{CARel}(\beta, \beta')$*.)*

In the following, we enumerate some properties of (computation-)admissible relations, especially that certain stylized methods of constructing them are available. The proofs are all fairly simple and can be found in the Appendix.

Given these properties, we can reason about related terms entirely within the base language, without referring to the semantic equations, chains, continuity, etc. explicitly. That is, although we will not consider other notions of (computation-)admissibility than (pointed) chain-completeness, the remainder of this section establishes all we actually need to require of admissible relations for establishing the results in Section 3.3.

Lemma 3.17 *Admissible relations are closed under inverse image by term contexts and under arbitrary intersection. That is,*

1. *When* $R \in \mathrm{ARel}(\alpha, \alpha')$, $(x_1{:}\,\alpha_1, \ldots, x_n{:}\,\alpha_n) \vdash M : \alpha$ *and* $(x'_1{:}\,\alpha'_1, \ldots, x'_{n'}{:}\,\alpha'_{n'}) \vdash M' : \alpha'$ *are terms of L and L' respectively, and for all* $i \geq 2$, $\sigma x_i \in \mathrm{Val}_{\mathcal{L}}(\alpha_i)$ *and* $\sigma' x'_i \in \mathrm{Val}_{\mathcal{L}'}(\alpha'_i)$, *the relation* $R_1 \in \mathrm{Rel}(\alpha_1, \alpha'_1)$ *given by*

   $$a_1\ R_1\ a'_1 \iff M^{(a_1/x_1,\sigma)}\ R\ M'^{(a'_1/x'_1,\sigma')}$$

   is admissible.

   Moreover, when α_1 and α'_1 are computation-types, R is computation-admissible, and the functions $\lambda x_1.\, M^{\sigma}$ and $\lambda x'_1.\, M'^{\sigma'}$ are rigid, then R_1 is also computation-admissible.

2. *When $(R_j)_{j \in J}$ is an arbitrary (not necessarily finite or even countable) family of admissible relations between α and α', the relation $\bigcap_{j \in J} R_j$ is admissible, where*

   $$a\ (\textstyle\bigcap_{j \in J} R_j)\ a' \iff \forall j \in J.\, a\ R_j\ a'$$

   Moreover, if each R_j is computation-admissible then so is $\bigcap_{j \in J} R_j$.

Proof. See Lemma A.9. ■

We also have a simple way of combining existing relations on individual types into relations on constructed ones:

Lemma 3.18 *The* standard relational actions *of the type constructors, defined by*

$$\begin{aligned}
i\ \iota^{\mathrm{r}}\ i' &\iff \exists n \in \mathbf{N}.\, i = \underline{n} \wedge i' = \underline{n} \\
u\ 1^{\mathrm{r}}\ u' &\iff \mathit{true} \\
p\ (R_1 \times^{\mathrm{r}} R_2)\ p' &\iff \mathsf{fst}\, p\ R_1\ \mathsf{fst}\, p' \wedge \mathsf{snd}\, p\ R_2\ \mathsf{snd}\, p' \\
s\ (R_1 +^{\mathrm{r}} R_2)\ s' &\iff (\exists a_1\ R_1\ a'_1.\, s = \mathsf{inl}\, a_1 \wedge s' = \mathsf{inl}\, a'_1) \\
&\qquad \vee (\exists a_2\ R_2\ a'_2.\, s = \mathsf{inr}\, a_2 \wedge s' = \mathsf{inr}\, a'_2) \\
f\ (R_1 \to^{\mathrm{r}} R_2)\ f' &\iff \forall a\ R_1\ a'.\, f\, a\ R_2\ f'\, a'
\end{aligned}$$

are admissible. Specifically, when all the R's are admissible, so are ι^{r}, 1^{r}, $R_1 \times^{\mathrm{r}} R_2$, $R_1 +^{\mathrm{r}} R_2$, *and* $R_1 \to^{\mathrm{r}} R_2$*. Moreover, when the S's are computation-admissible, so are* 1^{r}, $S_1 \times^{\mathrm{r}} S_2$, *and* $R \to^{\mathrm{r}} S$.

(We often omit the $-^{\mathrm{r}}$ when it is clear form the context that the action is on relations.) Note that the relational action of $^{\circ}$ is not in general explicitly definable within the language; we characterize it in Definition 3.20.

Proof. We can actually show admissibility of 1^{r}, $R_1 \times^{\mathrm{r}} R_2$ and $R_1 \to^{\mathrm{r}} R_2$ using only Lemma 3.17. The first case is simply an empty intersection of (computation-)admissible relations. The constructed relation for products is the intersection of the two admissible relations obtained by inverse images of the projections on the admissible relations R_1 and R_2. Moreover, since projections are rigid, $R_1 \times R_2$ is also computation-admissible if both R_1 and R_2 are. Finally, $R_1 \to R_2$ can be expressed as an intersection of the family $(R'_j)_{j \in R_1}$ of admissible relations, where each R'_j is given by an inverse-image construction: $f \; R'_{(a,a')} \; f' \iff f\, a \; R_2 \; f'\, a'$. And again, since application is rigid, computation-admissibility of R_2 implies computation-admissibility of $R_1 \to R_2$.

The cases for natural numbers and sums, on the other hand, depend on the specifics of the model; see Lemma A.10(1,2). ■

The reason for restricting attention to (computation-)admissible relations is that they validate the following binary version of *fixed-point induction*:

Lemma 3.19 *Let* $S \in \mathrm{CARel}(\beta, \beta')$*, and let* $f \in \mathrm{Val}_{\mathcal{L}}(\beta \to \beta)$ *and* $f' \in \mathrm{Val}_{\mathcal{L}'}(\beta' \to \beta')$ *be such that* $\forall b \; S \; b'.\, f\, b \; S \; f'\, b'$*. Then* $\mathsf{fix}_{\beta}\, f \; S \; \mathsf{fix}_{\beta'}\, f'$.

Proof. See Lemma A.11. ■

Effectively, this is saying that for any computation-admissible relation S, the two interpretations of fix are related by $(S \to^{\mathrm{r}} S) \to^{\mathrm{r}} S$.

3.2.3 Computation-extension of relations

A key concept we will make use of in the following is the *extension* of a value-relation to a relation on computations. Intuitively, two computations are considered related if they both have the same (or, more generally, related) effects, and if any results they pass on to further computations are related by the original relation.

For example, in the case of partiality, two computations are related if they either both diverge, or both converge to related values. For exceptions, two computations are related if they produce related successful answers, or raise the same exception. For state (given a fixed relation on states), they must map related initial states to related values and related final states. And for two control-computations to be related, when invoked with related continuations (i.e., mapping related values to related final answers), they must themselves produce related final answers.

Much as monads abstract out the common equational properties of effects into a simple set of axioms for the unit and extension functions, we can characterize the minimal requirements for a relation-extension as follows:

Definition 3.20 *A* computation-extension *of relations assigns to any pair of types α and α', and admissible relation $R \in \mathrm{ARel}(\alpha, \alpha')$, a computation-admissible relation $^\circ R \in \mathrm{CARel}(^\circ\alpha, {}^\circ\alpha')$, such that for all admissible R, R_1 and R_2, the following holds:*

1. $\forall a\ R\ a'.\ {}^\circ a\ (^\circ R)\ {}^\circ a'$.
2. *If* $\forall a\ R_1\ a'.\ f\,a\ (^\circ R_2)\ f'\,a'$ *then* $\forall m\ (^\circ R_1)\ m'.\ \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ f\,x\ (^\circ R_2)\ \mathbf{let}^\circ\ x' \Leftarrow m'\ \mathbf{in}\ f'\,x'$.

That is, if two terms are related as values, their inclusions into computations must also be related. And if two parameterized computations are related for every pair of related parameters, they must remain related when prefixed by related computations computing values for those parameters. A simple instance is given by the following:

Proposition 3.21 *In the standard partiality semantics (i.e., with $\mathcal{T}A = \mathcal{T}'A = A_\perp$), taking for any R, $^\circ R$ to be the* lifting *of R, i.e.,*

$$m\ (^\circ R)\ m' \iff (\exists v\ R\ v'.\ m = {}^\circ v \wedge m' = {}^\circ v') \vee (m = \perp_{^\circ\alpha} \wedge m' = \perp_{^\circ\alpha'})$$

determines a computation-extension.

Proof. Lemma A.10(4) shows that $^\circ R$ is computation-admissible. Condition (1) ($v\ R\ v' \Rightarrow {}^\circ v\ (^\circ R)\ {}^\circ v'$) is also immediate. For (2), assume $m\ (^\circ R_1)\ m'$ and $\forall a\ R_1\ a'.\ f\,a\ (^\circ R_2)\ f'\,a'$. There are two cases, one for each disjunct in the definition of $^\circ R_1$:

- $m = {}^\circ v$ and $m' = {}^\circ v'$ for some $v\ R_1\ v'$. Then we get the result directly by assumption on f and f':

  $$\mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ f\,x = f\,v\ (^\circ R_2)\ f'\,v' = \mathbf{let}^\circ\ x' \Leftarrow m'\ \mathbf{in}\ f'\,x'$$

- $m = \perp$, $m' = \perp$. Then, by the second disjunct in the definition of $^\circ R_2$,

  $$\mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ f\,x = \perp\ (^\circ R_2)\ \perp = \mathbf{let}^\circ\ x' \Leftarrow m'\ \mathbf{in}\ f'\,x'$$

  (using Proposition 2.13 to obtain the equalities)

■

We will see later (Proposition 3.40) how to systematically construct computation-extensions for effects defined by an explicit monadic translation.

3.3 The simulation proof

To avoid repetition in the following, we first define:

Definition 3.22 (Persistent assumption) *Throughout this section, we will assume that there is given a relational correspondence between interpretations $\mathcal{L}_\mathrm{s}$ of L_0 (the* specification language*) and $\mathcal{L}_\mathrm{i}$ of a signature $L_0^+ \supseteq L_0$ (the* implementation language*), with a fixed computation-extension of relations. In particular, all unqualified occurrences of* $\mathrm{ARel}(\alpha, \alpha')$, $\mathrm{CARel}(\beta, \beta')$, *and $^\circ R$ will refer to this correspondence.*

For concreteness, it may help to think of L_0^+ as L_0 extended with recursive types and embedding types (which is what we will use in the monad-continuation case in Section 3.3.4), $\mathcal{L}_{\mathrm{s}}$ as the partiality semantics and $\mathcal{L}_{\mathrm{i}}$ as a continuation semantics (for base computations, not to be confused with a continuation monad defined in $(L_0^+, \mathcal{L}_{\mathrm{i}})$; we will use such a continuation-based interpretation of ambient effects in Chapter 4).

3.3.1 Overview

A monad morphism i gives us a simple way of converting a T-computation t to the U-computation representing it, by taking $u = \mathsf{i}_\alpha t$. However, this simple relationship does not extend directly to *functions* on computations. For example, given a function $f : T\alpha \to T\alpha'$, how would we obtain the $g : U\alpha \to U\alpha'$, representing f?

If i has a left inverse j, we could try taking $g = \mathsf{i}_{\alpha'} \circ f \circ \mathsf{j}_\alpha$. This is not really satisfactory, however: for example, when $\alpha = \alpha'$ and f is the identify function, we get $g : U\alpha \to U\alpha = \mathsf{i}_\alpha \circ \mathsf{j}_\alpha$, meaning that the identity on $U\alpha$ would in general not be *the* correct representation of the identity on $T\alpha$.

A better approach, therefore, is to characterize instead what *a* correct U-based representation g of f would be, for example by requiring it to satisfy the equation $\mathsf{i}_{\alpha'} \circ f = g \circ \mathsf{i}_\alpha$. In general, then, instead of a *function* from higher-order values involving T-computations to the corresponding ones with U-computations, we get a (binary) *relation*.

The general outline of the proof is then as follows. First, for any type family α in L_0, we define a family of logical relations $\langle\!\langle \alpha \rangle\!\rangle$, and show that for any relational interpretation of the type parameters, the two interpretations of a term family M of L_0 are related by $\langle\!\langle \alpha \rangle\!\rangle$. In particular, this means that the term components of any monad-triple $\mathbf{T}$ in L_0 are related in the two interpretations. This gives us a way of talking about related T-effects in the two languages, even when $\mathbf{T}$ is not a monad in $\mathcal{L}_{\mathrm{i}}$.

We then define the general notion of a *monad relation* between a monad $\mathbf{T}$ in the specification language and a monad $\mathbf{U}$ in the implementation language. This is a more general notion than existence of a monad morphism from $\mathbf{T}$ to $\mathbf{U}$: instead of assigning to every α a *function* from $T\alpha$ to $U\alpha$, we only assign a binary *relation*. More precisely, to every relation $R \in \mathrm{ARel}(\alpha, \alpha')$, we assign a relation $\text{'}R \in \mathrm{ARel}(T\alpha, U\alpha')$.

However, if $\mathbf{U}$ is also a monad in the specification language, any monad morphism i from $\mathbf{T}$ to $\mathbf{U}$ induces in a canonical way a monad relation between $\mathbf{T}$ and $\mathbf{U}$, by taking $t\ (\text{'}R)\ u \iff \mathsf{i}t\ (UR)\ u$, where UR is the standard, syntactically-derived action of U on relations. This way of constructing monad relations covers most of our sample monad simulations – all except the general monad-continuation case.

Given a monad relation between $\mathbf{T}$ and $\mathbf{U}$, we can now exhibit a family of relations indexed by L_0^T-types α, $\trianglelefteq_\alpha \in \mathrm{ARel}([\![\alpha]\!]_T, [\![\alpha]\!]_U)$, defined in the usual inductive way for the standard type constructors, and taking $\trianglelefteq_{\text{'}\alpha} = \text{'}\trianglelefteq_\alpha \in \mathrm{ARel}(T[\![\alpha]\!]_T, U[\![\alpha]\!]_U)$.

Further, we show that the two translations, $[\![\text{-}]\!]_T : L_0^T \to L_0$ and $[\![\text{-}]\!]'_T : L_0^T \to L_0^+$ of an L_0^T-term of type α are related by $\trianglelefteq_\alpha$. Since in particular $\trianglelefteq_{\circ\iota} = {}^{\circ}(\iota^{\mathrm{r}})$, this says that the two translations coincide for complete programs.

Returning to the monad-continuation case, we show how to construct the appropriate monad relation between $\mathbf{T}$ and $\mathbf{K}_o$ explicitly. (Intuitively, we need the generalization from an equational to a relational characterization of the relationship between the monads for the same reason that forced us to adapt a relational approach for higher-order values: when we embed non-simple types in the answer type o of the continuation monad, the K_o-based representation of T-effects will in general not be unique.)

For this simulation, we also rely on the fact that our implementation language may contain additional types and terms beyond those in L_0; in particular, depending on how general a notion of T-reification we want to simulate, we will need recursive types and/or embedding-types to construct $\mathbf{K}_o$. Once we have established the monad relation, it is a simple consequence of the properties of $\trianglelefteq$ that the definitional T-translation and the variant, or *continuation-passing*, translation agree on complete programs.

Finally, we show how to lift the simulation result from a relationship between translations to a relationship between source terms. Specifically, the basic motivation for adding reflection and reification to our source language was precisely to permit programs to be written in direct style instead of in effect-passing style. And in fact, it is possible to express the simulation result at the source level, by defining the reflection and reification operators for $\mathbf{T}$ in terms of those for $\mathbf{K}_o$. Thus, we do not need a monad-specific variant translation for implementing $\mathbf{T}$-effects with continuations, but can use a fixed continuation-passing translation for all such effects.

3.3.2 Relating standard terms

We first show that a large collection of terms are related by the relations determined systematically from their types:

Definition 3.23 *Let Δ be a finite set of type variables, θ a substitution of closed L_0-types for variables in Δ, and θ' of closed L_0^+-types. Further, let ϱ assign to each type variable $\mathrm{a} \in \Delta$ a relation $\varrho\,\mathrm{a} \in \mathrm{ARel}(\theta\,\mathrm{a}, \theta'\,\mathrm{a})$. To every type α over Δ in L_0, we then assign a relation $\langle\!\langle \alpha \rangle\!\rangle^{\varrho} \in \mathrm{ARel}(\alpha\{\theta\}, \alpha\{\theta'\})$ (such that $\langle\!\langle \beta \rangle\!\rangle^{\varrho} \in \mathrm{CARel}(\beta\{\theta\}, \beta\{\theta'\})$) as follows:*

$$\begin{array}{rcl}
\langle\!\langle \mathrm{a} \rangle\!\rangle^{\varrho} & = & \varrho\,\mathrm{a} \\
\langle\!\langle \iota \rangle\!\rangle^{\varrho} & = & \iota^{\mathrm{r}} \\
\langle\!\langle 1 \rangle\!\rangle^{\varrho} & = & 1^{\mathrm{r}} \\
\langle\!\langle \alpha_1 \times \alpha_2 \rangle\!\rangle^{\varrho} & = & \langle\!\langle \alpha_1 \rangle\!\rangle^{\varrho} \times^{\mathrm{r}} \langle\!\langle \alpha_2 \rangle\!\rangle^{\varrho} \\
\langle\!\langle \alpha_1 + \alpha_2 \rangle\!\rangle^{\varrho} & = & \langle\!\langle \alpha_1 \rangle\!\rangle^{\varrho} +^{\mathrm{r}} \langle\!\langle \alpha_2 \rangle\!\rangle^{\varrho} \\
\langle\!\langle \alpha \to \beta \rangle\!\rangle^{\varrho} & = & \langle\!\langle \alpha \rangle\!\rangle^{\varrho} \to^{\mathrm{r}} \langle\!\langle \beta \rangle\!\rangle^{\varrho} \\
\langle\!\langle {}^{\circ}\alpha \rangle\!\rangle^{\varrho} & = & {}^{\circ}\langle\!\langle \alpha \rangle\!\rangle^{\varrho}
\end{array}$$

We extend this definition pointwise to relate value-substitutions σ and σ', i.e., if for each $(x_i\colon \alpha_i) \in \Gamma$, $(\sigma x_i)\ \langle\!\langle \alpha_i \rangle\!\rangle^{\varrho}\ (\sigma' x_i)$, we write $\sigma\ \langle\!\langle \Gamma \rangle\!\rangle^{\varrho}\ \sigma'$.

The (computation-)admissibility of these relation follows directly from Lemma 3.18. It is also easy to see that we have the usual weakening and substitution principles,

$$\langle\!\langle \alpha \rangle\!\rangle^{\varrho} = \langle\!\langle \alpha \rangle\!\rangle^{\varrho[\mathrm{a} \mapsto R]}\ (\mathrm{a} \notin FTV(\alpha)) \quad \text{and} \quad \langle\!\langle \alpha\{\alpha'/\mathrm{a}\} \rangle\!\rangle^{\varrho} = \langle\!\langle \alpha \rangle\!\rangle^{\varrho[\mathrm{a} \mapsto \langle\!\langle \alpha' \rangle\!\rangle^{\varrho}]}$$

Lemma 3.24 (logical relations lemma) *Let θ and θ' be substitutions of closed types for Δ-variables, and ϱ a relation assignment for θ and θ' (as in Definition 3.23). Further, let Γ and α be a type assignment and a type over Δ, both in L_0, and let M be a term of L_0 with $\Gamma \vdash_\Delta M : \alpha$. Finally, let σ and σ' be substitutions of values from $\mathcal{L}_\mathrm{s}$ and $\mathcal{L}_\mathrm{i}$ for variables in Γ, such that $\sigma \; \langle\!\langle \Gamma \rangle\!\rangle^\varrho \; \sigma'$. Then $M^{\theta\sigma} \; \langle\!\langle \alpha \rangle\!\rangle^\varrho \; M^{\theta'\sigma'}$.*

Proof. By induction on the structure of M:

- Case x_i, where $(x_i\colon \alpha_i) \in \Gamma$. To show: $x_i^{\theta\sigma} \; \langle\!\langle \alpha_i \rangle\!\rangle^\varrho \; x_i^{\theta'\sigma'}$, i.e., that $\sigma x_i \; \langle\!\langle \alpha_i \rangle\!\rangle^\varrho \; \sigma' x_i$, which follows directly from the assumption on σ and σ'.
- Case z. To show:
$$\exists n \in \mathbf{N}.\, \mathsf{z}^{\theta\sigma} = \underline{n} \wedge \mathsf{z}^{\theta'\sigma'} = \underline{n}$$
Since $\mathsf{z}^{\theta\sigma} = \mathsf{z}$, we can simply take $n = 0$.
- Case $\mathsf{s}\,M$. To show:
$$\exists n \in \mathbf{N}.\, \mathsf{s}\,(M^{\theta\sigma}) = \underline{n} \wedge \mathsf{s}\,(M^{\theta'\sigma'}) = \underline{n}$$
By IH on M, we already have
$$\exists m \in \mathbf{N}.\, M^{\theta\sigma} = \underline{m} \wedge M^{\theta'\sigma'} = \underline{m}$$
so we get the result by taking $n = m + 1$.
- Case $\mathsf{ifz}\,(M, M_z, x.\, M_s)$. To show:
$$\mathsf{ifz}\,(M^{\theta\sigma}, M_z^{\theta\sigma}, x.\, M_s^{\theta\sigma}) \; \langle\!\langle \alpha \rangle\!\rangle^\varrho \; \mathsf{ifz}\,(M^{\theta'\sigma'}, M_z^{\theta'\sigma'}, x.\, M_s^{\theta'\sigma'})$$
By IH on M, we know that (in $\mathcal{L}_\mathrm{s}$) $M^{\theta\sigma} = \underline{n}$ and (in $\mathcal{L}_\mathrm{i}$) $M^{\theta'\sigma'} = \underline{n}$ for some natural number n. There are then two cases:
  - Case $n = 0$. By IH on M_z, we get
  $$\mathsf{ifz}\,(\mathsf{z}, M_z^{\theta\sigma}, x_s.\, M_s^{\theta\sigma}) = M_z^{\theta\sigma} \; \langle\!\langle \alpha \rangle\!\rangle^\varrho \; M_z^{\theta'\sigma'} = \mathsf{ifz}\,(\mathsf{z}, M_z^{\theta'\sigma'}, x_s.\, M_s^{\theta'\sigma'})$$
  - Case $n = m + 1$. Then
  $$\mathsf{ifz}\,(\mathsf{s}\,\underline{m}, M_z^{\theta\sigma}, x.\, M_s^{\theta\sigma}) = M_s^{\theta\sigma}\{\underline{m}/x\} = M_s^{\theta(\sigma,(\underline{m}/x))}$$
  and analogously on the RHS, so the result follows by IH on M_s, in the type assignment $(\Gamma, x\colon \iota)$ and the extended substitutions $(\sigma, \underline{m}/x)$ and $(\sigma', \underline{m}/x)$.
- Case $\langle\rangle$. To show:
$$\langle\rangle^{\theta\sigma} \; \langle\!\langle 1 \rangle\!\rangle^\varrho \; \langle\rangle^{\theta'\sigma'}$$
which is trivially true by the definition of $\langle\!\langle 1 \rangle\!\rangle^\varrho$.
- Case $\langle M_1, M_2 \rangle$. To show:
$$\langle M_1, M_2 \rangle^{\theta\sigma} \; \langle\!\langle \alpha_1 \times \alpha_2 \rangle\!\rangle^\varrho \; \langle M_1, M_2 \rangle^{\theta'\sigma'}$$
That is,

$$\begin{aligned}&\mathsf{fst}\,(\langle M_1, M_2\rangle^{\theta\sigma})\ \langle\!\langle \alpha_1\rangle\!\rangle^{\varrho}\ \mathsf{fst}\,(\langle M_1, M_2\rangle^{\theta'\sigma'})\\ &\quad\wedge\ \mathsf{snd}\,(\langle M_1, M_2\rangle^{\theta\sigma})\ \langle\!\langle \alpha_2\rangle\!\rangle^{\varrho}\ \mathsf{snd}\,(\langle M_1, M_2\rangle^{\theta'\sigma'})\end{aligned}$$

Since we have

$$\mathsf{fst}\,(\langle M_1, M_2\rangle^{\theta\sigma}) = \mathsf{fst}\,\langle M_1^{\theta\sigma}, M_2^{\theta\sigma}\rangle = M_1^{\theta\sigma}$$

and analogously on the RHS and for snd, we get the result by IH on M_1 and M_2.

- Case $\mathsf{fst}\, M$ ($\mathsf{snd}\, M$ is analogous). To show:

  $$\mathsf{fst}\,(M^{\theta\sigma})\ \langle\!\langle \alpha_1\rangle\!\rangle^{\varrho}\ \mathsf{fst}\,(M^{\theta'\sigma'})$$

  which we get from IH on M and the first conjunct of the definition of $\langle\!\langle \alpha_1 \times \alpha_2\rangle\!\rangle^{\varrho}$.

- Case $\mathsf{inl}\, M$ ($\mathsf{inr}\, M$ is analogous). To show:

  $$\mathsf{inl}\, M^{\theta\sigma}\ \langle\!\langle \alpha_1 + \alpha_2\rangle\!\rangle^{\varrho}\ \mathsf{inl}\, M^{\theta'\sigma'}$$

  By the first disjunct in the definition of $\langle\!\langle \alpha_1 + \alpha_2\rangle\!\rangle^{\varrho}$, it suffices to show that

  $$M^{\theta\sigma}\ \langle\!\langle \alpha_1\rangle\!\rangle^{\varrho}\ M^{\theta'\sigma'}$$

  which we get from IH on M.

- Case $\mathsf{case}\,(M, x_1.M_1, x_2.M_2)$. To show:

  $$\mathsf{case}\,(M^{\theta\sigma}, x_1.M_1^{\theta\sigma}, x_2.M_2^{\theta\sigma})\ \langle\!\langle \alpha\rangle\!\rangle^{\varrho}\ \mathsf{case}\,(M^{\theta'\sigma'}, x_1.M_1^{\theta'\sigma'}, x_2.M_2^{\theta'\sigma'})$$

  By IH on M, we have $M^{\theta\sigma}\ \langle\!\langle \alpha_1 + \alpha_2\rangle\!\rangle^{\varrho}\ M^{\theta'\sigma'}$. Without loss of generality, assume that we are in the first case of the definition of $\langle\!\langle \alpha_1 + \alpha_2\rangle\!\rangle^{\varrho}$. That is, $M^{\theta\sigma} = \mathsf{inl}\, a_1$ and $M^{\theta'\sigma'} = \mathsf{inl}\, a_1'$ for some $a_1\ \langle\!\langle \alpha_1\rangle\!\rangle^{\varrho}\ a_1'$. Then

  $$\begin{aligned}&\mathsf{case}\,(M^{\theta\sigma}, x_1.M_1^{\theta\sigma}, x_2.M_2^{\theta\sigma}) = \mathsf{case}\,(\mathsf{inl}\, a_1, x_1.M_1^{\theta\sigma}, x_2.M_2^{\theta\sigma}) = M_1^{\theta\sigma}\{a_1/x_1\}\\ &\quad = M_1^{\theta(\sigma,(a_1/x_1))}\end{aligned}$$

  and analogously on the RHS, so we get the result by IH on M_1 using the extended substitutions $\sigma_1 = (\sigma, a_1/x_1)$ and $\sigma_1' = (\sigma', a_1'/x_1)$.

- Case $\lambda x^{\alpha}.\, M$. To show:

  $$(\lambda x^{\alpha}.\, M)^{\theta\sigma}\ \langle\!\langle \alpha \to \beta\rangle\!\rangle^{\varrho}\ (\lambda x^{\alpha}.\, M)^{\theta'\sigma'}$$

  I.e., that

  $$\lambda x^{\alpha\{\theta\}}.\, M^{\theta\sigma}\ \langle\!\langle \alpha \to \beta\rangle\!\rangle^{\varrho}\ \lambda x^{\alpha\{\theta'\}}.\, M^{\theta'\sigma'}$$

  Accordingly, let $a\ \langle\!\langle \alpha\rangle\!\rangle^{\varrho}\ a'$; we must show that

  $$(\lambda x^{\alpha\{\theta\}}.\, M^{\theta\sigma})\, a\ \langle\!\langle \beta\rangle\!\rangle^{\varrho}\ (\lambda x^{\alpha\{\theta'\}}.\, M^{\theta'\sigma'})\, a'$$

  And since

  $$(\lambda x^{\alpha\{\theta\}}.\, M^{\theta\sigma})\, a = M^{\theta\sigma}\{a/x\} = M^{\theta(\sigma,(a/x))}$$

  we get the result by IH on $\Gamma, x{:}\,\alpha \vdash_{\Delta} M : \beta$ using the extended substitutions $\sigma_1 = (\sigma, a/x)$ and $\sigma_1' = (\sigma', a'/x)$.

- Case $M_1\,M_2$. To show:

  $$(M_1^{\theta\sigma})\,(M_2^{\theta\sigma})\ \langle\!\langle\beta\rangle\!\rangle^{\varrho}\ (M_1^{\theta'\sigma'})\,(M_2^{\theta'\sigma'})$$

  This follows directly from IH on M_1 and M_2, and the definition of $\langle\!\langle\alpha\to\beta\rangle\!\rangle^{\varrho}$.

- Case ${}^\circ M$. To show:

  $${}^\circ(M^{\theta\sigma})\ \langle\!\langle{}^\circ\alpha\rangle\!\rangle^{\varrho}\ {}^\circ(M^{\theta'\sigma'})$$

  By IH on M, $M^{\theta\sigma}\ \langle\!\langle\alpha\rangle\!\rangle^{\varrho}\ M^{\theta'\sigma'}$, so the result follows from the definition of $\langle\!\langle{}^\circ\alpha\rangle\!\rangle^{\varrho}$ and the properties of a computation-extension (Definition 3.20(1)).

- Case $\mathbf{let}^\circ\ x \Leftarrow M_1\ \mathbf{in}\ M_2$. To show:

  $$\mathbf{let}^\circ\ x \Leftarrow M_1^{\theta\sigma}\ \mathbf{in}\ M_2^{\theta\sigma}\ \langle\!\langle{}^\circ\alpha_2\rangle\!\rangle^{\varrho}\ \mathbf{let}^\circ\ x \Leftarrow M_1^{\theta'\sigma'}\ \mathbf{in}\ M_2^{\theta'\sigma'}$$

  By IH on M_1, we have $M_1^{\theta\sigma}\ \langle\!\langle{}^\circ\alpha_1\rangle\!\rangle^{\varrho}\ M_1^{\theta'\sigma'}$, i.e.,

  $$M_1^{\theta\sigma}\ ({}^\circ\langle\!\langle\alpha_1\rangle\!\rangle^{\varrho})\ M_1^{\theta'\sigma'}$$

  Similarly, by IH on M_2 with appropriately extended substitutions, we get

  $$\forall a_1\ \langle\!\langle\alpha_1\rangle\!\rangle^{\varrho}\ a_1'.\, M_2^{\theta(\sigma,(a_1/x))}\ ({}^\circ\langle\!\langle\alpha_2\rangle\!\rangle^{\varrho})\ M_2^{\theta'(\sigma',(a_1'/x))}$$

  And from those two facts and Definition 3.20(2), we get the required result.

- Case $\mathsf{fix}_\beta\,M$. To show:

  $$\mathsf{fix}_{\beta\{\theta\}}\,M^{\theta\sigma}\ \langle\!\langle\beta\rangle\!\rangle^{\varrho}\ \mathsf{fix}_{\beta\{\theta'\}}\,M^{\theta'\sigma'}$$

  By IH on M, we have $M^{\theta\sigma}\ \langle\!\langle\beta\to\beta\rangle\!\rangle^{\varrho}\ M^{\theta'\sigma'}$, i.e.,

  $$\forall b\ \langle\!\langle\beta\rangle\!\rangle^{\varrho}\ b'.\, M^{\theta\sigma}\,b\ \langle\!\langle\beta\rangle\!\rangle^{\varrho}\ M^{\theta'\sigma'}\,b'$$

  The result then follows directly from fixed-point induction (Lemma 3.19), because $\langle\!\langle\beta\rangle\!\rangle^{\varrho}$ is computation-admissible.

■

As a simple corollary of Lemma 3.24, we obtain that the two interpretations of a monad-triple are related:

Lemma 3.25 *Let* $\mathbf{T}$ *be a monad-triple in* L_0. *Then the standard relational action of* T, *given by*

$$t\ (TR)\ t' \iff t\ \langle\!\langle Ta\rangle\!\rangle^{a\mapsto R}\ t'$$

respects the monad operations in the sense that for any R, R_1, *and* R_2, *the following conditions are satisfied:*

0. If $\forall a\ R_1\ a'.\, f\,a\ (TR_2)\ f'\,a'$
then $\forall m\ ({}^\circ R_1)\ m'.\, \mathbf{let}^\circ_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f\,x\ (TR_2)\ \mathbf{let}^\circ_{T\alpha_2'}\ x' \Leftarrow m'\ \mathbf{in}\ f'\,x'$.

1. $\forall a \; R \; a'.\, \eta_\alpha \, a \; (TR) \; \eta_{\alpha'} \, a'$.
2. *If* $\forall a \; R_1 \; a'.\, f\, a \; (TR_2) \; f'\, a'$ *then* $\forall t \; (TR_1) \; t'.\, f^* \, t \; (TR_2) \; f'^* \, t'$

Proof. All cases are simple:

0. By assumption on f and f', we have
$$f \; \langle\!\langle \mathrm{a}_1 \to T\mathrm{a}_2 \rangle\!\rangle^{\mathrm{a}_1 \mapsto R_1, \mathrm{a}_2 \mapsto R_2} \; f'$$
and by assumption on m and m',
$$m \; \langle\!\langle {}^{\circ}\mathrm{a}_1 \rangle\!\rangle^{\mathrm{a}_1 \mapsto R_1, \mathrm{a}_2 \mapsto R_2} \; m'$$
Then use Lemma 3.24 on the term
$$x_f \colon \mathrm{a}_1 \to T\mathrm{a}_2, x_m \colon {}^{\circ}\mathrm{a}_1 \vdash_{\{\mathrm{a}_1, \mathrm{a}_2\}} \mathbf{let}^{\circ}_{T\mathrm{a}_2} \; x \Leftarrow x_m \; \mathbf{in} \; x_f \, x : T\mathrm{a}_2$$
(which, recall, abbreviates a term of the core syntax, given by expanding the generalized let according to the shape of T) with the substitutions $\sigma = (f/x_f, m/x_m)$ and $\sigma' = (f'/x_f, m'/x_m)$.
1. Analogous to above, using the term $x_a \colon \mathrm{a} \vdash_{\{\mathrm{a}\}} \eta_{\mathrm{a}} \, x_a : T\mathrm{a}$.
2. Analogous to above, with $x_f \colon \mathrm{a}_1 \to T\mathrm{a}_2, x_t \colon T\mathrm{a}_1 \vdash_{\{\mathrm{a}_1, \mathrm{a}_2\}} x_f^* \, x_t : T\mathrm{a}_2$.

■

3.3.3 Relating computational structure

We are now ready to characterize what it means for two monads to be related:

Definition 3.26 *A* monad relation *between monads* $\mathbf{T} = (T, \eta, \text{-}^*)$ *in* $(L_0, \mathcal{L}_{\mathrm{s}})$ *and* $\mathbf{U} = (U, \varepsilon, \text{-}^+)$ *in* $(L_0^+, \mathcal{L}_{\mathrm{i}})$ *assigns to every admissible relation* $R \in \mathrm{ARel}(\alpha, \alpha')$ *a computation-admissible relation* ${}'R \in \mathrm{CARel}(T\alpha, U\alpha')$ *such that for all admissible relations* R, R_1, *and* R_2,

0. *If* $\forall a \; R_1 \; a.\, f\, a \; ({}'R_2) \; g\, a'$
then $\forall m \; ({}^{\circ}R_1) \; m'.\, \mathbf{let}^{\circ}_{T\alpha_2} \; x \Leftarrow m \; \mathbf{in} \; f\, x \; ({}'R_2) \; \mathbf{let}^{\circ}_{U\alpha'_2} \; x' \Leftarrow m' \; \mathbf{in} \; g\, x'$.
1. $\forall a \; R \; a'.\, \eta_\alpha \, a \; ({}'R) \; \varepsilon_{\alpha'} \, a'$.
2. *If* $\forall a \; R_1 \; a'.\, f\, a \; ({}'R_2) \; g\, a'$ *then* $\forall t \; ({}'R_1) \; u.\, f^* \, t \; ({}'R_2) \; g^+ \, u$.

Further, we say that an L_0^+*-term* $\mathsf{i}_{\alpha'} : T\alpha' \to U\alpha'$ *is a* reflection function *(with respect to the monad relation) if for any* L_0*-type* α *and relation* $R \in \mathrm{ARel}(\alpha, \alpha')$,

3. $\forall t \; (TR) \; t'.\, t \; ({}'R) \; \mathsf{i}_{\alpha'} t'$

Analogously, a reification function *is an* L_0^+*-term* $\mathsf{j}_{\alpha'} : U\alpha' \to T\alpha'$ *such that for any* α *and* $R \in \mathrm{ARel}(\alpha, \alpha')$,

4. $\forall t \; ({}'R) \; u.\, t \; (TR) \; \mathsf{j}_{\alpha'} u$

(Note that since $\mathbf{T}$ need not satisfy the monad laws in $\mathcal{L}_{\mathrm{i}}$, i cannot in general be a monad morphism. Nor do we explicitly require that $\mathsf{j}_{\alpha'} \circ \mathsf{i}_{\alpha'} = \mathsf{id}_{T\alpha'}$, although it often does hold.)

An important special case is when a monad-triple $\mathbf{T} = \mathbf{U}$ is a monad in both $\mathcal{L}_{\mathrm{s}}$ and $\mathcal{L}_{\mathrm{i}}$. Then Lemma 3.25 shows that the standard relational action of T induces a monad relation between the two copies of $\mathbf{T}$; moreover, for any α', we can simply take $\mathsf{i}_{\alpha'} = \mathsf{j}_{\alpha'} = \mathsf{id}_{T\alpha'}$. (There is still some non-trivial content to the definition in this case, because $\mathcal{L}_{\mathrm{s}}$ and $\mathcal{L}_{\mathrm{i}}$ could be different models, with their ambient effects related only by relation-extension.)

More generally, we have the following convenient way of obtaining monad relations directly from monad morphisms:

Proposition 3.27 *Let* $\mathbf{T}$ *and* $\mathbf{U}$ *be monad-triples in* L_0 *such that* $\mathbf{T}$ *is a monad in* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathbf{U}$ *is a monad in both* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathcal{L}_{\mathrm{i}}$. *Further, let* i *in* $(L_0, \mathcal{L}_{\mathrm{s}})$ *be a monad morphism from* $\mathbf{T}$ *to* $\mathbf{U}$. *Then the assignment to any* $R \in \mathrm{ARel}(\alpha, \alpha')$ *of the* ${}'R \in \mathrm{CARel}(T\alpha, U\alpha')$ *given by*

$$t\ ({}'R)\ u \iff \mathsf{i}_{\alpha}\, t\ (UR)\ u$$

establishes a monad relation between $\mathbf{T}$ *and* $\mathbf{U}$.

Moreover, for any α' *of* L_0^{+}, $\mathsf{i}_{\alpha'}$ *is a reflection function. And if* j *is a schematic retraction of* i *in* $(L_0, \mathcal{L}_{\mathrm{s}})$ *then for any* α' *in* L_0^{+}, $\mathsf{j}_{\alpha'}$ *is a reification function.*

Proof. First, ${}'R$ is computation-admissible, because it is defined as an inverse image of the computation-admissible UR by the rigid functions i_{α} and $\mathsf{id}_{U\alpha'}$. Further, we have:

0. Let $m\ ({}^{\circ}R_1)\ m'$ and $f\ (R_1 \to {}'R_2)\ g$ be given. To show:

   $$\mathsf{i}_{\alpha_2}\,(\mathbf{let}^{\circ}_{T\alpha_2}\ a \Leftarrow m\ \mathbf{in}\ f\,a)\ (UR_2)\ \mathbf{let}^{\circ}_{U\alpha'_2}\ a' \Leftarrow m'\ \mathbf{in}\ g\,a'$$

   By rigidity of i_{α_2} (Definition 3.1(0)) on the LHS, this amounts to showing

   $$\mathbf{let}^{\circ}_{U\alpha_2}\ a \Leftarrow m\ \mathbf{in}\ \mathsf{i}_{\alpha_2}\,(f\,a)\ (UR_2)\ \mathbf{let}^{\circ}_{U\alpha'_2}\ a' \Leftarrow m'\ \mathbf{in}\ g\,a'$$

   We get that from assumption on m and m', and the properties of U's relational action (Lemma 3.25(0)). if we can establish that

   $$\forall a\ R_1\ a'.\, \mathsf{i}_{\alpha_2}\,(f\,a)\ (UR_2)\ g\,a'$$

   And that was precisely the assumption on f and g.

1. Let $a\ R\ a'$. To show: $\mathsf{i}_{\alpha}\,(\eta_{\alpha}\,a)\ (UR)\ \varepsilon_{\alpha'}\,a'$. By Definition 3.1(1) on the LHS, this is equivalent to showing

   $$\varepsilon_{\alpha}\,a\ (UR)\ \varepsilon_{\alpha'}\,a'$$

   which we get from Lemma 3.25(1) for $\mathbf{U}$.

2. Let $t\ ({}'R_1)\ u$ and $f\ (R_1 \to {}'R_2)\ g$ be given as in the hypothesis. We must show that

   $$\mathsf{i}_{\alpha_2}\,(f^{*}\,t)\ (UR_2)\ g^{+}\,u$$

Again, using the property of a monad morphism 3.1(2) on the LHS, this amounts to showing

$$(\mathsf{i}_{\alpha_2} \circ f)^{+}\,(\mathsf{i}_{\alpha_1} t)\ (UR_2)\ g^{+} u$$

Now, by assumption on t and u, we have

$$\mathsf{i}_{\alpha_1} t\ (UR_1)\ u$$

and by assumption on f and g,

$$\forall a\ R_1\ a'.\,(\mathsf{i}_{\alpha_2} \circ f)\,a\ (UR_2)\ g\,a'$$

from which we get the desired result by 3.25(2).

3. Let $t\ (TR)\ t'$; we must show that $t\ (\text{'}R)\ \mathsf{i}_{\alpha'} t'$, i.e., that $\mathsf{i}_{\alpha} t\ (UR)\ \mathsf{i}_{\alpha'} t'$. And since i_{a} was a term of L_0, this follows from Lemma 3.24 by an argument analogous to those in Lemma 3.25.
4. Let $t\ (\text{'}R)\ u$; to show: $t\ (TR)\ \mathsf{j}_{\alpha'} u$. From the assumption on t and u, we have $\mathsf{i}_{\alpha} t\ (UR)\ u$, and hence again by Lemma 3.24 with $M = \mathsf{j}_{\mathrm{a}}$, we get $\mathsf{j}_{\alpha}\,(\mathsf{i}_{\alpha} t)\ (TR)\ \mathsf{j}_{\alpha'} u$. Then cancelling the j_{α} and i_{α} on the LHS gives us the result.

■

A monad relation with reflection and reification functions is exactly what we need to relate the definitional and the variant translation: the monad relation itself relates the computational structure, and the reflection and reification functions, where they exist, convert between effect representations:

Definition 3.28 *Let* $\mathbf{T}$ *be a monad in* $(L_0, \mathcal{L}_{\mathrm{s}})$ *and* $\mathbf{U}$ *a monad in* $(L_0^{+}, \mathcal{L}_{\mathrm{i}})$*, with a monad relation between* $\mathbf{T}$ *and* $\mathbf{U}$*. Then for any type* α *of* L_0^{T}*, the relation*

$$\trianglelefteq_{\alpha}\ \in \mathrm{ARel}(\llbracket\alpha\rrbracket_{T}, \llbracket\alpha\rrbracket_{U})$$

is given in the usual way by induction on the structure of α *(as in Definition 3.23, but without the type variables), and with* $\trianglelefteq_{\text{'}\alpha} = \text{'}\!\trianglelefteq_{\alpha}$ *from the monad relation.*

Note in particular that since the standard relational action of T in Lemma 3.25 is also given by Definition 3.23, we have $\trianglelefteq_{T\alpha} = T\!\trianglelefteq_{\alpha}$ for any L_0^{T}-type α. We can now state:

Proposition 3.29 *Let there be given a monad relation between* $\mathbf{T}$ *and* $\mathbf{U}$*, with a reflection function* i *at every type* $\llbracket\alpha\rrbracket_{U}$ *where* α *is a type of* L_0^{T}*. Further, let* $\aleph$ *be a family of* L_0^{T}*-types, with* j *a reification function at every* $\llbracket\aleph(i)\rrbracket_{U}$*. Let* $\Gamma \vdash M : \alpha$ *be a term of* $L_0^{T[\aleph]}$ *(i.e., with reifications only at types in* $\aleph$*). Then for any pair of substitutions of* $\trianglelefteq$*-related values for the variables in* Γ*,* $\sigma \trianglelefteq_{\Gamma} \sigma'$*, we have* $\llbracket M\rrbracket_{T}^{\sigma} \trianglelefteq_{\alpha} \llbracket M\rrbracket_{T}^{\prime\,\sigma'}$ *(where* $\llbracket\text{-}\rrbracket'_{T}$ *is the variant translation from Definition 3.9).*

Proof. Given the definitions, this is a simple induction on the structure of M. The cases for variables, numbers, products, sums, functions, and base computations are exactly as before (Lemma 3.24). For the remaining constructs, we have:

- Case $\mathtt{'}M$. To show:
$$\eta\,[\![M]\!]^{\sigma}_{T} \trianglelefteq_{\mathtt{'}\alpha} \varepsilon\,[\![M]\!]'^{\sigma'}_{T}$$
Follows from IH on M and Definition 3.26(1).

- Case $\mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ M_2$ (of type $\mathtt{'}\alpha_2$). To show:
$$\mathbf{let}^{\circ}_{T[\![\alpha_2]\!]_T}\ x \Leftarrow [\![M_1]\!]^{\sigma}_{T}\ \mathbf{in}\ [\![M_2]\!]^{\sigma}_{T} \trianglelefteq_{\mathtt{'}\alpha_2} \mathbf{let}^{\circ}_{U[\![\alpha_2]\!]_U}\ x \Leftarrow [\![M_1]\!]'^{\sigma'}_{T}\ \mathbf{in}\ [\![M_2]\!]'^{\sigma'}_{T}$$
By IH on M_1 and M_2, and 3.26(0).

- Case $\mathbf{let'}\ x \Leftarrow M_1\ \mathbf{in}\ M_2$. To show:
$$(\lambda x.\,[\![M_2]\!]^{\sigma}_{T})^{*}\,[\![M_1]\!]^{\sigma}_{T} \trianglelefteq_{\mathtt{'}\alpha_2} (\lambda x.\,[\![M_2]\!]'^{\sigma'}_{T})^{+}\,[\![M_1]\!]'^{\sigma'}_{T}$$
Follows from IH on M_1 and M_2 using 3.26(2).

- Case $\mu(M)$. To show:
$$[\![M]\!]^{\sigma}_{T} \trianglelefteq_{\mathtt{'}\alpha} \mathsf{i}\,[\![M]\!]'^{\sigma'}_{T}$$
Since $\trianglelefteq_{T\alpha} = T\trianglelefteq_{\alpha}$, this follows from IH on M and Definition 3.26(3).

- Case $\mathtt{[}M\mathtt{]}$. To show:
$$[\![M]\!]^{\sigma}_{T} \trianglelefteq_{T\alpha} \mathsf{j}\,[\![M]\!]'^{\sigma'}_{T}$$
As above, $\trianglelefteq_{T\alpha} = T\trianglelefteq_{\alpha}$, so we get the result by IH on M and Definition 3.26(4).

- Case fix. As before, we need to check that $\trianglelefteq_{\beta}$ is computation-admissible, where β may now also contain the type $\mathtt{'}\alpha$. And in that case, computation-admissibility of $\mathtt{'}\trianglelefteq_{\alpha}$ is ensured by the requirement of a monad relation (Definition 3.26).

■

Using the monad relation induced by a monad morphism (Proposition 3.27), Proposition 3.29 immediately gives us correctness of a number of effect-simulations. For example, the complexity-state monad morphism from Example 3.3 validates the state-based maintenance of complexities.

For our monad-continuation simulation in full generality, however, we have to work a little harder. To obtain reification at arbitrary types, we cannot use purely equational properties of the monad morphism and the standard relational action of U alone – we need to construct the appropriate monad relation explicitly.

3.3.4 Relating monads to continuation-passing

Recall from Example 3.10 that the main limitation of our monad-morphism formulation of the continuation-based variant translation was its incomplete treatment of T-reification. Specifically, it did not directly allow us to define a reification operation (1) at more than one type and (2) at types containing $\mathtt{'}$. We will now see how to overcome these problems by using a more elaborate monad relation.

At the same time, we will take care of an independent technical complication (3) with simulating a monad $\mathbf{T}$ with a continuation monad $\mathbf{K_o}$. Following Lemma 3.5, we would expect to take the answer type o to be $T\gamma$ for some γ. However, the constraints imposed by our eventual application in Section 3.3.5 will not always allow us to do this. Specifically, we will need the answer type to be expressible as ${}^{\circ}\omega$ for some ω, but $T\gamma$ is not necessarily of this form.

Fortunately, the relational approach allows us considerable latitude in picking the actual answer type o, as long as it is "larger" than the $T\gamma$ that we originally needed. (Because the answer type occurs both positively and negatively in the continuation monad, we cannot express this condition purely equationally.) For conciseness, we formulate the requirement in general terms:

Definition 3.30 *An* answer-embedding *of L_0^+-computation-types o_1 into o_2 consists of a pair of functions $\phi^\bullet : o_1 \to o_2$ and $\psi^\bullet : o_2 \to o_1$, such that in $\mathcal{L}_\mathrm{i}$, (1) $\psi^\bullet$ is rigid and (2) $\psi^\bullet \circ \phi^\bullet = \mathsf{id}_{o_1}$.*

(Taking $o_1 = o_2 = T\gamma$ and $\phi^\bullet = \psi^\bullet = \mathsf{id}_{T\gamma}$ certainly satisfies these requirements, and still gives us a result strong enough to solve problems (1) and (2) mentioned above. Thus, on a first reading, it may be helpful to simply ignore all occurrences of $\phi^\bullet$ and $\psi^\bullet$ throughout this section. However, for the purpose of the next section, it is important that the we only rely on the weaker properties guaranteed by Definition 3.30.)

We are now ready to state a central result about relating monads and continuations. The essential trick is that, although we commit to a fixed answer *type* for the continuation monad, we are still free to consider all possible *relational interpretations* of that type:

Lemma 3.31 (continuation-simulation of monads) *Let $\mathbf{T}$ be a monad in $(L_0, \mathcal{L}_\mathrm{s})$. Further, let γ be a type and o a computation-type of L_0^+, with an answer-embedding $\phi^\bullet : T\gamma \to o$ and $\psi^\bullet : o \to T\gamma$. Then the mapping of $R \in \mathrm{ARel}(\alpha, \alpha')$ to $\text{'}R \in \mathrm{CARel}(T\alpha, K_o\alpha')$ given by*

$$\begin{aligned}
&t\ (\text{'}R)\ u \\
&\quad \iff \forall \alpha_0\ \mathsf{type}_{L_0}, O \in \mathrm{ARel}(\alpha_0, \gamma), k \in \mathrm{Val}_{\mathcal{L}_\mathrm{s}}(\alpha \to T\alpha_0), k' \in \mathrm{Val}_{\mathcal{L}_\mathrm{i}}(\alpha' \to o). \\
&\qquad\qquad (\forall a\ R\ a'.\, k\, a\ (TO)\ \psi^\bullet\,(k'\, a')) \Rightarrow k^*\, t\ (TO)\ \psi^\bullet\,(u\, k')
\end{aligned}$$

is a monad relation between $\mathbf{T}$ and $\mathbf{K}_o$. Moreover,

$$\mathsf{i}_{\alpha'} = \lambda t^{T\alpha'}.\, \lambda k^{\alpha' \to o}.\, \phi^\bullet\,((\psi^\bullet \circ k)^*\, t) \qquad \textit{and} \qquad \mathsf{j}_\gamma = \lambda u^{(\gamma \to o) \to o}.\, \psi^\bullet\,(u\,(\phi^\bullet \circ \eta_\gamma))$$

form a reflection function for all α' and a reification function for γ.

(Intuitively, the outer quantification over α_0 allows us to overcome limitation (1) from above; if we only needed reification at a single L_0^T-type α, we could simply fix $\alpha_0 = [\![\alpha]\!]_T$. Further, the inner quantification over O takes care of (2), by replacing a fixed relation on answer types (where in particular γ may be recursive) with a stronger parametricity condition. And finally, as already mentioned, we need $\phi^\bullet$ and $\psi^\bullet$ for (3).

It is instructive to compare the cases of the following proof with the corresponding ones in Proposition 3.27. Although some common structure could clearly be abstracted out, it is probably easier to follow how the continuations are being passed around in a concrete formulation.)

Proof. First, we check that ${}^{\shortmid}R$ is computation-admissible when R is admissible: ${}^{\shortmid}R$ is defined as an intersection over inverse images of the computation-admissible relations TO by the rigid functions k^* and $\psi^\bullet \circ \lambda u.\, u\, k'$. For the specific requirements, we have:

0. Assume $\forall a\ R_1\ a'.\, f\, a\ ({}^{\shortmid}R_2)\ g\, a'$ and $m\ ({}^{\circ}R_1)\ m'$. To show:

   $$\mathbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f\, x\ ({}^{\shortmid}R_2)\ \mathbf{let}^{\circ}_{(\alpha_2' \to o) \to o}\ x' \Leftarrow m'\ \mathbf{in}\ g\, x'$$

   i.e., that

   $$\mathbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f\, x\ ({}^{\shortmid}R_2)\ \lambda k'.\, \mathbf{let}^{\circ}_{o}\ x' \Leftarrow m'\ \mathbf{in}\ g\, x'\, k'\,.$$

   Let O, k, and k' be given, with

   $$\forall a\ R_2\ a'.\, k\, a\ (TO)\ \psi^\bullet\, (k'\, a')\,.$$

   We must then show that

   $$k^*\, (\mathbf{let}^{\circ}_{T\alpha_2}\ x \Leftarrow m\ \mathbf{in}\ f\, x)\ (TO)\ \psi^\bullet\, (\mathbf{let}^{\circ}_{o}\ x' \Leftarrow m'\ \mathbf{in}\ g\, x'\, k')\,.$$

   Using rigidity of k^* (Definition 2.15(0)) on the LHS and rigidity of $\psi^\bullet$ (Definition 3.30(1)) on the RHS, this is equivalent to showing

   $$\mathbf{let}^{\circ}_{T\alpha_0}\ x \Leftarrow m\ \mathbf{in}\ k^*\, (f\, x)\ (TO)\ \mathbf{let}^{\circ}_{T\gamma}\ x' \Leftarrow m'\ \mathbf{in}\ \psi^\bullet\, (g\, x'\, k')\,.$$

   Now, by Lemma 3.25(0) and the assumption that $m\ ({}^{\circ}R_1)\ m'$, it suffices to show that

   $$\forall a\ R_1\ a'.\, k^*\, (f\, a)\ (TO)\ \psi^\bullet\, (g\, a'\, k')$$

   and that follows from the assumption that $f\, a\ ({}^{\shortmid}R_2)\ g\, a'$.

1. Assume $a\ R\ a'$. Then for O, k, and k' as above, we must show that

   $$k^*\, (\eta\, a)\ (TO)\ \psi^\bullet\, (\varepsilon\, a'\, k')$$

   i.e., using law 2.15(1) and the definition of ε, that

   $$k\, a\ (TO)\ \psi^\bullet\, (k'\, a')$$

   which was precisely the assumption on k and k'.

2. Assume $\forall a\ R_1\ a'.\, f\, a\ ({}^{\shortmid}R_2)\ g\, a'$ and $t\ ({}^{\shortmid}R_1)\ u$.

   Let O, k and k' be given as before; to show:

   $$k^*\, (f^*\, t)\ (TO)\ \psi^\bullet\, (g^+\, u\, k')$$

   using monad law 2.15(3) on the LHS and expanding the RHS, this amounts to showing

   $$(\lambda x.\, k^*\, (f\, x))^*\, t\ (TO)\ \psi^\bullet\, (u\, (\lambda x.\, g\, x\, k'))$$

   This follows from the definition of $t\ ({}^{\shortmid}R_2)\ u$ if we can show that

   $$\forall a\ R_1\ a'.\, (\lambda x.\, k^*\, (f\, x))\, a\ (TO)\ \psi^\bullet\, ((\lambda x.\, g\, x\, k')\, a')$$

   i.e., that

   $$\forall a\ R_1\ a'.\, k^*\, (f\, a)\ (TO)\ \psi^\bullet\, (g\, a'\, k')$$

   And that follows from the assumption that $f\, a\ ({}^{\shortmid}R_2)\ g\, a'$.

3. Again, let O, k, and k' be given with $\forall a\ R\ a'.\, k\, a\ (TO)\ \psi^\bullet\,(k'\,a')$, and $t\ (TR)\ t'$; we must show that
$$k^*\,t\ (TO)\ \psi^\bullet\,(\phi^\bullet\,((\psi^\bullet \circ k')^*\,t'))$$
i.e., cancelling the $\psi^\bullet$ and $\phi^\bullet$ (Definition 3.30(2)), that
$$k^*\,t\ (TO)\ (\psi^\bullet \circ k')^*\,t'\,.$$
By Lemma 3.25(2), it suffices to show that
$$\forall a\ R\ a'.\, k\, a\ (TO)\ (\psi^\bullet \circ k')\,a'\,.$$
And that was precisely the assumption on k and k'.

4. Let $R \in \mathrm{ARel}(\alpha, \gamma)$ be given, with $t\ (\text{'}R)\ u$. To show:
$$t\ (TR)\ \psi^\bullet\,(u\,(\phi^\bullet \circ \eta_\gamma))$$
Here we finally need to instantiate the O in the definition of $\text{'}R$. Take
$$\alpha_0 = \alpha, \qquad O = R, \qquad k = \eta_\alpha, \qquad k' = \phi^\bullet \circ \eta_\gamma\,.$$
Clearly this O is admissible, because R was assumed to be. Further, let $a\ R\ a'$. Then, because η respects the relational action of T (Lemma 3.25(1)), we have:
$$k\,a = \eta\,a\ (TO)\ \eta\,a' = \psi^\bullet\,(\phi^\bullet\,(\eta\,a')) = \psi^\bullet\,(k'\,a')$$
From the assumption on t and u, and monad law 2.15(2), we therefore obtain
$$t = \eta^*\,t = k^*\,t\ (TO)\ \psi^\bullet\,(u\,k') = \psi^\bullet\,(u\,(\phi^\bullet \circ \eta))$$
as required.

■

Although the construction only gives reification at γ directly, by choosing γ appropriately, we can define reification functions at other types:

Lemma 3.32 *Let there be a monad relation between* $\mathbf{T}$ *and* $\mathbf{U}$*, and let* $\mathsf{j}_\gamma : U\gamma \to T\gamma$ *be a reification function at (*L_0^+*-type)* γ*. Let* α' *be any type of* L_0^+ *with term constructors* $\epsilon : \alpha' \to \gamma$ *and* $\delta : \gamma \to {}^\circ\alpha'$ *such that (in* $\mathcal{L}_\mathrm{i}$*)* $a\colon \alpha' \vdash \delta\,(\epsilon\,a) = {}^\circ a$*. Then the term*
$$\mathsf{j}^\gamma_{\alpha'}(\epsilon,\delta) : U\alpha' \to T\alpha' \overset{\mathrm{def}}{=} \lambda u^{U\alpha'}.\,(\lambda s^\gamma.\,\mathbf{let}^\circ_{T\alpha'}\ a \Leftarrow \delta\,s\ \mathbf{in}\ \eta_{\alpha'}\,a)^*\,(\mathsf{j}_\gamma\,((\lambda a^{\alpha'}.\,\varepsilon_\gamma\,(\epsilon\,a))^+\,u))$$
is a reification function at α'*.*

Proof. Let α in L_0 and $R \in \mathrm{ARel}(\alpha, \alpha')$ be given, with $t\ (\text{'}R)\ u$; we must show that $t\ (TR)\ \mathsf{j}^\gamma_{\alpha'}\,(\epsilon,\delta)u$. Accordingly, define $R_\gamma \in \mathrm{ARel}(\alpha, \gamma)$ by
$$a\ R_\gamma\ s \iff {}^\circ a\ ({}^\circ R)\ \delta\,s\,.$$

This is clearly admissible, being given as an inverse image of the admissible ${}^{\circ}R$. Moreover, from the assumption on ϵ and δ, and the properties of the computation-extension ${}^{\circ}R$ (Definition 3.20(1)), we immediately get

$$\forall a\; R\; a'.\, a\; R_\gamma\; \epsilon\, a'\,,$$

which, together with the assumption on t and u and the properties of the monad relation (Definition 3.26(1,2)), gives us

$$t = (\lambda a.\,\eta\, a)^*\, t\; ({}^{\shortmid}R_\gamma)\; (\lambda a'.\,\varepsilon\,(\epsilon\, a'))^+\, u\,.$$

From this, we get by the assumption on j_γ that

$$t\; (TR_\gamma)\; \mathsf{j}_\gamma\,((\lambda a'.\,\varepsilon\,(\epsilon\, a'))^+\, u)\,.$$

And finally, using all three parts of Lemma 3.25 and the definition of R_γ,

$$\begin{aligned}&t = (\lambda a.\,\mathbf{let}^{\circ}_{T\alpha}\; a \Leftarrow {}^{\circ}a\; \mathbf{in}\; \eta\, a)^*\, t\\ &\quad (TR)\; (\lambda s.\,\mathbf{let}^{\circ}_{T\alpha'}\; a' \Leftarrow \delta\, s\; \mathbf{in}\; \eta\, a')^*\,(\mathsf{j}_\gamma\,((\lambda a'.\,\varepsilon\,(\epsilon\, a'))u^+\,)) = \mathsf{j}^{\gamma}_{\alpha'}(\epsilon, \delta)u\end{aligned}$$

as required. ■

For the j from Lemma 3.31 specifically, this works out to:

$$\mathsf{j}^{\gamma}_{\alpha'}(\epsilon, \delta) = \lambda u^{K_\circ \alpha'}.\,(\lambda s^{\gamma}.\,\mathbf{let}^{\circ}_{T\alpha'}\; a \Leftarrow \delta\, s\; \mathbf{in}\; \eta\, a)^*\,(\psi^{\bullet}\,(u\,(\lambda a^{\alpha'}.\,(\phi^{\bullet} \circ \eta)\,(\epsilon\, a))))$$

Consider now an L_0^T-program. Because our type system is monomorphic, every `[-]`-operator in that program can be uniquely labeled with a specific type. There is thus only ever a finite set $\aleph$ of L_0^T-types α such that $[\![\alpha]\!]'_T$ needs to be embedded in the γ from Lemma 3.32. (Note that this is a *static* property of the program, with the set of reification-types bounded linearly by the program size. This in contrast to, say, finite unrollings of fixed points, where we cannot a priori determine how deeply to unroll.)

Thus, we can simulate reification with a finite sum, if we are willing to construct the relevant type $\gamma = \Sigma_i [\![\aleph(i)]\!]'_T$ for each program. In fact, for any finite $\aleph$ covering all reifications in an L_0^T-program, we get the same overall result when using any larger $\aleph$ for defining γ. We can thus formally define the evaluation semantics of programs resulting from the variant translation to be the *unique* meaning determined by any "sufficiently large" finite collection $\aleph$.

Or, we can use a single, infinite embedding type that works for all programs. In particular, we can simply take I to be the countable set of *names* of closed L_0^T-types, with $\aleph(\text{‘}\alpha\text{’}) = \alpha$. Giving such an enumeration is unproblematic: the set of L_0^T-types does not itself contain any embedding-types. Also, the tags themselves are inherently unstructured; in particular, for a monadic translation, we have $[\![\mathsf{in}_{\text{‘}\alpha\text{’}} M]\!]_T = \mathsf{in}_{\text{‘}\alpha\text{’}} [\![M]\!]_T$, not $\mathsf{in}_{\text{‘}[\![\alpha]\!]_T\text{’}} [\![M]\!]_T$. (In the actual ML implementation, we use an extensible data type for $\Sigma\aleph$, with tags dynamically generated and assigned at each instance of reification.)

Embedding-types alone do not suffice to express a continuation-based simulation of reification, however: there is an independent problem with reifying at L_0^T-types containing the type constructor $\mathtt{'}$. Suppose for simplicity that we only needed reification at a single L_0^T-type α_0, and moreover that we could choose the answer type of the continuation monad freely. Then it would seem natural to simply take $\gamma = \llbracket \alpha_0 \rrbracket'_T$ and $o = T\gamma$.

But since the continuation-passing $\llbracket \text{-} \rrbracket'_T$-translation is itself defined in terms of the answer type o, this would require us to solve the recursive type equation $o = T\llbracket \alpha_0 \rrbracket'_T$ *exactly*, which is too strong a requirement in general. Instead, we still take $\gamma = \llbracket \alpha_0 \rrbracket'_T$, but only $o \cong T\gamma$. In fact, the latter need not even be a full isomorphism; an answer-embedding suffices.

(Alternatively, we could have broken up the recursion by taking $o = T\gamma$ and $\gamma \cong \llbracket \alpha_0 \rrbracket'_T$. This approach gives a slightly simpler abstract correspondence between monads and continuation-passing, but does not allow us to express the construction in the next section in full generality.)

3.3.5 Factorizing the variant translation

Although it translates from L^T to L, the $\llbracket \text{-} \rrbracket'_T$-translation using U-effects is actually much more like the standard U-monadic translation $\llbracket \text{-} \rrbracket_U$ from L^U to L, sharing the type translation and most of the term translation clauses with the latter. The only non-standard clauses are for reflection and reification of T-effects. And in fact, we can express the $\llbracket \text{-} \rrbracket'_T$-translation entirely in terms of the $\llbracket \text{-} \rrbracket_U$-translation by expanding T-reflection and reification into L^U-definable terms.

In practice, this means that if we have a good (efficient, convenient, etc.) way of implementing evaluators for $\mathcal{L}_U$ (whether using the definitional translation for U or some other technique, as long as it gives correct results for complete programs), we can obtain an evaluator for $\mathcal{L}_T$ by simply viewing T-reflection and -reification as definitional extensions of L^U.

When i and j are definable in L_0 (and hence invariant under the translations), this is immediate: we can simply take $\mu^T(M) = \mu^U(\mathsf{i}\,M)$ and $\texttt{[}M\texttt{]}^T = \mathsf{j}\,\texttt{[}M\texttt{]}^U$. When $\mathbf{U}$ is a continuation monad with a recursively-defined answer type, however, it will be more convenient to work with a formulation of U-effects that integrates the recursion isomorphisms in the continuation-passing translation.

First, since for any monad-triples $\mathbf{T}$ and $\mathbf{U}$, the sets of *types* of L^T and L^U are actually the same (given by the type constructors of L together with $\mathtt{'}$), we use the name $L^\star$-type for a type from the extended signature, independent of the actual monad (which only affects the types of $\mu(\text{-})$ and $\texttt{[-]}$). We can then define a suitable notion of "native" effects for an $L^\star$-continuation monad:

Definition 3.33 *Let L be a cll signature. Then for any closed $L^\star$-type ω, the signature $L^{K\circ\omega}$ extends L with a new computation-type constructor $\mathtt{'}\alpha$, the associated value-inclusion and two* **let***s (with types as in Definition 2.21), and the following two term constructors:*

$$\frac{\Gamma \vdash M : (\alpha \to {}^\circ\omega) \to {}^\circ\omega}{\Gamma \vdash \mu^K(M) : \mathtt{'}\alpha} \qquad \frac{\Gamma \vdash M : \mathtt{'}\alpha}{\Gamma \vdash \texttt{[}M\texttt{]}^K : (\alpha \to {}^\circ\omega) \to {}^\circ\omega}$$

Note that this strictly generalizes our previous definition of a monad-extended signature, because in the case where ω is actually a type of L (i.e., does not contain any $'$), the above is exactly what we get by taking the monad $\mathbf{T}$ in Definition 2.21 to be $\mathbf{K}_{{}^\circ\omega}$.

Unlike the case for a standard monad-extension of a signature, however, we will not always be able to translate $L^{K_{{}^\circ\omega}}$ back into L, because the corresponding monad now involves a recursive type definition. But when the target syntax includes μ-types, we can give such a translation:

Definition 3.34 *The translation* $\llbracket\text{-}\rrbracket_K : L^{K_{{}^\circ\omega}} \to L^\mu$ *is defined as follows: first take*

$$\hat{\omega} = \mu\mathrm{a}.\, \llbracket\omega\rrbracket_{K_{{}^\circ\mathrm{a}}} \qquad \textit{and} \qquad o = {}^\circ\hat{\omega}$$

(where the type translation $\llbracket\text{-}\rrbracket_{K_{{}^\circ\mathrm{a}}}$ *expands* $'\alpha$ *into* $(\llbracket\alpha\rrbracket_{K_{{}^\circ\mathrm{a}}} \to {}^\circ\mathrm{a}) \to {}^\circ\mathrm{a}$ *and preserves all type constructors of* L*). We abbreviate the associated isomorphisms as:*

$$\phi \overset{\text{def}}{=} \mathsf{roll}_{\mathrm{a}.\llbracket\omega\rrbracket_{K_{{}^\circ\mathrm{a}}}} : \llbracket\omega\rrbracket_{K_o} \overset{\sim}{\Rightarrow} \hat{\omega} \qquad \textit{and} \qquad \psi \overset{\text{def}}{=} \mathsf{unroll}_{\mathrm{a}.\llbracket\omega\rrbracket_{K_{{}^\circ\mathrm{a}}}} : \hat{\omega} \overset{\sim}{\Rightarrow} \llbracket\omega\rrbracket_{K_o}$$

(where we write $\varphi : \alpha \overset{\sim}{\Rightarrow} \alpha'$ *to summarize the typing rule of a term constructor* φ *building* α'*-terms from* α*-terms).*

Then the type and term translation is the standard monadic translation for the monad $\mathbf{K}_o$*, except with the clauses for reflection and reification reading:*

$$\begin{aligned}
\llbracket\mu^K(M)\rrbracket_K &= \lambda k.\, \mathbf{let}^\circ\, r \Leftarrow \llbracket M\rrbracket_K\, (\lambda a.\, \mathbf{let}^\circ\, o \Leftarrow k\, a \;\mathbf{in}\; {}^\circ(\psi\, o)) \;\mathbf{in}\; {}^\circ(\phi\, r) \\
\llbracket[M]^K\rrbracket_K &= \lambda k.\, \mathbf{let}^\circ\, o \Leftarrow \llbracket M\rrbracket_K\, (\lambda a.\, \mathbf{let}^\circ\, r \Leftarrow k\, a \;\mathbf{in}\; {}^\circ(\phi\, r)) \;\mathbf{in}\; {}^\circ(\psi\, o)
\end{aligned}$$

Note that when the isomorphisms are identities, as we can always trivially ensure when ω is only an L-type, this reduces to the original definition of a monadic translation (Definition 2.23) because of the law $\mathbf{let}^\circ\, x \Leftarrow M \;\mathbf{in}\; {}^\circ x = M$.

The usual direct-style reasoning principles for reflection and reification from Definition 2.27 still hold for the more general notion of monadic translation. Specifically:

Lemma 3.35 *In addition to the equations for let and inclusion from Definition 2.27, the following equations are sound for the* $\llbracket\text{-}\rrbracket_K$*-translation from Definition 3.34:*

$$\begin{aligned}
\mu^K([M]^K) &= M \\
[\mu^K(M)]^K &= \lambda k.\, M\, k \;=\; M \\
['M]^K &= \lambda k.\, k\, M \\
[\mathbf{let}'\, x \Leftarrow M_1 \;\mathbf{in}\; M_2]^K &= \lambda k.\, [M_1]^K\, (\lambda x.\, [M_2]^K\, k) \\
[\mathbf{let}^\circ\, x \Leftarrow M_1 \;\mathbf{in}\; M_2]^K &= \lambda k.\, \mathbf{let}^\circ\, x \Leftarrow M_1 \;\mathbf{in}\; [M_2]^K\, k
\end{aligned}$$

(where k *does not occur free in any of the* M*s).*

Proof. Simple calculation; we mainly have to verify that the isomorphisms cancel out. For example, for the third equation:

$$\begin{aligned}
[\![\texttt{['}M\texttt{]}^{K}]\!]_{K} &= \lambda k.\,\mathbf{let}^{\circ}\ o \Leftarrow [\![\texttt{'}M]\!]_{K}\,(\lambda a.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\phi r))\ \mathbf{in}\ {}^{\circ}(\psi\, o)\\
&= \lambda k.\,\mathbf{let}^{\circ}\ o \Leftarrow (\lambda k'.\,k'\,[\![M]\!]_{K})\,(\lambda a.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\phi r))\ \mathbf{in}\ {}^{\circ}(\psi\, o)\\
&= \lambda k.\,\mathbf{let}^{\circ}\ o \Leftarrow (\mathbf{let}^{\circ}\ r \Leftarrow k\,[\![M]\!]_{K}\ \mathbf{in}\ {}^{\circ}(\phi r))\ \mathbf{in}\ {}^{\circ}(\psi\, o)\\
&= \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,[\![M]\!]_{K}\ \mathbf{in}\ \mathbf{let}^{\circ}\ o \Leftarrow {}^{\circ}(\phi r)\ \mathbf{in}\ {}^{\circ}(\psi\, o) = \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,[\![M]\!]_{K}\ \mathbf{in}\ {}^{\circ}(\psi\,(\phi r))\\
&= \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,[\![M]\!]_{K}\ \mathbf{in}\ {}^{\circ}r = \lambda k.\,k\,[\![M]\!]_{K} = [\![\lambda k.\,k\,M]\!]_{K}
\end{aligned}$$

The others are similar. ■

Note again the similarity of the direct-style equations characterizing $\texttt{[-]}$ to those of an explicit continuation-passing translation. The operational intuition is that $\mu^{K}(M)$ passes to M a functional representation of the current evaluation context, i.e., the continuation waiting for the result of $\mu^{K}(M)$. Conversely, $\texttt{[}M\texttt{]}^{K}$ evaluates M with a given continuation and returns the answer. For example, taking $\omega = \iota$, we have

$$\begin{aligned}
&\texttt{[}\mathbf{let}\texttt{'}\ x \Leftarrow \mu^{K}(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,\underline{3}\ \mathbf{in}\ k\,r)\ \mathbf{in}\ \texttt{'}(\mathsf{s}\,x)\texttt{]}^{K}\,(\lambda a.\,{}^{\circ}a)\\
&\quad = \texttt{[}\mu^{K}(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,\underline{3}\ \mathbf{in}\ k\,r)\texttt{]}^{K}\,(\lambda x.\,\texttt{['}(\mathsf{s}\,x)\texttt{]}^{K}\,(\lambda a.\,{}^{\circ}a))\\
&\quad = (\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,\underline{3}\ \mathbf{in}\ k\,r)\,(\lambda x.\,(\lambda a.\,{}^{\circ}a)\,(\mathsf{s}\,x)) = (\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,\underline{3}\ \mathbf{in}\ k\,r)\,(\lambda x.\,{}^{\circ}(\mathsf{s}\,x))\\
&\quad = \mathbf{let}^{\circ}\ r \Leftarrow (\lambda x.\,{}^{\circ}(\mathsf{s}\,x))\,\underline{3}\ \mathbf{in}\ (\lambda x.\,{}^{\circ}(\mathsf{s}\,x))\,r = \mathbf{let}^{\circ}\ r \Leftarrow {}^{\circ}(\mathsf{s}\,\underline{3})\ \mathbf{in}\ (\lambda x.\,{}^{\circ}(\mathsf{s}\,x))\,r\\
&\quad = (\lambda x.\,{}^{\circ}(\mathsf{s}\,x))\,\underline{4} = {}^{\circ}\underline{5}
\end{aligned}$$

That is, k gets bound to the function $\lambda x.\,{}^{\circ}(\mathsf{s}\,x)$ and applied twice to $\underline{3}$.

Remark 3.36 The circularity inherent in allowing ω to be an $L^{\star}$-type is genuine: even without fix in the language, it is possible to write non-terminating programs in $L^{K_{{}^{\circ}\omega}}$. For perhaps the simplest example, take $\omega = \texttt{'}1$. Then reflection and reification (as shown above, they are still two-sided inverses) give us an isomorphism

$$\omega = \texttt{'}1 \cong (1 \to {}^{\circ}\omega) \to {}^{\circ}\omega \cong {}^{\circ}\omega \to {}^{\circ}\omega$$

And indeed, we can define a diverging term $\Omega_{{}^{\circ}\omega}$ by the usual double self-application made type-correct by the isomorphisms:

$$\begin{aligned}
d : \omega \to {}^{\circ}\omega &= \lambda x^{\omega}.\,\texttt{[}x\texttt{]}\,(\lambda\langle\rangle.\,{}^{\circ}x)\\
d' : \omega &= \mu(\lambda k^{1\to{}^{\circ}\omega}.\,\mathbf{let}^{\circ}\ x \Leftarrow k\,\langle\rangle\ \mathbf{in}\ d\,x)\\
\Omega : {}^{\circ}\omega &= d\,d'
\end{aligned}$$

■

It turns out that picking a fixed *shape* for the answer type (i.e., requiring it to be of the form ${}^{\circ}\omega$) necessitates a slight twist when simulating monads whose type constructors do not contain an outermost ${}^{\circ}$; this is why we allowed the answer-embedding in Definition 3.30 to be a retraction, rather than a full isomorphism. First, we slightly transform the monad to be simulated:

Definition 3.37 *Let* $\mathbf{T} = (T, \eta, \text{-}^{*})$ *be a monad-triple. We then define a new monad-triple* $\hat{\mathbf{T}} = (\hat{T}, \hat{\eta}, \text{-}^{\hat{*}})$ *as follows:*

$$\begin{aligned}
\hat{T}\alpha &= {}^{\circ}(T\alpha)\\
\hat{\eta} &= \lambda a.\,{}^{\circ}(\eta\,a)\\
f^{\hat{*}} &= \lambda m.\,\mathbf{let}^{\circ}\ t \Leftarrow m\ \mathbf{in}\ {}^{\circ}((\lambda a.\,\mathbf{let}^{\circ}_{T\alpha_2}\ r \Leftarrow f\,a\ \mathbf{in}\ r)^{*}\,t)
\end{aligned}$$

For example, for state (Example 2.18), this gives

$$
\begin{aligned}
\hat{T}\alpha &= {}^{\circ}(\sigma \to {}^{\circ}(\alpha \times \sigma)) \\
\hat{\eta} &= \lambda a.\, {}^{\circ}(\lambda s.\, {}^{\circ}\langle a, s\rangle) \\
f^{*} &= \lambda m.\, \mathbf{let}^{\circ}\; t \Leftarrow m \;\mathbf{in}\; {}^{\circ}(\lambda s.\, \mathbf{let}^{\circ}\; \langle a, s'\rangle \Leftarrow t\, s \;\mathbf{in}\; \mathbf{let}^{\circ}\; r \Leftarrow f\, a \;\mathbf{in}\; r\, s)
\end{aligned}
$$

Now $\hat{T}\alpha$ does have an outermost $^{\circ}$. On the other hand, $\hat{\mathbf{T}}$ is not in general a *monad*, even if $\mathbf{T}$ was. In particular, for law 2.15(1) we only get

$$
\begin{aligned}
& f^{\hat{*}}\,(\hat{\eta}\, a) = \mathbf{let}^{\circ}\; t \Leftarrow {}^{\circ}(\eta\, a) \;\mathbf{in}\; {}^{\circ}((\lambda a.\, \mathbf{let}^{\circ}_{T\alpha_2}\; r \Leftarrow f\, a \;\mathbf{in}\; r)^{*}\, t) \\
& \quad = {}^{\circ}((\lambda a.\, \mathbf{let}^{\circ}_{T\alpha_2}\; r \Leftarrow f\, a \;\mathbf{in}\; r)^{*}\,(\eta\, a)) = {}^{\circ}(\mathbf{let}^{\circ}_{T\alpha_2}\; r \Leftarrow f\, a \;\mathbf{in}\; r) \overset{?}{=} f\, a
\end{aligned}
$$

$\hat{\mathbf{T}}$ is, however, a monad "up to extensionality": when evaluated and applied to a value, $f\, a$ and $f^{\hat{*}}\,(\hat{\eta}\, a)$ do behave identically:

$$
\begin{aligned}
& \mathbf{let}^{\circ}\; r \Leftarrow f^{\hat{*}}\,(\hat{\eta}\, a) \;\mathbf{in}\; r\, s = \mathbf{let}^{\circ}\; r \Leftarrow {}^{\circ}(\mathbf{let}^{\circ}_{T\alpha_2}\; r \Leftarrow f\, a \;\mathbf{in}\; r) \;\mathbf{in}\; r\, s \\
& \quad = (\mathbf{let}^{\circ}_{T\alpha_2}\; r \Leftarrow f\, a \;\mathbf{in}\; r)\, s = \mathbf{let}^{\circ}\; r \Leftarrow f\, a \;\mathbf{in}\; r\, s
\end{aligned}
$$

And in fact, our construction will ensure that functions like f are always "fully applied", so that we can use $\hat{\eta}$ and $\text{-}^{\hat{*}}$ instead of η and -^{*}.

We can now define reflection and reification for $\mathbf{T}$ in terms of the corresponding operators for continuations as follows:

Theorem 3.38 *Let* $\mathbf{T}$ *be a monad in* $(L_0, \mathcal{L}_{\mathrm{s}})$*, and let* M *be an* L_0^T*-program without top-level focus effects, (i.e.,* $\cdot \vdash M : {}^{\circ}\iota$*). Further, let* $\aleph$ *be a family of types containing at least all* L_0^T*-types for which* M *contains a reification-operator. Take* $\omega = T(\Sigma\aleph)$*, a well-formed* $(L_0^{\Sigma})^{\star}$*-type, and in* $(L_0^{\Sigma})^{K_{\circ\omega}}$ *define the term constructors* $\mu^{T}(\text{-})$ *and* $\mathtt{[}\text{-}\mathtt{]}^{T}$ *by:*

$$
\begin{aligned}
\mu^{T}(M) &= \mu^{K}(\lambda k.\, k^{\hat{*}}\,({}^{\circ}M)) \\
\mathtt{[}M\mathtt{]}^{T} &= \mathbf{let}^{\circ}_{T\alpha}\; t \Leftarrow (\lambda r.\, \mathbf{let}^{\circ}\; a \Leftarrow \mathsf{out}_i\, r \;\mathbf{in}\; \hat{\eta}\, a)^{\hat{*}}\,(\mathtt{[}M\mathtt{]}^{K}\,(\lambda a.\, \hat{\eta}\,(\mathsf{in}_i\, a))) \;\mathbf{in}\; t \qquad {\scriptstyle (\aleph(i)=\alpha)}
\end{aligned}
$$

Now take $L_0^{+} = (L_0^{\Sigma})^{\mu}$*, with* $\mathcal{L}_{\mathrm{i}}$ *a model of* $(\mathcal{E}_0^{\Sigma})^{\mu}$ *from Section 3.2.1. Then replacing all* T*-reflection and -reification operators in* M *with the definitions above (picking* i *for each reification arbitrarily, subject to the constraint), yields an* $(L_0^{\Sigma})^{K_{\circ\omega}}$*-program* M' *such that* $[\![M]\!]_{T}\; ({}^{\circ}\iota^{\mathrm{r}})\; [\![M']\!]_{K}$*.*

Proof. Let $\hat{\omega} \cong [\![T(\Sigma\aleph)]\!]_{K}$ with associated isomorphisms be as in Definition 3.34, and take $\gamma = \Sigma_i [\![\aleph(i)]\!]_{K}$; then $[\![\omega]\!]_{K} = [\![T(\Sigma\aleph)]\!]_{K} = T[\![\Sigma\aleph]\!]_{K} = T(\Sigma[\![\aleph]\!]_{K}) = T\gamma$. It is also easy to see that the functions defined by

$$
\begin{aligned}
\phi^{\bullet} : T\gamma \to {}^{\circ}\hat{\omega} &= \lambda r.\, {}^{\circ}(\phi\, r) \\
\psi^{\bullet} : {}^{\circ}\hat{\omega} \to T\gamma &= \lambda m.\, \mathbf{let}^{\circ}_{T\gamma}\; o \Leftarrow m \;\mathbf{in}\; \psi\, o
\end{aligned}
$$

form an answer-embedding in the sense of Definition 3.30.

Now, using the i from Lemma 3.31 directly, we get (omitting a few tedious let-simplification steps):

$$\begin{aligned}
\llbracket \mu^T(M) \rrbracket_K &= \llbracket \mu^K(\lambda k.\, k^{\hat{*}}\,({}^{\circ}M)) \rrbracket_K = \llbracket \mu^K(\lambda k.\, {}^{\circ}((\lambda a.\, \mathbf{let}^{\circ}_{\omega}\ r \Leftarrow k\,a\ \mathbf{in}\ r)^*\, M)) \rrbracket_K \\
&= \lambda k.\, \mathbf{let}^{\circ}\ r \Leftarrow \llbracket \lambda k.\, {}^{\circ}((\lambda a.\, \mathbf{let}^{\circ}_{\omega}\ r \Leftarrow k\,a\ \mathbf{in}\ r)^*\, M) \rrbracket_K\, (\lambda a.\, \mathbf{let}^{\circ}\ o \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\psi\, o))\ \mathbf{in}\ {}^{\circ}(\phi r) \\
&= \lambda k.\, \mathbf{let}^{\circ}\ r \Leftarrow {}^{\circ}((\lambda a.\, \mathbf{let}^{\circ}_{T\gamma}\ o \Leftarrow k\,a\ \mathbf{in}\ \psi\, o)^*\, \llbracket M \rrbracket_K)\ \mathbf{in}\ {}^{\circ}(\phi r) \\
&= \lambda k.\, {}^{\circ}(\phi\,((\lambda a.\, \mathbf{let}^{\circ}_{T\gamma}\ o \Leftarrow k\,a\ \mathbf{in}\ \psi\, o)^*\, \llbracket M \rrbracket_K)) = \lambda k.\, \phi^{\bullet}\,((\lambda a.\, \psi^{\bullet}\,(k\,a))^*\, \llbracket M \rrbracket_K) \\
&= \mathsf{i}_{\llbracket \alpha \rrbracket_K}\, \llbracket M \rrbracket_K = \llbracket \mu(M) \rrbracket'_T
\end{aligned}$$

Similarly, using j from 3.31 as extended by Lemma 3.32 (with $\epsilon = \mathsf{in}_i$ and $\delta = \mathsf{out}_i$, satisfying the retraction condition by definition), we get:

$$\begin{aligned}
\llbracket [M]^T \rrbracket_K &= \llbracket \mathbf{let}^{\circ}_{T\alpha}\ t \Leftarrow (\lambda r.\, \mathbf{let}^{\circ}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \hat{\eta}\, a)^{\hat{*}}\, ([M]^K\, (\lambda a.\, \hat{\eta}\,(\mathsf{in}_i\, a)))\ \mathbf{in}\ t \rrbracket_K \\
&= \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ t \Leftarrow \llbracket [M]^K \rrbracket_K\, (\lambda a.\, {}^{\circ}(\eta\,(\mathsf{in}_i\, a)))\ \mathbf{in}\ (\lambda r.\, \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \eta\, a)^*\, t \\
&= \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ o \Leftarrow \llbracket M \rrbracket_K\, (\lambda a.\, {}^{\circ}(\phi\,(\eta\,(\mathsf{in}_i\, a))))\ \mathbf{in}\ (\lambda r.\, \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \eta\, a)^*\, (\psi\, o) \\
&= (\lambda r.\, \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \eta\, a)^*\, (\mathbf{let}^{\circ}_{T\gamma}\ o \Leftarrow \llbracket M \rrbracket_K\, (\lambda a.\, \phi^{\bullet}\,(\eta\,(\mathsf{in}_i\, a)))\ \mathbf{in}\ \psi\, o) \\
&= (\lambda r.\, \mathbf{let}^{\circ}_{T\llbracket \alpha \rrbracket_K}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \eta\, a)^*\, (\psi^{\bullet}\,(\llbracket M \rrbracket_K\, (\lambda a.\, (\phi^{\bullet} \circ \eta)\,(\mathsf{in}_i\, a)))) \\
&= \mathsf{j}^{\gamma}_{\llbracket \alpha \rrbracket_K}(\mathsf{in}_i, \mathsf{out}_i)\, \llbracket M \rrbracket_K = \llbracket [M] \rrbracket'_T
\end{aligned}$$

We can thus apply Proposition 3.29 (with empty σ and σ') to get the result. ■

Remark 3.39 When the monad $\mathbf{T}$ is already of the form $T\alpha = {}^{\circ}(T'\alpha)$ for some type constructor T' (e.g., for exceptions, $T'\alpha = \alpha + \chi$), a slightly simpler construction is possible. Take $\omega = T'(\Sigma\aleph)$ so that $o = {}^{\circ}\hat{\omega} \cong {}^{\circ}(T'\gamma) = T\gamma$, $\phi^{\bullet}\, t = \mathbf{let}^{\circ}\ s \Leftarrow t\ \mathbf{in}\ {}^{\circ}(\phi\, s)$, and $\psi^{\bullet}\, m = \mathbf{let}^{\circ}\ r \Leftarrow m\ \mathbf{in}\ {}^{\circ}(\psi r)$. (Here $\phi^{\bullet}$ and $\psi^{\bullet}$ are actually two-sided inverses.) Then we get the analog of Theorem 3.38 by defining reflection and reification as follows:

$$\begin{aligned}
\mu^T(M) &= \mu^K(\lambda k.\, k^*\, M) \\
[M]^T &= (\lambda r.\, \mathbf{let}^{\circ}\ a \Leftarrow \mathsf{out}_i\, r\ \mathbf{in}\ \eta\, a)^*\, ([M]^K\, (\lambda a.\, \eta\,(\mathsf{in}_i\, a))) \quad {\scriptstyle (\aleph(i)=\alpha)}
\end{aligned}$$

The actual ML code in Section 4.5 takes advantage of this optimization by not including an explicit suspension in the definition of monads like $\mathbf{T}$, but instead having it implicitly inserted by the CBV elaboration from Section 2.1.6. In other words, the type constructor `T` in the ML signature of such a monad actually corresponds to the T' above, so that, e.g., $(\!(\sigma_1 \to \mathtt{T}\sigma_2)\!)^{\mathsf{v}} = (\!(\sigma_1)\!)^{\mathsf{v}} \to {}^{\circ}(\!(\mathtt{T}\sigma_2)\!)^{\mathsf{v}} = (\!(\sigma_1)\!)^{\mathsf{v}} \to T(\!(\sigma_2)\!)^{\mathsf{v}}$. We could, however, simply use Theorem 3.38 directly in all cases. ■

We have thus reduced the problem of implementing a language with monadic effects for an arbitrary definable monad $\mathbf{T}$ to that of implementing a language with reflection and reification operators for a continuation monad with an answer type ${}^{\circ}\omega$ for some value-type ω. In the next chapter, we will show how this can itself be achieved by embedding the continuation-effect language into a Scheme-like one.

3.3.6 Induced relational correspondence

We finally show how the relational correspondence between $\mathcal{L}_{\mathrm{s}}$ and $\mathcal{L}_{\mathrm{i}}$ can be generalized to the case where the base language is itself given by a monadic translation. That is, we consider the language where we take ' as the distinguished computation-type constructor, while ° and its related operations become simply additional type and term constructors. Corollary 2.29 showed that this new language is also a model of $\mathcal{E}_0$; the following shows that this equational characterization extends to a relational one:

Proposition 3.40 *Let there be given a monad relation between* $\mathbf{T}$ *in* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathbf{U}$ *in* $\mathcal{L}_{\mathrm{i}}$ *(Definition 3.26). Then the relation assignment determined by the monad relation is a computation-extension in the sense of Definition 3.20 for* '*-computations in the languages* $(L_0^T, \mathcal{L}_{\mathrm{s}}^T)$ *and* $(L_0^{+U}, \mathcal{L}_{\mathrm{i}}^U)$*, as given by Definition 2.24.*

Proof. For the first condition, let $R \in \mathrm{ARel}_{\mathcal{L}_{\mathrm{s}}^T, \mathcal{L}_{\mathrm{i}}^U}(\alpha, \alpha')$ be a relation, and let $a \; R \; a'$; we must show that $\text{'}a \; (\text{'}R) \; \text{'}a'$, i.e., that in the original correspondence,

$$\llbracket \text{'}a \rrbracket_T = \eta \, a \; (\text{'}R) \; \varepsilon \, a = \llbracket \text{'}a \rrbracket_U$$

and that follows immediately from Definition 3.26(1).

Similarly, assume that $\forall a \; R_1 \; a'. \, f \, a \; (\text{'}R_2) \; f' \, a'$ and $m \; (\text{'}R_1) \; m'$. We must show that

$$\textbf{let'} \; x \Leftarrow m \; \textbf{in} \; f \, x \; (\text{'}R_2) \; \textbf{let'} \; x' \Leftarrow m' \; \textbf{in} \; f' \, x'$$

in the new correspondence, i.e., that

$$\begin{aligned} &\llbracket \textbf{let'} \; x \Leftarrow m \; \textbf{in} \; f \, x \rrbracket_T = (\lambda x. \, f \, x)^* \, m = f^* \, m \\ &\quad (\text{'}R_2) \; f'^{+} \, m' = (\lambda x'. \, f' \, x')^{+} \, m' = \llbracket \textbf{let'} \; x' \Leftarrow m' \; \textbf{in} \; f' \, x' \rrbracket_U \end{aligned}$$

in the original one, which is precisely the statement of 3.26(2). ■

3.4 Related work

The study of relationships between direct and continuation semantics has a long history. Early investigations [Rey74a, ST80, Sto81] were set in a domain-theoretic framework where the main difficulties concerned reflexive domains; as a result, these methods and results were closely tied to specific semantic models. On the other hand, Meyer and Wand's more abstract approach [MW85] applied to all models of simply-typed λ-calculi, but did not encompass computational effects – not even nontermination.

The present work, while formulated in a simply-typed setting, and using mostly axiomatic reasoning, is nevertheless closer conceptually to the domain-theoretic results. In particular, it explicitly handles general recursion in computations by fixed-point induction, and should extend to recursively-defined types without too many obstacles. (The initial version in [Fil94] was based on the Meyer-Wand approach, but it is not clear how well that would scale to ambient effects and especially recursion.)

A possible correspondence between monads and continuation-passing style (CPS) was conjectured by Danvy and Filinski [DF90], and more concretely presented by Wadler [Wad92b]. (The general idea of using a monad morphism to simulate one monadic effect with another is also due to Wadler [Wad90].) However, this work was largely informal. Most notably, the problems with reification (needed, e.g., to express **handle** for continuation-based exceptions) in a typed setting were not addressed at all.

Peyton Jones and Wadler [PW93] probe the relationship between monads and CPS further, and Wadler [Wad94] analyzes composable continuations from a monadic perspective, but in both cases the restriction to Hindley-Milner typable translations obscures the general correspondence; properly expressing the answer-type parametricity in a simulation of general monads by continuation-passing requires a more flexible type system.

Finally, another glimmer of the connection between monads and continuations can be seen in Sabry and Felleisen's result that $\beta\eta$-equivalence of CPS terms coincides with direct-style equivalence in Moggi's computational λ-calculus [SF93, Mog89]; the latter captures exactly the equivalences holding in the presence of arbitrary monadic effects. While this does not by itself imply that any monadic effect can be simulated by a continuation monad, it does indicate that continuations form a *maximally* (but not necessarily *most*) general notion of effect.

Chapter 4

Implementing Continuation-Effects

In this chapter, we continue the simulation of effects by showing that a language with reflection and reification operators for a continuation monad can itself be embedded in a language with a more traditional set of effects: Scheme-style first-class continuations and typed state.

That is, in the previous chapter we showed that continuations are in a precise sense a *universal effect*: any definable monad can be simulated by a continuation monad with a suitable answer type. Now we show that this universal effect can itself be expressed in terms of two specific, low-level effects. Thus, we can program directly with monadic effects in a language such as Scheme, or ML with continuations.

The development consists of three major steps. First, we re-express reflection and reification for continuations in terms of an alternative, more operationally motivated pair of control operators. These implement a control abstraction known as *composable continuations*. We further decompose the composable-continuations operators into a standard *escape-operator*, an *abort-operator*, and a *control delimiter*.

Then we show that the level-tags ($^{\circ}$ and ' on value-inclusions and lets) introduced by the monadic translation are actually unnecessary for evaluation-purposes: the *level-erasure* of a program evaluates to the same result as the original one. The proof involves another set of logical relations, indexed by types of the original two-level language, and relating original and level-erased terms at each type.

Finally, in the level-erased language, we define the control operators in terms of Scheme-like primitives. The key step here is to re-express the sequencing of already continuation-passing terms in *meta-continuation-passing style*, then observe that the metacontinuation is used in a single-threaded way throughout the translation and can hence be maintained in a fixed cell of the store. Again, a simple logical-relations argument shows the equivalence of the original definitions of the control operators to their escape-state simulation.

We conclude the chapter with an actual implementation of the construction in Standard ML of New Jersey, which includes language support for first-class continuations. In addition to the complete code implementing monadic reflection and reification in terms of escapes and state, we show a few simple programming examples. In particular, we illustrate how definable monadic effects, such as nondeterministic or probabilistic computations, fit very naturally into a traditional call-by-value setting.

4.1 Continuation-reflection and composable continuations

For convenience, we will assume that all signatures in the following contain an *empty type* 0 with no value constructors. Our implementation signature $L_0^+ = L_0^{\Sigma,\mu}$ (i.e., L_0 with recursive types and embedding-types) certainly does, for example defining 0 as μa. a or as an embedding-type with empty index set. We need not require that 0 has a counterpart in the specification language L_0 as well, although adding it there would be unproblematic.

Further, for any computation-type β there is a (unique) function from 0 to β, expressible as, e.g., $\lambda z.\, \bot_\beta$, which we write as $\mathcal{V}_\beta$. In fact, 0 and $\mathcal{V}$ are simply the zero-ary analogs of sums and `case` (except that we do not require a $\mathcal{V}_\alpha$ for arbitrary α); in particular, for any monadic translation we have $[\![0]\!]_T = 0$ and $[\![\mathcal{V}_\beta\, M]\!]_T = \mathcal{V}_{[\![\beta]\!]_T}\, [\![M]\!]_T$.

Unlike Chapter 3, where the simulation results were parameterized by the fairly complex notion of a definable monad (which included a type constructor and type-indexed families of term constructors), all the constructions in this chapter are parameterized by a fixed, closed value-type ω. Accordingly, except within definitions of recursive types with μ, we only need to consider type-closed types and terms.

The continuation-passing translation allows us to define a wide range of control operators in the source language. We have already seen reflection and reification, but many others are possible. In particular, we have:

Definition 4.1 *Let L be a cll signature with a 0-type, and let ω be a type of $L^\star$. Then in $L^{K{\circ}\omega}$ (Definition 3.33), we define the operations*

$$\frac{\Gamma \vdash M : (\alpha \to \text{'}0) \to \text{'}0}{\Gamma \vdash \mathcal{C}\, M : \text{'}\alpha} \qquad \frac{\Gamma \vdash M : \omega}{\Gamma \vdash \mathcal{A}\, M : \text{'}0} \qquad \frac{\Gamma \vdash M : \text{'}\omega}{\Gamma \vdash \#M : {}^\circ\omega} \qquad \frac{\Gamma \vdash M : (\alpha \to {}^\circ\omega) \to \text{'}\omega}{\Gamma \vdash \mathcal{S}\, M : \text{'}\alpha}$$

by the expansions:

$$\begin{aligned}
\mathcal{C}\, M &= \mu(\lambda k^{\alpha \to {}^\circ\omega}.\, [M\, (\lambda a^\alpha.\, \mu(\lambda q'^{0 \to {}^\circ\omega}.\, k\, a))]\, (\lambda z^0.\, \mathcal{V}_{{}^\circ\omega} z)) \\
\mathcal{A}\, M &= \mu(\lambda q^{0 \to {}^\circ\omega}.\, {}^\circ M) \\
\#M &= [M]\, (\lambda r^\omega.\, {}^\circ r) \\
\mathcal{S}\, M &= \mu(\lambda k^{\alpha \to {}^\circ\omega}.\, [M\, k]\, (\lambda r^\omega.\, {}^\circ r))
\end{aligned}$$

The operational intuition is as follows: $\mathcal{C}\, M$ (*escape*) invokes M with a representation of the current evaluation context as a procedure $q : \alpha \to \text{'}0$ that, when applied to a value $a : \alpha$, will abandon the then current context of evaluation and return a as the result of $\mathcal{C}\, M$, e.g.,

$$\mathcal{C}\, (\lambda q.\, \textbf{let'}\ z \Leftarrow q\, \underline{3}\ \textbf{in} \cdots) = \text{'}\underline{3}$$

(regardless of what happens in $\cdots$). The 0-"returning" q can invoked in a context expecting an $\text{'}\alpha'$-typed result by writing **let'** $z \Leftarrow q\, a$ **in** $\mathcal{V}_{\text{'}\alpha'}\, z$. $\mathcal{C}$ thus acts very much like Scheme's `call/cc`, except that the M must explicitly invoke q in order to return a value from $\mathcal{C}\, M$. Each variant can be used to define the other, however.

$\mathcal{A}M$ (*abort*) immediately terminates the current computation, returning M as the answer. Like a q supplied by $\mathcal{C}$, it can be used in combination with $\mathcal{V}$ to break out of any computation-typed context.

$\#M$ (*reset* or *prompt*) evaluates M in an empty evaluation context and returns the final answer of that evaluation, thus delimiting any control effects M might have. For example, when $\omega = \iota$, we get

$$\#(\mathbf{let}'\ z \Leftarrow \mathcal{A}\underline{5}\ \mathbf{in}\ '\underline{7}) = {}^{\circ}\underline{5}$$

Finally, $\mathcal{S}M$ (*shift*) captures and erases the current evaluation context up to (but not including) the innermost enclosing $\#$, passing this context to M as a composable function. For example, still with $\omega = \iota$, we have

$$\#(\mathbf{let}'\ x \Leftarrow \mathcal{S}(\lambda k.\,\mathbf{let}^{\circ}\ r_1 \Leftarrow k\,\underline{4}\ \mathbf{in}\ \mathbf{let}^{\circ}\ r_2 \Leftarrow k\,r_1\ \mathbf{in}\ 'r_2)\ \mathbf{in}\ '(\mathsf{s}\,x)) = {}^{\circ}\underline{6}$$

Note also that we could define $\mathcal{A}M = \mathcal{S}(\lambda k^{0 \to {}^{\circ}\omega}.\,'M)$

Although probably not as well known as call/cc, control operators like $\mathcal{S}$, $\mathcal{A}$, and $\#$ have already seen a fair amount of study, e.g., [Fel88, SF90, DF92, Wad94, Fil94, GRR95]; we will briefly compare the various approaches in Section 4.6.

For reference, and since we will need it later (in Definition 4.5), let us note:

Lemma 4.2 *The translations of the derived control operators using Definition 3.34 work out to:*

$$\begin{array}{rcl}
[\![\mathcal{C}\,M]\!]_K & = & \lambda k.\,[\![M]\!]_K\,(\lambda a.\,\lambda q.\,k\,a)\,(\lambda z.\,\mathcal{V}\,z) \\
[\![\mathcal{A}\,M]\!]_K & = & \lambda q.\,{}^{\circ}(\phi\,[\![M]\!]_K) \\
[\![\#M]\!]_K & = & \mathbf{let}^{\circ}\ o \Leftarrow [\![M]\!]_K\,(\lambda r.\,{}^{\circ}(\phi\,r))\ \mathbf{in}\ {}^{\circ}(\psi\,o) \\
[\![\mathcal{S}\,M]\!]_K & = & \lambda k.\,[\![M]\!]_K\,(\lambda a.\,\mathbf{let}^{\circ}\ o \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\psi\,o))\,(\lambda r.\,{}^{\circ}(\phi\,r))
\end{array}$$

Proof. Straightforward. For example,

$$\begin{aligned}
&[\![\mathcal{A}\,M]\!]_K = [\![\mu^K(\lambda q.\,{}^{\circ}M)]\!]_K = \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow [\![\lambda q.\,{}^{\circ}M]\!]_K\,(\lambda a.\,\mathbf{let}^{\circ}\ o \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\psi\,o))\ \mathbf{in}\ {}^{\circ}(\phi\,r) \\
&\quad = \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow (\lambda q.\,{}^{\circ}[\![M]\!]_K)\,(\lambda a.\,\mathbf{let}^{\circ}\ o \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}(\psi\,o))\ \mathbf{in}\ {}^{\circ}(\phi\,r) \\
&\quad = \lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow {}^{\circ}[\![M]\!]_K\ \mathbf{in}\ {}^{\circ}(\phi\,r) = \lambda k.\,{}^{\circ}(\phi\,[\![M]\!]_K)
\end{aligned}$$

The others are similar. ■

The reason why we can concentrate on the composable-continuations operators instead of the seemingly more general $\mu^K(\text{-})$ and $\texttt{[}\text{-}\texttt{]}^K$ is the following property:

Lemma 4.3 *Shift and reset form a* complete *set of control operators, in the sense that we can use them to express reflection and reification as follows:*

$$\begin{array}{rcl}
\mu^K(M) & = & \mathcal{S}(\lambda k^{\alpha \to {}^{\circ}\omega}.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ 'r) \\
\texttt{[}M\texttt{]}^K & = & \lambda k^{\alpha \to {}^{\circ}\omega}.\,\#(\mathbf{let}'\ a \Leftarrow M\ \mathbf{in}\ \mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ 'r)
\end{array}$$

and get equivalent translations under $[\![\text{-}]\!]_K$.

Proof. The actual translations of the terms contain explicit isomorphisms, which clutter up the equational proofs. It is thus more convenient to use the standard direct-style reasoning principles for reflection and reification, whose soundness with respect to the $[\![\text{-}]\!]_K$-translation was established in Lemma 3.35:

$$\begin{aligned}
&\mathcal{S}\,(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ \texttt{'}r) = \mu(\lambda k.\,\texttt{[}(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ \texttt{'}r)\,k\texttt{]}\,(\lambda r.\,{}^{\circ}r))\\
&\quad = \mu(\lambda k.\,\texttt{[}\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ \texttt{'}r\texttt{]}\,(\lambda r.\,{}^{\circ}r)) = \mu(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ \texttt{[}\texttt{'}r\texttt{]}\,(\lambda r.\,{}^{\circ}r))\\
&\quad = \mu(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ (\lambda k.\,k\,r)\,(\lambda r.\,{}^{\circ}r)) = \mu(\lambda k.\,\mathbf{let}^{\circ}\ r \Leftarrow M\,k\ \mathbf{in}\ {}^{\circ}r) = \mu(\lambda k.\,M\,k)\\
&\quad = \mu(M)
\end{aligned}$$

$$\begin{aligned}
&\lambda k.\,\#(\mathbf{let'}\ a \Leftarrow M\ \mathbf{in}\ \mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ \texttt{'}r) = \lambda k.\,\texttt{[}\mathbf{let'}\ a \Leftarrow M\ \mathbf{in}\ \mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ \texttt{'}r\texttt{]}\,(\lambda r.\,{}^{\circ}r)\\
&\quad = \lambda k.\,\texttt{[}M\texttt{]}\,(\lambda a.\,\texttt{[}\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ \texttt{'}r\texttt{]}\,(\lambda r.\,{}^{\circ}r)) = \lambda k.\,\texttt{[}M\texttt{]}\,(\lambda a.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ \texttt{[}\texttt{'}r\texttt{]}\,(\lambda r.\,{}^{\circ}r))\\
&\quad = \lambda k.\,\texttt{[}M\texttt{]}\,(\lambda a.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ (\lambda r.\,{}^{\circ}r)\,r) = \lambda k.\,\texttt{[}M\texttt{]}\,(\lambda a.\,\mathbf{let}^{\circ}\ r \Leftarrow k\,a\ \mathbf{in}\ {}^{\circ}r)\\
&\quad = \lambda k.\,\texttt{[}M\texttt{]}\,(\lambda a.\,k\,a) = \lambda k.\,\texttt{[}M\texttt{]}\,k = \texttt{[}M\texttt{]}
\end{aligned}$$

■

That is, reflection essentially captures the current continuation and passes it to M, while reification evaluates M in a delimited control context containing only k. This simple reading is somewhat obscured by the explicit lets and value-inclusions used to coerce between the two kinds of computation. However, when we eliminate the operational distinction between $^{\circ}$ and ' in the next section, the two mixed-level lets (binding r) can actually be replaced with just $M\,k$ and $k\,a$.

Somewhat surprisingly, the reflect-like $\mathcal{S}$ can itself be decomposed into a standard escape-operator and two simpler constructs:

Lemma 4.4 *$\mathcal{S}$ is definable in terms of $\mathcal{C}$, $\#$, and $\mathcal{A}$ by*

$$\mathcal{S}\,M = \mathcal{C}\,(\lambda c^{\alpha\to\texttt{'}0}.\,\mathbf{let'}\ r \Leftarrow M\,(\lambda a^{\alpha}.\,\#(\mathbf{let'}\ z \Leftarrow c\,a\ \mathbf{in}\ \mathcal{V}_{\texttt{'}\omega}\,z))\ \mathbf{in}\ \mathcal{A}\,r)$$

Proof. As in Lemma 4.3, we can use direct-style reasoning for the actual verification:

$$\begin{aligned}
&\mathcal{C}\,(\lambda c.\,\mathbf{let'}\ r \Leftarrow M\,(\lambda a.\,\#(\mathbf{let'}\ z \Leftarrow c\,a\ \mathbf{in}\ \mathcal{V}\,z))\ \mathbf{in}\ \mathcal{A}\,r)\\
&\quad = \mu(\lambda k.\,\texttt{[}(\lambda c.\,\mathbf{let'}\ r \Leftarrow M\,(\lambda a.\,\#(\mathbf{let'}\ z \Leftarrow c\,a\ \mathbf{in}\ \mathcal{V}\,z))\ \mathbf{in}\ \mathcal{A}\,r)\,(\lambda a.\,\mu(\lambda q.\,k\,a))\texttt{]}\,(\lambda z.\,\mathcal{V}\,z))\\
&\quad = \mu(\lambda k.\,\texttt{[}\mathbf{let'}\ r \Leftarrow M\,(\lambda a.\,\#(\mathbf{let'}\ z \Leftarrow (\lambda a.\,\mu(\lambda q.\,k\,a))\,a\ \mathbf{in}\ \mathcal{V}\,z))\ \mathbf{in}\ \mathcal{A}\,r\texttt{]}\,(\lambda z.\,\mathcal{V}\,z))\\
&\quad = \mu(\lambda k.\,\texttt{[}M\,(\lambda a.\,\#(\mathbf{let'}\ z \Leftarrow \mu(\lambda q.\,k\,a)\ \mathbf{in}\ \mathcal{V}\,z))\texttt{]}\,(\lambda r.\,\texttt{[}\mathcal{A}\,r\texttt{]}\,(\lambda z.\,\mathcal{V}\,z)))\\
&\quad = \mu(\lambda k.\,\texttt{[}M\,(\lambda a.\,\texttt{[}\mathbf{let'}\ z \Leftarrow \mu(\lambda q.\,k\,a)\ \mathbf{in}\ \mathcal{V}\,z\texttt{]}\,(\lambda x.\,{}^{\circ}x))\texttt{]}\,(\lambda r.\,\texttt{[}\mu(\lambda q.\,{}^{\circ}r)\texttt{]}\,(\lambda z.\,\mathcal{V}\,z)))\\
&\quad = \mu(\lambda k.\,\texttt{[}M\,(\lambda a.\,\texttt{[}\mu(\lambda q.\,k\,a)\texttt{]}\,(\lambda z.\,\texttt{[}\mathcal{V}\,z\texttt{]}\,(\lambda x.\,{}^{\circ}x)))\texttt{]}\,(\lambda r.\,(\lambda q.\,{}^{\circ}r)\,(\lambda z.\,\mathcal{V}\,z)))\\
&\quad = \mu(\lambda k.\,\texttt{[}M\,(\lambda a.\,(\lambda q.\,k\,a)\,(\lambda z.\,\texttt{[}\mathcal{V}\,z\texttt{]}\,(\lambda x.\,{}^{\circ}x)))\texttt{]}\,(\lambda r.\,{}^{\circ}r)) = \mu(\lambda k.\,\texttt{[}M\,(\lambda a.\,k\,a)\texttt{]}\,(\lambda r.\,{}^{\circ}r))\\
&\quad = \mu(\lambda k.\,\texttt{[}M\,k\texttt{]}\,(\lambda r.\,{}^{\circ}r)) = \mathcal{S}\,M
\end{aligned}$$

■

Here, we wrap the escaping continuation c provided by $\mathcal{C}$ in a control delimiter, making it into a composable function that can be passed to M. Since $\mathcal{S}$ also needs to erase the continuation after capturing it, we explicitly abort with the result r returned by M.

Because $\mathcal{C}$, $\mathcal{A}$ and $\#$ were themselves defined in terms of $\mu^{K}(\text{-})$ and $\texttt{[-]}^{K}$, in principle it does not matter which set we use in the following. Pragmatically, however, $(\mathcal{C}, \mathcal{A}, \#)$ have the advantage that their types contain no negative occurrences of $^{\circ}$, which slightly simplifies the arguments in Section 4.2. We therefore now switch attention to the new set:

Definition 4.5 *Let L be a cll signature with a* 0*-type, and let ω be a type of $L^\star$. The signature $L^{K^{cc}_{\circ\omega}}$ extends L with a computation-type constructor* ', *associated value-inclusion and lets, and the term constructors $\mathcal{C}$, $\mathcal{A}$, and #, typed as in Definition 4.1. There is an evident translation from $L^{K_{\circ\omega}}$ to $L^{K^{cc}_{\circ\omega}}$, given by the statements of Lemmas 4.3 and 4.4.*

The definitional translation $[\![-]\!]_K$ from $L^{K^{cc}_{\circ\omega}}$ to L^μ is identical to the one for $L^{K_{\circ\omega}}$ from Definition 3.34, except that the clauses for $\mu^K(\text{-})$ and `[-]`^{K} *are replaced with the clauses for $\mathcal{C}$, $\mathcal{A}$, and # from Lemma 4.2.*

Note that, since we consider complete programs to be terms of type $^\circ\iota$, the type system ensures that all control effects in an $L^{K^{cc}_{\circ\omega}}$-program occur within the dynamic scope of some #.

4.2 Level-erasure

Between Chapter 3 and Section 4.1, we have now reduced monadic effects to shift/reset and further to escapes, abort, and reset; all dependencies on the original monadic translation are gone from the translation equations, with the monad simulation being performed entirely by expanding T-reflection and -reification into simple control operators and components of the monad-triple.

However, our effect-enriched language L_0^T still has a significant practical limitation: we need to explicitly indicate the levels on all value-inclusions and lets. From a specification perspective, this is reasonable; with general reflection and reification available, we must distinguish properly between °- and '-computations in order to even define the monadic translation.

Moreover, most programs can actually be written in terms of '-computations alone, with uses of ° restricted to the definitions of monad-specific effects from reflection and reification, such as the **raise** and **handle** in Example 2.26. Thus, for particular computational effects, we may not need to explicitly expose °-computations to the language as a whole.

On the other hand, if we are to provide reflection and reification for arbitrary, programmer-defined monads, we do need general °-computations to be directly expressible in the language. In languages such as Effect-PCF (i.e., our L_0), where computation-sequencing is already explicit, adding level-annotations to all inclusions and lets may not be too problematic. But in an ML-like language, implicitly elaborated into Effect-PCF as in Section 2.1.6, there is no room for significant effect-annotations of source terms. And fortunately, as far as program evaluation is concerned, the levels can actually be safely elided.

The idea is to view $^\circ\alpha$ as a *subtype* of $'\alpha$, rather than as an entirely separate type. Membership in $^\circ\alpha$ then becomes a semantic property on values, with the type system guaranteeing absence of effects in certain terms, but not playing an active role in the actual evaluation process. Accordingly, we now define a new language with a *unified* notion of control-effects:

Definition 4.6 *Take L_1 to be L_0^Σ extended with an empty type* 0 *and associated $\mathcal{V}$. We further define the set of $L_1^{\bar\star}$-types to be the same as that of L_1, but with the type constructor* ° *replaced by a new constructor* ¡.

When ω is an $L_1^{\bar{\star}}$-type, the signature $L_1^{CC_\omega}$ of the composable-continuations language *then consists of L_1 with all instances of ${}^{\circ}$ replaced by † (so in particular, for computations we now have:*

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash {}^{\dagger}M : {}^{\dagger}\alpha} \qquad \frac{\Gamma \vdash M_1 : {}^{\dagger}\alpha_1 \qquad \Gamma, x{:}\,\alpha_1 \vdash M_2 : {}^{\dagger}\alpha_2}{\Gamma \vdash \mathbf{let}^{\dagger}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : {}^{\dagger}\alpha_2}$$

but no ${}^{\circ}$-computations), together with the following additional term constructors:

$$\frac{\Gamma \vdash M : (\alpha \to {}^{\dagger}0) \to {}^{\dagger}0}{\Gamma \vdash \mathcal{C}\, M : {}^{\dagger}\alpha} \qquad \frac{\Gamma \vdash M : \omega}{\Gamma \vdash \mathcal{A} M : {}^{\dagger}0} \qquad \frac{\Gamma \vdash M : {}^{\dagger}\omega}{\Gamma \vdash \# M : {}^{\dagger}\omega}$$

We can define a translation from our two-level control-effect language into the unified one by simply dropping the distinctions between the levels:

Definition 4.7 *The* level-erasure *translation $|\text{-}|$ is defined as follows. First, for any $L_1^{\star}$-type α, $|\alpha|$ is the $L_1^{\bar{\star}}$-type obtained by replacing all occurrences of ${}^{\circ}$ and ${}^{\shortmid}$ in α with †:*

$$|{}^{\circ}\alpha| = |{}^{\shortmid}\alpha| = {}^{\dagger}|\alpha|$$

with the other type constructors unaffected. For terms, level-erasure likewise conflates all uses of ${}^{\circ}$ and ${}^{\shortmid}$ into † (e.g., $|{}^{\circ}M| = |{}^{\shortmid}M| = {}^{\dagger}|M|$), and maps the constructors $\mathcal{C}$, $\mathcal{A}$, and $\#$ to their counterparts from Definition 4.6.

It is easy to see that if $\Gamma \vdash M : \alpha$ in $L_1^{K^{\mathrm{cc}}_{\omega}}$ then $|\Gamma| \vdash |M| : |\alpha|$ in $L_1^{CC_{|\omega|}}$.

We can also give a definitional translation of the one-level language:

Definition 4.8 *Let ω be an $L_1^{\bar{\star}}$-type. We then define the continuation-passing translation $\llbracket\text{-}\rrbracket_{\overline{K}}$ from $L_1^{CC_\omega}$ to L_1^{μ} as follows. First, let the auxiliary $\llbracket\text{-}\rrbracket_{\overline{K}{}^{\circ}\mathrm{a}}$ on types be the syntactic expansion of ${}^{\dagger}\alpha$ into $(\llbracket\alpha\rrbracket_{\overline{K}{}^{\circ}\mathrm{a}} \to {}^{\circ}\mathrm{a}) \to {}^{\circ}\mathrm{a}$, and take*

$$\bar{\omega} = \mu\mathrm{a}.\, \llbracket\omega\rrbracket_{\overline{K}{}^{\circ}\mathrm{a}}\, .$$

We also write

$$\bar{\phi} : \llbracket\omega\rrbracket_{\overline{K}{}^{\circ}\bar{\omega}} \overset{\sim}{\to} \bar{\omega} = \mathsf{roll}_{\mathrm{a}.\llbracket\omega\rrbracket_{\overline{K}{}^{\circ}\mathrm{a}}} \qquad \text{and} \qquad \bar{\psi} : \bar{\omega} \overset{\sim}{\to} \llbracket\omega\rrbracket_{\overline{K}{}^{\circ}\bar{\omega}} = \mathsf{unroll}_{\mathrm{a}.\llbracket\omega\rrbracket_{\overline{K}{}^{\circ}\mathrm{a}}}$$

for the associated isomorphisms. The translation of types is then given by:

$$\llbracket{}^{\dagger}\alpha\rrbracket_{\overline{K}} = (\llbracket\alpha\rrbracket_{\overline{K}} \to {}^{\circ}\bar{\omega}) \to {}^{\circ}\bar{\omega}$$

(with the other type constructors not affected). Correspondingly, the non-identity clauses of the term translation are:

$$\begin{aligned}
\llbracket{}^{\dagger}M\rrbracket_{\overline{K}} &= \lambda k.\, k\, \llbracket M\rrbracket_{\overline{K}} \\
\llbracket\mathbf{let}^{\dagger}\ x \Leftarrow M_1\ \mathbf{in}\ M_2\rrbracket_{\overline{K}} &= \lambda k.\, \llbracket M_1\rrbracket_{\overline{K}}\, (\lambda x.\, \llbracket M_2\rrbracket_{\overline{K}}\, k) \\
\llbracket\mathcal{C}\, M\rrbracket_{\overline{K}} &= \lambda k.\, \llbracket M\rrbracket_{\overline{K}}\, (\lambda a.\, \lambda q.\, k\, a)\, (\lambda z.\, \mathcal{V}\, z) \\
\llbracket\mathcal{A}\, M\rrbracket_{\overline{K}} &= \lambda q.\, {}^{\circ}(\bar{\phi}\, \llbracket M\rrbracket_{\overline{K}}) \\
\llbracket\# M\rrbracket_{\overline{K}} &= \lambda k.\, \mathbf{let}^{\circ}\ o \Leftarrow \llbracket M\rrbracket_{\overline{K}}\, (\lambda r.\, {}^{\circ}(\bar{\phi}\, r))\ \mathbf{in}\ k\, (\bar{\psi}\, o)
\end{aligned}$$

As usual, this translation is easily seen to be type-preserving: if $\Gamma \vdash M : \alpha$ in $L_1^{CC_\omega}$ then $\llbracket\Gamma\rrbracket_{\overline{K}} \vdash \llbracket M\rrbracket_{\overline{K}} : \llbracket\alpha\rrbracket_{\overline{K}}$ in L_1^{μ}.

We can now state our goal concretely: we want to show that the original $[\![\text{-}]\!]_K$-translation from Definition 4.5 is equivalent for evaluation purposes to a level-erasure followed by the $[\![\text{-}]\!]_{\overline{K}}$-translation. To establish this equivalence, we will define a collection of relations between the types arising from the two translations. The key ingredient here is a suitable relational action of the type constructor ${}^\circ$ in the two translations:

Definition 4.9 *Let there be given a relational correspondence between two interpretations of L_1^μ, with a computation-extension of relations $R \mapsto {}^\circ R$ (Definition 3.20). Then for any relation $R \in \mathrm{ARel}(\alpha, \alpha')$, we define the relation $\overline{{}^\circ} R \in \mathrm{CARel}({}^\circ\alpha, (\alpha' \to {}^\circ\bar{\omega}) \to {}^\circ\bar{\omega})$ by:*

$$\begin{aligned}
&m\ (\overline{{}^\circ} R)\ u \\
&\quad \iff \forall \alpha_0\ \mathsf{type}_{L_1^\mu}, O \in \mathrm{ARel}(\alpha_0, \bar{\omega}).\, \lambda k.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ k\,x\ ((R \to {}^\circ O) \to {}^\circ O)\ u \\
&\quad \iff \forall \alpha_0\ \mathsf{type}_{L_1^\mu}, O \in \mathrm{ARel}(\alpha_0, \bar{\omega}), k : \alpha \to {}^\circ\alpha_0, k' : \alpha' \to {}^\circ\bar{\omega}. \\
&\qquad\qquad (\forall a\ R\ a'.\, k\,a\ ({}^\circ O)\ k'\,a') \Rightarrow \mathbf{let}^\circ\ a \Leftarrow m\ \mathbf{in}\ k\,a\ ({}^\circ O)\ u\,k'
\end{aligned}$$

(Note that this is essentially the monad relation from Lemma 3.31, with $\mathbf{T}$ taken as the identity monad, $\gamma = \bar{\omega}$, $o = {}^\circ\bar{\omega}$, and $\phi^\bullet = \psi^\bullet = \mathsf{id}$. That is, we are using a continuation monad to simulate a trivial notion of focus effects.)

We can now define our system of relations:

Definition 4.10 *Let ω be an $L_1^\star$-type, take $\hat{\omega} \cong [\![\omega]\!]_K$ (Definition 3.34) and $\bar{\omega} \cong [\![|\omega|]\!]_{\overline{K}}$ (Definition 4.8), and let*

$$\succeq\ \in \mathrm{ARel}(\hat{\omega}, \bar{\omega})$$

be an admissible relation on final answers; for the moment we leave its definition unspecified. For any type α of $L_1^\star$, the relation

$$\succ_\alpha\ \in \mathrm{ARel}([\![\alpha]\!]_K, [\![|\alpha|]\!]_{\overline{K}})$$

is then given in the usual way for base types, sums, products, and functions. For the remaining $L_1^\star$-type constructors, we take:

$$\begin{aligned}
z \succ_0 z' &\iff z\ 0^{\mathrm{r}}\ z' \iff \mathit{false} \\
s \succ_{\Sigma\aleph} s' &\iff s\ (\Sigma_i^{\mathrm{r}} {\succ_{\aleph(i)}})\ s' \iff \exists i \in \mathrm{dom}\,\aleph, a \succ_{\aleph(i)} a'.\, s = \mathsf{in}_i\,a \wedge s' = \mathsf{in}_i\,a' \\
m \succ_{{}^\circ\alpha} u &\iff m\ (\overline{{}^\circ}{\succ_\alpha})\ u \\
u \succ_{{}^{\shortmid}\alpha} u' &\iff u\ ((\succ_\alpha \to {}^\circ{\succeq}) \to {}^\circ{\succeq})\ u' \\
&\iff \forall k, k'.\, (\forall a \succ_\alpha a'.\, k\,a\ ({}^\circ{\succeq})\ k'\,a') \Rightarrow u\,k\ ({}^\circ{\succeq})\ u'\,k'
\end{aligned}$$

It is easy to see that all $\succ_\alpha$ are admissible (we can view $\succ_0$ as being defined by an inverse image of constant functions: $z \succ_0 z' \iff \underline{0}\ \iota^{\mathrm{r}}\ \underline{1}$; for $\succ_{\Sigma\aleph}$, see Lemma A.10(3)), and that $\succ_{{}^\circ\alpha}$ and $\succ_{{}^{\shortmid}\alpha}$ are also computation-admissible.

The representation of a ${}^\circ\alpha$-value in the $[\![|\text{-}|]\!]_{\overline{K}}$-translation will always be of the form $\lambda k.\, \mathbf{let}^\circ\ x \Leftarrow m'\ \mathbf{in}\ k\,x$ for some m', and thus in particular must be parametric in the answer type. Hence, we could define $\succ_{{}^\circ\alpha}$ without committing to any particular relational interpretation of answers: $\succeq$ does not occur in the definition of $\succ_{{}^\circ\alpha}$ from $\succ_\alpha$.

On the other hand, for $\succ_{\text{'}\alpha}$ the identity of the answer type is explicitly exposed to the source language, because $\mathcal{A}$ takes an arbitrary value of type ω to be an answer, while # allows answers to be inspected as (control-effect-free) computations of type ω. Thus, if we want to relate terms containing $\mathcal{A}$ and #, we cannot choose $\succeq$ arbitrarily: it must match up with $\succ_\omega$. Fortunately, this circular dependency can be resolved, because of the following important result:

Theorem 4.11 *Let F and F' be type constructors, and let $^\bullet$ be a formal relation constructor, built out of (1) the standard relational actions of L_1-type constructors, (2) constant admissible relations (computation-admissible for computation-types), and (3) the relation constructor $\bar{^\circ}$; so that $^\bullet$ maps any relation $R \in \mathrm{ARel}(\alpha, \alpha')$ to $^\bullet R \in \mathrm{ARel}(F\alpha, F'\alpha')$.*

Then $^\bullet$ has an invariant relation *$\mu R.\, ^\bullet R \in \mathrm{ARel}(\mu \mathrm{a}.\, F\mathrm{a}, \mu \mathrm{a}.\, F'\mathrm{a})$, such that*

$$a\ (\mu R.\, ^\bullet R)\ a' \iff \mathsf{unroll}_{\mathrm{a}.F\mathrm{a}}\, a\ ^\bullet(\mu R.\, ^\bullet R)\ \mathsf{unroll}_{\mathrm{a}.F'\mathrm{a}}\, a'\,.$$

Proof. See Corollary A.20 in Section A.4. ■

Form this we immediately obtain:

Lemma 4.12 *There exists an admissible relation $\succeq\ \in \mathrm{ARel}(\hat{\omega}, \bar{\omega})$ such that*

$$o \succeq o' \iff \psi\, o \succ_\omega \bar{\psi}\, o'$$

(where $\succ_\omega$ is defined in terms of $\succeq$ by Definition 4.10).

Proof. The existence of $\succeq$ hinges on $\succ_\omega$ being defined from it using only the operations enumerated in Theorem 4.11. Thus, we can directly take $\succeq$ to be the invariant relation for the action $^\bullet$ mapping $\succeq$ to $\succ_\omega$. ■

Note that even though $\bar{\omega}$ is genuinely recursive when ω contains *any* computation-type constructors, the circularity in the definition of $\succeq$ still only occurs when ω contains a '. Otherwise, $\succ_\alpha$ becomes just an unparameterized definition by induction on α, and in particular does not depend on $\succeq$. We can then simply take Lemma 4.12 as the *definition* of $\succeq$; there is nothing to prove in that case.

We can now state the correctness result for level-erasure:

Lemma 4.13 *If $\Gamma \vdash M : \alpha$ is a term of $L_1^{K^{\circ cc}_{\omega}}$ and $\sigma \succ_\Gamma \sigma'$ then $[\![M]\!]^\sigma_K \succ_\alpha [\![|M|]\!]^{\sigma'}_{\overline{K}}$.*

Proof. The proof is by induction on M. The interesting cases are:

- Case $^\circ M$. To show:

  $$^\circ[\![M]\!]^\sigma_K \succ_{^\circ\alpha} \lambda k'.\, k'\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  I.e., that for any O and $k\ (\succ_\alpha \to {^\circ O})\ k'$,

  $$\mathbf{let}^\circ\ a \Leftarrow {^\circ[\![M]\!]^\sigma_K}\ \mathbf{in}\ k\, a\ ({^\circ O})\ k'\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  With a simplification of the LHS, this reduces to showing

  $$k\, [\![M]\!]^\sigma_K\ ({^\circ O})\ k'\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  which we get immediately from IH on M and the assumption on k and k'.

- Case $\mathbf{let}^\circ\; x \Leftarrow M_1 \;\mathbf{in}\; M_2 : {}^\circ\alpha_2$. To show:

  $$\mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; [\![M_2]\!]^{\sigma}_{K} \succ_{{}^\circ\alpha_2} \lambda k'.\, [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')$$

  That is, for any O and $k\ (\succ_{\alpha_2} \to {}^\circ O)\ k'$, we must show

  $$\mathbf{let}^\circ\; a_2 \Leftarrow (\mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; [\![M_2]\!]^{\sigma}_{K}) \;\mathbf{in}\; k\, a_2\ ({}^\circ O)\ [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')$$

  Again, by a simple rewriting of the LHS, this is equivalent to

  $$\begin{aligned}&\mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; (\lambda x.\, \mathbf{let}^\circ\; a_2 \Leftarrow [\![M_2]\!]^{\sigma}_{K} \;\mathbf{in}\; k\, a_2)\, x\\ &\quad ({}^\circ O)\ [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')\end{aligned}$$

  By IH on M_1 and the definition of $\succ_{{}^\circ\alpha_1}$, it suffices to show that

  $$\forall a_1 \succ_{\alpha_1} a_1'.\, (\lambda x.\, \mathbf{let}^\circ\; a_2 \Leftarrow [\![M_2]\!]^{\sigma}_{K} \;\mathbf{in}\; k\, a_2)\, a_1\ ({}^\circ O)\ (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')\, a_1'$$

  i.e., that for all $a_1 \succ_{\alpha_1} a_1'$,

  $$\mathbf{let}^\circ\; a_2 \Leftarrow [\![M_2]\!]^{\sigma}_{K}\{a_1/x\} \;\mathbf{in}\; k\, a_2\ ({}^\circ O)\ [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\{a_1'/x\}\, k'$$

  And that follows immediately from the IH on M_2, with extended substitutions $(\sigma, a_1/x)$ and $(\sigma', a_1'/x)$.

- Case $\mathbf{let}^\circ\; x \Leftarrow M_1 \;\mathbf{in}\; M_2 : {}^{\shortmid}\alpha_2$. To show:

  $$\lambda k.\, \mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; [\![M_2]\!]^{\sigma}_{K}\, k \succ_{{}^{\shortmid}\alpha_2} \lambda k'.\, [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')$$

  That is, for $k\ (\succ_{\alpha_2} \to {}^\circ{\succeq})\ k'$,

  $$\mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; [\![M_2]\!]^{\sigma}_{K}\, k\ ({}^\circ{\succeq})\ [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')$$

  Again, by a simple rewriting of the LHS, this is equivalent to

  $$\mathbf{let}^\circ\; x \Leftarrow [\![M_1]\!]^{\sigma}_{K} \;\mathbf{in}\; (\lambda x.\, [\![M_2]\!]^{\sigma}_{K}\, k)\, x\ ({}^\circ{\succeq})\ [\![|M_1|]\!]^{\sigma'}_{\overline{K}}\, (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')$$

  As above, by IH on M_1 and the definition of $\succ_{{}^\circ\alpha_1}$, taking $\alpha_0 = \hat{\omega}$ and $O = {\succeq}$, it suffices to show that

  $$\forall a_1 \succ_{\alpha_1} a_1'.\, (\lambda x.\, [\![M_2]\!]^{\sigma}_{K}\, k)\, a_1\ ({}^\circ{\succeq})\ (\lambda x.\, [\![|M_2|]\!]^{\sigma'}_{\overline{K}}\, k')\, a_1'$$

  which follows immediately from the IH on M_2, with extended substitutions $(\sigma, a_1/x)$ and $(\sigma', a_1'/x)$.

- Case $\mathcal{A}\, M$. To show:

  $$\lambda q.\, {}^\circ(\phi\, [\![M]\!]^{\sigma}_{K}) \succ_{{}^{\shortmid}0} \lambda q.\, {}^\circ(\bar{\phi}\, [\![|M|]\!]^{\sigma'}_{\overline{K}})$$

  I.e., that when $q\ (\succ_0 \to {}^\circ{\succeq})\ q'$ (vacuously true for any q and q') then

  $${}^\circ(\phi\, [\![M]\!]^{\sigma}_{K})\ ({}^\circ{\succeq})\ {}^\circ(\bar{\phi}\, [\![|M|]\!]^{\sigma'}_{\overline{K}})$$

By property 3.20(1) of ${}^{\circ}{\succeq}$, it suffices to show that

$$\phi\,\llbracket M\rrbracket^{\sigma}_{K} \succeq \bar{\phi}\,\llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}$$

which by Lemma 4.12 is equivalent to

$$\psi\,(\phi\,\llbracket M\rrbracket^{\sigma}_{K}) \succ_{\omega} \bar{\psi}\,(\bar{\phi}\,\llbracket|M|\rrbracket^{\sigma'}_{\overline{K}})$$

and that we get by IH on M after cancelling out the isomorphisms.

- Case $\#M$. To show:

  $$\begin{aligned}&\mathbf{let}^{\circ}\ o \Leftarrow \llbracket M\rrbracket^{\sigma}_{K}\,(\lambda r.\,{}^{\circ}(\phi\,r))\ \mathbf{in}\ {}^{\circ}(\psi\,o)\\ &\quad \succ_{{}^{\circ}\omega} \lambda k'.\,\mathbf{let}^{\circ}\ o' \Leftarrow \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}\,(\lambda r'.\,{}^{\circ}(\bar{\phi}\,r'))\ \mathbf{in}\ k'\,(\bar{\psi}\,o')\end{aligned}$$

  I.e., that for any O and $k\ (\succ_{\omega} \to {}^{\circ}O)\ k'$,

  $$\begin{aligned}&\mathbf{let}^{\circ}\ x \Leftarrow (\mathbf{let}^{\circ}\ o \Leftarrow \llbracket M\rrbracket^{\sigma}_{K}\,(\lambda r.\,{}^{\circ}(\phi\,r))\ \mathbf{in}\ {}^{\circ}(\psi\,o))\ \mathbf{in}\ k\,x\\ &\quad ({}^{\circ}O)\ \mathbf{let}^{\circ}\ o' \Leftarrow \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}\,(\lambda r'.\,{}^{\circ}(\bar{\phi}\,r'))\ \mathbf{in}\ k'\,(\bar{\psi}\,o')\end{aligned}$$

  which simplifies to

  $$\begin{aligned}&\mathbf{let}^{\circ}\ o \Leftarrow \llbracket M\rrbracket^{\sigma}_{K}\,(\lambda r.\,{}^{\circ}(\phi\,r))\ \mathbf{in}\ k\,(\psi\,o)\\ &\quad ({}^{\circ}O)\ \mathbf{let}^{\circ}\ o' \Leftarrow \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}\,(\lambda r'.\,{}^{\circ}(\bar{\phi}\,r'))\ \mathbf{in}\ k'\,(\bar{\psi}\,o')\end{aligned}$$

  By definition of $\succeq$ and the assumption on k and k', we have

  $$\forall o \succeq o'.\, k\,(\psi\,o)\ ({}^{\circ}O)\ k\,(\bar{\psi}\,o')$$

  so by 3.20(2), it suffices to show that

  $$\llbracket M\rrbracket^{\sigma}_{K}\,(\lambda r.\,{}^{\circ}(\phi\,r))\ ({}^{\circ}{\succeq})\ \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}\,(\lambda r'.\,{}^{\circ}(\bar{\phi}\,r'))$$

  By IH on M, we have $\llbracket M\rrbracket^{\sigma}_{K} \succ_{'\omega} \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}$, so we only need to show that

  $$\forall r \succ_{\omega} r'.\, {}^{\circ}(\phi\,r)\ ({}^{\circ}{\succeq})\ {}^{\circ}(\bar{\phi}\,r')$$

  which follows from 3.20(1) if we have

  $$\forall r \succ_{\omega} r'.\, \phi\,r \succeq \bar{\phi}\,r'$$

  and that is again an immediate consequence of the definition of $\succeq$, as in the case for $\mathcal{A}\,M$ above.

- Case $\mathcal{C}\,M$. Simple – same translation on both sides.

- Case $\mathcal{V}_{\beta}\,M$. To show:

  $$\mathcal{V}_{\llbracket\beta\rrbracket_K}\,\llbracket M\rrbracket^{\sigma}_{K} \succ_{\beta} \mathcal{V}_{\llbracket|\beta|\rrbracket_{\overline{K}}}\,\llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}$$

  But by IH on M, $\llbracket M\rrbracket^{\sigma}_{K} \succ_0 \llbracket|M|\rrbracket^{\sigma'}_{\overline{K}}$, so this case can never actually occur (indeed, there are no closed values of type 0).

- Case $\mathsf{in}_i\, M$. To show:

  $$\mathsf{in}_i\, [\![M]\!]^{\sigma}_{K} \succ_{\Sigma\aleph} \mathsf{in}_i\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  By IH on M, we have $[\![M]\!]^{\sigma}_{K} \succ_{\aleph(i)} [\![|M|]\!]^{\sigma'}_{\overline{K}}$, so we get the result directly from the definition of $\succ_{\Sigma\aleph}$.

- Case $\mathsf{outd}_i M$. To show:

  $$\mathsf{outd}_i\, [\![M]\!]^{\sigma}_{K} \succ_{\aleph(i)+1} \mathsf{outd}_i\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  I.e., that

  $$\mathsf{outd}_i\, [\![M]\!]^{\sigma}_{K}\ (\succ_{\aleph(i)} + 1)\ \mathsf{outd}_i\, [\![|M|]\!]^{\sigma'}_{\overline{K}}$$

  By IH on M and the definition of $\succ_{\Sigma\aleph}$, $[\![M]\!]^{\sigma}_{K} = \mathsf{in}_{i'}\, a$ and $[\![|M|]\!]^{\sigma'}_{\overline{K}} = \mathsf{in}_{i'}\, a'$ for some $i' \in \operatorname{dom} \aleph$ and $a \succ_{\aleph(i')} a'$. There are two possibilities:

  - $i' = i$. Then by definition of $+^{\mathrm{r}}$ we have

    $$\mathsf{outd}_i\, (\mathsf{in}_i\, a) = \mathsf{inl}\, a\ (\succ_{\aleph(i)} + 1)\ \mathsf{inl}\, a' = \mathsf{outd}_i\, (\mathsf{in}_i\, a')$$

  - $i' \neq i$. Then, again by definition of the relational actions of $+$ and 1,

    $$\mathsf{outd}_i\, (\mathsf{in}_{i'}\, a) = \mathsf{inr}\, \langle\rangle\ (\succ_{\aleph(i)}\ {+1})\ \mathsf{inr}\, \langle\rangle = \mathsf{outd}_i\, (\mathsf{in}_{i'}\, a')$$

■

4.3 Composable continuations from escapes and state

We now only have to implement a *one-level* language with escapes, prompts, and abort, specified by a simple continuation-passing transform. Since we may want to perform the continuation-passing translation anyway, e.g., for cps-based code generation [App92], we seem to be on the right track. On closer inspection, however, the translation does not quite produce "proper" continuation-passing terms: there is still a little bit of explicit sequencing left in the output.

Recall the equations for $[\![-]\!]_{\overline{K}}$ from Definition 4.8. The problem is with $\mathcal{A}$, and especially with #, which introduce an explicit notion of sequencing of already continuation-passing terms. By a stroke of good luck, however, we can express this sequencing in terms of another standard effect, namely state.

The key idea is to eliminate the remaining traces of explicit sequencing by performing *another* continuation-passing transformation, using a new *metacontinuation* γ to keep track of the nested $^{\circ}$-computations. That is, we take the implementation interpretation of *ambient* effects to also be given by a continuation monad. (We do not constrain the answer type of this monad, so we retain the full range of possible ambient computational effects.) While this may at first seem to move us farther away from a direct implementation, we will see that the "properly continuation-passing" terms are effectively unaffected by this second translation, while the translations of $\mathcal{A}$ and # change in a useful way.

Definition 4.14 *In L_1^{μ}, let there be given a computation-type θ of* ultimate answers*; we will often abbreviate $\alpha \to \theta$ as $\neg\alpha$. We then define $[\![\text{-}]\!]_C : L_1^{\mu} \to L_1^{\mu}$ to be the translation expanding ${}^{\circ}$-computations into continuation-passing with answer type θ, i.e.,*

$$\begin{aligned} [\![{}^{\circ}\alpha]\!]_C &= ([\![\alpha]\!]_C \to \theta) \to \theta \\ [\![{}^{\circ}M]\!]_C &= \lambda\gamma.\,\gamma\,[\![M]\!]_C \\ [\![\mathbf{let}^{\circ}\ x \Leftarrow M_1\ \mathbf{in}\ M_2]\!]_C &= \lambda\gamma.\,[\![M_1]\!]_C\,(\lambda x.\,[\![M_2]\!]_C\,\gamma) \end{aligned}$$

(with other type and term constructors unaffected as usual).

We also define a new continuation-passing translation of $L_1^{CC_\omega}$, where the answer type is itself explicitly a type of continuation-passing computations (as opposed to the unspecified notion of ambient effects in ${}^{\circ}\bar{\omega}$):

Definition 4.15 *Let ω be an $L_1^{\bar{\star}}$-type, and take $\breve{\omega} = \mu\mathrm{a}.\,[\![\omega]\!]_{\overline{K}_{\neg\neg\mathrm{a}}}$ with isomorphisms*

$$\phi^{\mathrm{u}} : [\![\omega]\!]_{\overline{K}_{\neg\neg\breve{\omega}}} \xrightarrow{\sim} \breve{\omega} = \mathsf{roll}_{\mathrm{a}.[\![\omega]\!]_{\overline{K}_{\neg\neg\mathrm{a}}}} \qquad \textit{and} \qquad \psi^{\mathrm{u}} : \breve{\omega} \xrightarrow{\sim} [\![\omega]\!]_{\overline{K}_{\neg\neg\breve{\omega}}} = \mathsf{unroll}_{\mathrm{a}.[\![\omega]\!]_{\overline{K}_{\neg\neg\mathrm{a}}}}$$

We then define $[\![\text{-}]\!]_{K^{\mathrm{u}}} : L_1^{CC_\omega} \to L_1^{\mu}$ as follows:

$$\begin{aligned} [\![{}^{¡}\alpha]\!]_{K^{\mathrm{u}}} &= K_{\neg\neg\breve{\omega}}[\![\alpha]\!]_{K^{\mathrm{u}}} \ = \ ([\![\alpha]\!]_{K^{\mathrm{u}}} \to \neg\neg\breve{\omega}) \to \neg\neg\breve{\omega} \\ [\![{}^{¡}M]\!]_{K^{\mathrm{u}}} &= \lambda k.\,\lambda\gamma.\,k\,[\![M]\!]_{K^{\mathrm{u}}}\,\gamma \\ [\![\mathbf{let}^{¡}\ x \Leftarrow M_1\ \mathbf{in}\ M_2]\!]_{K^{\mathrm{u}}} &= \lambda k.\,\lambda\gamma.\,[\![M_1]\!]_{K^{\mathrm{u}}}\,(\lambda x.\,\lambda\gamma'.\,[\![M]\!]_{K^{\mathrm{u}}}\,k\,\gamma')\,\gamma \\ [\![\mathcal{C}\,M]\!]_{K^{\mathrm{u}}} &= \lambda k.\,\lambda\gamma.\,[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda a.\,\lambda q.\,\lambda\gamma'.\,k\,a\,\gamma')\,(\lambda z.\,\lambda\gamma''.\,\mathcal{V}\,z\,\gamma'')\,\gamma \\ [\![\mathcal{A}\,M]\!]_{K^{\mathrm{u}}} &= \lambda q.\,\lambda\underline{\gamma}.\,\underline{\gamma}\,(\phi^{\mathrm{u}}\,[\![M]\!]_{K^{\mathrm{u}}}) \\ [\![\#M]\!]_{K^{\mathrm{u}}} &= \lambda k.\,\lambda\underline{\gamma}.\,[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\,\lambda\underline{\gamma'}.\,\underline{\gamma'}\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.\,k\,(\psi^{\mathrm{u}}\,o)\,\underline{\gamma}) \end{aligned}$$

Note that all but the underlined occurrences of γ can be η-reduced away, so the translations for value-inclusions, lets, and escapes form a completely standard continuation-passing transformation.

It is also easy to see that this translation consolidates the two nested continuation-passing translations into one:

Lemma 4.16 *For any type α of $L_1^{CC_\omega}$, $[\![[\![\alpha]\!]_{\overline{K}}]\!]_C = [\![\alpha]\!]_{K^{\mathrm{u}}}$ and for any term M, $[\![[\![M]\!]_{\overline{K}}]\!]_C = [\![M]\!]_{K^{\mathrm{u}}}$ (in the predomain interpretation of L_1^{μ}).*

Proof. The only complication is the type-recursion in the definition of the translations. Recall the key cases:

- $[\![{}^{¡}\alpha]\!]_{\overline{K}} = K_{{}^{\circ}\bar{\omega}}[\![\alpha]\!]_{\overline{K}}$ where $\bar{\omega} = \mu\mathrm{a}.\,[\![\omega]\!]_{\overline{K}_{{}^{\circ}\mathrm{a}}}$.
- $[\![{}^{¡}\alpha]\!]_{K^{\mathrm{u}}} = K_{\neg\neg\breve{\omega}}[\![\alpha]\!]_{K^{\mathrm{u}}}$ where $\breve{\omega} = \mu\mathrm{a}.\,[\![\omega]\!]_{\overline{K}_{\neg\neg\mathrm{a}}}$.
- $[\![{}^{\circ}\alpha]\!]_C = K_{\theta}[\![\alpha]\!]_C = \neg\neg[\![\alpha]\!]_C$.

We first strengthen the relationship for the type translation to, for any o of L_1^{μ},

$$\llbracket \llbracket \alpha \rrbracket_{\overline{K}_o} \rrbracket_C = \llbracket \alpha \rrbracket_{\overline{K}_{\llbracket o \rrbracket_C}} \tag{*}$$

The proof of (*) is a simple induction on α; the only interesting case is

$$\begin{aligned}&\llbracket \llbracket {}^{\bullet}\alpha \rrbracket_{\overline{K}_o} \rrbracket_C = \llbracket (\llbracket \alpha \rrbracket_{\overline{K}_o} \to o) \to o \rrbracket_C = (\llbracket \llbracket \alpha \rrbracket_{\overline{K}_o} \rrbracket_C \to \llbracket o \rrbracket_C) \to \llbracket o \rrbracket_C \\ &\quad \overset{\text{ih}}{=} (\llbracket \alpha \rrbracket_{\overline{K}_{\llbracket o \rrbracket_C}} \to \llbracket o \rrbracket_C) \to \llbracket o \rrbracket_C = K_{\llbracket o \rrbracket_C} \llbracket \alpha \rrbracket_{\overline{K}_{\llbracket o \rrbracket_C}} = \llbracket {}^{\bullet}\alpha \rrbracket_{\overline{K}_{\llbracket o \rrbracket_C}}\end{aligned}$$

Now, first take $o = {}^{\circ}\mathrm{a}$ in (*) to get

$$\llbracket \bar{\omega} \rrbracket_C = \llbracket \mu \mathrm{a}.\, \llbracket \omega \rrbracket_{\overline{K}_{{}^{\circ}\mathrm{a}}} \rrbracket_C = \mu \mathrm{a}.\, \llbracket \llbracket \omega \rrbracket_{\overline{K}_{{}^{\circ}\mathrm{a}}} \rrbracket_C = \mu \mathrm{a}.\, \llbracket \omega \rrbracket_{\overline{K}_{\llbracket {}^{\circ}\mathrm{a} \rrbracket_C}} = \mu \mathrm{a}.\, \llbracket \omega \rrbracket_{\overline{K}_{\neg\neg \mathrm{a}}} = \check{\omega}$$

and then, with $o = {}^{\circ}\bar{\omega}$,

$$\llbracket \llbracket \alpha \rrbracket_{\overline{K}} \rrbracket_C = \llbracket \llbracket \alpha \rrbracket_{\overline{K}_{{}^{\circ}\bar{\omega}}} \rrbracket_C = \llbracket \alpha \rrbracket_{\overline{K}_{\llbracket {}^{\circ}\bar{\omega} \rrbracket_C}} = \llbracket \alpha \rrbracket_{\overline{K}_{\neg\neg \llbracket \bar{\omega} \rrbracket_C}} = \llbracket \alpha \rrbracket_{\overline{K}_{\neg\neg \check{\omega}}}$$

Given the equalities on types, the equality on terms is completely straightforward. The cases for value-inclusion, let, and escape are immediate since their $\llbracket \text{-} \rrbracket_K$-translations do not contain any sequencing; we obtain the result by simple η-conversion. For $\mathcal{A}$ and #, we use that

$$\llbracket \bar{\phi} M \rrbracket_C = \llbracket \mathsf{roll}_{\mathrm{a}.\llbracket \omega \rrbracket_{\overline{K}_{{}^{\circ}\mathrm{a}}}} M \rrbracket_C = \mathsf{roll}_{\mathrm{a}.\llbracket \llbracket \omega \rrbracket_{\overline{K}_{{}^{\circ}\mathrm{a}}} \rrbracket_C} \llbracket M \rrbracket_C = \mathsf{roll}_{\mathrm{a}.\llbracket \omega \rrbracket_{\overline{K}_{\neg\neg\mathrm{a}}}} \llbracket M \rrbracket_C = \phi^{\mathrm{u}} \llbracket M \rrbracket_C$$

and analogously for $\bar{\psi}$. Then, for example,

$$\begin{aligned}&\llbracket \llbracket \mathcal{A} M \rrbracket_{\overline{K}} \rrbracket_C = \llbracket \lambda q.\, {}^{\circ}(\bar{\phi} \llbracket M \rrbracket_{\overline{K}}) \rrbracket_C = \lambda q.\, \lambda \gamma.\, \gamma\, (\phi^{\mathrm{u}} \llbracket \llbracket M \rrbracket_{\overline{K}} \rrbracket_C) \overset{\text{ih}}{=} \lambda q.\, \lambda \gamma.\, \gamma\, (\phi^{\mathrm{u}} \llbracket M \rrbracket_{K^{\mathrm{u}}}) \\ &\quad = \llbracket \mathcal{A} M \rrbracket_{K^{\mathrm{u}}}\end{aligned}$$

■

4.3.1 Re-tying the recursive knot

Our metacontinuation translation $\llbracket \text{-} \rrbracket_{K^{\mathrm{u}}}$ was derived directly from the original $\llbracket \text{-} \rrbracket_{\overline{K}}$. However, to match it up with he state-passing translation later, we first need to relate $\llbracket \text{-} \rrbracket_{K^{\mathrm{u}}}$ to an equivalent formulation, using an isomorphic answer type:

Definition 4.17 *Let* $\varsigma = \mu \mathrm{a}'.\, \neg \llbracket \omega \rrbracket_{\overline{K}_{\neg \mathrm{a}'}}$ *with isomorphisms*

$$\phi^{\mathrm{n}} : \neg \llbracket \omega \rrbracket_{\overline{K}_{\neg\varsigma}} \xrightarrow{\sim} \varsigma = \mathsf{roll}_{\mathrm{a}'.\neg \llbracket \omega \rrbracket_{\overline{K}_{\neg \mathrm{a}'}}} \qquad \textit{and} \qquad \psi^{\mathrm{n}} : \varsigma \xrightarrow{\sim} \neg \llbracket \omega \rrbracket_{\overline{K}_{\neg\varsigma}} = \mathsf{unroll}_{\mathrm{a}'.\neg \llbracket \omega \rrbracket_{\overline{K}_{\neg \mathrm{a}'}}}$$

Then define $\llbracket \text{-} \rrbracket_{K^{\mathrm{n}}}$ *to be the continuation-passing translation with answer type* $\neg\varsigma$*, and with translation equations for escape, abort and reset now reading:*

$$\begin{aligned}\llbracket \mathcal{C} M \rrbracket_{K^{\mathrm{n}}} &= \lambda k.\, \lambda g.\, \llbracket M \rrbracket_{K^{\mathrm{n}}} (\lambda a.\, \lambda q.\, \lambda g'.\, k\, a\, g') (\lambda z.\, \lambda g''.\, \mathcal{V}\, z\, g'')\, g \\ \llbracket \mathcal{A} M \rrbracket_{K^{\mathrm{n}}} &= \lambda k.\, \lambda g.\, \psi^{\mathrm{n}} g\, \llbracket M \rrbracket_{K^{\mathrm{n}}} \\ \llbracket \# M \rrbracket_{K^{\mathrm{n}}} &= \lambda k.\, \lambda g.\, \llbracket M \rrbracket_{K^{\mathrm{n}}} (\lambda r.\, \lambda g'.\, \psi^{\mathrm{n}}\, g'\, r) (\phi^{\mathrm{n}} (\lambda a.\, k\, a\, g))\end{aligned}$$

It should be intuitively plausible that this definition is equivalent to the one in Definition 4.15 above; we state this precisely in Corollary 4.22 below, to which one may proceed without loss of continuity.

From Section A.3, we include:

Definition 4.18 *The functorial action of L_1-type constructors on isomorphisms is given as follows, so that for any $\varphi : \alpha_1 \overset{\sim}{\rightarrow} \alpha_2$, $\Phi^i_{\mathrm{a}.\alpha}(\varphi) : \alpha\{\alpha_1/\mathrm{a}\} \overset{\sim}{\rightarrow} \alpha\{\alpha_2/\mathrm{a}\}$ and $\Psi^i_{\mathrm{a}.\beta}(\varphi) : \beta\{\alpha_1/\mathrm{a}\} \overset{\sim}{\rightarrow} \beta\{\alpha_2/\mathrm{a}\}$:*

$$\begin{aligned}
\Phi^i_{\mathrm{a}.\mathrm{a}}(\varphi)\, a &= \varphi\, a \\
\Phi^i_{\mathrm{a}.\iota}(\varphi)\, n &= n \\
\Phi^i_{\mathrm{a}.1}(\varphi)\, u &= \langle\rangle \\
\Phi^i_{\mathrm{a}.\alpha_1\times\alpha_2}(\varphi)\, p &= \langle \Phi^i_{\mathrm{a}.\alpha_1}(\varphi)\,(\mathsf{fst}\, p), \Phi^i_{\mathrm{a}.\alpha_2}(\varphi)\,(\mathsf{snd}\, p)\rangle \\
\Phi^i_{\mathrm{a}.0}(\varphi)\, z &= z \\
\Phi^i_{\mathrm{a}.\alpha_1+\alpha_2}(\varphi)\, s &= \mathsf{case}\,(s, a_1.\, \mathsf{inl}\,(\Phi^i_{\mathrm{a}.\alpha_1}(\varphi)\, a_1), a_2.\, \mathsf{inl}\,(\Phi^i_{\mathrm{a}.\alpha_2}(\varphi)\, a_2)) \\
\Phi^i_{\mathrm{a}.\Sigma\aleph}(\varphi)\, s &= \mathsf{case}\,(s, i.\, a_i.\, \mathsf{in}_i\,(\Phi^i_{\mathrm{a}.\aleph(i)}(\varphi)\, a_i)) \\
\Phi^i_{\mathrm{a}.\beta}(\varphi)\, b &= \Psi^i_{\mathrm{a}.\beta}(\varphi)\, b
\end{aligned}$$

$$\begin{aligned}
\Psi^i_{\mathrm{a}.{}^\circ\alpha}(\varphi)\, m &= \mathbf{let}^\circ\; x \Leftarrow m \;\mathbf{in}\; {}^\circ(\Phi^i_{\mathrm{a}.\alpha}(\varphi)\, x) \\
\Psi^i_{\mathrm{a}.1}(\varphi)\, o &= \langle\rangle \\
\Psi^i_{\mathrm{a}.\beta_1\times\beta_2}(\varphi)\, p &= \langle \Psi^i_{\mathrm{a}.\beta_1}(\varphi)\,(\mathsf{fst}\, p), \Psi^i_{\mathrm{a}.\beta_2}(\varphi)\,(\mathsf{snd}\, p)\rangle \\
\Psi^i_{\mathrm{a}.\alpha\to\beta}(\varphi)\, g &= \lambda x.\, \Psi^i_{\mathrm{a}.\beta}(\varphi)\,(g\,(\Phi^i_{\mathrm{a}.\alpha}(\varphi^{-1})\, x))
\end{aligned}$$

We then take advantage of the fact that our chosen solutions to recursive type equations are unique up to isomorphism, so that in particular it does not matter where we break up the recursion when defining a pair of mutually recursive types:

Lemma 4.19 *Let F and G be type constructors of L_1 (not necessarily covariant), and let $\alpha = \mu\mathrm{a}.\, F(G\mathrm{a})$ and $\alpha' = \mu\mathrm{a}'.\, G(F\mathrm{a}')$ be the solutions to the corresponding recursive type equations. Then in the predomain model, there exists an isomorphism $\chi : G\alpha \overset{\sim}{\rightarrow} \alpha'$, which further satisfies the following two (equivalent)* coherence *equations:*

$$x\colon G\alpha \vdash \mathsf{roll}_{\mathrm{a}'.G(F\mathrm{a}')}\,(\Phi^i_{\mathrm{a}'.G(F\mathrm{a}')}(\chi)\,(\Phi^i_{\mathrm{a}.G\mathrm{a}}(\mathsf{unroll}_{\mathrm{a}.F(G\mathrm{a})})\, x)) = \chi\, x : \alpha'$$

$$y\colon \alpha' \vdash \Phi^i_{\mathrm{a}.G\mathrm{a}}(\mathsf{roll}_{\mathrm{a}.F(G\mathrm{a})})\,(\Phi^i_{\mathrm{a}'.G(F\mathrm{a}')}(\chi^{-1})\,(\mathsf{unroll}_{\mathrm{a}'.G(F\mathrm{a}')}\, y)) = \chi^{-1}\, y : G\alpha$$

Proof. See Lemma A.14 in the appendix. ■

In our case, we obtain from this the following instance:

Lemma 4.20 *There exists an isomorphism $\chi : \neg\breve{\omega} \overset{\sim}{\rightarrow} \varsigma$. This induces for any type α of $L_1^{CC_\omega}$ an isomorphism $\Upsilon_\alpha : [\![\alpha]\!]_{K^{\mathrm{u}}} \overset{\sim}{\rightarrow} [\![\alpha]\!]_{K^{\mathrm{n}}} = \Phi^i_{\mathrm{a}'.[\![\alpha]\!]_{\overline{K}_{\neg\mathrm{a}'}}}(\chi)$, and moreover*

$$g\colon \varsigma, r\colon [\![\omega]\!]_{K^{\mathrm{u}}} \vdash \chi^{-1}\, g\,(\phi^{\mathrm{u}}\, r) = \psi^{\mathrm{n}}\, g\,(\Upsilon_\omega\, r) : \theta$$

Proof. Define the type constructors $F\mathrm{a}' = [\![\omega]\!]_{\overline{K}_{\neg\mathrm{a}'}}$ and $G\mathrm{a} = \neg\mathrm{a}$. Then we have

$$\breve{\omega} = \mu\mathrm{a}.\, [\![\omega]\!]_{\overline{K}_{\neg\neg\mathrm{a}}} = \mu\mathrm{a}.\, F(G\mathrm{a}) \qquad \text{and} \qquad \varsigma = \mu\mathrm{a}'.\, \neg[\![\omega]\!]_{\overline{K}_{\neg\mathrm{a}'}} = \mu\mathrm{a}'.\, G(F\mathrm{a}')$$

We thus get the isomorphism $\chi : \neg\breve{\omega} \to \varsigma$ directly from Lemma 4.19. Moreover, we can write $\Upsilon_\omega = \Phi^i_{\mathrm{a}'.F\mathrm{a}'}(\chi)$. Now, first note that

$$\begin{aligned}&\Phi^i_{\mathrm{a}.\neg\alpha}(\varphi)\,h x = \Psi^i_{\mathrm{a}.\alpha\to\theta}(\varphi)\,h x = \Psi^i_{\mathrm{a}.\theta}(\varphi)\,(h\,(\Phi^i_{\mathrm{a}.\alpha}(\varphi^{-1})\,x)) = \mathsf{id}\,(h\,(\Phi^i_{\mathrm{a}.\alpha}(\varphi^{-1})\,x))\\ &\quad = h\,(\Phi^i_{\mathrm{a}.\alpha}(\varphi^{-1})\,x)\end{aligned}$$

And then, using the second form of the coherence equation from 4.19, we get

$$\begin{aligned}&\chi^{-1}\,g\,(\phi^{\mathrm{u}}\,r) = \Phi^i_{\mathrm{a}.\neg\mathrm{a}}(\phi^{\mathrm{u}})\,[\Phi^i_{\mathrm{a}'.\neg\llbracket\omega\rrbracket_{\overline{K}_{\neg\mathrm{a}'}}}(\chi^{-1})\,(\psi^{\mathrm{n}}\,g)]\,(\phi^{\mathrm{u}}\,r)\\ &\quad = [\Phi^i_{\mathrm{a}'.\neg\llbracket\omega\rrbracket_{\overline{K}_{\neg\mathrm{a}'}}}(\chi^{-1})\,(\psi^{\mathrm{n}}\,g)]\,(\Phi^i_{\mathrm{a}.\mathrm{a}}(\psi^{\mathrm{u}})\,(\phi^{\mathrm{u}}\,r)) = [\Phi^i_{\mathrm{a}'.\neg\llbracket\omega\rrbracket_{\overline{K}_{\neg\mathrm{a}'}}}(\chi^{-1})\,(\psi^{\mathrm{n}}\,g)]\,(\psi^{\mathrm{u}}\,(\phi^{\mathrm{u}}\,r))\\ &\quad = \Phi^i_{\mathrm{a}'.\neg\llbracket\omega\rrbracket_{\overline{K}_{\neg\mathrm{a}'}}}(\chi^{-1})\,(\psi^{\mathrm{n}}\,g)\,r = \psi^{\mathrm{n}}\,g\,(\Phi^i_{\mathrm{a}'.\llbracket\omega\rrbracket_{\overline{K}_{\neg\mathrm{a}'}}}(\chi)\,r) = \psi^{\mathrm{n}}\,g\,(\Upsilon_\omega\,r)\end{aligned}$$

■

Because the translations from Definitions 4.15 and 4.17 are both continuation-passing translations with isomorphic answer types, they are very closely related: instead of the usual logical relation, we get a simple equational correspondence:

Lemma 4.21 *Let* $\Gamma = (x_1{:}\,\alpha_1,\ldots,x_n{:}\,\alpha_n)$*, and let us write* Υ_Γ *for the substitution* $(\Upsilon_{\alpha_1}\,x_1/x_1,\ldots,\Upsilon_{\alpha_n}\,x_n/x_n)$*. Then for any* $L_1^{CC_\omega}$*-term* $\Gamma\vdash M:\alpha$*,*

$$\Upsilon_\alpha\,\llbracket M\rrbracket_{K^{\mathrm{u}}} = \llbracket M\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}}$$

Proof. The cases for the standard terms (products, sums, and functions) are straightforward. For example, for abstractions and applications, we have

$$\begin{aligned}&\Upsilon_{\alpha\to\beta}\,\llbracket\lambda x.\,M\rrbracket_{K^{\mathrm{u}}} = \lambda a.\,\Upsilon_\beta\,((\lambda x.\,\llbracket M\rrbracket_{K^{\mathrm{u}}})\,(\Upsilon_\alpha^{-1}\,a)) = \lambda a.\,(\lambda x.\,\Upsilon_\beta\,\llbracket M\rrbracket_{K^{\mathrm{u}}})\,(\Upsilon_\alpha^{-1}\,a)\\ &\quad \overset{\mathrm{ih}}{=} \lambda a.\,(\lambda x.\,\llbracket M\rrbracket^{\Upsilon_\Gamma,\Upsilon_\alpha x/x}_{K^{\mathrm{n}}})\,(\Upsilon_\alpha^{-1}\,a) = \lambda a.\,\llbracket M\rrbracket^{\Upsilon_\Gamma,\Upsilon_\alpha x/x}_{K^{\mathrm{n}}}\{\Upsilon_\alpha^{-1}\,a/x\} = \lambda a.\,\llbracket M\rrbracket^{\Upsilon_\Gamma,\Upsilon_\alpha\,(\Upsilon_\alpha^{-1}\,a)/x}_{K^{\mathrm{n}}}\\ &\quad = \lambda a.\,\llbracket M\rrbracket^{\Upsilon_\Gamma,a/x}_{K^{\mathrm{n}}} = (\lambda x.\,\llbracket M\rrbracket_{K^{\mathrm{n}}})^{\Upsilon_\Gamma} = \llbracket\lambda x.\,M\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}}\end{aligned}$$

$$\begin{aligned}&\Upsilon_\beta\,\llbracket M_1\,M_2\rrbracket_{K^{\mathrm{u}}} = \Upsilon_\beta\,(\llbracket M_1\rrbracket_{K^{\mathrm{u}}}\,\llbracket M_2\rrbracket_{K^{\mathrm{u}}}) = (\lambda a.\,\Upsilon_\beta\,(\llbracket M_1\rrbracket_{K^{\mathrm{u}}}\,a))\,\llbracket M_2\rrbracket_{K^{\mathrm{u}}}\\ &\quad = (\lambda a'.\,\Upsilon_\beta\,(\llbracket M_1\rrbracket_{K^{\mathrm{u}}}\,(\Upsilon_\alpha^{-1}\,a)))\,(\Upsilon_\alpha\,\llbracket M_2\rrbracket_{K^{\mathrm{u}}}) = (\Upsilon_{\alpha\to\beta}\,\llbracket M_1\rrbracket_{K^{\mathrm{u}}})\,(\Upsilon_\alpha\,\llbracket M_2\rrbracket_{K^{\mathrm{u}}})\\ &\quad \overset{\mathrm{ih}}{=} \llbracket M_1\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}}\,\llbracket M_2\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}} = (\llbracket M_1\rrbracket_{K^{\mathrm{n}}}\,\llbracket M_2\rrbracket_{K^{\mathrm{n}}})^{\Upsilon_\Gamma} = \llbracket M_1\,M_2\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}}\end{aligned}$$

For computations, let us first name the induced isomorphisms on meta-computations

$$\pi : \neg\neg\breve{\omega} \overset{\sim}{\to} \neg\varsigma = \Phi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi) \qquad \text{and} \qquad \pi^{-1} : \neg\varsigma \overset{\sim}{\to} \neg\neg\breve{\omega} = \Phi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi^{-1})$$

The value isomorphism for computations then becomes:

$$\begin{aligned}&\Upsilon_{{}^\bullet\alpha}\,u = \Phi^i_{\mathrm{a}.\llbracket{}^\bullet\alpha\rrbracket_{\overline{K}_{\neg\mathrm{a}}}}(\chi)\,u = \Phi^i_{\mathrm{a}.(\llbracket\alpha\rrbracket_{\overline{K}_{\neg\mathrm{a}}}\to\neg\mathrm{a})\to\neg\mathrm{a}}(\chi)\,u\\ &\quad = \lambda k.\,\Psi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi)\,(u\,(\Phi^i_{\mathrm{a}.\llbracket\alpha\rrbracket_{\overline{K}_{\neg\mathrm{a}}}\to\neg\mathrm{a}}(\chi^{-1})\,k))\\ &\quad = \lambda k.\,\pi\,(u\,(\lambda a.\,\Psi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi^{-1})\,(k\,(\Phi^i_{\mathrm{a}.\llbracket\alpha\rrbracket_{\overline{K}_{\neg\mathrm{a}}}}(\chi)\,a)))) = \lambda k.\,\pi\,(u\,(\lambda a.\,\pi^{-1}\,(k\,(\Upsilon_\alpha\,a))))\end{aligned}$$

With this, the cases for inclusion and let are also simple, e.g.,

$$\begin{aligned}&\Upsilon_{{}^\bullet\alpha}\,\llbracket{}^\bullet M\rrbracket_{K^{\mathrm{u}}} = \Upsilon_{{}^\bullet\alpha}\,\llbracket\lambda k.\,k\,\llbracket M\rrbracket_{\overline{K}}\rrbracket_C = \Upsilon_{{}^\bullet\alpha}\,(\lambda k.\,k\,\llbracket M\rrbracket_{K^{\mathrm{u}}})\\ &\quad = \lambda k.\,\pi\,((\lambda k.\,k\,\llbracket M\rrbracket_{K^{\mathrm{u}}})\,(\lambda a.\,\pi^{-1}\,(k\,(\Upsilon_\alpha\,a)))) = \lambda k.\,\pi\,(\pi^{-1}\,(k\,(\Upsilon_\alpha\,\llbracket M\rrbracket_{K^{\mathrm{u}}})))\\ &\quad = \lambda k.\,k\,(\Upsilon_\alpha\,\llbracket M\rrbracket_{K^{\mathrm{u}}}) \overset{\mathrm{ih}}{=} \lambda k.\,k\,\llbracket M\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}} = \llbracket{}^\bullet M\rrbracket^{\Upsilon_\Gamma}_{K^{\mathrm{n}}}\end{aligned}$$

The interesting cases are for $\mathcal{A}$ and $\#$, which actually depend on the answer type. Here we need to expand the π and π^{-1},

$$\pi\,f = \Phi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi)\,f = \lambda g.\,f\,(\chi^{-1}\,g) \qquad \text{and} \qquad \pi^{-1}\,f = \Phi^i_{\mathrm{a}.\neg\mathrm{a}}(\chi^{-1})\,f = \lambda\gamma.\,f\,(\chi\,\gamma)$$

in $\Upsilon_{{}^\bullet\alpha}$ to get

$$\Upsilon_{{}^\bullet\alpha}\,u = \lambda k.\,\lambda g.\,u\,(\lambda a.\,\lambda\gamma.\,k\,(\Upsilon_\alpha\,a)\,(\chi\,\gamma))\,(\chi^{-1}\,g)\,.$$

We then check:

$$\begin{aligned}
&\Upsilon_{{}^{\bullet}0}\,[\![\mathcal{A}\,M]\!]_{K^{\mathrm{u}}} = \Upsilon_{{}^{\bullet}0}\,[\![\lambda q.{}^{\circ}(\phi^{\mathrm{u}}\,[\![M]\!]_{K})]\!]_{C} = \Upsilon_{{}^{\bullet}0}\,(\lambda q.\lambda\gamma.\gamma\,(\phi^{\mathrm{u}}\,[\![M]\!]_{K^{\mathrm{u}}}))\\
&\quad = \lambda q.\lambda g.\chi^{-1}\,g\,(\phi^{\mathrm{u}}\,[\![M]\!]_{K^{\mathrm{u}}}) =^{\dagger} \lambda q.\lambda g.\psi^{\mathrm{n}}\,g\,(\Upsilon_{\omega}\,[\![M]\!]_{K^{\mathrm{u}}}) \overset{\mathrm{ih}}{=} \lambda q.\lambda g.\psi^{\mathrm{n}}\,g\,[\![M]\!]^{\Upsilon_{\Gamma}}_{K^{\mathrm{n}}}\\
&\quad = [\![\mathcal{A}\,M]\!]^{\Upsilon_{\Gamma}}_{K^{\mathrm{n}}}
\end{aligned}$$

where the step marked with † uses the coherence equation from Lemma 4.20.

For reset, the calculation is a little more involved, since here we do not simply discard the continuation:

$$\begin{aligned}
&\Upsilon_{{}^{\bullet}\omega}\,[\![\#M]\!]_{K^{\mathrm{u}}} = \Upsilon_{{}^{\bullet}\omega}\,[\![\lambda k.\mathbf{let}^{\circ}\ o \Leftarrow M\,(\lambda r.{}^{\circ}(\phi^{\mathrm{u}}\,r))\ \mathbf{in}\ k\,(\psi^{\mathrm{u}}\,o)]\!]_{C}\\
&\quad = \Upsilon_{{}^{\bullet}\omega}\,(\lambda k.\lambda\gamma.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\gamma'\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.k\,(\psi^{\mathrm{u}}\,o)\,\gamma))\\
&\quad = \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\gamma'\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.k\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o))\,(\chi\,(\chi^{-1}\,g)))\\
&\quad = \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\gamma'\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.k\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o))\,g)\\
&\quad = \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\gamma'\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.(\lambda r.k\,r\,g)\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o)))\\
&\quad = \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\chi^{-1}\,(\chi\,\gamma')\,(\phi^{\mathrm{u}}\,r))\,(\lambda o.\psi^{\mathrm{n}}\,(\phi^{\mathrm{n}}\,(\lambda r.k\,r\,g))\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o)))\\
&\quad =^{\dagger} \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\psi^{\mathrm{n}}\,(\chi\,\gamma')\,(\Upsilon_{\omega}\,r))\,(\lambda o.\chi^{-1}\,(\phi^{\mathrm{n}}\,(\lambda r.k\,r\,g))\,(\phi^{\mathrm{u}}\,(\psi^{\mathrm{u}}\,o)))\\
&\quad = \lambda k.\lambda g.[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda r.\lambda\gamma'.\psi^{\mathrm{n}}\,(\chi\,\gamma')\,(\Upsilon_{\omega}\,r))\,(\chi^{-1}\,(\phi^{\mathrm{n}}\,(\lambda r.k\,r\,g)))\\
&\quad = \lambda k.\lambda g.(\Upsilon_{{}^{\bullet}\omega}\,[\![M]\!]_{K^{\mathrm{u}}})\,(\lambda r.\lambda g'.\psi^{\mathrm{n}}\,g'\,r)\,(\phi^{\mathrm{n}}\,(\lambda r.k\,r\,g))\\
&\quad \overset{\mathrm{ih}}{=} \lambda k.\lambda g.[\![M]\!]^{\Upsilon_{\Gamma}}_{K^{\mathrm{n}}}\,(\lambda r.\lambda g'.\psi^{\mathrm{n}}\,g'\,r)\,(\phi^{\mathrm{n}}\,(\lambda r.k\,r\,g)) = [\![\#M]\!]^{\Upsilon_{\Gamma}}_{K^{\mathrm{n}}}
\end{aligned}$$

where again the † marks two applications of the coherence equation. ■

We can now state the observable consequence of the above result, expressed using only constructs of L_1^{μ}, i.e., without the "helper" isomorphisms χ and Υ_{α}:

Corollary 4.22 *Let M be a closed $L_1^{CC_{\omega}}$-term of type ${}^{\bullet}\iota$. Then for any $a_0 : \theta$ and $p : \iota \to \theta$,*

$$[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.a_0) = [\![M]\!]_{K^{\mathrm{n}}}\,(\lambda n.\lambda g'.p n)\,(\phi^{\mathrm{n}}\,(\lambda r.a_0))$$

Proof. Simple equational verification, using Lemma 4.21 (with empty Γ), $\Upsilon_{\iota} = \Phi^{i}_{\mathrm{a}'.\iota}(\chi) = \mathsf{id}$, and the coherence equation:

$$\begin{aligned}
&[\![M]\!]_{K^{\mathrm{n}}}\,(\lambda n.\lambda g'.p n)\,(\phi^{\mathrm{n}}\,(\lambda r.a_0)) = (\Upsilon_{{}^{\bullet}\iota}\,[\![M]\!]_{K^{\mathrm{u}}})\,(\lambda n.\lambda g'.p n)\,(\phi^{\mathrm{n}}\,(\lambda r.a_0))\\
&\quad = [\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.[\lambda n.\lambda\gamma.p n]\,(\Upsilon_{\iota}\,n)\,(\chi\,\gamma'))\,(\chi^{-1}\,(\phi^{\mathrm{n}}\,(\lambda r.a_0)))\\
&\quad = [\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p\,(\Upsilon_{\iota}\,n))\,(\lambda o.\chi^{-1}\,(\phi^{\mathrm{n}}\,(\lambda r.a_0))\,(\phi^{\mathrm{u}}\,(\psi^{\mathrm{u}}\,o)))\\
&\quad =^{\dagger} [\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.\psi^{\mathrm{n}}\,(\phi^{\mathrm{n}}\,(\lambda r.a_0))\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o)))\\
&\quad = [\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.(\lambda r.a_0)\,(\Upsilon_{\omega}\,(\psi^{\mathrm{u}}\,o))) = [\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.a_0)
\end{aligned}$$

■

Although this corollary may at first appear too specialized, it actually covers exactly what we need. In particular, if M is a term without escaping effects, it must be equivalent to an included numeral, ${}^{\bullet}\underline{n}$ in the $[\![\text{-}]\!]_{K^{\mathrm{u}}}$-translation, and we get:

$$\begin{aligned}
&[\![M]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.a_0) = [\![{}^{\bullet}\underline{n}]\!]_{K^{\mathrm{u}}}\,(\lambda n.\lambda\gamma'.p n)\,(\lambda o.a_0)\\
&\quad = [\![{}^{\bullet}\underline{n}]\!]_{K^{\mathrm{n}}}\,(\lambda n.\lambda g'.p n)\,(\phi^{\mathrm{n}}\,(\lambda r.a_0)) = [\lambda k.\lambda g.k\,\underline{n}\,g]\,(\lambda n.\lambda g'.p n)\,(\phi^{\mathrm{n}}\,(\lambda r.a_0)) = p\,\underline{n}
\end{aligned}$$

On the other hand, if M actually invokes the metacontinuation (through an $\mathcal{A}$ not protected by an enclosing #), both translations return the "error answer" a_0.

4.3.2 The continuation-state language

Let us now assume that we have available a language with Scheme-like escapes and state as the effects. For simplicity, we consider the state to consist of only a single, typed cell (additional state could still be accommodated by choosing θ appropriately):

Definition 4.23 *Let σ be an $L_1^{\bar{\star}}$-type. Then the signature $L_1^{CS_\sigma}$ of the* continuation-state language *consists of L_1 with all occurrences of ${}^{\circ}$ replaced by ${}^{\bullet}$ (as in Definition 4.6), and extended with the following term constructors:*

$$\frac{\Gamma \vdash M : (\alpha \to {}^{\bullet}0) \to {}^{\bullet}0}{\Gamma \vdash \mathcal{C}\, M : {}^{\bullet}\alpha} \qquad \frac{}{\Gamma \vdash \mathsf{!st} : {}^{\bullet}\sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{st} := M : {}^{\bullet}1}$$

Note that the type σ can be a complex type, such as $\iota \to {}^{\bullet}\iota$, so that values stored in the cell can be procedures that themselves read or modify the state. This again introduces a reflexivity in the types, and it is well known that one can define a fixed-point operator using higher-order state (as actually done for **letrec** in Scheme [CR91]). As usual, we give a definitional translation of the new language:

Definition 4.24 *Let θ be a computation-type of L_1^{μ}, and let σ be a $L_1^{\bar{\star}}$-type. Then the translation $[\![\text{-}]\!]_S$ from $L_1^{CS_\sigma}$ to L_1^{μ} is given by, on types:*

$$[\![{}^{\bullet}\alpha]\!]_S = K_{\neg\hat{\sigma}}[\![\alpha]\!]_S \qquad \textit{where } \hat{\sigma} = \mu \mathrm{a}.\, [\![\sigma]\!]_{\overline{K}_{\neg \mathrm{a}}}$$

We have $\phi^{\mathrm{s}} : [\![\sigma]\!]_S \overset{\sim}{\to} \hat{\sigma}$ and $\psi^{\mathrm{s}} : \hat{\sigma} \overset{\sim}{\to} [\![\sigma]\!]_S$ in the two directions. (As usual, these can be taken as identities if σ does not contain ${}^{\bullet}$.) Then we can give the term translations of the new constructs:

$$\begin{aligned}
[\![{}^{\bullet}M]\!]_S &= \lambda k.\, \lambda s.\, k\, [\![M]\!]_S\, s \\
[\![\mathbf{let}^{\bullet}\ x \Leftarrow M_1\ \mathbf{in}\ M_2]\!]_S &= \lambda k.\, \lambda s.\, [\![M_1]\!]_S\, (\lambda x.\, \lambda s'.\, [\![M_2]\!]_S\, k\, s')\, s \\
[\![\mathcal{C}\, M]\!]_S &= \lambda k.\, \lambda s.\, [\![M]\!]_S\, (\lambda a.\, \lambda q.\, \lambda s'.\, k\, a\, s')\, (\lambda z.\, \lambda s''.\, \mathcal{V}\, z\, s'')\, s \\
[\![\mathsf{!\,st}]\!]_S &= \lambda k.\, \lambda \underline{s}.\, k\, (\psi^{\mathrm{s}}\, \underline{s})\, \underline{s} \\
[\![\mathsf{st} := M]\!]_S &= \lambda k.\, \lambda \underline{s}.\, k\, \langle\rangle\, (\phi^{\mathrm{s}}\, [\![M]\!]_S)
\end{aligned}$$

Again, all but the underlined instances of state-passing in the above can be eta-reduced away. In other words, for the core computational structure, this is a standard continuation-translation with answer type $\hat{\sigma} \to \theta$.

Now pick $\sigma = \omega \to {}^{\bullet}0$. We then have

$$\hat{\sigma} \cong [\![\omega \to {}^{\bullet}0]\!]_S = [\![\omega]\!]_S \to (0 \to \hat{\sigma} \to \theta) \to \hat{\sigma} \to \theta$$

We will use $\hat{\sigma}$ to represent our metacontinuation γ. Although this state-based encoding of γ now also gets passed a continuation $0 \to \neg\hat{\sigma}$ and a state $\hat{\sigma}$, it will use neither of these. That is, we informally have "$\hat{\sigma} \cong [\![\omega]\!]_S \to \theta$".

Having chosen a suitable state type, we also need to express the relevant operations on the metacontinuation in terms of the constructs of our continuation-state language. For conciseness, we introduce the abbreviation:

$$\frac{\Gamma \vdash M_1 : {}^{\intercal}1 \qquad \Gamma \vdash M_2 : {}^{\intercal}\alpha}{\Gamma \vdash M_1; M_2 : {}^{\intercal}\alpha}$$

with expansion

$$M_1; M_2 \stackrel{\text{def}}{=} \mathbf{let}^{\intercal}\ \langle\rangle \Leftarrow M_1\ \mathbf{in}\ M_2$$

We already have value-inclusion, computation-sequencing, and escapes directly available in $L_1^{CS_\sigma}$. For the remaining two constructs of $L_1^{CC_\omega}$, we take:

Definition 4.25 *Let ω be an $L_1^{\bar\star}$-type. Then in $L_1^{CS_{\omega \to {}^{\intercal}0}}$, we define operators $\mathcal{A}$ and $\#$, typed as in Definition 4.6, as follows:*

$$\begin{aligned}
\mathcal{A}M &\stackrel{\text{def}}{=} \mathbf{let}^{\intercal}\ g \Leftarrow {!}\mathsf{st}\ \mathbf{in}\ g\,M \\
\#M &\stackrel{\text{def}}{=} \mathcal{C}\,(\lambda c^{\omega \to {}^{\intercal}0}.\,\mathbf{let}^{\intercal}\ g \Leftarrow {!}\mathsf{st}\ \mathbf{in}\ (\mathsf{st} := (\lambda v^{\omega}.\,(\mathsf{st} := g; c\,v)); \mathbf{let}^{\intercal}\ x \Leftarrow M\ \mathbf{in}\ \mathcal{A}x))
\end{aligned}$$

Note in particular that the procedure stored into st in # does not use the previous value of st, nor does it return to its point of call (not that it could, since its return type is empty).

We now set up a system of logical relations suitable for showing that the above state-based definitions of the control operators capture the behavior of the metacontinuation-based translation:

Lemma 4.26 *Let there be given a relational correspondence between two interpretations $\mathcal{L}$ and $\mathcal{L}'$ of L_1^{μ}, with a computation-extension of relations $R \mapsto {}^{\circ}R$. Let $\simeq\, \in \mathrm{CARel}(\theta, \theta)$ be an arbitrary computation-admissible relation on ultimate answers. Then there exists a collection of relations with the following properties:*

- *On "wrapped" ultimate answers: $\simeq^* \in \mathrm{CARel}(\theta, (0 \to \neg\hat\sigma) \to \hat\sigma \to \theta)$*

$$m \simeq^* m' \iff \forall q \in \mathrm{Val}_{\mathcal{L}'}(0 \to \neg\hat\sigma), s \in \mathrm{Val}_{\mathcal{L}'}(\hat\sigma).\, m \simeq m'\,q\,s$$

- *On metacontinuations/state: $\asymp\, \in \mathrm{ARel}(\varsigma, \hat\sigma)$,*

$$g \asymp s \iff \psi^{\mathrm{n}}\,g\ (\sim_\omega \!\to \simeq^*)\ \psi^{\mathrm{s}}\,s \iff \forall r \sim_\omega r'.\, \psi^{\mathrm{n}}\,g\,r \simeq^* \psi^{\mathrm{s}}\,s\,r'$$

- *On meta-computations: $\approx\, \in \mathrm{CARel}(\varsigma \to \theta, \hat\sigma \to \theta) = (\asymp \to \simeq)$, i.e.,*

$$x \approx x' \iff \forall g \asymp s.\, x\,g \simeq x'\,s$$

- *On values: for any $L_1^{\bar\star}$-type α, $\sim_\alpha\, \in \mathrm{ARel}([\![\alpha]\!]_{K^{\mathrm{n}}}, [\![\alpha]\!]_S)$, defined in the usual way for the standard type constructors, and in particular,*

$$u \sim_{{}^{\intercal}\alpha} u' \iff u\ ((\sim_\alpha \to \approx) \to \approx)\ u'$$

Proof. First assume that the relation $\asymp$ is given independently, and define the others in terms of it (thus satisfying all of the equivalences in the lemma except the one characterizing $\asymp$). Further, define ${}^{\bullet}\!\asymp \in \mathrm{CARel}([\![\omega]\!]_{K^{\mathrm{n}}} \to \theta, [\![\omega]\!]_S \to (0 \to \neg\hat{\sigma}) \to \hat{\sigma} \to \theta)$ by

$$\gamma\;({}^{\bullet}\!\asymp)\;\gamma' \iff \gamma\;(\sim_\omega \to \simeq^*)\;\gamma' \iff \forall r \sim_\omega r'.\, \gamma\, r \simeq^* \gamma'\, r'$$

$\simeq^*$ is a computation-admissible relation, being given as an intersection over inverse images of $\simeq$ by the (rigid) functions id on the LHS and $\lambda m.\, m\, q\, s$ on the RHS. ($\simeq^*$ does not depend on $\asymp$, so there is no concern about admissibility of the *action* defining it.) All of the relational actions defining ${}^{\bullet}\!\asymp$ from $\asymp$ are thus standard, so by Theorem 4.11 we can take $\asymp$ as the invariant relation for the overall action, i.e.,

$$g \asymp s \iff \psi^{\mathrm{n}}\, g\;({}^{\bullet}\!\asymp)\;\psi^{\mathrm{s}}\, s$$

giving us the remaining equivalence of the lemma. ∎

Having established existence of the appropriate relations, we can now easily show correctness of the state-based representation of the metacontinuation:

Lemma 4.27 *Let $\Gamma \vdash M : \alpha$ be a term of $L_1^{CC_\omega}$ and $\sigma \sim_\Gamma \sigma'$. Then $[\![M]\!]^{\sigma}_{K^{\mathrm{n}}} \sim_\alpha [\![M]\!]^{\sigma'}_S$ (where on the RHS we use the expansions of $\mathcal{A}$ and $\#$ from Definition 4.25.*

Proof. By induction on M. Most cases are immediate, with the term constructors having the same expansions in the two translations. The only exceptions are:

- Case $\mathcal{A}\, M$. We first compute

  $$\begin{aligned}[\![\mathcal{A}\, M]\!]_S &= [\![\mathbf{let}^{\dagger}\; g \Leftarrow \mathsf{!st}\;\mathbf{in}\; g\, M]\!]_S = \lambda q.\, \lambda s.\, [\![\mathsf{!st}]\!]_S\, (\lambda g.\, \lambda s'.\, [\![g\, M]\!]_S\, q\, s')\, s \\ &= \lambda q.\, \lambda s.\, (\lambda g.\, \lambda s'.\, g\, [\![M]\!]_S\, q\, s')\, (\psi^{\mathrm{s}}\, s)\, s = \lambda q.\, \lambda s.\, (\psi^{\mathrm{s}}\, s)\, [\![M]\!]_S\, q\, s\end{aligned}$$

  We must then show that $[\![\mathcal{A}\, M]\!]^{\sigma}_{K^{\mathrm{n}}} \sim_{{}^{\dagger}0} [\![\mathcal{A}\, M]\!]^{\sigma'}_S$, i.e., that

  $$\lambda q.\, \lambda g.\, (\psi^{\mathrm{n}}\, g)\, [\![M]\!]_{K^{\mathrm{n}}} \sim_{{}^{\dagger}0} \lambda q'.\, \lambda s.\, (\psi^{\mathrm{s}}\, s)\, [\![M]\!]^{\sigma'}_S\, q'\, s$$

  So let $q\;(0 \to \approx)\;q'$. Then we must show

  $$\lambda g.\, (\psi^{\mathrm{n}}\, g)\, [\![M]\!]^{\sigma}_{K^{\mathrm{n}}} \approx \lambda s.\, (\psi^{\mathrm{s}}\, s)\, [\![M]\!]^{\sigma'}_S\, q'\, s$$

  Accordingly, let $g \asymp s$; it then suffices to show that

  $$(\psi^{\mathrm{n}}\, g)\, [\![M]\!]^{\sigma}_{K^{\mathrm{n}}} \simeq^* (\psi^{\mathrm{s}}\, s)\, [\![M]\!]^{\sigma'}_S$$

  which follows from the definition of $g \asymp s$ and the IH that $[\![M]\!]^{\sigma}_{K^{\mathrm{n}}} \sim_\omega [\![M]\!]^{\sigma'}_S$.

- Case $\#M$. Again, we first expand the RHS:

  $$\begin{aligned}[\![\#M]\!]_S &= [\![\mathcal{C}\, (\lambda c.\, \mathbf{let}^{\dagger}\; g \Leftarrow \mathsf{!st}\;\mathbf{in}\; (\mathsf{st} := (\lambda v.\, (\mathsf{st} := g; c\, v)); \mathbf{let}^{\dagger}\; x \Leftarrow M\;\mathbf{in}\;\mathcal{A}\, x))]\!]_S \\ &= \cdots = \lambda k.\, \lambda s.\, [\![M]\!]_S\, (\lambda x.\, \lambda s'.\, (\psi^{\mathrm{s}}\, s')\, x\, (\lambda z.\, \mathcal{V}\, z)\, s')\, (\phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k\, v\, (\phi^{\mathrm{s}}\, (\psi^{\mathrm{s}}\, s)))) \\ &= \lambda k.\, \lambda s.\, [\![M]\!]_S\, (\lambda x.\, \lambda s'.\, (\psi^{\mathrm{s}}\, s')\, x\, (\lambda z.\, \mathcal{V}\, z)\, s')\, (\phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k\, v\, s))\end{aligned}$$

  We must now show that $[\![\#M]\!]^{\sigma}_{K^{\mathrm{n}}} \sim_{{}^{\dagger}\omega} [\![\#M]\!]^{\sigma'}_S$, i.e., that

$$\begin{aligned}&\lambda k.\, \lambda g.\, \llbracket M \rrbracket^{\sigma}_{K^{\mathrm{n}}} \, (\lambda r.\, \lambda g'.\, \psi^{\mathrm{n}}\, g'\, r)\, (\phi^{\mathrm{n}}\, (\lambda a.\, k\, ag)) \\ &\quad \sim_{{}^{\text{¡}}\omega} \lambda k'.\, \lambda s.\, \llbracket M \rrbracket^{\sigma'}_{S} \, (\lambda x.\, \lambda s'.\, (\psi^{\mathrm{s}}\, s')\, x\, (\lambda z.\, \mathcal{V}\, z)\, s')\, (\phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k'\, v\, s))\end{aligned}$$

As usual, assume $k\ (\sim_{\omega} \to \approx)\ k'$ and $g \asymp s$; we must then show

$$\begin{aligned}&\llbracket M \rrbracket^{\sigma}_{K^{\mathrm{n}}} \, (\lambda r.\, \lambda g'.\, \psi^{\mathrm{n}}\, g'\, r)\, (\phi^{\mathrm{n}}\, (\lambda a.\, k\, ag)) \\ &\quad \simeq \llbracket M \rrbracket^{\sigma'}_{S} \, (\lambda x.\, \lambda s'.\, (\psi^{\mathrm{s}}\, s')\, x\, (\lambda z.\, \mathcal{V}\, z)\, s')\, (\phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k'\, v\, s))\end{aligned}$$

By IH on M, it suffices to show that the continuations and metacontinuations passed to the two translations are related. For the continuations, we must show that if $r \sim_{\omega} x$ and $g' \asymp s'$ then

$$\psi^{\mathrm{n}}\, g'\, r \simeq (\psi^{\mathrm{s}}\, s')\, x\, (\lambda z.\, \mathcal{V}\, z)\, s'$$

which follows from the definition of $g' \asymp s'$. Similarly, for the metacontinuations, we must show that

$$\phi^{\mathrm{n}}\, (\lambda a.\, k\, a\, g) \asymp \phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k'\, v\, s)$$

Again, by definition of $\asymp$, this requires showing that for $r \sim_{\omega} r'$.

$$\psi^{\mathrm{n}}\, (\phi^{\mathrm{n}}\, (\lambda a.\, k\, ag))\, r \simeq^{*} \psi^{\mathrm{s}}\, (\phi^{\mathrm{s}}\, (\lambda v.\, \lambda q.\, \lambda s''.\, k'\, v\, s))\, r'$$

i.e., cancelling the isomorphisms, that

$$k\, r\, g \simeq^{*} \lambda q.\, \lambda s''.\, k'\, r'\, s$$

which follows immediately from the definition of $\simeq^{*}$ and the assumption on k and k'.

■

4.4 Putting it all together

Summarizing the results of this chapter, we can state:

Theorem 4.28 *Let there be given a relational correspondence between a language* $(L_1^{\mu}, \mathcal{L})$ *and itself, with a computation-extension of relations* $R \mapsto {}^{\circ}R$ *such that* $\simeq\, =\, {}^{\circ}(\iota^{\mathrm{r}})$ *is an equivalence relation.*

Further, let $\llbracket\text{-}\rrbracket_{C} : L_1^{\mu} \to L_1^{\mu}$ *be the translation of ambient effects using the continuation monad with answer type* $\theta = {}^{\circ}\iota$ *from Definition 4.14,* $p \in \mathrm{Val}_{\mathcal{L}}(\iota \to \theta)$ *a* printing function*, and* $a_0 \in \mathrm{Val}_{\mathcal{L}}(\theta)$ *an* error answer*.*

Finally, let ω *be a type of* $L_1^{\star}$*,* $\llbracket\text{-}\rrbracket_{K}$ *the continuation-passing transform with answer type* ω *from Definition 3.34, and* $\llbracket\text{-}\rrbracket_{S}$ *the continuation-state transform from Definition 4.24, with state type* $\sigma = |\omega| \to {}^{\text{¡}}0$ *and ultimate-answer type* θ*.*

Then for any complete program $\cdot \vdash M : {}^{\circ}\iota$ *in* $L_1^{K_{\circ}\omega}$*,*

$$\llbracket \llbracket M \rrbracket_{K} \rrbracket_{C}\, p \simeq \llbracket M_1' \rrbracket_{S}\, (\lambda n.\, \lambda s'.\, p n)\, (\phi^{\mathrm{s}}\, (\lambda x.\, \lambda q.\, \lambda s.\, a_0))$$

where $\cdot \vdash M_1' : {}^{\text{¡}}\iota$ *is a term of* $L_1^{CS_{\sigma}}$ *obtained syntactically from* M *by (1) erasing all level-annotations on value-inclusions and lets, and (2) defining* $\mu^{K}(\text{-})$ *and* `[-]`^{K} *in terms of escapes and state as detailed in Lemmas 4.3 and 4.4, and Definition 4.25.*

(For a correctly effect-stratified program M, the initial error-metacontinuation on the right-hand side will never be invoked. When the implementation is hosted in an ML-like language, however, the system cannot statically verify that M is typable in our stricter system, only that the level-erasure of M is ML-typable. Pragmatically, to give a more useful behavior for effect-typing errors (notably if M has *escaping* control-effects, i.e., if it effectively has type $\text{'}\iota$ rather than $^\circ\iota$), we therefore take the initial metacontinuation to produce a distinct answer a_0 when invoked; we want to show that the simulation is still correct with this error-catching extension.)

Proof. First, let $\mathcal{L}_\mathcal{C}$ be the predomain semantics for the ambient-effect monad induced by $\mathbf{K}_{\circ\iota}$ (which is easily checked to be a uniform monad in the predomain semantics) as in Proposition 2.20. Proposition 2.25 (straightforwardly extended to the additional term constructors of L_1^μ) then gives us that

$$\mathcal{L}_\mathcal{C}\llbracket\text{-}\rrbracket = \mathcal{L}\llbracket\llbracket\text{-}\rrbracket_C\rrbracket$$

for types and terms. Moreover, the standard relational action of $K_{\circ\iota}$ in $\mathcal{L}$, i.e.,

$$\begin{aligned} &m\ (CR)\ m' \iff m\ ((R \to {^\circ\iota}) \to {^\circ\iota})\ m' \\ &\quad \iff \forall\gamma,\gamma'.\, (\forall a\ R\ a'.\, \gamma\, a \simeq \gamma'\, a') \Rightarrow m\, \gamma \simeq m'\, \gamma' \end{aligned}$$

is easily seen to be a computation-extension for the notion of ambient effects determined by the continuation monad: for any a and a' such that $a\ R\ a'$,

$$\llbracket{^\circ a}\rrbracket_C = \lambda k.\, k\, a\ ((R \to \simeq) \to \simeq)\ \lambda k.\, k\, a' = \llbracket{^\circ a'}\rrbracket_C\ ,$$

and similarly for $\mathbf{let}^\circ$.

Let M' now be the $L_1^{K^{\mathrm{cc}}_{\circ\omega}}$-program obtained from M by defining $\mu^K(\text{-})$ and $\texttt{[-]}^K$ in terms of $\mathcal{C}$, $\mathcal{A}$, and # (still with their two-level types). Then from Definition 4.5 (with associated lemmas) we get that in $\mathcal{L}_\mathcal{C}$, $\llbracket M\rrbracket_K = \llbracket M'\rrbracket_K$, and hence in $\mathcal{L}$ that

$$\llbracket\llbracket M\rrbracket_K\rrbracket_C\, p = \llbracket\llbracket M'\rrbracket_K\rrbracket_C\, p \tag{*}$$

We can now use the level-erasure Lemma 4.13 to get, in the relational correspondence between the two copies of $\mathcal{L}_\mathcal{C}$:

$$\llbracket M'\rrbracket_K \succ_{\circ\iota} \llbracket|M'|\rrbracket_{\overline{K}}$$

Since $\succ_\iota$ is simply equality of numerals, this expands to

$$\begin{aligned} &\forall\alpha_0\ \mathsf{type}_{L_1^\mu}, O \in \mathrm{ARel}_{\mathcal{L}_\mathcal{C},\mathcal{L}_\mathcal{C}}(\alpha_0, \bar\omega), k \in \mathrm{Val}_{\mathcal{L}_\mathcal{C}}(\iota \to {^\circ\alpha_0}), k' \in \mathrm{Val}_{\mathcal{L}_\mathcal{C}}(\iota \to {^\circ\bar\omega}). \\ &\quad (\forall n \in \mathbf{N}.\, k\, \underline{n}\ ({^\circ O})\ k'\, \underline{n}) \Rightarrow \mathbf{let}^\circ\ x \Leftarrow \llbracket M'\rrbracket_K\ \mathbf{in}\ k\, x\ ({^\circ O})\ \llbracket|M'|\rrbracket_{\overline{K}}\, k' \end{aligned}$$

Or, in the original correspondence:

$$\begin{aligned} &\forall\alpha_0\ \mathsf{type}_{L_1^\mu}, O \in \mathrm{ARel}_{\mathcal{L},\mathcal{L}}(\llbracket\alpha_0\rrbracket_C, \llbracket\bar\omega\rrbracket_C), \\ &\qquad k \in \mathrm{Val}_\mathcal{L}(\iota \to (\llbracket\alpha_0\rrbracket_C \to \theta) \to \theta), k' \in \mathrm{Val}_\mathcal{L}(\iota \to (\llbracket\bar\omega\rrbracket_C \to \theta) \to \theta). \\ &\quad (\forall n \in \mathbf{N}.\, k\, \underline{n}\ (CO)\ k'\, \underline{n}) \Rightarrow \lambda\gamma.\, \llbracket\llbracket M'\rrbracket_K\rrbracket_C\, (\lambda x.\, k\, x\, \gamma)\ (CO)\ \llbracket\llbracket|M'|\rrbracket_{\overline{K}}\rrbracket_C\, k' \end{aligned}$$

Somewhat surprisingly, the actual choice of the relation O does not matter much; it is the use of C to computation-extend O that is important. In fact, we can simply take

$\alpha_0 = 0$, and O to vacuously relate every element of 0 to every element of $[\![\bar{\omega}]\!]_C$. Then consider the two continuations

$$k = \lambda n^{\iota}.\, \lambda \gamma^{0 \to \theta}.\, p\, n \quad \text{and} \quad k' = \lambda n'^{\iota}.\, \lambda \gamma'^{[\![\bar{\omega}]\!]_C \to \theta}.\, p\, n' \,.$$

Let n be a natural number; we must show that $k\, \underline{n}\; (CO)\; k'\, \underline{n}$, i.e., that

$$\forall \gamma, \gamma'.\, (\forall o\; O\; o'.\, \gamma\, o \simeq \gamma'\, o') \Rightarrow k\, \underline{n}\, \gamma \simeq k'\, \underline{n}\, \gamma' \,.$$

Since both k and k' ignore their metacontinuation arguments, this reduces to $p\, \underline{n} \simeq p\, \underline{n}$, which we get from reflexivity of $\simeq$. We thus have:

$$\begin{aligned} &\lambda \gamma.\, [\![[\![M']\!]_K]\!]_C\, p = \lambda \gamma.\, [\![[\![M']\!]_K]\!]_C\, (\lambda x.\, k\, x\, \gamma) \\ &\quad (CO)\; [\![[\![|M'|]\!]_{\overline{K}}]\!]_C\, k' = \lambda \gamma'.\, [\![[\![|M'|]\!]_{\overline{K}}]\!]_C\, (\lambda n.\, \lambda \gamma''.\, p\, n)\, \gamma' \end{aligned}$$

Take $\gamma = \lambda z.\, \mathcal{V}_{\theta}\, z$ and $\gamma' = \lambda o.\, a_0$; they vacuously map all O-related values to $\simeq$-related results. Expanding the definition of CO, we therefore get:

$$[\![[\![M']\!]_K]\!]_C\, p \simeq [\![[\![|M'|]\!]_{\overline{K}}]\!]_C\, (\lambda n.\, \lambda \gamma''.\, p\, n)\, (\lambda o.\, a_0) \qquad (*)$$

We can now take the step to escapes and state. Let $M_1 = |M'|$. First, Lemma 4.16 gives us

$$[\![[\![M_1]\!]_{\overline{K}}]\!]_C\, (\lambda n.\, \lambda \gamma''.\, p\, n)\, (\lambda x.\, a_0) = [\![M_1]\!]_{K^{\mathrm{u}}}\, (\lambda n.\, \lambda \gamma''.\, p\, n)\, (\lambda x.\, a_0) \qquad (*)$$

and then Corollary 4.22,

$$[\![M_1]\!]_{K^{\mathrm{u}}}\, (\lambda n.\, \lambda \gamma''.\, p\, n)\, (\lambda x.\, a_0) = [\![M_1]\!]_{K^{\mathrm{n}}}\, (\lambda n.\, \lambda g.\, p\, n)\, (\phi^{\mathrm{n}}\, (\lambda x.\, a_0)) \qquad (*)$$

From Lemma 4.27, we get, in the $\mathcal{L}$-correspondence:

$$[\![M_1]\!]_{K^{\mathrm{n}}} \sim^{\dagger}_{\iota} [\![M_1']\!]_S$$

where M_1' is obtained from M_1 by defining $\mathcal{A}$ and $\#$ in terms of $\mathcal{C}$, $!\mathsf{st}$ and $\mathsf{st} := \text{-}$ as in Definition 4.25. We want to get from this that

$$[\![M_1]\!]_{K^{\mathrm{n}}}\, (\lambda x.\, \lambda g'.\, p\, x)\, (\phi^{\mathrm{n}}\, (\lambda x.\, a_0)) \simeq [\![M_1']\!]_S\, (\lambda x.\, \lambda s'.\, p\, x)\, (\phi^{\mathrm{s}}\, (\lambda x.\, \lambda q.\, \lambda s.\, a_0)) \qquad (*)$$

Expanding the definition of $\sim^{\dagger}_{\iota}$, we need to verify that the continuations are related, i.e., that for every $n \sim_{\iota} n'$ (i.e., $n = n'$), $\lambda g'.\, p\, n \approx \lambda s'.\, p\, n'$, which again reduces to just $p\, n \simeq p\, n'$. We must also check that the initial metacontinuation and state are related by $\asymp$, i.e., that for $r \sim_{\omega} r'$, and q, s arbitrary,

$$\psi^{\mathrm{n}}\, (\phi^{\mathrm{n}}\, (\lambda x.\, a_0))\, r \simeq (\psi^{\mathrm{s}}\, (\phi^{\mathrm{s}}\, (\lambda x.\, \lambda q.\, \lambda s.\, a_0)))\, r'\, q\, s$$

And that is true since both sides simplify to a_0.

Finally, taking the lines marked with $(*)$ above together in sequence, using the transitivity of $\simeq$, gives us the desired result. ■

And finally, taking this theorem together with Chapter 3, with nontermination as the notion of ambient effects, and simply diverging for effect-typing errors, we get:

Corollary 4.29 *Let $\mathcal{L}_\perp$ be the partiality interpretation of L_1^{μ}, and let $\mathbf{T}$ be a monad in $(L_0, \mathcal{L}_\perp)$. Then we can pick a state type σ in $L_1^{\bar{\star}}$ such that for any complete L_0^{T}-program $\cdot \vdash M : {}^{\circ}\iota$,*

$$\mathcal{L}_\perp\llbracket \llbracket M \rrbracket_{T} \rrbracket(\bullet) = \mathcal{L}_\perp\llbracket \llbracket M' \rrbracket_{S}\, (\lambda n.\, \lambda s'.\, {}^{\circ}n)\, (\phi^{\mathrm{s}}\, (\lambda x.\, \perp)) \rrbracket(\bullet)$$

where M' is a term of $L_1^{CS_\sigma}$ obtained syntactically from M by (1) erasing all the levels on value-inclusions and lets, and (2) defining $\mu^{T}(\text{-})$ and $\mathtt{[-]}^{T}$ in terms of escapes, state, embeddings, and the term constructors of $\mathbf{T}$.

Proof. In the partiality semantics, with relation lifting as the computation-extension, two closed terms of type ${}^{\circ}\iota$ are related by $\simeq\, = {}^{\circ}(\iota^{\mathrm{r}})$ iff their denotations are equal in the model (so in particular, $\simeq$ is an equivalence relation).

First, let $\mathbf{T}_0 = \mathbf{I}$ be the identity monad (Example 2.16) and $\mathbf{U}_0 = \mathbf{K}_{{}^{\circ}\iota}$ the continuation monad with answer type ${}^{\circ}\iota$ (Definition 3.4). Then by Lemma 3.5 there is a monad morphism h from $\mathbf{T}_0$ to $\mathbf{U}_0$ defined as follows:

$$\mathsf{h}_{\alpha} = \lambda m^{{}^{\circ}\alpha}.\, \lambda \gamma^{\alpha \to {}^{\circ}\iota}.\, \mathbf{let}^{\circ}\ a \Leftarrow m\ \mathbf{in}\ \gamma\, a$$

From this, Proposition 3.27 gives us a monad relation between $\mathbf{I}$ and $\mathbf{K}_{{}^{\circ}\iota}$, mapping a relation $R \in \mathrm{ARel}(\alpha, \alpha')$ to $\text{'}R \in \mathrm{CARel}({}^{\circ}\alpha, (\alpha \to {}^{\circ}\iota) \to {}^{\circ}\iota)$ by:

$$\begin{aligned} m\ (\text{'}R)\ m' \iff{}& \mathsf{h}\, m = \lambda\gamma.\, \mathbf{let}^{\circ}\ x \Leftarrow m\ \mathbf{in}\ \gamma\, x\ ((R \to \simeq) \to \simeq)\ m' \\ \iff{}& \forall \gamma, \gamma'.\, (\forall a\ R\ a'.\, \gamma\, a \simeq \gamma'\, a') \Rightarrow \mathbf{let}^{\circ}\ x \Leftarrow m\ \mathbf{in}\ \gamma\, x \simeq m'\, \gamma' \end{aligned}$$

Hence, by Proposition 3.40, we get a relational correspondence between the interpretations given by $\mathcal{L}_{\mathrm{s}}\llbracket\text{-}\rrbracket = \mathcal{L}_\perp\llbracket\text{-}\rrbracket$ and $\mathcal{L}_{\mathrm{i}}\llbracket\text{-}\rrbracket = \mathcal{L}_\perp\llbracket\llbracket\text{-}\rrbracket_{C}\rrbracket$, with computation-extension ${}^{\circ}R$ taken as the $\text{'}R$ defined above.

Theorem 3.38, with $\aleph$ taken as an enumeration of all closed L_0^{T}-types, now gives us that, in the correspondence between $\mathcal{L}_{\mathrm{s}}$ and $\mathcal{L}_{\mathrm{i}}$,

$$\llbracket M \rrbracket_{T}\ ({}^{\circ}\iota^{\mathrm{r}})\ \llbracket M_1 \rrbracket_{K}$$

where M_1 in $L_1^{K_{{}^{\circ}T(\Sigma\aleph)}}$ is obtained from M by defining $\mu^{T}(\text{-})$ and $\mathtt{[-]}^{T}$ in terms of $\mu^{K}(\text{-})$ and $\mathtt{[-]}^{K}$, the components of $\mathbf{T}$, and the operations for embedding-types. Using ι^{r} as our R above, with $\gamma = \gamma' = \lambda x.\, {}^{\circ}x$, we thus get in the original correspondence:

$$\llbracket M \rrbracket_{T} = \mathbf{let}^{\circ}\ x \Leftarrow \llbracket M \rrbracket_{T}\ \mathbf{in}\ (\lambda x.\, {}^{\circ}x)\, x \simeq \llbracket \llbracket M_1 \rrbracket_{K} \rrbracket_{C}\, (\lambda x.\, {}^{\circ}x)$$

And from this, we get the desired result directly by Theorem 4.28 with $p = \lambda x.\, {}^{\circ}x$ and $a_0 = \perp_{{}^{\circ}\iota}$. ■

4.5 ML implementation and examples

In this section we illustrate how the abstract construction presented so far can be transcribed into runnable code. To emphasize the typing issues involved, we use the New Jersey dialect of Standard ML [AM91] as our concrete language, but the operational content should translate straightforwardly into Scheme as well (though instantiation to different monads may be less convenient without a "parameterized module" facility). We also give several examples; the reader may want to compare these with Wadler's presentation [Wad92b].

4.5.1 Composable continuations

In SML/NJ, first-class continuations have a type distinct from the type of general procedures. Let us therefore first set up a Scheme-style representation of such continuations as non-returning procedures (this is not essential but makes for a more direct correspondence with the development in Section 4.3):

```
signature ESCAPE =
   sig
      type void
      val coerce : void -> 'a
      val escape : (('1a -> void) -> void) -> '1a
   end;

structure Escape : ESCAPE =
   struct
      datatype void = VOID of void
      fun coerce (VOID v) = coerce v
      fun escape f = callcc (fn k => coerce (f (fn x => throw k x)))
   end;
```

For example, we can write

```
let open Escape
in 3 + escape (fn k => k (6 + coerce (k 1))) end;
(* val it = 4 : int *)
```

(The use of `void` and `coerce` instead of an unconstrained type variable in `Escape` permits continuations to be stored in `ref`-cells while staying within the ML type system [HDM93].)

Now we can define a composable-continuations facility, parameterized by the type of final answers (using Definition 4.25 and Lemma 4.4):

```
signature CONTROL =
   sig
      type ans
      val reset : (unit -> ans) -> ans
      val shift : (('1a -> ans) -> ans) -> '1a
   end;

functor Control (type ans) : CONTROL =
   struct
```

```
      open Escape
      exception MissingReset
      val mk : (ans -> void) ref = ref (fn _ => raise MissingReset)
      fun abort x = (!mk x)

      type ans = ans
      fun reset t =
         escape (fn k => let val m = !mk
                         in mk := (fn r => (mk := m; k r));
                            abort (t()) end)
      fun shift h =
         escape (fn k => abort (h (fn v => reset (fn ()=>coerce (k v)))))
   end;
```

For example,

```
structure IntCtrl = Control (type ans = int);

let open IntCtrl
in 1 + reset (fn () => 2 * shift (fn k => k (k 10))) end;
(* val it = 41 : int *)
```

4.5.2 Monadic reflection

Building on the composable-continuations package, we implement the construction of Section 3.3.5. The signature of a monad is simple:

```
signature MONAD =
   sig
      type 'a t
      val unit : 'a -> 'a t
      val ext : ('a -> 'b t) -> 'a t -> 'b t
      val show : string t -> string
   end;
```

(The monad laws have to be verified manually, though.) The component `show` is included in the signature for convenience only. We require it to satisfy $\texttt{show} \circ \texttt{unit} = \mathsf{id}$; on terms that do not factor through unit, it provides an informal string-based representation of the effect if possible. It might at first seem more general to parameterize over types, i.e., have a `show': ('a -> string) -> 'a t -> string`, but we can recover that as `fn ms=>fn t=>show (ext (unit o ms) t)`. Our goal is to define reflection and reification operations for an arbitrary monad `M` to get

```
signature RMONAD =
   sig
      structure M : MONAD
      val reflect : '1a M.t -> '1a
      val reify : (unit -> '1a) -> '1a M.t
      val run : (unit -> string) -> string
   end;
```

Here, `run` is again mostly for illustration purposes: it takes a suspended `string`-returning computation and returns the result of executing it, annotated by an external representation of its computational effects, if any.

Using `Control` we can now define a representation of the continuation monad for an arbitrary answer type (Lemma 4.3, but simplified because of level-erasure):

```
functor ContMonad (type answer) : MONAD =
   struct
      type 'a t = ('a -> answer) -> answer
      fun unit a = fn k => k a
      fun ext f t = fn k => t (fn a => f a k)
      fun show t = raise Fail "show not defined"
   end;

functor ContRep (type answer) : RMONAD =
   struct
      structure C = Control (type ans = answer)

      structure M = ContMonad (type answer = answer)
      val reflect = C.shift
      fun reify t = fn k => C.reset (fn () => k (t ()))
      fun run t = raise Fail "run not defined"
    end;
```

(where `show` and `run` cannot be defined when the answer type is unknown).

To implement the general construction, we also need to somehow represent the infinitary embedding type from Section 3.3.4. This might at first seem fundamentally incompatible with SML's type system, especially if we want a "parametric" solution, independent of the collection of available base types and type constructors. But the construction only requires us to exhibit an embedding for those types at which we actually perform a reification. Thus, all we need is what could be called a "generative type `dynamic`": a structure matching

```
signature DYNAMIC =
   sig
      type dyn
      val newdyn : unit -> ('1a -> dyn) * (dyn -> '1a)
   end;
```

such that for any monotype `'1a`, an invocation of `newdyn ()` returns a pair of functions `(to_d, from_d)` with `from_d` ∘ `to_d` equal to the identity on `'1a`. This signature can actually be implemented type-safely in SML, by exploiting the fact that the standard datatype `exn` (nominally of *exception names*, but useful for other purposes as well) can be dynamically extended with new summands:

```
structure Dynamic : DYNAMIC =
   struct
      exception Dynamic
      abstype dyn = DYN of exn
      with fun newdyn () =
         let exception E of '1a
         in (fn a => DYN (E a), fn DYN (E a) => a | _ => raise Dynamic) end
```

```
      end
   end;
```

Note that we never actually raise or handle the exception `E` anywhere; we only use it as a dynamically-allocated tag.

Remark 4.30 Encoding dynamic types in terms of exception names is probably the most efficient approach in SML/NJ (short of bypassing the type system entirely via `System.Unsafe.cast`), but we do not actually depend on existence of an "extensible datatype" for the construction. In fact, we can get the same effect by representing a value of type `dyn` as a procedure `unit -> unit`, setting a specific cell to the desired value:

```
structure Dynamic' : DYNAMIC =
   struct
      exception Dynamic
      abstype dyn = DYN of unit -> unit
      with fun newdyn () =
         let val r = ref NONE
         in (fn a => DYN (fn () => r := SOME a),
             fn (DYN d) =>
               (r := NONE; d ();
                case !r of SOME a => a | NONE => raise Dynamic)) end
      end
   end;
```

However, this needlessly builds a closure for the dynamic value, and is perhaps a bit more obscure than the `exn`-based definition above. ■

We can now complete the construction (Theorem 3.38):

```
functor Represent (structure M : MONAD) : RMONAD =
   struct
      structure CR = ContRep (type answer = Dynamic.dyn M.t)

      structure M = M
      fun reflect m = CR.reflect (fn k => M.ext k m)
      fun reify t =
         let val (to_d, from_d) = Dynamic.newdyn ()
         in M.ext (M.unit o from_d) (CR.reify t (M.unit o to_d)) end
      fun run t = M.show (reify t)
   end;
```

4.5.3 Example: exceptions

Example 1.5 from the Introduction becomes, in the concrete setting of our ML-based implementation:

```
structure ErrorMonad =
   struct
      datatype 'a t = SUC of 'a | ERR of string
      val unit = SUC
```

```
      fun ext f (SUC a) = f a
        | ext f (ERR s) = (ERR s)
      fun show (SUC a) = a
        | show (ERR s) = "<Error: " ^ s ^ ">"
   end;

functor ErrorOps (structure R : RMONAD sharing R.M = ErrorMonad) :
   sig
      val myraise : string -> '1a
      val myhandle : (unit -> '2a) -> (string -> '2a) -> '2a
   end =
   struct
      open ErrorMonad
      fun myraise e = R.reflect (ERR e)
      fun myhandle t h = case R.reify t of SUC a => a | ERR s => h s
   end;
```

Note that the operations `myhandle` and `myraise` are defined generically in terms of *any* valid implementation of reflection and reification for the exception monad. For example, since SML already has exceptions we could simply take

```
structure ErrorRep' : RMONAD =
   struct
      exception Exc of string;

      structure M = ErrorMonad open M
      fun reflect (SUC a) = a
        | reflect (ERR e) = raise Exc e
      fun reify t = SUC (t ()) handle Exc e => ERR e
      fun run t = show (reify t)
   end;
```

We can, however, also plug in the "canonical" definitions obtained from `Represent`:

```
structure ErrorRep = Represent (structure M = ErrorMonad)
structure FX = ErrorOps (structure R = ErrorRep) open FX;

fun mydiv (x,y) = if y = 0 then myraise "Div0" else x div y;
(* val mydiv : int * int -> int *)

ErrorRep.run (fn () => makestring (1 + mydiv (100, 3)));
(* val it = "34" : string *)

ErrorRep.run (fn () => makestring (1 + mydiv (100, 0)));
(* val it = "<Error: Div0>" *)

ErrorRep.run (fn () => myhandle (fn () => makestring (1 + mydiv (100, 0)))
                                (fn s => "Oops: " ^ s));
(* val it = "Oops: Div0" *)
```

The type inferred for `myraise` above is actually overly conservative wrt. weakness: since an exception-raising operation never returns normally in the first place, it is safe to give it a fully polymorphic type. We can achieve this by simply changing the definition of `myraise` to `Escape.coerce (reflect (ERR e))`. Unfortunately, `myhandle` is also only weakly polymorphic, which can be traced back to the fact that `reify` in the functor `Represent` has a weakly polymorphic type (and that itself is a consequence of its definition in terms of `Dynamic.newdyn`).

It is instructive to inspect the expansion of `myraise` and `myhandle` into the underlying state and continuation manipulations: the cell allocated for the metacontinuation in `Control` effectively contains the "current handler continuation", which is invoked by a **raise** and temporarily rebound in the scope of each new **handle**. This is very much like the way exceptions are actually implemented in SML/NJ, although the details are not quite the same: an exception-specific implementation can take advantage of particular operational properties of the monad (notably that handler continuations are invoked at most once) to optimize the generic construction a bit.

4.5.4 Example: state

The state monad is straightforward:

```
functor StateMonad (type state) : MONAD =
   struct
      type 'a t = state -> 'a * state
      fun unit a = fn s => (a,s)
      fun ext f t = fn s => let val (a,s') = t s in f a s' end
      fun show t = raise Fail "not defined"
   end;

structure IntStateMonad : MONAD =
   struct
      structure S = StateMonad (type state = int) open S
      fun show t =
         let val (a,s') = t 42
         in if s' = 42 then a else "<s: " ^ makestring s' ^ "> " ^ a end
   end

functor IntStateOps (structure R : RMONAD sharing R.M = IntStateMonad) :
   sig
      val store : int -> unit
      val fetch : unit -> int
      val tick : unit -> unit
   end =
   struct
      fun store n = R.reflect (fn s => ((),n))
      fun fetch () = R.reflect (fn s => (s,s))
      fun tick () = R.reflect (fn s => ((),s+1))
   end

structure IntStateRep = Represent (structure M = IntStateMonad)
structure FX = IntStateOps (structure R = IntStateRep) open FX;

IntStateRep.run (fn () => (store 5; tick ();
                           let val x = fetch ()
                           in tick (); makestring (2 * x) end));
(* val it = "<s: 7> 12" *)
```

Here the general construction is clearly wasteful, however: we could easily have represented the state monad without using `callcc` at all. This is also true for many other "state-like" monads, such as I/O or complexity. Thus, the real value of the general construction is when the decomposition into escapes and state is not immediately apparent, as in the following examples.

4.5.5 Example: nondeterminism

A nondeterministic computation can be represented as a *list* of answers. (Formally, this goes beyond the monads considered in Chapter 3, but extending the proof to a language with inductive datatypes such as lists is straightforward.)

```
structure ListMonad : MONAD =
   struct
      type 'a t = 'a list
      fun unit a = [a]
      fun ext f [] = []
        | ext f (h::t) = f h @ ext f t
      fun show [] = "<fail>"
        | show [x] = x
        | show (h::t) = h ^ " <or> " ^ show t
   end;

functor ListOps (structure R : RMONAD sharing R.M = ListMonad) :
   sig
      val pick : '1a list -> '1a
      val fail : unit -> '1a
      val results : (unit -> '1a) -> '1a list
   end =
   struct
      fun pick l = R.reflect l
      fun fail () = R.reflect []
      fun results t = R.reify t
   end;

structure ListRep = Represent (structure M = ListMonad)
structure FX = ListOps (structure R = ListRep) open FX;

ListRep.run (fn () => let val x = pick [3,4] * pick [5,7]
                      in if x >= 20 then makestring x else fail () end);
(* val it = "21 <or> 20 <or> 28" : string *)
```

More generally, we get Haskell-style list comprehensions "for free", in that the schema

$$[E \mid x_1 \leftarrow E_1;\ \ldots;\ x_n \leftarrow E_n]$$

(where each x_i may be used in $E_{i+1}, \ldots, E_n$ and in E) can be expressed directly as

$$\texttt{[}\mathbf{let}\ x_1 = \mu(E_1)\ \mathbf{in} \ldots \mathbf{let}\ x_n = \mu(E_n)\ \mathbf{in}\ E\texttt{]}$$

Of course, this is probably not the most efficient way of implementing list comprehensions in ML. As observed by Wadler, however, list comprehensions can be generalized to arbitrary monads [Wad92a]; similarly we get general monad comprehensions in ML simply by supplying the appropriate `[-]` and $\mu(\text{-})$ operations.

4.5.6 Example: probability

A slight refinement of the nondeterminism monad permits us to keep track not only of the possible outcomes of a nondeterministic evaluation, but also their relative probabilities, given a distribution on the individual choice operations. That is, a probabilistic computation of type α is represented by a finite set of pairs (a_i, p_i), where a_i is a value of type α, $p_i \in (0, 1]$, all the a_i are distinct, and the p_i sum to 1.

However, this example also illustrates a technical problem with monads as a structuring tool for functional programs, as opposed to describing programming language semantics: the definition of a monad requires that the operations η and -^* be defined uniformly at all types, but in general we cannot properly implement sets of higher-order values because elements of such type cannot be tested for equality.

For example, we cannot algorithmically identify two probabilistic computations like $\{(\lambda x.\, x, 1)\}$ and $\{(\lambda x.\, x, 0.5), (\lambda x.\, x, 0.5)\}$, even though both represent the same "definite" identity function. Note that the latter variant can easily arise even if we do not allow explicit non-deterministic choice at higher types – consider a source term like **let** $y = \mathsf{amb}\,(3, 4)$ **in** $\lambda x.\, x + y - y$.

While this non-uniqueness is not in itself a problem – after all, we cannot observe functions directly – we need to ensure that any *ground-type* result we may obtain by a series of applications of potentially higher-order probabilistic functions is still uniquely represented. An easy way of achieving this is to always represent "active" probabilistic computations non-uniquely using list-nondeterminism, but then only expose *reification* at types for which we can eliminate duplicates:

```
abstraction ProbMonad :
   sig
      include MONAD
      val to_t : (’a * real) list -> ’a t
      val from_t : ’’a t -> (’’a * real) list
   end =
   struct
      type ’a t = (’a * real) list  (* 0.0<p<=1.0; sum(p) = 1.0 *)
      fun unit a = [(a,1.0)]
      fun ext f ([]:’a t) = []
        | ext f ((a,p) :: t) = map (fn (b,q) => (b,p*q)) (f a) @ ext f t
      fun show’ [(a, 1.0)] = a
        | show’ [] = ""
        | show’ ((a,p) :: t) = "<p: " ^ makestring p ^ ">" ^ a ^ show’ t

      fun to_t l = l  (* could do some sanity checking here *)
      fun tally (a,p) ([]:’’a t) = [(a,p)]
        | tally (a,p) ((a’,p’) :: t) =
             if a = a’ then (a,p+p’) :: t else (a’,p’) :: tally (a,p) t
      fun from_t t = fold (fn (h,l) => tally h l) t []
      fun show t = show’ (from_t t)
   end;

functor ProbOps (structure R : RMONAD sharing R.M = ProbMonad) :
   sig
      val choose : (’1a * real) list -> ’1a
```

```
      val flip : real -> bool
      val distribution : (unit -> ''1a) -> (''1a * real) list
   end =
   struct
      fun choose l = R.reflect (ProbMonad.to_t l)
      fun flip p = if p <= 0.0 then false
                   else if p >= 1.0 then true
                   else choose [(true,p), (false,1.0-p)]
      fun distribution t = ProbMonad.from_t (R.reify t)
   end;

structure ProbRep = Represent (structure M = ProbMonad)
structure FX = ProbOps (structure R = ProbRep) open FX;

ProbRep.run (fn () => if flip 0.3 = flip 0.3 then "same" else "diff");
(* val it = "<p: 0.58>same<p: 0.42>diff" : string *)
```

Here we have used the SML/NJ `abstraction` extension to hide the implementation of the type `t`; an analogous effect could be achieved, slightly more verbosely, using the standard `abstype` construct. Also, strictly speaking, the above only gives us uniqueness up to permutation; to get a truly unique representation we actually need the type `''a` to be linearly orderable, not only supporting an equality predicate.

We can use probabilistic effects to solve "textbook problems" such as finding the distribution of the total number of heads in n tosses of a biased coin:

```
fun toss p 0 = 0
  | toss p n = if flip p then 1+toss p (n-1) else toss p (n-1);
(* val toss = fn : real -> int -> int *)

distribution (fn () => toss 0.3 5);
(* val it = [(0,0.16807), (1,0.36015), (2,0.3087), (3,0.1323),
             (4,0.02835), (5,0.00243)] : (int * real) list *)
```

Of course, in this particular case, there already exists a simple analytic solution, but the "probabilistic execution" approach also handles less regular experiment protocols, where very dissimilar branches may be taken depending on outcomes of probabilistic choices.

Note that the simulation keeps track of all possible computation paths, at a potentially exponential cost in computation time. In cases where the same net outcome can be achieved in many different ways (as in the example above), it is therefore often useful to add an explicit wrapper,

`choose (distribution (fn () =>` E`))`

around such a subcomputation E. This has no effect on the result computed (almost by definition: it is an instance of the principle $\mu([E]) = E$), but it improves efficiency by consolidating computation paths in a manner analogous to dynamic programming.

4.5.7 Example: continuations

Finally, let us consider the continuation monad (for an arbitrary but fixed answer type). This lets us define both escapes and composable first-class continuations. We already have the functor `ContMonad`. Let us create a specific instantiation:

```
structure StringContMonad : MONAD =
   struct
      structure S = ContMonad (type answer = string) open S
      fun show t = t (fn x => x)
   end

functor StringContOps (structure R : RMONAD sharing R.M = StringContMonad):
   sig
      val mycallcc : (('1a -> '1b) -> '1a) -> '1a
      val myshift : (('1a -> string) -> string) -> '1a
      val myreset : (unit -> string) -> string
   end =
   struct
      fun mycallcc h =
        R.reflect (fn k => let fun c a = R.reflect (fn k' => k a)
                           in R.reify (fn () => h c) k end)
      fun myshift h =
        R.reflect (fn k => R.reify (fn () => h k) (fn x => x))
      fun myreset t =  R.reify t (fn x => x)
   end;

structure StringContRep = Represent (structure M = StringContMonad)
structure FX = StringContOps (structure R = StringContRep) open FX;

StringContRep.run (fn () => makestring (3 + mycallcc (fn k => 6 * k 1)));
(* val it = "4" : string *)

StringContRep.run (fn () => "a" ^ myreset (fn () =>
                                     "b" ^ myshift (fn k => k (k "c"))));
(* val it = "abbc" : string *)
```

4.6 Related work

Different notions of *functional* or *composable continuations* have been studied by a number of researchers. Early work [JD88, FWFD88, DF90] presumed explicit support from the compiler or runtime system for the actual implementation, such as the ability to mark or splice together delimited stack segments. However, an encoding in standard Scheme of one variant was devised by Sitaram and Felleisen [SF90]. Still, this embedding was quite complex, relying on dynamically-allocated, mutable data structures, `eq?`-tests, and the dynamic typing of Scheme.

Another explicitly Scheme-implementable notion of partial continuations was proposed by Queinnec and Serpette [QS91]; the code required is perhaps even more intricate. And more recently, an implementation of a related construct in Standard ML of New Jersey was presented by Gunter, Rémy and Riecke [GRR95].

At least initially, most of these operators appear more general than monadic reflection for continuations, but it is not clear if the additional expressive power is sufficiently useful in practice to justify their fairly complex implementations. The much simpler construction presented in this chapter uses only a single, statically-typed cell holding a continuation, perhaps the minimal increment over `call/cc` alone.

Much more significantly, however, this implementation is directly derived from and related to the original specification; other efforts gave at most an informal argument that the (usually operationally specified) control construct was correctly implemented by the code. Given the relatively complex correctness proof for even the very simple control operators used in this chapter ($\mathcal{A}$ and $\#$), it is not likely that any of the alternatives would be easier to verify.

The term *metacontinuation*, with a fairly broad meaning, was first used in giving a formal semantics to a notion of computational reflection by Wand and Friedman [WF88]. The more restrictive usage of the term, where the metacontinuation actually arises from a standard continuation-passing transform of an "almost-cps" term, is due to Danvy and Filinski [DF90].

The further observation that the metacontinuation can be represented by a storage cell was first exploited in a preliminary version of the present work [Fil94]. An application of this technique for continuation-based partial evaluation was reported by Lawall and Danvy, who found that a `call/cc`-based implementation of composable continuations uniformly outperformed the equivalent explicit continuation-passing translation, especially with respect to heap usage [LD94].

The main difference between the variant of composable continuations considered in this chapter and the previous formulations is that we start with an even more abstract specification of the original operators, distinguishing in the type system between computations with and without control effects. Correspondingly, the definitional translation only has a non-trivial effect on computations of the former kind.

This distinction gives us a very simple correspondence between composable continuations and monadic reflection for the continuation monad, further motivating composable continuations as the *canonical* control effect. (The change was also partially necessitated by the introduction of ambient effects; in [Fil94], the target language of the definitional translation was assumed to be effect-free in the present terminology.)

Chapter 5

Conclusions

5.1 Summary

We have analyzed a new approach to incorporating computational effects in a functional language. In many ways, it combines the best features of the existing "purely functional" and "imperative" models for effects, as well as providing a basis for introducing effects incrementally. Let us recapitulate the main properties of the construction:

Convenience. An important advantage of monadic reflection is the ease with which it fits into the familiar programming paradigm of ML-like languages. There is essentially no up-front cost: programs do not have to be (re)written in any particular style, the effects used do not have to be settled upon in advance, and we can directly use the existing type checker, module system, etc.

In fact, there is no need to even explicitly mention monads when writing the bulk of the program. Typically, the programmer simply defines the desired operations (such as raise and handle for exceptions, pick and results for nondeterminism, or spawn and yield for resumptions) using monadic reflection for a suitable monad, then expresses the program in terms of those new primitives alone.

The visible difference from a "manual" implementation of the effects in terms of continuations and state (or, even more markedly, as part of the compiler) is the amount of effort and ingenuity required. Usually, the monad specification consists of only a few lines of simple, effect-free code. Likewise, the exported operations are generally a simple combination of the reflection and/or reification operators. We never have to think about capturing, storing, retrieving, and invoking continuations to implement, say, a backtracking search; all the required low-level code is synthesized mechanically from an abstract specification of nondeterministic choice.

Ease of reasoning. Despite its apparent "imperative" nature, monadic reflection can equally well be viewed as a technique for writing "purely functional" programs in a more concise notation, much like monad comprehensions [Wad92a]. In fact, any imperative program fragment is extensionally equivalent to its monadic-style counterpart, in the sense that there exist language-definable isomorphisms between the two representations.

A crucial point, however, is that this correspondence to monadic style is a *means*, not an *end*, for reasoning about programs. Simply "being expressible with a monad", or "having a translation into purely functional code" are vacuous properties, true of *any* program using continuations and state (since both are monadic effects), and do not help us prove anything new. Rather, we must exploit the knowledge that a program is expressible with a *particular* monad, with a more restrictive notion of effects than the continuation-state monad into which it happens to be embedded for implementation purposes.

For example, in an ML-like language defined by exception-passing on top of partiality, it is easy to argue correctness of a source-level transformation such as $f\,x + f\,x = 2 \times f\,x$: the subcomputation $f\,x$ must either succeed with a value, raise an exception, or diverge; in all three cases, the two expressions are equivalent. On the other hand, if we examine only a hand-coded implementation of exceptions in terms of escapes and state – even if the latter effects are used for no other purpose in the program – we cannot argue nearly as directly that common-subexpression elimination is a valid optimization principle.

Efficiency. Execution efficiency is an important concern for practical uses of effects, and monadic reflection usually fares significantly better than an actual translation into monadic style. If effects are rare, programs run at full speed without the overhead of explicitly performing the administrative manipulations specified by the monad, such as tagging and checking return values for exceptions.

To ensure good performance of the reflection and reification operators as well, we do need to assume a reasonably efficient implementation of call/cc in the host language. In cps-based compilers, providing a cheap first-class continuation facility is generally straightforward [App92]. And even in stack-based implementations, good techniques exist for keeping at least the amortized cost per call/cc acceptably low [HDB90].

Still, if a particular effect is heavily used, it may be preferable to rewrite the program in the corresponding monadic style. For example, if the parameter provided by an environment monad changes very frequently, we should make it an explicit argument to all functions using it. Not only is this likely to be faster than going through the store on every access, but it will probably result in a clearer program as well. Conversely, of course, rarely-used arguments can be made implicit, improving both execution speed and clarity – the latter by focusing attention on the few cases where some value changes, rather than on all the ones where it is merely propagated.

In either case, however, the changeover need not be done all at once, because we can use reflection and reification to interface between program fragments using the two approaches. Indeed, the best solution may well be to make the effect explicit in parts of the program that use it heavily, and implicit in those that are not directly affected by it.

5.2 Future work

Several opportunities for extensions and future investigation arise naturally:

Recursive types in the specification language. Even though our language for defining monadic effects was simply typed, there do not appear to be any fundamental problems in allowing general recursive types. In fact, the logical-relations proofs in Chapter 4 already handle recursion in the answer type for the continuation monads using invariant relations, and similar techniques could in all likelihood be used in Chapter 3 as well.

However, a proper treatment of recursive types would probably include more than merely adding the μ-types from Section 3.2.1 to the specification language. For example, it might be appropriate to also allow *recursively-defined computation-types*, i.e., types of the form $\mu b.\, \beta$, with an explicit notion of computation-type variables and the associated extensions to generalized let, etc.

Even more important, we would want a general treatment of *recursive monad specifications*, such as used in the continuation-passing translation of Definition 3.34. The required structure seems to be an L_0-monad in the usual sense, but *parameterized by an* $L_0^\star$*-type.* This would allow us to express, for example, ML-style `ref`-cells storing procedures, or exceptions carrying non-ground data, without introducing explicit isomorphisms.

Layering effects. Although its potential was not fully realized in this thesis, the organization in terms of ambient and focus effects should generalize directly to multiple, layered effects. In other words, we should be able to integrate different notions of effects in a single language by a series of nested monadic translations, at each step taking the previous focus effect as the new notion of ambient effect.

Moreover, this layered strategy for modularly *specifying* effects promises to generalize to a modular *implementation* of such effects in terms of continuations and state. More specifically, we would first relate a heterogeneous tower of monads to a tower of continuation-monads (applying at each level the construction in Chapter 3), then flatten this cps tower into a single-level implementation (as in Chapter 4), with a collection of cells, each holding one meta-continuation of the hierarchy.

Indeed, an apparently-working implementation based on this strategy already exists, and preliminary investigations into both its theoretical justification and practical usefulness have been very encouraging. However, time constraints made it infeasible to include a treatment this generalized construction in the thesis. Fully formalizing and analyzing the multiple-effect case is therefore left as future work.

Practical effect-typing for monadic effects. While one of the goals of the construction was to permit a direct embedding of the effect-enriched language into ML, this does not mean that we could not take advantage of a more refined type system. Some discipline is required when writing programs with effects, and it would be useful to detect violations of effect-stratification statically, rather than during program execution.

Accordingly, there should be a way to optionally make the effects used by a piece of code manifest in its type, especially at module boundaries. We could of course achieve this by always exporting procedures in their "fully reified" form. Such an approach, however, tends to be impractically verbose, and the additional conversions, although semantically transparent, may impose a non-negligible overhead. We would want a concise and unobtrusive way of representing that same information in direct style.

Existing work in this area tends to consider mainly low-level notions of effects (jumps and state manipulation) [JG89, KJLS87], rather than application-specific, higher-level concepts. But given the often complex relationship between a monadic specification and its imperative implementation, it seems highly unlikely that an automated analysis based on the latter would be able to detect a higher-level pattern such as an exception-handling system.

Moreover, current effect-type systems are generally phrased in terms of Curry-style *type inference* (i.e., with the semantics of a program given a priori, and independently of its type). The reflection-based approach to effects, on the other hand, also seems well suited for Church-style *type reconstruction* (where type information is considered an inherent part of the program, only elided for conciseness), as already advocated for ML in [HM93].

5.3 Closing remarks

Perhaps the most concise way of stating the main conclusion of this work is that a functional program can and should *distinguish between specification and implementation* of computational effects – as it already would for any other abstract data type. Oversimplifying grossly, we could summarize the alternatives by following Hegelian triad:

- **Thesis:** the implementation *is* the specification. The meaning of an effect is fully determined by a *reference implementation.* For example, a Scheme program could be written with intuitive but informal abstractions such as error handlers, backtracking, or threads, ultimately defined only by their expansions into `call/cc` and `set!`.
- **Antithesis:** the specification *is* the implementation. The behavior of an effect is fully determined by a purely functional *executable specification.* For example, a Haskell program could be written in monadic style, expanding into explicit exception-passing, success lists, or resumptions.
- **Synthesis:** the implementation *is related to* the specification. An effect has a declarative meaning and an imperative behavior, with the latter obtained from the former in a systematic, but not necessarily direct way. For example (but by no means exclusively), a program could be written and analyzed in terms of monadic reflection, but eventually executed using effects built out of escapes and state.

In other words, the tension between Haskell-style monads and Scheme-style primitive effects need not and should not be resolved in unilateral favor of one or the other; it is precisely through their interplay that the best qualities of both are exposed.

Appendix A

Properties of the Predomain Model

In this chapter we summarize a few auxiliary results about the predomain semantics, needed in Chapters 3 and 4, but somewhat tangential to the main development. Most are fairly simple adaptations of standard domain-theoretic results to our predomain setting.

A.1 Recursive type definitions

The proof that all recursive type equations have solutions in the predomain semantics hinges on exhibiting for any type constructor a suitable functorial action in the category of domains and strict continuous functions. That is, in addition to the evident action on objects, we need an action on morphisms.

Although we could construct such functors directly in the model, using the standard notation for continuous functions, it seems more convenient and consistent to use the existing term syntax for effects (fixed to be partiality) in the definitions, and only consider the denotations of the constructed terms in the end.

For the purposes of this appendix only, let us therefore extend our term syntax by introducing the additional computation-type constructor ${}^{\perp}\alpha$ and term constructors ${}^{\perp}M$ and $\mathbf{let}^{\perp}\ x \Leftarrow M_1\ \mathbf{in}\ M_2$ with types:

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash {}^{\perp}M : {}^{\perp}\alpha} \qquad \frac{\Gamma \vdash M_1 : {}^{\perp}\alpha \qquad \Gamma, x{:}\,\alpha \vdash M_2 : \beta}{\Gamma \vdash \mathbf{let}^{\perp}\ x \Leftarrow M_1\ \mathbf{in}\ M_2 : \beta}$$

analogous to the existing ambient effects, but always referring to the partiality monad. The $\mathbf{let}^{\perp}$ is actually more like the generalized $\mathbf{let}^{\circ}_{\beta}$ than like $\mathbf{let}^{\circ}$, because the result can be of any computation-type. We omit the explicit type subscript in $\mathbf{let}^{\perp}$, however, because we already have a uniform semantic characterization of its meaning at all pointed types:

$$\begin{aligned} \mathcal{L}[\![{}^{\perp}\alpha]\!]^{\theta} &= \mathcal{L}[\![\alpha]\!]^{\theta}{}_{\perp} \\ \mathcal{L}[\![{}^{\perp}M]\!]^{\theta}(\rho) &= \mathsf{up}(\mathcal{L}[\![{}^{\perp}M]\!]^{\theta}(\rho)) \\ \mathcal{L}[\![\mathbf{let}^{\perp}\ x \Leftarrow M_1\ \mathbf{in}\ M_2]\!]^{\theta}(\rho) &= (\underline{\lambda} a.\, \mathcal{L}[\![M_2]\!]^{\theta}(\rho[x \mapsto a]))^{\ddagger}(\mathcal{L}[\![M_1]\!]^{\theta}(\rho)) \end{aligned}$$

where $f^{\ddagger}$ is the generalized strict extension from Section 2.1.3(lifting).

For embedding-types, we also define a general case-construct, dispatching among all the possibilities in $\aleph$:

$$\frac{\Gamma \vdash M : \Sigma\aleph \qquad \forall i \in I.\ \Gamma, x{:}\,\aleph(i) \vdash \mathcal{M}(i) : \alpha}{\Gamma \vdash \mathsf{case}(M, i.\, x_i.\, \mathcal{M}(i)) : \alpha}$$

with semantics

$$\mathcal{L}[\![\mathsf{case}(i.\, M, i.\, x_i.\, \mathcal{M}(i))]\!]^{\theta}(\rho) = \mathcal{L}[\![\mathcal{M}(i)]\!]^{\theta}(\rho[x_i \mapsto a_i]) \text{ when } \mathcal{L}[\![M]\!]^{\theta}(\rho) = (i, a_i)$$

Again, this case is never used in writing actual programs; it merely gives us a convenient way of referring to semantic entities in the predomain model.

Definition A.1 *For any value-type α and computation-type β over $\{\mathrm{a}\}$ in L_1 (i.e., L_0^{Σ} extended with an empty type), we define a type constructor $\Phi_{\mathrm{a}.\alpha}(\text{-},\text{-})$ and a computation-type constructor $\Psi_{\mathrm{a}.\beta}(\text{-},\text{-})$ by*

$$\Phi_{\mathrm{a}.\alpha}(\alpha^-, \alpha^+) = \alpha\{\alpha^-/\mathrm{a}^-, \alpha^+/\mathrm{a}^+\} \qquad \textit{and} \qquad \Psi_{\mathrm{a}.\beta}(\alpha^-, \alpha^+) = \beta\{\alpha^-/\mathrm{a}^-, \alpha^+/\mathrm{a}^+\}$$

where $\alpha\{\alpha'/\mathrm{a}^+\}$ means α with α' substituted for all positive occurrences of a*, and analogously for negative occurrences.*

Further, we define term constructors $\Phi_{\mathrm{a}.\alpha}(\text{-},\text{-})$ and $\Psi_{\mathrm{a}.\beta}(\text{-},\text{-})$ with types:

$$\frac{f^- : \alpha_1^- \to {}^{\perp}\alpha_2^- \qquad f^+ : \alpha_1^+ \to {}^{\perp}\alpha_2^+}{\Phi_{\mathrm{a}.\alpha}(f^-, f^+) : \Phi_{\mathrm{a}.\alpha}(\alpha_2^-, \alpha_1^+) \to {}^{\perp}\Phi_{\mathrm{a}.\alpha}(\alpha_1^-, \alpha_2^+)}$$

$$\frac{f^- : \alpha_1^- \to {}^{\perp}\alpha_2^- \qquad f^+ : \alpha_1^+ \to {}^{\perp}\alpha_2^+}{\Psi_{\mathrm{a}.\beta}(f^-, f^+) : \Psi_{\mathrm{a}.\beta}(\alpha_2^-, \alpha_1^+) \to \Psi_{\mathrm{a}.\beta}(\alpha_1^-, \alpha_2^+)}$$

as follows:

$$\begin{array}{rcl}
\Phi_{\mathrm{a}.\mathrm{a}}(f^-, f^+) & = & \lambda a.\, f^+\, a \\
\Phi_{\mathrm{a}.\iota}(f^-, f^+) & = & \lambda n.\, {}^{\perp}n \\
\Phi_{\mathrm{a}.1}(f^-, f^+) & = & \lambda u.\, {}^{\perp}\langle\rangle \\
\Phi_{\mathrm{a}.\alpha_1 \times \alpha_2}(f^-, f^+) & = & \lambda p.\, \mathbf{let}^{\perp}\ x_1 \Leftarrow \Phi_{\mathrm{a}.\alpha_1}(f^-, f^+)\, (\mathsf{fst}\, p) \\
 & & \qquad \mathbf{in}\ \mathbf{let}^{\perp}\ x_2 \Leftarrow \Phi_{\mathrm{a}.\alpha_2}(f^-, f^+)\, (\mathsf{snd}\, p)\ \mathbf{in}\ {}^{\perp}\langle x_1, x_2\rangle \\
\Phi_{\mathrm{a}.0}(f^-, f^+) & = & \lambda z.\, {}^{\perp}z \\
\Phi_{\mathrm{a}.\alpha_1 + \alpha_2}(f^-, f^+) & = & \lambda s.\, \mathsf{case}(s, x_1.\mathbf{let}^{\perp}\ y_1 \Leftarrow \Phi_{\mathrm{a}.\alpha_1}(f^-, f^+)x_1\ \mathbf{in}\ {}^{\perp}(\mathsf{inl}\, y_1), \\
 & & \qquad\qquad\quad x_2.\mathbf{let}^{\perp}\ y_2 \Leftarrow \Phi_{\mathrm{a}.\alpha_2}(f^-, f^+)x_2\ \mathbf{in}\ {}^{\perp}(\mathsf{inr}\, y_2)) \\
\Phi_{\mathrm{a}.\Sigma\aleph}(f^-, f^+) & = & \lambda s.\, \mathsf{case}(s, i.\, x_i.\, \mathbf{let}^{\perp}\ y_i \Leftarrow \Phi_{\mathrm{a}.\aleph(i)}(f^-, f^+)\, x_i\ \mathbf{in}\ {}^{\perp}(\mathsf{in}_i\, y_i)) \\
\Phi_{\mathrm{a}.\beta}(f^-, f^+) & = & \lambda b.\, {}^{\perp}(\Psi_{\mathrm{a}.\beta}(f^-, f^+)\, b) \\
 & & \\
\Psi_{\mathrm{a}.{}^{\circ}\alpha}(f^-, f^+) & = & \lambda m.\, \mathbf{let}^{\circ}\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^{\perp}\ y \Leftarrow \Phi_{\mathrm{a}.\alpha}(f^-, f^+)\, x\ \mathbf{in}\ {}^{\circ}y \\
\Psi_{\mathrm{a}.1}(f^-, f^+) & = & \lambda u.\, \langle\rangle \\
\Psi_{\mathrm{a}.\beta_1 \times \beta_2}(f^-, f^+) & = & \lambda p.\, \langle \Psi_{\mathrm{a}.\beta_1}(f^-, f^+)\, (\mathsf{fst}\, p), \Psi_{\mathrm{a}.\beta_2}(f^-, f^+)\, (\mathsf{snd}\, p)\rangle \\
\Psi_{\mathrm{a}.\alpha \to \beta}(f^-, f^+) & = & \lambda g.\, \lambda x.\, \mathbf{let}^{\perp}\ y \Leftarrow \Phi_{\mathrm{a}.\alpha}(f^+, f^-)\, x\ \mathbf{in}\ \Psi_{\mathrm{a}.\beta}(f^-, f^+)\, (g\, y)
\end{array}$$

A few simple properties of these definitions are:

Lemma A.2 *(1) When* a *does not occur free in* α *or* β *then*

$$\Phi_{\mathrm{a}.\alpha}(f^-, f^+) = \lambda a.\, {}^{\perp}a \qquad\qquad \Psi_{\mathrm{a}.\beta}(f^-, f^+) = \lambda b.\, b$$

More generally, (2) the type-directed actions are compositional:

$$\Phi_{\mathrm{a}.\alpha}(\Phi_{\mathrm{a}.\alpha'}(f^+, f^-), \Phi_{\mathrm{a}.\alpha'}(f^-, f^+)) = \Phi_{\mathrm{a}.\alpha\{\alpha'/\mathrm{a}\}}(f^-, f^+)$$

$$\Psi_{\mathrm{a}.\beta}(\Phi_{\mathrm{a}.\alpha'}(f^+, f^-), \Phi_{\mathrm{a}.\alpha'}(f^-, f^+)) = \Psi_{\mathrm{a}.\beta\{\alpha'/\mathrm{a}\}}(f^-, f^+)$$

Finally, (3) the definitions are functorial in the following sense:

$$\Phi_{\mathrm{a}.\alpha}(\lambda x.\, {}^{\perp}x, \lambda x.\, {}^{\perp}x) = \lambda a.\, {}^{\perp}a \qquad\qquad \Psi_{\mathrm{a}.\beta}(\lambda x.\, {}^{\perp}x, \lambda x.\, {}^{\perp}x) = \lambda b.\, b$$

$$\begin{aligned}&\Phi_{\mathrm{a}.\alpha}(\lambda x.\, \mathbf{let}^{\perp}\ y \Leftarrow f_1^- x \ \mathbf{in}\ f_2^- y, \lambda x.\, \mathbf{let}^{\perp}\ y \Leftarrow f_1^+ x \ \mathbf{in}\ f_2^+ y)\\ &\quad = \lambda a.\, \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(f_2^-, f_1^+)\, a \ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(f_1^-, f_2^+)\, r\end{aligned}$$

$$\begin{aligned}&\Psi_{\mathrm{a}.\beta}(\lambda x.\, \mathbf{let}^{\perp}\ y \Leftarrow f_1^- x \ \mathbf{in}\ f_2^- y, \lambda x.\, \mathbf{let}^{\perp}\ y \Leftarrow f_1^+ x \ \mathbf{in}\ f_2^+ y)\\ &\quad = \lambda b.\, \Psi_{\mathrm{a}.\beta}(f_1^-, f_2^+)\, (\Psi_{\mathrm{a}.\beta}(f_2^-, f_1^+)\, b)\end{aligned}$$

(i.e., $\Phi_{\mathrm{a}.\alpha}(\text{-},\text{-})$ *is an endofunctor in the Kleisli category of the lifting monad, while* $\Psi_{\mathrm{a}.\beta}(\text{-},\text{-})$ *is a functor from the Kleisli category to the underlying one).*

Proof. Simple induction on α and β in all cases. Note, however, that verification of the value-product case of (3) relies on partiality being a commutative effect. ■

We can also define an "ordinary" functorial operation on functions between lifted types:

Definition A.3 *When* g^- *and* g^+ *are strict functions (i.e., rigid with respect to* ${}^{\perp}$*-effects), we define the term constructor*

$$\frac{g^- : {}^{\perp}\alpha_1^- \to {}^{\perp}\alpha_2^- \qquad g^+ : {}^{\perp}\alpha_1^+ \to {}^{\perp}\alpha_2^+}{\Phi^d_{\mathrm{a}.\alpha}(g^-, g^+) : {}^{\perp}\Phi_{\mathrm{a}.\alpha}(\alpha_2^-, \alpha_1^+) \to {}^{\perp}\Phi_{\mathrm{a}.\alpha}(\alpha_1^-, \alpha_2^+)}$$

by

$$\Phi^d_{\mathrm{a}.\alpha}(g^-, g^+) = \lambda m.\, \mathbf{let}^{\perp}\ x \Leftarrow m \ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda x.\, g^-\, ({}^{\perp}x), \lambda x.\, g^+\, ({}^{\perp}x))\, x$$

We then have:

Lemma A.4 $\Phi^d_{\mathrm{a}.\alpha}(\text{-},\text{-})$ *is functorial in the following sense:*

$$\Phi^d_{\mathrm{a}.\alpha}(\mathsf{id}, \mathsf{id}) = \mathsf{id}$$

$$\Phi^d_{\mathrm{a}.\alpha}(g_2^- \circ g_1^-, g_2^+ \circ g_1^+) = \Phi^d_{\mathrm{a}.\alpha}(g_1^-, g_2^+) \circ \Phi^d_{\mathrm{a}.\alpha}(g_2^-, g_1^+)$$

Proof. Simple verification, using Lemma A.2(3):

$$\begin{aligned}&\Phi^d_{\mathrm{a}.\alpha}(\mathsf{id}, \mathsf{id}) = \lambda m.\, \mathbf{let}^{\perp}\ x \Leftarrow m \ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda x.\, {}^{\perp}x, \lambda x.\, {}^{\perp}x)\, x = \lambda m.\, \mathbf{let}^{\perp}\ x \Leftarrow m \ \mathbf{in}\ (\lambda a.\, {}^{\perp}a)\, x\\ &\quad = \lambda m.\, \mathbf{let}^{\perp}\ x \Leftarrow m \ \mathbf{in}\ {}^{\perp}x = \lambda m. m = \mathsf{id}\end{aligned}$$

$$\begin{aligned}
\Phi^d_{\mathrm{a}.\alpha}(g_2^- \circ g_1^-, g_2^+ \circ g_1^+) &= \lambda m.\,\mathbf{let}^\perp\ x \Leftarrow m\ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda a.\, g_2^-\,(g_1^-\,({}^\perp a)), \lambda a.\, g_2^+\,(g_1^+\,({}^\perp a)))\,x \\
&=^\dagger \lambda m.\,\mathbf{let}^\perp\ x \Leftarrow m \\
&\qquad \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda a.\,\mathbf{let}^\perp\ y \Leftarrow g_1^-\,({}^\perp a)\ \mathbf{in}\ g_2^-\,({}^\perp y), \lambda a.\,\mathbf{let}^\perp\ y \Leftarrow g_1^+\,({}^\perp a)\ \mathbf{in}\ g_2^+\,({}^\perp y))\,x \\
&= \lambda m.\,\mathbf{let}^\perp\ x \Leftarrow m \\
&\qquad \mathbf{in}\ \mathbf{let}^\perp\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(\lambda y.\, g_2^-\,({}^\perp y), \lambda a.\, g_1^+\,({}^\perp a))\,x\ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda a.\, g_1^-\,({}^\perp a), \lambda y.\, g_2^+\,({}^\perp y))\,r \\
&= \lambda m.\,\mathbf{let}^\perp\ r \Leftarrow (\mathbf{let}^\perp\ x \Leftarrow m\ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda y.\, g_2^-\,({}^\perp y), \lambda a.\, g_1^+\,({}^\perp a))\,x) \\
&\qquad \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda a.\, g_1^-\,({}^\perp a), \lambda y.\, g_2^+\,({}^\perp y))\,r \\
&= \lambda m.\,\Phi^d_{\mathrm{a}.\alpha}(g_1^-, g_2^+)\,(\Phi^d_{\mathrm{a}.\alpha}(g_2^-, g_1^+)\,m) = \Phi^d_{\mathrm{a}.\alpha}(g_1^-, g_2^+) \circ \Phi^d_{\mathrm{a}.\alpha}(g_2^-, g_1^+)
\end{aligned}$$

where † uses that for a strict g,

$$g\,m = g\,(\mathbf{let}^\perp\ x \Leftarrow m\ \mathbf{in}\ {}^\perp x) = \mathbf{let}^\perp\ x \Leftarrow m\ \mathbf{in}\ g\,({}^\perp x)$$

■

Let us also recall some elementary properties of least fixed points:

Lemma A.5 *Let* $\mathsf{fix}_B : (B \to B) \to B$ *denote the least-fixed-point operator for a pointed cpo* B*. Then*

1. *For any continuous* $f : B \to B$ *and* $g : B' \to B'$*, and* strict *continuous* $h : B' \to B$ *with* $f \circ h = h \circ g$*,* $\mathsf{fix}_B(f) = h(\mathsf{fix}_{B'}(g))$*.*
2. *For any continuous* $f : B \to B'$ *and* $g : B' \to B$*,* $\mathsf{fix}_B(g \circ f) = g(\mathsf{fix}_{B'}(f \circ g))$*.*

Proof.

1. Follows directly from the definition of fix:

$$\begin{aligned}
h(\mathsf{fix}_{B'}(g)) &= h(\textstyle\bigsqcup_i g^i(\bot_{B'})) = \bigsqcup_i h(g^i(\bot_{B'})) = \bigsqcup_i f^i(h(\bot_{B'})) = \bigsqcup_i f^i(\bot_B) \\
&= \mathsf{fix}_B(f)
\end{aligned}$$

2. (We cannot simply use the above result here, because g is not necessarily strict.) Let $x = \mathsf{fix}_{B'}(f \circ g)$ and $y = \mathsf{fix}_B(g \circ f)$. First, since $g(x)$ is *a* fixed point of $g \circ f$ (because $(g \circ f)(g(x)) = g((f \circ g)(x)) = g(x)$), we have $y \sqsubseteq g(x)$. Analogously, since $f(y)$ is a fixed point of $f \circ g$, $x \sqsubseteq f(y)$, and hence by monotonicity of g, $g(x) \sqsubseteq g(f(y)) = y$. And thus, since $\sqsubseteq$ is a partial order, we get $g(x) = y$.

■

Although for the purposes of Chapter 3, all we need is *a* solution to the type equation (not necessarily the least one), for Chapter 4 we will also need that the relevant isomorphism satisfies an additional equational property:

Definition A.6 *Let* $\mathbf{Cpo}_\perp$ *be the category of pointed cpos (domains) and strict continuous functions. Let* $F : \mathbf{Cpo}_\perp^{op} \times \mathbf{Cpo}_\perp \to \mathbf{Cpo}_\perp$ *be a functor (we call such an* F *a* mixed *functor in* $\mathbf{Cpo}_\perp$*); it is* locally continuous *if its action on morphisms is continuous. A* minimal invariant *for* F *is an object* X *together with an isomorphism* $i : F(X, X) \to X$ *such that*

$$\mathsf{fix}_{X \to X}\,(\underline{\lambda} h.\, i \circ F(h, h) \circ i^{-1}) = \mathsf{id}_X$$

One can show that the standard inverse-limit construction for solving recursive domain equation actually yields minimal invariants:

Theorem A.7 *Every locally continuous mixed functor in* **Cpo**$_\perp$ *has a minimal invariant.*

Proof. See [Pit99]. ■

Using this, we get for our predomain language:

Corollary A.8 *Every recursive type equation in* L_1 *has a solution in the predomain model, i.e., for any parameterized type* $\vdash_{\{\mathrm{a}\}} \alpha\ \mathsf{type}$*, there exists a cpo* A *with an isomorphism* $i : \mathcal{L}\llbracket\alpha\rrbracket^{\mathrm{a}\mapsto A} \to A$*. Moreover, interpreting* $\mu\mathrm{a}.\,\alpha$ *as* A*,* $\mathsf{roll}_{\mathrm{a}.\alpha}$ *as* i*, and* $\mathsf{unroll}_{\mathrm{a}.\alpha}$ *as* i^{-1}*, the following equation is satisfied in the model:*

$$\mathsf{fix}_{(\mu\mathrm{a}.\alpha)\to^{\perp}\mu\mathrm{a}.\alpha}\,(\lambda f.\,\lambda a.\,\mathbf{let}^{\perp}\ x \Leftarrow \Phi_{\mathrm{a}.\alpha}(f,f)\,(\mathsf{unroll}_{\mathrm{a}.\alpha}\,a)\ \mathbf{in}\ {}^{\perp}(\mathsf{roll}_{\mathrm{a}.\alpha}\,x)) = \lambda a.\,{}^{\perp}a$$

Proof. Every pointed cpo is isomorphic to a lifted cpo. So we can use Theorem A.7 with the functor given by

$$\begin{aligned}
&F(A^-_\perp, A^+_\perp) = (\mathcal{L}\llbracket\Phi_{\mathrm{a}.\alpha}(\mathrm{a}^-,\mathrm{a}^+)\rrbracket^{\mathrm{a}^-\mapsto A^-,\mathrm{a}^+\mapsto A^+})_\perp,\\
&F(g^- : A^-_{1\,\perp} \to A^-_{2\,\perp}, g^+ : A^+_{1\,\perp} \to A^+_{2\,\perp})\\
&\quad = \mathcal{L}\llbracket\Phi^d_{\mathrm{a}.\alpha}(x^-,x^+)\rrbracket^{\mathrm{a}^-_1\mapsto A^-_1,\ldots,\mathrm{a}^+_2\mapsto A^+_2}(\bullet[x^-\mapsto g^-, x^+\mapsto g^+])
\end{aligned}$$

to obtain a pointed cpo $A_\perp$ with an isomorphism $j : (\mathcal{L}\llbracket\alpha\rrbracket^{\mathrm{a}\mapsto A})_\perp \to A_\perp$. Moreover, since j is an isomorphism, it both preserves and reflects $\perp$, and must hence be expressible as $j = i_\perp = \lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ {}^{\perp}(i\,x)$ for some isomorphism $i : \mathcal{L}\llbracket\alpha\rrbracket^{\mathrm{a}\mapsto A} \to A$.

Further, we get the minimal invariant property for i from the minimal invariant property of j wrt. $\Phi^d_{\mathrm{a}.\alpha}(\text{-},\text{-})$, using Lemma A.5(2) to rearrange the fix body:

$$\begin{aligned}
&\mathsf{fix}\,(\lambda f.\,\lambda x.\,\mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(f,f)\,(\psi x)\ \mathbf{in}\ {}^{\perp}(\phi r))\\
&\quad = \mathsf{fix}\,((\lambda g.\,\lambda x.\,g\,({}^{\perp}x)) \circ (\lambda f.\,\lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(f,f)\,(\psi x)\ \mathbf{in}\ {}^{\perp}(\phi r)))\\
&\quad = (\lambda g.\,\lambda x.\,g\,({}^{\perp}x))\\
&\qquad\quad (\mathsf{fix}\,((\lambda f.\,\lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(f,f)\,(\psi x)\ \mathbf{in}\ {}^{\perp}(\phi r)) \circ (\lambda g.\,\lambda x.\,g\,({}^{\perp}x))))\\
&\quad = \lambda x.\,\mathsf{fix}\,(\lambda g.\,\lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.\alpha}(\lambda x.\,g\,({}^{\perp}x), \lambda x.\,g\,({}^{\perp}x))\,(\psi x)\ \mathbf{in}\ {}^{\perp}(\phi r))\,({}^{\perp}x)\\
&\quad = \lambda x.\,\mathsf{fix}\,(\lambda g.\,\lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ \mathbf{let}^{\perp}\ r \Leftarrow \Phi^d_{\mathrm{a}.\alpha}(g,g)\,({}^{\perp}(\psi x))\ \mathbf{in}\ {}^{\perp}(\phi r))\,({}^{\perp}x)\\
&\quad = \lambda x.\,\mathsf{fix}\,(\lambda g.\,\lambda m.\,(\lambda m.\,\mathbf{let}^{\perp}\ r \Leftarrow m\ \mathbf{in}\ {}^{\perp}(\phi r))\,(\Phi^d_{\mathrm{a}.\alpha}(g,g)\,(\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ {}^{\perp}(\psi x))))\,({}^{\perp}x)\\
&\quad = \lambda x.\,\mathsf{fix}\,(\lambda g.\,\lambda m.\,j\,(\Phi^d_{\mathrm{a}.\alpha}(g,g)\,(j^{-1}\,m)))\,({}^{\perp}x) = \lambda x.\,\mathsf{id}\,({}^{\perp}x) = \lambda x.\,{}^{\perp}x
\end{aligned}$$

■

A.2 Admissible relations

In this section we review the properties of (computation-)admissbile relations in the cpo semantics; in particular, we show admissibility of the key relation-forming constructs.

Let there be given a relational correspondence between predomain interpretations $\mathcal{L}$ of L and $\mathcal{L}'$ of L' (Definition 3.16). Most notably, admissibility is then preserved by formation of *inverse images* and *intersections* (e.g., [Pit99]):

Lemma A.9 *Recall that a (computation-)admissibile relations between closed types α and α' is a (pointed) chain-complete relation between cpos $\mathcal{L}[\![\alpha]\!]$ and $\mathcal{L}'[\![\alpha']\!]$. We then have:*

1. *When $R \in \mathrm{ARel}(\alpha, \alpha')$, $(x_1{:}\,\alpha_1, \ldots, x_n{:}\,\alpha_n) \vdash M : \alpha$ and $(x'_1{:}\,\alpha'_1, \ldots, x'_{n'}{:}\,\alpha'_{n'}) \vdash M' : \alpha'$ are terms of L and L' respectively, and for all $i \geq 2$, $\sigma x_i \in \mathrm{Val}_{\mathcal{L}}(\alpha_i)$ and $\sigma' x'_i \in \mathrm{Val}_{\mathcal{L}'}(\alpha'_i)$, the relation $R_1 \in \mathrm{Rel}(\alpha_1, \alpha'_1)$ given by*

   $$a_1 \; R_1 \; a'_1 \iff M^{(a_1/x_1, \sigma)} \; R \; M'^{(a'_1/x'_1, \sigma')}$$

   is admissible.

   Moreover, when α_1 and α'_1 are computation-types, R is computation-admissible, and the functions $\lambda x_1 .\, M^{\sigma}$ and $\lambda x'_1 .\, M'^{\sigma'}$ are rigid, then R_1 is also computation-admissible.

2. *When $(R_j)_{j \in J}$ is an arbitrary (not necessarily finite or even countable) family of admissible relations between α and α', the relation $\bigcap_{j \in J} R_j$ is admissible, where*

   $$a \; (\textstyle\bigcap_{j \in J} R_j) \; a' \iff \forall j \in J .\, a \; R_j \; a'$$

   Moreover, if each R_j is computation-admissible then so is $\bigcap_{j \in J} R_j$.

Proof. Both parts are fairly simple:

1. Define the continuous functions $f : \mathcal{L}[\![\alpha_1]\!] \to \mathcal{L}[\![\alpha]\!]$ and $f' : \mathcal{L}'[\![\alpha'_1]\!] \to \mathcal{L}'[\![\alpha']\!]$ by

   $$f = \underline{\lambda} a_1 .\, \mathcal{L}[\![M]\!](\bullet[x_1 \mapsto a_1, x_2 \mapsto \sigma x_2 \ldots, x_n \mapsto \sigma x_n])$$

   and

   $$f' = \underline{\lambda} a'_1 .\, \mathcal{L}'[\![M']\!](\bullet[x'_1 \mapsto a'_1, x'_2 \mapsto \sigma' x'_2, \ldots, x_{n'} \mapsto \sigma' x'_{n'}])$$

   Then $a_1 \; R_1 \; a'_1$ iff $f(a_1) \; R \; f'(a'_1)$.

   Now let $(a_{1i})_{i \in \omega}$ and $(a'_{1i})_{i \in \omega}$ be chains in $\mathcal{L}[\![\alpha_1]\!]$ and $\mathcal{L}'[\![\alpha'_1]\!]$, respectively, such that for all $i \in \omega$, $a_{1i} \; R_1 \; a'_{1i}$, i.e., $f(a_{1i}) \; R \; f'(a'_{1i})$. By monotonicity of f, the sequences $(f(a_{1i}))_{i \in \omega}$ and $(f'(a'_{1i}))_{i \in \omega}$ then form chains in $\mathcal{L}[\![\alpha]\!]$ and $\mathcal{L}'[\![\alpha']\!]$, and since R was assumed chain-complete, we have

   $$\textstyle\bigsqcup_i f(a_{1i}) \; R \; \bigsqcup_i f'(a'_{1i})$$

   By continuity of f and f', this is equivalent to

   $$f(\textstyle\bigsqcup_i a_{1i}) \; R \; f'(\bigsqcup_i a'_{1i})$$

   which says that $\bigsqcup_i a_{1i} \; R_1 \; \bigsqcup a'_{1i}$, meaning that R_1 is chain-complete.

   Further, to show R_1 pointed, we must show $f(\bot) \; R \; f'(\bot)$, which follows from pointedness of R and the fact (Proposition 2.13) that a rigid function is strict.

2. Let R_j be a family of chain-complete relations between cpos $\mathcal{L}[\![\alpha]\!]$ and $\mathcal{L}'[\![\alpha']\!]$, and let $(a_i)_{i\in\omega}$ and $(a'_i)_{i\in\omega}$ be chains componentwise related by the intersection R of all the R_j. That is,

$$\forall i \in \omega.\, \forall j \in J.\, a_i \; R_j \; a'_i$$

Exchanging the two universal quantifiers, we get

$$\forall j \in J.\, \forall i \in \omega.\, a_i \; R_j \; a'_i$$

Now, since each R_j was assumed chain-complete, this implies

$$\forall j \in J.\, \bigsqcup_i a_i \; R_j \; \bigsqcup_i a'_i$$

And thus, $\bigsqcup_i a_i \; (\bigcap_{j\in J} R_j) \; \bigsqcup_i a'_i$ as required to show chain-completeness of R.

Similarly, if each R_j relates $\bot_{\mathcal{L}[\![\beta]\!]}$ and $\bot_{\mathcal{L}'[\![\beta']\!]}$, then so must their intersection, and thus $\bigcap_{j\in J} R_j$ is pointed.

■

We can also show admissibility of the standard relational actions of the type constructors (where it does not already follow directly from the previous lemma):

Lemma A.10 *If the R's are admissible relations, then so are (1)* ι^{r} *and (2)* $R_1 +^{\mathrm{r}} R_2$ *(defined as in Lemma 3.18), and (3) the* $\Sigma^{\mathrm{r}}_i R_i$*, given by*

$$s \; (\Sigma^{\mathrm{r}}_i R_i) \; s' \iff \exists i \in I.\, \exists a_i \; R_i \; a'_i.\, s = \mathsf{in}_i \, a_i \wedge s' = \mathsf{in}_i \, a'_i$$

Moreover, if relation extension is taken as relation-lifting (from Proposition 3.21), then (4) ${}^{\circ}R$ *is computation-admissible.*

Proof. The relations determined by the definitions in each case can be written as:

$$\begin{aligned}
\iota^{\mathrm{r}} &= \{(n,n) \mid n \in \mathbf{N}\}\\
R_1 +^{\mathrm{r}} R_2 &= \{((1,a_1),(1,a'_1)) \mid (a_1,a'_1) \in R_1\} \cup \{((2,a_2),(2,a'_2)) \mid (a_2,a'_2) \in R_2\}\\
\Sigma^{\mathrm{r}}_i R_i &= \{((i,a_i),(i,a'_i)) \mid i \in I, (a_i,a'_i) \in R_i\}\\
{}^{\circ}R &= \{(\mathsf{up}(a),\mathsf{up}(a')) \mid (a,a') \in R\} \cup \{(\bot,\bot)\}
\end{aligned}$$

We then check each case:

1. Case ι^{r}. Since the cpo $\mathbf{N}$ of natural numbers was discretely ordered, any chain in $\mathbf{N}$ must be constant, so least upper bounds of componentwise related chains are obviously also related.

2. Case $R_1 +^{\mathrm{r}} R_2$. A chain in $A_1 + A_2$ must lie entirely within one of the injects. Assume wlog. that it is of the form $((1,a_i))_{i\in\omega}$ for some chain $(a_i)_{i\in\omega}$ in A_1. Analogously, the related chain must be of the form $((1,a'_i))_{i\in\omega}$, with $a_i \; R_1 \; a'_i$. By assumption on R_1, $\bigsqcup_i a_i \; R_1 \; \bigsqcup_i a'_i$. The result then follows by observing that

$$\bigsqcup_i (1,a_i) = (1, \bigsqcup_i a_i) \; (R_1 + R_2) \; (1, \bigsqcup_i a'_i) = \bigsqcup_i (1,a'_i)$$

3. Case $\Sigma^{\mathrm{r}}_i R_i$ is analogous to $+^{\mathrm{r}}$, only with the set of tags taken as I instead of $\{1, 2\}$.

4. Case ${}^\circ R$. Let $(m_i)_{i\in\omega}$ and $(m'_i)_{i\in\omega}$ be chains such that $m_i\ ({}^\circ R)\ m'_i$. There are then two possibilities. It could be that for all i, $m_i = \bot_{A_\bot}$ and $m'_i = \bot_{A'_\bot}$. In this case,

   $$\textstyle\bigsqcup_i m_i = \bot_{A_\bot}\ ({}^\circ R)\ \bot_{A'_\bot} = \bigsqcup_i m'_i$$

   by the second disjunct of the definition, and we are done. Or there exists an $i_0 \geq 0$, such that for all $i \geq i_0$, $m_i = \mathsf{up}(a_i)$ and $m'_i = \mathsf{up}(a'_i)$ for some $a_i\ R\ a'_i$. By definition of the ordering in $A_\bot$, the a_i form a chain (if $\mathsf{up}(a_i) \sqsubseteq \mathsf{up}(a_{i+1})$ then $a_i \sqsubseteq a_{i+1}$). Analogously for the a'_i. Because R was assumed chain-complete, we have $\bigsqcup_{i\geq i_0} a_i\ R\ \bigsqcup_{i\geq i_0} a'_i$. And since the initial segment of $\bot$s in a chain does not affect its least upper bound, we get

   $$\textstyle\bigsqcup_i m_i = \bigsqcup_{i\geq i_0} \mathsf{up}(a_i) = \mathsf{up}(\bigsqcup_{i\geq i_0} a_i)\ ({}^\circ R)\ \mathsf{up}(\bigsqcup_{i\geq i_0} a'_i) = \bigsqcup_{i\geq i_0} \mathsf{up}(a'_i) = \bigsqcup_i m'_i$$

   Also, directly by the second component of the definition, ${}^\circ R$ is pointed.

■

And finally, we can verify our fixed-point induction principle:

Lemma A.11 *Let S be a computation-admissible relation between β and β'. Let $f \in \mathrm{Val}_{\mathcal{L}}(\beta \to \beta)$ and $f' \in \mathrm{Val}_{\mathcal{L}'}(\beta' \to \beta')$ be such that $\forall b\ S\ b'.\, f\, b\ S\ f'\, b'$. Then $\mathsf{fix}_\beta\, f\ S\ \mathsf{fix}_{\beta'}\, f'$.*

Proof. We have $\mathcal{L}[\![\mathsf{fix}_\beta\, x]\!](\bullet[x \mapsto f]) = \bigsqcup_i f^i(\bot_{\mathcal{L}[\![\beta]\!]})$, and analogously for f'. Since in particular S is admissible, i.e., chain-complete in the model, it suffices to show that for all $i \geq 0$,

$$f^i(\bot_{\mathcal{L}[\![\beta]\!]})\ S\ f'^i(\bot_{\mathcal{L}[\![\beta']\!]})$$

This follows by a simple induction on i. For $i = 0$, we get the result from pointedness (*computation*-admissibility) of S. And for the inductive step, we use that if $f^i(\bot)\ S\ f'^i(\bot)$ then by assumption on f and f',

$$f^{i+1}(\bot) = f(f^i(\bot))\ S\ f'(f'^i(\bot)) = f'^{i+1}(\bot)$$

■

A.3 Isomorphisms of recursive types

The definition of $\Phi_{\mathrm{a}.\alpha}(f^-, f^+)$ allows it to act on partial functions (i.e., total functions into a lifted cpo); we need this generality for solving recursive type equations, because the approximants will not be total. But when the type-directed functor acts on known *isomorphisms* between cpos (not necessarily pointed), a simpler definition is possible:

Definition A.12 *When φ is a term constructor denoting an isomorphism (e.g.,* roll *or* unroll*), we define term constructors $\Phi^i_{\mathrm{a}.\alpha}(\varphi)$ and $\Psi^i_{\mathrm{a}.\beta}(\varphi)$ with types:*

$$\frac{\varphi : \alpha_1 \overset{\sim}{\to} \alpha_2}{\Phi^i_{\mathrm{a}.\alpha}(\varphi) : \alpha\{\alpha_1/\mathrm{a}\} \overset{\sim}{\to} \alpha\{\alpha_2/\mathrm{a}\}} \quad \textit{and} \quad \frac{\varphi : \alpha_1 \overset{\sim}{\to} \alpha_2}{\Psi^i_{\mathrm{a}.\beta}(\varphi) : \beta\{\alpha_1/\mathrm{a}\} \overset{\sim}{\to} \beta\{\alpha_2/\mathrm{a}\}}$$

as follows:

$$\begin{array}{rcl}
\Phi^i_{\mathrm{a.a}}(\varphi)\,a & = & \varphi\,a \\
\Phi^i_{\mathrm{a.}\iota}(\varphi)\,n & = & n \\
\Phi^i_{\mathrm{a.1}}(\varphi)\,u & = & \langle\rangle \\
\Phi^i_{\mathrm{a.}\alpha_1\times\alpha_2}(\varphi)\,p & = & \langle\Phi^i_{\mathrm{a.}\alpha_1}(\varphi)\,(\mathsf{fst}\,p),\Phi^i_{\mathrm{a.}\alpha_2}(\varphi)\,(\mathsf{snd}\,p)\rangle \\
\Phi^i_{\mathrm{a.0}}(\varphi)\,z & = & z \\
\Phi^i_{\mathrm{a.}\alpha_1+\alpha_2}(\varphi)\,s & = & \mathsf{case}\,(s, a_1.\,\mathsf{inl}\,(\Phi^i_{\mathrm{a.}\alpha_1}(\varphi)\,a_1), a_2.\,\mathsf{inl}\,(\Phi^i_{\mathrm{a.}\alpha_2}(\varphi)\,a_2)) \\
\Phi^i_{\mathrm{a.}\Sigma\aleph}(\varphi)\,s & = & \mathsf{case}(s, i.\,a_i.\,\mathsf{in}_i\,(\Phi^i_{\mathrm{a.}\aleph(i)}(\varphi)\,a_i)) \\
\Phi^i_{\mathrm{a.}\beta}(\varphi)\,b & = & \Psi^i_{\mathrm{a.}\beta}(\varphi)\,b
\end{array}$$

$$\begin{array}{rcl}
\Psi^i_{\mathrm{a.}^\circ\alpha}(\varphi)\,m & = & \mathbf{let}^\circ\;x \Leftarrow m\;\mathbf{in}\;{}^\circ(\Phi^i_{\mathrm{a.}\alpha}(\varphi)\,x) \\
\Psi^i_{\mathrm{a.1}}(\varphi)\,o & = & \langle\rangle \\
\Psi^i_{\mathrm{a.}\beta_1\times\beta_2}(\varphi)\,p & = & \langle\Psi^i_{\mathrm{a.}\beta_1}(\varphi)\,(\mathsf{fst}\,p),\Psi^i_{\mathrm{a.}\beta_2}(\varphi)\,(\mathsf{snd}\,p)\rangle \\
\Psi^i_{\mathrm{a.}\alpha\to\beta}(\varphi)\,g & = & \lambda x.\,\Psi^i_{\mathrm{a.}\beta}(\varphi)\,(g\,(\Phi^i_{\mathrm{a.}\alpha}(\varphi^{-1})\,x))
\end{array}$$

Lemma A.13 *The functorial actions on isomorphisms are related to their general counterparts as follows:*

$$\begin{array}{rcl}
{}^\perp(\Phi^i_{\mathrm{a.}\alpha}(\varphi)\,a) & = & \Phi_{\mathrm{a.}\alpha}(\lambda x.\,{}^\perp(\varphi^{-1}\,x), \lambda y.\,{}^\perp(\varphi\,y))\,a \\
\Psi^i_{\mathrm{a.}\beta}(\varphi)\,b & = & \Psi_{\mathrm{a.}\beta}(\lambda x.\,{}^\perp(\varphi^{-1}\,x), \lambda y.\,{}^\perp(\varphi\,y))\,b
\end{array}$$

Proof. Straightforward induction on α and β. ■

Note also that we have $\Phi^i_{\mathrm{a.}\alpha}(\Phi^i_{\mathrm{a}'.\alpha'}(\varphi)) = \Phi^i_{\mathrm{a}'.\alpha\{\alpha'/\mathrm{a}\}}(\varphi)$.

Lemma A.14 *Let F and G be type constructors of L_1 (not necessarily covariant), and let $\alpha = \mu\mathrm{a}.\,F(G\mathrm{a})$ and $\alpha' = \mu\mathrm{a}'.\,G(F\mathrm{a}')$ be the solutions to the corresponding recursive type equations. Then in the predomain model, there exists an isomorphism $\chi : G\alpha \overset{\sim}{\rightarrow} \alpha'$, which further satisfies the following two (equivalent)* coherence equations*:*

$$x\colon G\alpha \vdash \mathsf{roll}_{\mathrm{a}'.G(F\mathrm{a}')}\,(\Phi^i_{\mathrm{a}'.G(F\mathrm{a}')}(\chi)\,(\Phi^i_{\mathrm{a}.G\mathrm{a}}(\mathsf{unroll}_{\mathrm{a}.F(G\mathrm{a})})\,x)) = \chi\,x : \alpha'$$

$$y\colon \alpha' \vdash \Phi^i_{\mathrm{a}.G\mathrm{a}}(\mathsf{roll}_{\mathrm{a}.F(G\mathrm{a})})\,(\Phi^i_{\mathrm{a}'.G(F\mathrm{a}')}(\chi^{-1})\,(\mathsf{unroll}_{\mathrm{a}'.G(F\mathrm{a}')}\,y)) = \chi^{-1}\,y : G\alpha$$

Proof. When $\varphi : \alpha_1 \overset{\sim}{\rightarrow} \alpha_2$ is a term constructor, we define the function

$$\varphi^\sharp : {}^\perp\alpha_1 \to {}^\perp\alpha_2 = \lambda m^{{}^\perp\alpha_1}.\,\mathbf{let}^\perp\;x \Leftarrow m\;\mathbf{in}\;{}^\perp(\varphi\,x)$$

For terms, take $F(f^-, f^+) = \Phi^d_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)$ and also write $\hat{F}(f)$ for $F(f, f)$. Analogously for G. Further, we define the usual abbreviations $\phi = \mathsf{roll}_{\mathrm{a}.F(G\mathrm{a})}$ and $\phi' = \mathsf{roll}_{\mathrm{a}'.G(F\mathrm{a}')}$, with ψ and ψ' for the inverses.

Now let $i = \phi^{\sharp} : {}^{\perp}F(G\alpha) \rightarrow {}^{\perp}\alpha$ and $j = \phi'^{\sharp} : {}^{\perp}G(F\alpha') \rightarrow {}^{\perp}\alpha'$ be the minimal invariants for the corresponding functors. We first show that there exists an isomorphism $l : {}^{\perp}G(\alpha) \rightarrow {}^{\perp}\alpha'$. Take

$$\begin{aligned}(l', l) = \mathsf{fix}\,(\lambda(h^{{}^{\perp}\alpha' \rightarrow {}^{\perp}G\alpha}, k^{{}^{\perp}G\alpha \rightarrow {}^{\perp}\alpha'}).\\ (G(i^{-1}, i) \circ G(F(h,k), F(k,h)) \circ j^{-1},\ j \circ G(F(k,h), F(h,k)) \circ G(i, i^{-1})))\end{aligned}$$

We want to show that l' is actually the two-sided inverse of l. Accordingly, consider the strict function $c = \lambda(h,k).\, k \circ h$. Lemma A.4 then gives us:

$$\begin{aligned}
&c \circ (\lambda(h,k).\,(G(i^{-1},i) \circ G(F(h,k),F(k,h)) \circ j^{-1}, j \circ G(F(k,h),F(h,k)) \circ G(i,i^{-1})))\\
&\quad = \lambda(h,k).\, j \circ G(F(k,h),F(h,k)) \circ G(i,i^{-1}) \circ G(i^{-1},i) \circ G(F(h,k),F(k,h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\, j \circ G(F(k,h),F(h,k)) \circ G(i^{-1} \circ i, i^{-1} \circ i)) \circ G(F(h,k),F(k,h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\, j \circ G(F(k,h),F(h,k)) \circ G(\mathsf{id}_{{}^{\perp}FG\alpha}, \mathsf{id}_{{}^{\perp}FG\alpha}) \circ G(F(h,k),F(k,h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\, j \circ G(F(k,h),F(h,k)) \circ \mathsf{id}_{{}^{\perp}GFG\alpha} \circ G(F(h,k),F(k,h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\, j \circ G(F(h,k) \circ F(k,h), F(h,k) \circ F(k,h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\, j \circ G(F(k \circ h, k \circ h), F(k \circ h, k \circ h)) \circ j^{-1} = \lambda(h,k).\, j \circ \hat{G}(\hat{F}(k \circ h)) \circ j^{-1}\\
&\quad = \lambda(h,k).\,(\lambda f.\, j \circ \hat{G}(\hat{F}(f)) \circ j^{-1})\,(k \circ h) = (\lambda f.\, j \circ \hat{G}(\hat{F}(f)) \circ j^{-1}) \circ c
\end{aligned}$$

From Lemma A.2(2) and a let-simplification, we obtain that

$$\Phi^{d}_{\mathrm{a}.G\mathrm{a}}(\Phi^{d}_{\mathrm{a}.F\mathrm{a}}(f,f), \Phi^{d}_{\mathrm{a}.F\mathrm{a}}(f,f)) = \Phi^{d}_{\mathrm{a}.G(F\mathrm{a})}(f,f)$$

so by Lemma A.5(1) and the fact that j is a minimal invariant for GF, we get

$$l' \circ l = c\,(\mathsf{fix}\,(\lambda(h,k).\cdots)) = \mathsf{fix}\,(\lambda f.\, j \circ \hat{G}(\hat{F}(f)) \circ j^{-1}) = \mathsf{id}_{{}^{\perp}\alpha'} = \mathsf{id}^{\sharp}_{\alpha'}$$

In the other direction, taking $c' = \lambda(h,k).\, h \circ k$, we similarly get:

$$\begin{aligned}
&c' \circ (\lambda(h,k).\,(G(i^{-1},i) \circ G(F(h,k),F(k,h)) \circ j^{-1}, j \circ G(F(k,h),F(h,k)) \circ G(i,i^{-1})))\\
&\quad = \lambda(h,k).\, G(i^{-1},i) \circ G(F(h,k),F(k,h)) \circ j^{-1} \circ j \circ G(F(k,h),F(h,k)) \circ G(i,i^{-1})\\
&\quad = \lambda(h,k).\, G(i^{-1},i) \circ G(F(h,k),F(k,h)) \circ G(F(k,h),F(h,k)) \circ G(i,i^{-1})\\
&\quad = \lambda(h,k).\, G(i^{-1},i) \circ G(F(h \circ k, h \circ k), F(h \circ k, h \circ k)) \circ G(i,i^{-1})\\
&\quad = (\lambda g.\, G(i^{-1},i) \circ G(F(g,g),F(g,g)) \circ G(i,i^{-1})) \circ c'\\
&\quad = (\lambda g.\, G(i \circ F(g,g) \circ i^{-1}, i \circ F(g,g) \circ i^{-1})) \circ c' = (\lambda g.\, \hat{G}(i \circ \hat{F}(g) \circ i^{-1})) \circ c'
\end{aligned}$$

And thus we have, using both parts of Lemma A.5 and the minimal-invariant property of i:

$$\begin{aligned}
&l \circ l' = c'\,(\mathsf{fix}\,(\lambda(h,k).\cdots)) = \mathsf{fix}\,(\lambda g.\, \hat{G}(i \circ \hat{F}(g) \circ i^{-1})) = \hat{G}(\mathsf{fix}\,(\lambda f.\, i \circ \hat{F}(\hat{G}(f)) \circ i^{-1}))\\
&\quad = \hat{G}(\mathsf{id}_{{}^{\perp}\alpha}) = \mathsf{id}_{{}^{\perp}G\alpha} = \mathsf{id}^{\sharp}_{G\alpha}
\end{aligned}$$

We can thus take χ to be the unique isomorphism such that $\chi^{\sharp} = l$.

Further, knowing that l and l' are actually inverses, we get the second part of the result by unrolling their fixed-point definition once:

$$(l', l) = (G(i^{-1},i) \circ G(F(l',l),F(l,l')) \circ j^{-1}, j \circ G(F(l,l'),F(l',l)) \circ G(i,i^{-1}))$$

Now, take advantage of the following simple relationship between the functorial actions on an isomorphisms:

$$\begin{aligned}
&\Phi^{d}_{\mathrm{a}.\alpha}(\varphi^{-1\sharp}, \varphi^{\sharp}) = \Phi^{d}_{\mathrm{a}.\alpha}(\lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ {}^{\perp}(\varphi^{-1}x), \lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ {}^{\perp}(\varphi x))\\
&\quad = \lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda x.\,\mathbf{let}^{\perp}\ x \Leftarrow {}^{\perp}x\ \mathbf{in}\ {}^{\perp}(\varphi^{-1}x), \cdots\varphi\cdots)\,x\\
&\quad = \lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ \Phi_{\mathrm{a}.\alpha}(\lambda x.\,{}^{\perp}(\varphi^{-1}x), \lambda x.\,{}^{\perp}(\varphi x))\,x\\
&\quad = \lambda m.\,\mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ {}^{\perp}(\Phi^{i}_{\mathrm{a}.\alpha}(\varphi)\,x) = \Phi^{i}_{\mathrm{a}.\alpha}(\varphi)^{\sharp}
\end{aligned}$$

From this we get:

$$\begin{aligned}\chi^\sharp &= l = j \circ G(F(l,l'),F(l',l)) \circ G(i,i^{-1})\\ &= \phi'^\sharp \circ \Phi^d_{\mathrm{a}.G\mathrm{a}}(\Phi^d_{\mathrm{a}.F\mathrm{a}}(\chi^\sharp,\chi^{-1\sharp}),\Phi^d_{\mathrm{a}.F\mathrm{a}}(\chi^{-1\sharp},\chi^\sharp)) \circ \Phi^d_{\mathrm{a}.G\mathrm{a}}(\phi^\sharp,\psi^\sharp)\\ &= \phi'^\sharp \circ \Phi^d_{\mathrm{a}.G\mathrm{a}}(\Phi^i_{\mathrm{a}.F\mathrm{a}}(\chi^{-1})^\sharp,\Phi^i_{\mathrm{a}.F\mathrm{a}}(\chi)^\sharp) \circ \Phi^i_{\mathrm{a}.G\mathrm{a}}(\psi)^\sharp = \phi'^\sharp \circ \Phi^i_{\mathrm{a}.G\mathrm{a}}(\Phi^i_{\mathrm{a}.F\mathrm{a}}(\chi))^\sharp \circ \Phi^i_{\mathrm{a}.G\mathrm{a}}(\psi)^\sharp\end{aligned}$$

So, in particular,

$$\begin{aligned}{}^\perp(\phi'\,(\Phi^i_{\mathrm{a}.G(F\mathrm{a})}(\chi)\,(\Phi^i_{\mathrm{a}.G\mathrm{a}}(\psi)\,x))) &= (\phi'^\sharp \circ \Phi^i_{\mathrm{a}.G\mathrm{a}}(\Phi^i_{\mathrm{a}.F\mathrm{a}}(\chi))^\sharp \circ \Phi^i_{\mathrm{a}.G\mathrm{a}}(\psi)^\sharp)\,({}^\perp x) = \chi^\sharp\,({}^\perp x)\\ &= {}^\perp(\chi x)\,,\end{aligned}$$

and since lifting is injective, we get the first coherence equation. The second one is analogous. ■

A.4 Invariant relations over recursive types

We now want to show that certain principles for constructing relations over recursive types are valid; specifically, that a class of well-behaved relational actions allows us to solve "recursive relation equations". The following presents only the specific results we need for the proofs in Chapter 4; for a general treatment of the subject, see [Pit99].

Throughout this section, let us assume a fixed relational correspondence between predomain interpretations $\mathcal{L}$ of L and $\mathcal{L}'$ of L', with a computation-extension of relations. In keeping with the general convention in this appendix, we also write ${}^\perp R$ for relation-lifting. We first characterize a particularly well-behaved way of constructing admissible relations:

Definition A.15 *Let F and F' be type constructors. A (mixed) relational action $\mathcal{F}$ for F and F' assigns to every pair of relations $R^- \in \mathrm{ARel}(\alpha^-,\alpha'^-)$ and $R^+ \in \mathrm{ARel}(\alpha^+,\alpha'^+)$ a relation $\mathcal{F}(R^-,R^+) \in \mathrm{ARel}(\Phi_{\mathrm{a}.F\mathrm{a}}(\alpha^-,\alpha^+),\Phi_{\mathrm{a}.F'\mathrm{a}}(\alpha'^-,\alpha'^+))$. We say that this* action *is admissible if it satisfies:*

$$\begin{aligned}&(\forall x\ R_1^-\ x'.\, f^-x\ ({}^\perp R_2^-)\ f'^-x') \wedge (\forall x\ R_1^+\ x'.\, f^+x\ ({}^\perp R_2^+)\ f'^+x')\\ &\quad\Rightarrow \forall y\ \mathcal{F}(R_2^-,R_1^+)\ y'.\, \Phi_{\mathrm{a}.F\mathrm{a}}(f^-,f^+)\,y\ ({}^\perp\mathcal{F}(R_1^-,R_2^+))\ \Phi_{\mathrm{a}.F'\mathrm{a}}(f'^-,f'^+)\,y'\end{aligned}$$

Likewise, for computation-type constructors G and G', a relational action *is called computation-admissible if it maps $R^- \in \mathrm{ARel}(\alpha^-,\alpha'^-)$ and $R^+ \in \mathrm{ARel}(\alpha^+,\alpha'^+)$ to a relation $\mathcal{G}(R^-,R^+) \in \mathrm{CARel}(\Psi_{\mathrm{a}.G\mathrm{a}}(\alpha^-,\alpha^+),\Psi_{\mathrm{a}.G'\mathrm{a}}(\alpha'^-,\alpha'^+))$ such that*

$$\begin{aligned}&(\forall x\ R_1^-\ x'.\, f^-x\ ({}^\perp R_2^-)\ f'^-x') \wedge (\forall x\ R_1^+\ x'.\, f^+x\ ({}^\perp R_2^+)\ f'^+x')\\ &\quad\Rightarrow \forall y\ \mathcal{G}(R_2^-,R_1^+)\ y'.\, \Psi_{\mathrm{a}.G\mathrm{a}}(f^-,f^+)\,y\ \mathcal{G}(R_1^-,R_2^+)\ \Psi_{\mathrm{a}.G'\mathrm{a}}(f'^-,f'^+)\,y'\end{aligned}$$

It is easy to see from the definition of $\Phi_{\mathrm{a}.\beta}(f^-,f^+)$ that any computation-admissible action is also admissible. Moreover, we have a number of standard ways of constructing (computation-)admissible relational actions:

Lemma A.16 *The following relational actions are all admissible (and computation-admissible where noted):*

1. *$\mathcal{F}(R^-,R^+) = R^+$ for $F\mathrm{a} = F'\mathrm{a} = \mathrm{a}$.*

2. $\mathcal{F}(R^-,R^+) = R_0$ *for* $F\mathrm{a} = \alpha_0$ *and* $F'\mathrm{a} = \alpha'_0$*, where* α_0 *and* α'_0 *do not depend on* a*, and* $R_0 \in \mathrm{ARel}(\alpha_0, \alpha'_0)$ *is an arbitrary admissible relation.* $\mathcal{F}$ *is also computation-admissible if* R_0 *is.*

3. $\mathcal{F}(R^-,R^+) = \mathcal{F}_1(R^-,R^+) \times^{\mathrm{r}} \mathcal{F}_2(R^-,R^+)$ *for* $F\mathrm{a} = F_1\mathrm{a} \times F_2\mathrm{a}$ *and* $F'\mathrm{a} = F'_1\mathrm{a} \times F'_2\mathrm{a}$*, where* $\mathcal{F}_1$ *is admissible for* F_1 *and* F'_1*, and* $\mathcal{F}_2$ *is admissible for* F_2 *and* F'_2*.* $\mathcal{F}$ *is also computation-admissible if both* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *are.*

4. $\mathcal{F}(R^-,R^+) = \mathcal{F}_1(R^-,R^+) +^{\mathrm{r}} \mathcal{F}_2(R^-,R^+)$ *for* $F\mathrm{a} = F_1\mathrm{a} + F_2\mathrm{a}$ *and* $F'\mathrm{a} = F'_1\mathrm{a} + F'_2\mathrm{a}$*, where* $\mathcal{F}_1$ *is admissible for* F_1 *and* F'_1*, and* $\mathcal{F}_2$ *is admissible for* F_2 *and* F'_2*.*

5. $\mathcal{F}(R^-,R^+) = \Sigma^{\mathrm{r}}_i \mathcal{F}_i(R^-,R^+)$ *for* $F\mathrm{a} = \Sigma_i F_i\mathrm{a}$ *and* $F\mathrm{a} = \Sigma_i F'_i\mathrm{a}$*, where for every* $i \in I$*,* $\mathcal{F}_i$ *is admissible for* F_i *and* F'_i*.*

6. $\mathcal{G}(R^-,R^+) = \mathcal{F}(R^+,R^-) \rightarrow^{\mathrm{r}} \mathcal{G}_1(R^-,R^+)$ *for* $G\mathrm{a} = F\mathrm{a} \rightarrow G_1\mathrm{a}$ *and* $G'\mathrm{a} = F'\mathrm{a} \rightarrow G'_1\mathrm{a}$*, where* $\mathcal{F}$ *is admissible for* F *and* F'*, and* $\mathcal{G}_1$ *is computation-admissible for* G_1 *and* G'_1*.* $\mathcal{G}$ *is also computation-admissible.*

7. $\mathcal{G}(R^-,R^+) = {}^{\circ}\mathcal{F}(R^-,R^+)$ *for* $G\mathrm{a} = {}^{\circ}(F\mathrm{a})$ *and* $G'\mathrm{a} = {}^{\circ}(F'\mathrm{a})$*, where* $\mathcal{F}$ *is admissible for* F *and* F'*.* $\mathcal{G}$ *is also computation-admissible.*

8. $\mathcal{F}(R^-,R^+) = \bigcap_{j \in J} \mathcal{F}_j(R^-,R^+)$ *for any* F *and* F'*, when for every* $j \in J$*,* $\mathcal{F}_j$ *is admissible for* F *and* F'*.* $\mathcal{F}$ *is also computation-admissible if each* $\mathcal{F}_j$ *is.*

Proof. Simple verification in all cases. For example, and since it is somewhat non-standard, let us go through the details of (7), i.e., ${}^{\circ}\mathcal{F}(R^-,R^+)$. First, we note that for any computation-extension of relations we have that if $\forall a\ R_1\ a'.\, f\,a\ ({}^{\circ}R_2)\ f'\,a'$ then

$$\forall m\ ({}^{\perp}R_1)\ m'.\, \mathbf{let}^{\perp}\ x \Leftarrow m\ \mathbf{in}\ f\,x\ ({}^{\circ}R_2)\ \mathbf{let}^{\perp}\ x' \Leftarrow m'\ \mathbf{in}\ f'\,x'$$

(Note that this is simply condition (0) of a monad relation from Definition 3.26 in the case where the ambient effect is partiality, and ${}^{\circ}R_2$ is taken as the ${}^{\prime}R_2$ of the definition.) This holds because ${}^{\circ}R_2$ is by definition computation-admissible and hence pointed. Thus, since $m\ ({}^{\perp}R_1)\ m'$ means that either both m and m' are $\perp$, or both are liftings of R_1-related elements, we get the required relationship in either case.

We can now verify admissibility of the action $(R^-,R^+) \mapsto {}^{\circ}\mathcal{F}(R^-,R^+)$ for the type constructors $G\mathrm{a} = {}^{\circ}(F\mathrm{a})$ and $G'\mathrm{a} = {}^{\circ}(F'\mathrm{a})$ when $\mathcal{F}$ is an admissible action for F and F'. Let the Rs and fs be as in Definition A.15; we must then show:

$$\forall y\ ({}^{\circ}\mathcal{F}(R_2^-,R_1^+))\ y'.\, \Psi_{\mathrm{a}.{}^{\circ}F\mathrm{a}}(f^-,f^+)\,y\ ({}^{\circ}\mathcal{F}(R_1^-,R_2^+))\ \Psi_{\mathrm{a}.{}^{\circ}F'\mathrm{a}}(f'^-,f'^+)\,y'$$

So assume $y\ ({}^{\circ}\mathcal{F}(R_2^-,R_1^+))\ y'$. Expanding $\Psi_{\mathrm{a}.{}^{\circ}F\mathrm{a}}(f^-,f^+)$ according to Definition A.1, we must then establish that

$$\mathbf{let}^{\circ}\ x \Leftarrow y\ \mathbf{in}\ \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-,f^+)\,x\ \mathbf{in}\ {}^{\circ}r\ ({}^{\circ}\mathcal{F}(R_1^-,R_2^+))\ \mathbf{let}^{\circ}\ x' \Leftarrow y'\ \mathbf{in}\ \cdots$$

By property (2) of relation-extension and the assumption on y and y', it suffices to show that

$$\begin{array}{l} \forall x\ \mathcal{F}(R_2^-, R_1^+)\ x'. \\ \quad \mathbf{let}^{\perp}\ r \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)x\ \mathbf{in}\ {}^{\circ}r\ ({}^{\circ}\mathcal{F}(R_1^-, R_2^+))\ \mathbf{let}^{\perp}\ r' \Leftarrow \Phi_{\mathrm{a}.F'\mathrm{a}}(f'^-, f'^+)x'\ \mathbf{in}\ {}^{\circ}r' \end{array}$$

Let $x\ \mathcal{F}(R_2^-, R_1^+)\ x'$ be given. By assumption on $\mathcal{F}$, we then have that

$$\Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)x\ ({}^{\perp}\mathcal{F}(R_1^-, R_2^+))\ \Phi_{\mathrm{a}.F'\mathrm{a}}(f'^-, f'^+)x'$$

And hence, by the observation at the beginning of the proof, it suffices to show

$$\forall r\ \mathcal{F}(R_1^-, R_2^+)\ r'.\ {}^{\circ}r\ ({}^{\circ}\mathcal{F}(R_1^-, R_2^+))\ {}^{\circ}r'$$

which follows immediately from property (1) of relation extension. ■

We also have the following principle for constructing new computation-admissible actions from old ones:

Lemma A.17 *Let $\mathcal{G}_1$ be a computation-admissible relational action for G_1 and G_1', and let G and G' be another pair of computation-type constructors. Further let h be a rigid natural transformation from G to G_1, i.e., satisfying*

$$h\,(\Psi_{\mathrm{a}.G\mathrm{a}}(f^-, f^+)x) = \Psi_{\mathrm{a}.G_1\mathrm{a}}(f^-, f^+)\,(h\,x)$$

and analogously for h'. Then the relational action $\mathcal{G}$ for G and G' defined by

$$x\ \mathcal{G}(R^-, R^+)\ x' \iff h\,x\ \mathcal{G}_1(R^-, R^+)\ h'\,x'$$

is computation-admissible.

Proof. We first note that $\mathcal{G}(R^-, R^+)$ is a computation-admissible relation by assumption on $\mathcal{G}_1$ and rigidity of h and h'. Further, let the Rs and fs be as in Definition A.15; we must show that

$$\forall y\ \mathcal{G}(R_2^-, R_1^+)\ y'.\ \Psi_{\mathrm{a}.G\mathrm{a}}(f^-, f^+)y\ \mathcal{G}(R_1^-, R_2^+)\ \Psi_{\mathrm{a}.G'\mathrm{a}}(f'^-, f'^+)y'$$

I.e., that

$$\forall y, y'.\ h\,y\ \mathcal{G}_1(R_2^-, R_1^+)\ h'\,y' \Rightarrow h\,(\Psi_{\mathrm{a}.G\mathrm{a}}(f^-, f^+)y)\ \mathcal{G}_1(R_1^-, R_2^+)\ h'\,(\Psi_{\mathrm{a}.G'\mathrm{a}}(f'^-, f'^+)y')$$

Now, by assumption on h and h', this is equivalent to

$$\forall y, y'.\ h\,y\ \mathcal{G}_1(R_2^-, R_1^+)\ h'\,y' \Rightarrow \Psi_{\mathrm{a}.G_1\mathrm{a}}(f^-, f^+)\,(h\,y)\ \mathcal{G}_1(R_1^-, R_2^+)\ \Psi_{\mathrm{a}.G_1'\mathrm{a}}(f'^-, f'^+)\,(h'\,y')$$

and that follows form the assumption that $\mathcal{G}_1$ was admissible (taking y and y' in the definition of admissibility to be the $h\,y$ and $h'\,y'$ above). ■

From this, we get admissibility of the action relating ambient computations to their continuation-passing counterparts:

Lemma A.18 *Let γ be a type of L', and for any $R \in \mathrm{ARel}(\alpha, \alpha')$ let the relation ${}^{\bar\circ}R \in \mathrm{CARel}({}^\circ\alpha, (\alpha' \to {}^\circ\gamma) \to {}^\circ\gamma)$ be given by:*

$$\begin{aligned} &m\ ({}^{\bar\circ}R)\ u \\ &\quad\iff \forall\alpha_0\ \mathsf{type}_L, O \in \mathrm{ARel}(\alpha_0, \gamma).\, \lambda k^{\alpha \to {}^\circ\alpha_0}.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ k\,x\ ((R \to {}^\circ O) \to {}^\circ O)\ u \end{aligned}$$

Further, let F and F' be type constructors, with an admissible relational action $\mathcal{F}$. Then the relational action $\mathcal{G}$ given by

$$\mathcal{G}(R^-, R^+) = {}^{\bar\circ}\mathcal{F}(R^-, R^+)$$

is computation-admissible for $G\mathrm{a} = {}^\circ(F\mathrm{a})$ and $G'\mathrm{a} = (F'\mathrm{a} \to {}^\circ\gamma) \to {}^\circ\gamma$.

Proof. First note that for any α_0 and $O \in \mathrm{ARel}(\alpha_0, \gamma)$, the action $\mathcal{G}_1^O$ given by

$$\mathcal{G}_1^O(R^-, R^+) = (\mathcal{F}(R^-, R^+) \to^{\mathrm{r}} {}^\circ O) \to^{\mathrm{r}} {}^\circ O$$

is computation-admissible for $G_1^{\alpha_0}\mathrm{a} = (F\mathrm{a} \to {}^\circ\alpha_0) \to {}^\circ\alpha_0$ and G' by Lemma A.16(2,6).

Let $h = \lambda m.\, \lambda k.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ k\,x$. This mapping is a natural transformation between the functors derived from G and $G_1^{\alpha_0}$:

$$\begin{aligned} &h\,(\Psi_{\mathrm{a}.{}^\circ F\mathrm{a}}(f^-, f^+)\,m) = h\,(\mathbf{let}^\circ\ r \Leftarrow m\ \mathbf{in}\ \mathbf{let}^\perp\ y \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)\,r\ \mathbf{in}\ {}^\circ y) \\ &\quad= \lambda k.\, \mathbf{let}^\circ\ x \Leftarrow (\mathbf{let}^\circ\ r \Leftarrow m\ \mathbf{in}\ \mathbf{let}^\perp\ y \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)\,r\ \mathbf{in}\ {}^\circ y)\ \mathbf{in}\ k\,x \\ &\quad= \lambda k.\, \mathbf{let}^\circ\ r \Leftarrow m\ \mathbf{in}\ \mathbf{let}^\perp\ y \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)\,r\ \mathbf{in}\ k\,y \\ &\quad= \lambda k.\, (h\,m)\,(\lambda r.\, \mathbf{let}^\perp\ y \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(f^-, f^+)\,r\ \mathbf{in}\ k\,y) = \lambda k.\, (h\,m)\,(\Psi_{\mathrm{a}.F\mathrm{a} \to {}^\circ\alpha_0}(f^+, f^-)\,k) \\ &\quad= \lambda k.\, \mathbf{let}^\perp\ y \Leftarrow {}^\perp(\Psi_{\mathrm{a}.F\mathrm{a} \to {}^\circ\alpha_0}(f^+, f^-)\,k)\ \mathbf{in}\ (h\,m)\,y = \Psi_{\mathrm{a}.(F\mathrm{a} \to {}^\circ\alpha_0) \to {}^\circ\alpha_0}(f^-, f^+)\,(h\,m) \end{aligned}$$

(using that $\Psi_{\mathrm{a}.{}^\circ\alpha_0}(f^-, f^+) = \mathsf{id}_{{}^\circ\alpha_0}$ by Lemma A.2(1), because a cannot appear free in α_0). Thus, taking h' as the identity in Lemma A.17, we get that the action given by $\mathcal{G}^O$,

$$m\ (\mathcal{G}^O(R^-, R^+))\ u = \lambda k.\, \mathbf{let}^\circ\ x \Leftarrow m\ \mathbf{in}\ k\,a\ (\mathcal{G}_1^O(R^-, R^+))\ u$$

is computation-admissible for G and G'. And finally, since $\mathcal{G}(R^-, R^+)$ is the intersection of all $\mathcal{G}^O(R^-, R^+)$, we get the result by Lemma A.16(8). ■

We can now state the main result motivating the definition of admissible actions:

Theorem A.19 *Let $\mathcal{F}$ be an admissible relational action for type constructors F and F'. Then there exists an* invariant relation *for $\mathcal{F}$, i.e., a relation $\Delta \in \mathrm{ARel}(\mu\mathrm{a}.\, F\mathrm{a}, \mu\mathrm{a}.\, F'\mathrm{a})$ such that $a\ \Delta\ a' \iff \mathsf{unroll}_{\mathrm{a}.F\mathrm{a}}\, a\ \mathcal{F}(\Delta, \Delta)\ \mathsf{unroll}_{\mathrm{a}.F'\mathrm{a}}\, a'$.*

Proof. (The proof technique is due to Pitts and can essentially be found in [Pit99]. However, since we are working with binary relations instead of unary ones, and a few details are slightly more involved for predomains than for domains, it seems worth spelling out the construction.)

As usual, we abbreviate roll as ϕ and unroll as ψ.

We first note that a functorial action on relations preserves inclusions. For let $R_1^- \subseteq R_2^-$ and $R_1^+ \subseteq R_2^+$. Take $f^- = f^+ = \lambda a.\, {}^\perp a$ and $f'^- = f'^+ = \lambda a'.\, {}^\perp a'$. Then clearly

$f^-\ (R_1^- \to {}^\perp R_2^-)\ f'^-$ and $f^+\ (R_1^+ \to {}^\perp R_2^+)\ f'^+$. Hence for any $y\ \mathcal{F}(R_2^-, R_1^+)\ y$, we get from Lemma A.2(3) that

$${}^\perp y = \Phi_{\mathrm{a}.F\mathrm{a}}(\lambda a.\,{}^\perp a, \lambda a.\,{}^\perp a)\, y\ ({}^\perp \mathcal{F}(R_1^-, R_2^+))\ \Phi_{\mathrm{a}.F'\mathrm{a}}(\lambda a'.\,{}^\perp a', \lambda a'.\,{}^\perp a')\, y' = {}^\perp y'\,.$$

Moreover, we easily see that ${}^\perp x\ ({}^\perp R)\ {}^\perp x'$ iff $x\ R\ x'$ (by definition of relation-lifting specifically, not true for computation-extension of relations in general), so we get $y\ \mathcal{F}(R_1^-, R_2^+)\ y'$, i.e., $\mathcal{F}(R_2^-, R_1^+) \subseteq \mathcal{F}(R_1^-, R_2^+)$. In other words, $\mathcal{F}$ is monotone in its second argument and antimonotone in the first one.

Let R^- and R^+ be arbitrary relations in $\mathrm{ARel}(\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a})$, and define the relations $R^\mp, R^\pm \in \mathrm{ARel}(\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a})$ by

$$a\ R^\mp\ a' \iff \psi\, a\ \mathcal{F}(R^+, R^-)\ \psi'\, a' \quad \text{and} \quad a\ R^\pm\ a' \iff \psi\, a\ \mathcal{F}(R^-, R^+)\ \psi'\, a'\,.$$

We can then define an operator Θ, mapping (R^-, R^+) to $(R^\mp, R^\pm)$, antimonotone in the first position and monotone in the second. Further, the set of admissible relations between two types is closed under arbitrary intersection, so $\mathrm{ARel}(\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a})^{op} \times \mathrm{ARel}(\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a})$ forms a complete lattice. Hence, by the Knaster-Tarski fixed-point theorem, Θ has a least fixed point (Δ^-, Δ^+), with $\Delta^-, \Delta^+ \in \mathrm{ARel}(\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a})$ satisfying

$$a\ \Delta^-\ a' \iff \psi\, a\ \mathcal{F}(\Delta^+, \Delta^-)\ \psi'\, a' \qquad \text{and} \qquad a\ \Delta^+\ a' \iff \psi\, a\ \mathcal{F}(\Delta^-, \Delta^+)\ \psi'\, a'\,.$$

Moreover, (Δ^+, Δ^-) is clearly *also* a fixed point of Θ, and so must be greater than the least one, giving (for both components) the inclusion $\Delta^+ \subseteq \Delta^-$.

It thus remains to show containment in the other direction. Consider the relation $\nabla \in \mathrm{CARel}(\mu\mathrm{a}.\,F\mathrm{a} \to {}^\perp\mu\mathrm{a}.\,F\mathrm{a}, \mu\mathrm{a}.\,F'\mathrm{a} \to {}^\perp\mu\mathrm{a}.\,F'\mathrm{a})$ determined by

$$h\ \nabla\ h' \iff \forall a\ \Delta^-\ a'.\, h\, a\ ({}^\perp\Delta^+)\ h'\, a'$$

(∇ is computation-admissible because it is given by an intersection over inverse images (by application, which is rigid) of the computation-admissible ${}^\perp\Delta^+$.) Now define the functional $H : (\mu\mathrm{a}.\,F\mathrm{a} \to {}^\perp\mu\mathrm{a}.\,F\mathrm{a}) \to (\mu\mathrm{a}.\,F\mathrm{a} \to {}^\perp\mu\mathrm{a}.\,F\mathrm{a})$ by:

$$H = \lambda h^{\mu\mathrm{a}.F\mathrm{a} \to {}^\perp\mu\mathrm{a}.F\mathrm{a}}.\, \lambda a^{\mu\mathrm{a}.F\mathrm{a}}.\, \mathbf{let}^\perp\ x \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(h,h)\,(\psi\, a)\ \mathbf{in}\ {}^\perp(\phi\, x)$$

and analogously for H'. We want to show that when $h\ \nabla\ h'$ then also $H\,h\ \nabla\ H'\,h'$, i.e., that

$$\begin{aligned} \forall a\ \Delta^-\ a'.\ & \mathbf{let}^\perp\ x \Leftarrow \Phi_{\mathrm{a}.F\mathrm{a}}(h,h)\,(\psi\, a)\ \mathbf{in}\ {}^\perp(\phi\, x) \\ & \qquad ({}^\perp\Delta^+)\ \mathbf{let}^\perp\ x' \Leftarrow \Phi_{\mathrm{a}.F'\mathrm{a}}(h',h')\,(\psi'\, a')\ \mathbf{in}\ {}^\perp(\phi'\, x') \end{aligned}$$

This follows from the usual properties of relation-extension, the equations defining Δ^- and Δ^+ above, and the functorial action of $\mathcal{F}$ (taking $f^- = f^+ = h$, $f'^- = f'^+ = h'$, $R_1^- = R_1^+ = \Delta^-$, and $R_2^- = R_2^+ = \Delta^+$).

Thus, since the relation ∇ was computation-admissible, we get by fixed-point induction (Lemma 3.19) that $\mathsf{fix}\, H\ \nabla\ \mathsf{fix}\, H'$. And because $\mathsf{fix}\, H = \lambda x.\,{}^\perp x$ by the minimal-invariant property (Corollary A.8), we have

$$\forall a\ \Delta^-\ a'.\, {}^\perp a\ ({}^\perp\Delta^+)\ {}^\perp a'\,.$$

Finally, by the same argument about ${}^{\perp}R$ as at the beginning of the proof, this simplifies to $\Delta^{-} \subseteq \Delta^{+}$, completing the proof that we can take $\Delta = \Delta^{-} = \Delta^{+}$ as the invariant relation for $\mathcal{F}$. ■

From this, we immediately get that – much like recursive type equations – a large class of recursive relation equations has solutions:

Corollary A.20 *Let F and F' be type constructors, and let ${}^{\bullet}$ be a formal relation constructor, built out of (1) the standard relational actions of L_1-type constructors, (2) constant admissible relations (computation-admissible for computation-types), and (3) the relation constructor ${}^{\overline{\circ}}$ (for any γ); so that ${}^{\bullet}$ maps any relation $R \in \mathrm{ARel}(\alpha, \alpha')$ to ${}^{\bullet}R \in \mathrm{ARel}(F\alpha, F'\alpha')$.*

Then there exists a relation $\mu R.\, {}^{\bullet}R \in \mathrm{ARel}(\mu \mathrm{a}.\, F\mathrm{a}, \mu \mathrm{a}.\, F'\mathrm{a})$ such that

$$a\ (\mu R.\, {}^{\bullet}R)\ a' \iff \mathsf{unroll}_{\mathrm{a}.F\mathrm{a}}\, a\ {}^{\bullet}(\mu R.\, {}^{\bullet}R)\ \mathsf{unroll}_{\mathrm{a}.F'\mathrm{a}}\, a' \,.$$

Proof. By induction on ${}^{\bullet}$, using Lemmas A.16 and A.18, we directly obtain an admissible mixed relational action $\mathcal{F}$, such that

$$\mathcal{F}(R, R) = {}^{\bullet}R$$

By Theorem A.19, this $\mathcal{F}$ has an invariant relation Δ. And because of the equation above, we can simply take $\mu R.\, {}^{\bullet}R$ to be Δ. ■

Bibliography

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[CM93] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of the Conference on Category Theory and Computer Science*, Amsterdam, September 1993. CWI Tech. Report.

[CR91] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, July 1991.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[Esp95] David A. Espinosa. *Semantic Lego.* PhD thesis, Graduate School of Arts and Sciences, Columbia University, May 1995.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.

[Fis72] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109. SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14, January 1972. Revised version in *Lisp and Symbolic Computation*, 6(3/4), 1993.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'état, Université Paris VII, 1972.

[GRR95] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming and Computer Architecture*, 1995.

[Hay87] Christopher T. Haynes. Logic continuations. *The Journal of Logic Programming*, 4(2):157–176, June 1987.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[HDM93] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. (A preliminary version appeared in *Proceedings of the 1991 Symposium on Principles of Programming Languages*).

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[JD88] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 158–168, San Diego, California, January 1988.

[JG89] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Languages Design and Implementation*, pages 218–226, Portland, Oregon, June 1989.

[KJLS87] David K.Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 reference manual (edition 1.0). Technical Report MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.

[KW93] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 134–143, Ayr, Scotland, 1993. Springer-Verlag.

[LD94] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, June 1994.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[M$^+$62] John McCarthy et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996. To appear.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1990.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Mor93] J. Gregory Morrisett. Generalizing first-class stores. In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 73–87, Copenhagen, Denmark, June 1993. (Technical report YALEU/DCS/RR-968, Department of Computer Science, Yale University).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.

[Pit99] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 199? Revised version of Cambridge Computer Laboratory Technical Report Number 321. To appear.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993.

[QS91] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, August 1972.

[Rey74a] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.

[Rey74b] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, number 19 in Lecture Notes in Computer Science, pages 408–425, Paris, France, April 1974.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[SF90] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993. (An earlier version appeared in *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*).

[Smi82] Brian C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1982. MIT-LCS-TR-272.

[ST80] Ravi Sethi and Adrian Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, Oregon, January 1994.

[Sto81] Joseph E. Stoy. The congruence of two programming language definitions. *Theoretical Computer Science*, 13(2):151–174, February 1981.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, Nancy, France, September 1985.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990.

[Wad92a] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992. (An earlier version appeared in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*).

[Wad92b] Philip Wadler. The essence of functional programming (invited talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992.

[Wad94] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, January 1994.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford, California, August 1980.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1), May 1988.