RECURSION THEORETIC SEMANTICS, FULLY ABSTRACT TERM MODELS, AND IMPERATIVE CONSTRUCTS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ramesh Viswanathan
July 1995

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Vaughan R. Pratt

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Martín Abadi

Approved for the University Committee on Graduate Studies:

Acknowledgements

My doctoral study has been a unique educational experience for me which has been the result of many people who played a significant role in shaping my existing thinking and state of knowledge. First and foremost, I would like to express a deep gratitude to my advisor, John Mitchell. His rich and insightful guidance ensured that I was never at a loss for avenues to explore while his patience allowed me to explore them to the level that I felt satisfied with. Through his personal example, he has taught me the process of understanding new ideas and conducting research and I feel extremely privileged to have had the opportunity to interact with him. My research activity and awareness of existing work on programming language semantics received a significant impetus through the course of a summer spent at AT&T Bell Labs under the able supervision of Jon Riecke. The progress I made then was entirely due to the problems that he suggested and the discussions with him, and I am truly thankful to him for that fruitful period of time. Through courses taught by Martín Abadi, Sol Feferman, Grigori Mints, and Vaughan Pratt and discussions with them, my thinking and understanding of concepts has been enriched by a glimpse of their unique perspectives. I would also like to thank Vineet Gupta for the numerous interactions with him, in which I either learned several mathematical ideas directly from him or was motivated to study them.

Coming to the dissertation itself, Chapter 2 describes joint work with John Mitchell, Chapters 3 and 4 are joint work with Jon Riecke, and Chapter 5 is joint work with My Hoang and John Mitchell. All the chapters are revised and expanded versions of papers written jointly with them. Thus, I owe significant debt to them both for the genesis and expression of ideas found in those chapters. I had many useful discussions with Brian Howard and Gordon Plotkin which influenced the work described in Chapter 2. I would also like to thank Martín Abadi for his thorough reading of a draft of this thesis and his extremely useful comments.

I was introduced to the beauty of logic and computation by Somnath Biswas in my undergraduate studies. Without consciously realizing it, he was singlehandedly responsible for my interest in pursuing doctoral studies. Finally, a deep and heartfelt thanks to my parents for their unbridled support and encouragement of everything that I have wanted to do. With great humility and affection, this thesis is dedicated to them.

Contents

List of Tables

Chapter 1

Introduction

This thesis is a collection of four problems related to the theory of typed programming languages. In this introduction, we provide a brief overview of the lines of investigation pursued in the remaining chapters. More detailed motivations and explanations of results are provided in the introduction sections of the respective chapters.

There have been two essentially distinct traditions that have aimed at providing a "mathematical theory of computation". Classical recursion theory proceeds by defining the computable functions on natural numbers via suitable programs, *e.g.*, Turing Machines. Computability on arbitrary sets is considered by encoding the elements of the set as natural numbers — computing with elements of the set is modeled as computing with their natural number codes. In particular, functions are represented by the Turing Machines that compute them and since turing machines can be coded as natural numbers, one can define the computable higher-order functions. A distinct view of which higher-order functions are computable is provided by typed λ-calculi in which programs of the appropriate type are taken to define the set of computable functions of that type. The mathematical explanation of the syntax of these λ-calculi (their semantics) has typically proceeded by using partially ordered structures called domains and taking the denotation of programs to be continuous functions on the domains. In the domain-theoretic view, computability is therefore understood as continuity. In Chapter 2, we attempt to understand the connection between these two unrelated views of computation provided by recursion theory and domain theory. More precisely, we develop and study a semantic framework based on recursion theory, in which we can interpret the programming primitive of recursion, and the type constructions of polymorphism, subtyping, and recursive types. The semantic framework is based on identifying an inherent order structure on sets, whose elements are encoded as natural numbers, using the (recursion-theoretic) computable functions for such sets and taking the interpretation of types to be sets for which this ordering is (effectively) "complete". A significant property of this semantics is that such sets are then coded by the natural numbers that stand for the turing machines exhibiting the effectivity of the completeness of their orderings, *i.e.*, types are themselves encoded as natural numbers. This leads to new interpretations for the concepts of polymorphism and subtyping. The chapter also exhibits a surprising fundamental distinction between on the one hand classical recursion theory based on using natural numbers as codes of functions and on the other hand λ-calculus. Specifically, recursion can be computed uniformly by a single algorithm at all types when we construct the model over untyped call-by-value lambda terms, but not when we use numbers to encode computable functions.

Chapter 3 is concerned with the notion of operational equivalence in programming languages. Two expressions in a programming language are said to be *operationally equivalent* if they are interchangeable in arbitrary programs without affecting the observable outcome of the program. From a programming perspective, one needs to reason about operational equivalences in many situations. For example, if a programmer is refining some module or component of a piece of software, to be more efficient, then the refinement is guaranteed to be correct if the refined version of the module is equivalent to the old module. Similarly, it is important to establishing the correctness of local optimizations that may be performed during compilation. One of the purposes of a model of a programming language is to provide a mathematical structure in which it is simpler to analyse and reason about the programming language. We focus on two aspects of models that make them useful frameworks for reasoning. The first, which we term *extensionality*, is the property that the elements of functional types are akin to mathematical functions in that they are characterized by their extensional behavior. In more detail, the identity of any element of the model (at a functional type) is determined by the results it produces for each argument in its domain. The second, which we term *continuity*, is the property that the interpretation of types possess the structure of a complete partial order. A continuous model then allows recursion to be interpreted as least fixed points, *i.e.*, as least upper bounds of certain chains. Extensionality supports the proof principle that in establishing the equality of two elements of a functional type it suffices to prove that they have the same extensional behavior, and continuity allows us to reason about recursion using least upper bounds. A model is said to *fully-abstract* if it captures the operational equivalence of two expressions as equality of their meanings in the model. A fully-abstract model then provides a framework in which one can analyse the operational equivalences of a language. In Chapter 3, we show how to construct fully abstract models that are extensional and continuous. The distinguishing aspect of the languages considered is that they are *purely sequential* for which obtaining such models have been typically difficult. There is a well-known construction, due to Milner, of a fully abstract model for languages under call-by-name evaluation mechanism. We simplify this construction and generalize it in two ways: first, we consider languages with a call-by-value evaluation mechanism and second, we show how to obtain fully-abstract interpretations when the language includes recursive type definitions.

The subject of the last two chapters is the study of programming languages that include side-effecting features such as imperative memory assignment. Chapter 4 concerns operational equivalence for languages with such imperative constructs. For purely functional languages, there is a simple set of sound reasoning principles for determining when two pieces of code are operationally equivalent. (This is not in contradiction with the elusiveness of fully abstract models because the latter difficulty corresponds to obtaining a *complete* set of reasoning principles for operational equivalence.) This is an attractive aspect of functional languages because of the connections of operational equivalence with modular development of programs outlined earlier. However, the vast majority of programming languages are not purely functional — they include imperative constructs since they allow more efficient programs to be written. Unfortunately, this extension of the functional language with imperative constructs is not conservative with respect to operational equivalence, *i.e.*, there can be purely functional pieces of code that would be equivalent if the language were purely functional but are inequivalent in its extension with imperative constructs. In particular, this implies that functional reasoning principles cannot be used to reason even about purely functional code in the presence of imperative constructs. What might be useful is some way of using the simple functional reasoning principles even when the language includes imperative constructs. The solution we explore in Chapter 4 is to define a coercion operation encap which can be applied to functional code to restore operational equivalences. More precisely, we design the executional behavior of the operator encap in such a way that that for any two purely functional pieces of code M and N, M and N would be operationally equivalent in the purely functional fragment if and only if $\mathsf{encap}\,M$ and $\mathsf{encap}\,N$ are operationally equivalent in the full language with imperative features.

The execution rules of encap are designed using a denotational semantics of the language consisting of imperative constructs. Imperative constructs are seen as expressing a particular kind of side-effect — other examples of side-effects include exceptions, continuations, and non-determinism. The key technical device is the formulation of a framework (call-by-value monad) in which any kind of side-effect can be expressed by specifying some semantic primtives for manipulating the particular side-effect. The denotational semantics of any side-effecting operations can then be expressed uniformly using these primitives. Using these abstract semantic primitives, it is also easy to define a function that restores the *denotational* equivalences of purely functional expressions. By using the fully abstract models constructed in Chapter 3, we can then take the execution behavior of encap to be this function and mirror the denotational equivalences as the desired operational equivalences. The resulting execution rules for encap is such that encap provides a mechanism to force the side-effects of parts of programs to be invisible to the rest of the program. The advantages of such a mechanism for localizing the side-effects of parts of programs are further explored in Chapter 4.

Finally, Chapter 5 concerns the design of flexible type systems for languages with imperative constructs. Typed programming languages have the well known advantages of efficient error-free execution of programs and enhanced modularity of program code. For purely functional languages, there is a simple set of type checking rules that is *sound* (*i.e.*, ensures the absence of run-time errors for any typable program) and *polymorphic* (*i.e.*, allows pieces of code to be used at more than one type). However, for non-functional features such as assignment, exceptions, and continuations, the formulation of set of typing rules that allows code to be used polymorphically and yet sound is subtle. The most flexible type system that has found wide practical usage is the one used in the New Jersey implementation of the language Standard ML (SML-NJ). Although the SML-NJ compiler has been used for several years, its type system was not completely understood and had not been shown to be sound. Chapter 5 describes an axiomatic formulation of a type system expressing the concepts underlying weak polymorphism and and proves it correct. The resulting type system seems to be simpler and more powerful than the algorithm implemented in the SML-NJ compiler.

Chapter 2

Recursion Theory in the Semantics of Programming Languages

2.1 Introduction

The work described in this chapter began with a very simple-minded objective. This was to develop an elementary class of semantic models for polymorphism, subtyping, and recursion based on ideas from recursion theory instead of domain theory. Our driving intuition was that the compiler for a language such as ML compiles each expression into a bit string that could be regarded as the Gödel code for some function, pair, number, or other kind of datum, as appropriate. For each type, we therefore have a set of Gödel numbers representing the elements of the type. Since semantic models are intended to provide some basis for equational reasoning, we must also define an equivalence relation on the set of Gödel codes for each type. Our intention was therefore to define a membership predicate and equivalence relation, which together give us a so-called partial equivalence relation, for each type and take the resulting quotients of subsets of the natural numbers as the "domains" of our model.

A representative sample of recent studies using partial equivalence relations were presented at the 1990 IEEE Logic in Computer Science Conference. The first of three papers describes a special class of partial equivalence relations over the natural numbers called *extensional pers* [FRMS92], the second uses partial equivalence relations over partially-ordered domains instead of natural numbers [AP90], and the third works in the constructive setting of the *effective topos,* which boils down to partial equivalence relations over the natural numbers [Pho90b]. Since our goal is to develop a semantic model that can be viewed as a quotient of the result of compilation, the use of pers over special partially-ordered domains does not meet our objectives. While extensional pers do give us quotients of sets of bit-strings, the standard and expected representations of basic datatypes such as natural numbers and booleans, as well as type constructors such as products, function spaces, and lifting, require modification if they are to be "extensional" (in the technical sense of extensional pers). Therefore, as elaborated later in the chapter, this model too does not have the basic properties we would like. Our work is close in spirit to [Pho90b], although our "bottom-up" development yields a model that seems quite different in technical detail.

We begin with the category **Per** whose objects (interpretation of types) are partial equivalence relations over the natural numbers and whose maps (interpretation of terms) are computable functions, in the usual sense of computability theory. While this category is a well-known model of polymorphism and subtyping, it is not useful for interpreting recursion. This is shown by defining a per A of partial functions and a total recursive f on A that has no fixed point. (In other words, the recursion theorem does not hold for arbitrary quotients of subsets of $\mathcal{N}$.) The counter example is based on intuition from domain theory: if we order A in the natural way, it is not "complete". This suggests that the appropriate way to extend the recursion theorem to subquotients is by borrowing ideas from domain theory. In the resulting combination of recursion-theoretic and domain-theoretic techniques, it turns out that we need far fewer ideas from domain theory than [AP90, BM92], for example. (Section 2.11 contains a more detailed comparison with related work.)

Using essentially the same "intrinsic order" as [AP90, Pho90b], we identify a class of "cpo-like" pers, each such per R having a computable function sup_R that gives the least upper bound of any constructible chain in R. We call these *effective* *cpo's;* although the name is similar, these are completely different from the effectively presented cpo's described in [GS90], for example. By a generalization of the Myhill-Shepherdson theorem from recursion theory, all computable functions between effective cpo's are continuous. If an effective cpo has a least element, then the usual domain theoretic arguments give us an effective least fixed point operator.

Serendipitously, almost all of our basic desiderata are satisfied by this construction. The natural numbers (discretely ordered) and other basic data types are easily seen to be effective cpo's. The usual product, function space and other constructions preserve the property of being an effective cpo. Moreover, the intrinsic orderings turn out to be the usual "pointwise orderings" of domain theory. Since it is easy to use the supremum operator to compute a least fixed point, the full subcategory **ECpo** of effective cpo's and computable functions gives us an elementary semantics of first-order type operators and recursion.

In the model using Gödel numbers of partial recursive functions, the interpretation of higher-order types requires that we code types as natural numbers. The traditional approach, when we are not concerned with recursion, is to make the coding relation trivial, leading to an interpretation of polymorphic types as intersection of infinite indexed families. However, the intersection of a constructive family of effective cpo's is not necessarily an effective cpo. We therefore represent a type by any Gödel code of its supremum operator, producing an interpretation of a polymorphic type $\forall t.A(t)$ as the collection of recursive functions mapping any supremum operator sup_R to an element of the per $A(R)$. Fortunately, this interpretation of polymorphism still satisfies the expected parametricity principles. For example, we can show that the per $\forall t.t$ is empty and that the only element of the per $\forall t.t \to t$ is the identity function. We can also define a $\forall^p$ quantifier where we interpret the domain of quantification to be pointed effective cpos. The difference from the $\forall$ quantifier arises because the code for a pointed effective cpo naturally gives both its least element and its supremum operator. Continuing our examination of parametricity properties, we show that the least fixed point operator is the only fixed point operator of type $\forall^p t.(t \to t) \to t$. Due to least elements, the type $\forall^p t.t$ becomes nonempty and $\forall^p t.(t \to t)$ contains more than the identity function. In short, the $\forall$ quantifier supports the parametricity principles of the system **F** while the $\forall^p$ quantifier supports the parametricity principles of system **F** extended with fixed points.

A standard method for obtaining recursive types, described in [SP82], involves showing that suitable functors on **Cpo**-enriched categories have initial F-algebras. To obtain analogous results, we work with **ECpo**-enriched categories whose homsets each form an effective cpo. In addition, and in a departure from applications of the Smyth-Plotkin method we have seen before, we ask for functors that are effective on objects in addition to morphisms. This means that a functor F on **ECpo** is *effective* if the supremum function on $F(R)$ is uniformly computable from the supremum function on R. If each homset is also pointed, we have a **PECpo**-enriched category. As for morphisms in **Per**, we can show that any effective functor is automatically locally continuous (in the sense of [SP82]) and thus any effective functor between *effectively complete* **PECpo**-enriched categories has an initial F-algebra. Our categories **ECpo** and **PECpo** can be shown to be effectively complete, as well as **ECpo**-enriched and **PECpo**-enriched, respectively. This gives us initial F-algebras for all effective functors. All of this feels relatively routine, once we have the idea of realizers for effective cpos; we are simply working out effective versions of the usual arguments.

In other per models, the interpretation of subtyping has always been the subset relation on pers, namely, R is a subtype of S iff the membership predicate and equality relations of R are subsets of those for S. Intuitively, this means that a supertype may have more elements than a subtype, but also distinct elements of a subtype may become identified at a supertype. (It is argued elsewhere that this is very intuitive and natural.) In our model, we cannot simply follow this approach since we need coherence between the supremum operators at subtype and supertype, and between least elements if both are pointed. More specifically, we say an effective cpo R is a subtype of effective cpo S if in addition to being a subset, the least upper bound of any increasing chain in R is also a least upper bound in S. Moreover, if R is pointed then any index of the least element of R must also be one for S. All the usual examples of subtyping hold with our modified definition of subtyping. We also prove a strong form of coherence, namely, if two terms of the same type yield the same untyped term when types are erased, then their meanings are equal.

With the properties we have discussed so far, we have a semantic model, based on a combination of simple recursion-theoretic and simple domain-theoretic ideas, for common basic type operators, impredicative polymorphism, recursive types, and subtyping. We can also extend this structure to higher-order polymorphism. If we consider the category of all effective categories and effective functors, then this category essentially corresponds to the category **Per**. One can then show that the subcategory of **ECpo**-enriched categories and **PECpo**-enriched categories contain all the relevant internal structure of the categories **ECPo** and **PECpo** respectively. This implies that there are natural interpretations for all higher-order quantifiers of $\mathbf{F}\boldsymbol{\omega}$ and that one can also define recursive type constructors. The main technical difficulty in directly applying these ideas is that type operators such as function spaces do not yield covariant functors. We solve this problem by considering "doubles" of categories and symmetric effective functors. This idea has already been given in [AP90]; our main technical contribution is that we exhibit a cartesian closed structure on the category of doubles of **ECpo**-enriched categories and symmetric functors thus obtaining a categorical semantics of $\mathbf{F}\boldsymbol{\omega}$ extended with recursive values, recursive type constructors, and subtyping.

We started off with the intent of forming an extensional model by taking quotients of sets of bit strings (or Gödel numbers of functions). One reason for doing this was to obtain a close correspondence with compilation, since the denotation of an expression could be viewed as the equivalence class of the bit string produced by compiling this expression. However, a compiler for the kind of language we are interested in may generate code without using type information. A reasonable compiler could parse expressions, annotate the parse trees during type checking, but then discard or ignore the type annotations during code generation. A consequence is that we expected to find a uniform, type-independent algorithm computing the fixed point operator at each type. We can also consider the intuition of type-independent interpertations from a more mathematical standpoint. Since every lambda-definable function is computable we expect a heirarchy containing all computable functions to contain all the λ-definable functions. However, a difference between recursion theory and models of lambda-calculus is that a model must satisfy equations between terms making it an extensional theory whereas recursion theory admits many possible algorithms coding the same function making it an intensional theory. Since this seems to be the only difference, we would expect to obtain a subquotient structure over the natural number structure of recursion theory which is a model by suitably collapsing the intensional distinctions, in which the meaning of any term would be the equivalence class of the meaning of the untyped that results by erasing all type information from the term. Thus the existence of such a structure would seem to be true if the only difference between λ-calculus and recursion theory is some suitable notion of extensionality.

It is one of the most surprising aspects of our effort that we were unable to accomplish this. One way we might obtain a uniform fixed point algorithm is by giving a uniform algorithm for computing the suprema in any effective cpo. (This would not correspond with the usual compilation process, since compilers do not seem to use such order-theoretic primitives, but should at least settle the mathematical question.) In Section 2.9, we use computability arguments to show that there is no single algorithm computing suprema for both the per of natural numbers and the per of functions on natural numbers. The category **Per** as a model of simply-typed lambda calculus without fixed points has the property that the meaning of every term is the equivalence class of the meaning of the untyped term that results by erasing all type information from the term. A more natural candidate for a uniform fixed point algorithm is then an untyped fixed point operator such as the call-by-value version of Y (also given by the first recursion theorem) which also corresponds to the process of unwinding that a compiler interprets fixed points by. In Section 2.9, we show that no index of any untyped fixed point operator (and in particular Y) computes fixed points for any type that is a strict subquotient of the natural numbers. Because of the generality of this argument, it also shows that it is not just our choice of subcategory which does not work in this respect (*e.g.*, the operator Y would not be a fixed point operator for extensional pers, either). Fortunately, since our fixed point operator is polymorphic and is coherent with respect to subtyping, the lack of a uniform algorithm for all types does not seem to cause any observable problems with the model. For those familiar with the effective topos, the unimportance of a uniform algorithm seems attributable to the fact that uniformity cannot be expressed in the internal logic.

The proof of the fact that there is no single realizer for the fixed point operator at all types essentially depends on the fact that, in classical recursion theory, algorithms computing with functional arguments can use natural number operations (such as equality tests) on their function argument, since functions are also coded as natural numbers. We show that this is the only reason for the failure of obtaining a single realizer for uniform fixed points by considering an untyped lambda-calculus that corresponds closely to Turing machines except for introducing a distinction between natural numbers and codes of functions. If we consider effective cpos, defined exactly as before, but with respect to the application operation of this partial combinatory algebra, we can still inherit all the good properties of effective cpos in the effective topos since all computable functions on natural numbers are expressible. In addition, we can now show that the call-by-value fixed point operator Y computes least fixed points for all partial function spaces. This suggests that compilation is more profitably viewed as a translation to untyped lambda terms. While we initially regarded bit strings as the result of compilation, we realize upon reflection that the result of compiling a typed program is always a "typed algorithm." In other words, although machine language allows us to do bit string operations on function code, for example, no such manipulation need occur in the object code produced by compiling a typed program. Since natural numbers are distinct from function expressions in the untyped call-by-value calculus, there is just enough of an inherent type distinction to provide a uniform fixed-point algorithm. It also shows that in the presence of recursive values, the correspondence between untyped lambda terms and classical recursion theory does not extend to one when a type structure is imposed. While this may be a slight surprise to computer scientists, our eventual preference for call-by-value terms appears consistent with the move away from natural number coding in the field of recursion theory (e.g., [Bar75]).

2.2 Preliminaries

Let $\varphi_0, \varphi_1, \ldots$ be any enumeration of the partial recursive functions. We use $n \cdot m$ to denote $\varphi_n(m)$. From standard results in classical recursion theory, namely the existence of universal turing machines and the *s-m-n* theorem, any untyped lambda term can be interpreted as the index of a turing machine; we often use such terms to denote the obvious indices. It is easy to define computable pairing and projection functions on the natural numbers; to fix notation, we use pr for some bijective pairing function and $(n)_1$ and $(n)_2$ to denote the first and second projections of n. We use $n \cdot m\!\downarrow$ to mean that it is defined and $n \cdot m\!\uparrow$ to mean that it is undefined. We recall that the set $K = \{z \mid z \cdot z\!\downarrow\}$ is not recursive.

Let $\mathcal{N}$ denote the set of natural numbers. A partial equivalence relation (per) $R \subseteq \mathcal{N} \times \mathcal{N}$ is a symmetric transitive relation. For any per R, we write $n\colon R$ to denote that $n \mathrel{R} n$. For any $n\colon R$, we use $[n]_R$ to denote the equivalence class $\{m \mid n \mathrel{R} m\}$ and the set represented by R is defined to be $[R] = \{[n]_R \mid nRn\}$. We say that $a \in R$ to mean that $a \in [R]$ and say that a natural number n realizes an element a of R, $n \vdash_R a$ if $n \in a$. A set-theoretic function f from $[R]$ to $[S]$ is realized by a natural number n, denoted by $n \vdash_{R,S} f$, if for all $a \in [R]$, $m \vdash_R a$ we have that $n \cdot m \vdash_S f(a)$, where we use $n \cdot m \vdash_S f(a)$ to mean that $(n \cdot m)\!\downarrow$ and $n \cdot m \vdash_S f(a)$. We call a function $f\colon [R] \to [S]$ effective or computable if it has a realizer. We often omit the subscripts R, S from $n \vdash_{R,S} f$ when they are clear from the context. We use $[n]^{R,S}$ to denote the unique map from $[R]$ to $[S]$ that n realizes, if it realizes one. The category **Per** has as objects pers, and as morphisms from R to S the effective maps from $[R]$ to $[S]$. We say that two pers R and S are recursively isomorphic, denoted $R \cong S$, if they are isomorphic as objects of **Per**.

For any set $B \subseteq \mathcal{N}$, we define the per $D_B = \{\langle n, n\rangle \mid n \in B\}$, the identity relation on B. We use N to denote the per $D_{\mathcal{N}}$ and $\mathbf{1}$ to denote the per $D_{\{0\}}$. If R, S are pers, we define the pers $R \times S$, $R \to S$, and $R \rightharpoonup S$ as follows:

$$\begin{array}{rll} n\, R \times S\, m & \text{iff} & (n)_1\, R\, (m)_1 \text{ and } (n)_2\, S\, (m)_2 \\ n\, R \to S\, m & \text{iff} & \forall x, y \in \mathcal{N}.\; x\, R\, y \;\Rightarrow\; (n \cdot x)\, S\, (m \cdot y) \\ n\, R \rightharpoonup S\, m & \text{iff} & \forall x, y \in \mathcal{N}.\; x\, R\, y \;\Rightarrow\; (n \cdot x)\!\downarrow \text{ or } (m \cdot y)\!\downarrow \;\Rightarrow\; (n \cdot x)\, S\, (m \cdot y) \end{array}$$

where we use $(n \cdot x)\, S\, (m \cdot y)$ to mean that $(n \cdot x)\!\downarrow$ and $(m \cdot y)\!\downarrow$ and they are related in S. We can define the lifting of a per S as

$$S_\perp = \mathbf{1} \rightharpoonup S$$

It is the unique (up to recursive isomorphism) per such that

$$R \rightharpoonup S \;\cong\; R \to S_\perp$$

for any per R.

Let A be any set and $\vdash_A \subseteq \mathcal{N} \times A$ be a binary relation which we often call a realization relation. We say that $\vdash_A$ is *onto* if for all $a \in A$, there is an n with $n \vdash_A a$ and a *function* if whenever $n \vdash_A a, n \vdash_A b$ we have that $a = b$. Every onto function $\vdash_A$ on a set A gives rise to a per, namely,

$$\{\langle n, m\rangle \mid n, m \vdash_A a,\; a \in A\}$$

Suppose $\vdash_A$ is some fixed onto realization relation on a set A and $\vdash_B$ is another fixed onto realization relation on a set B. We call a function $f\colon A \to B$ *effective* if there is an n such that for all $a \in A, m \vdash_A a$ we have that $n \cdot m \vdash_B f(a)$; we take n to be the realizer of f. We also say that $f(a)$ is *effectively realizable* from a to mean that the function f is effective.

2.3 Fixed Points and Effective Cpos

Our goal is to give a programming language semantics based on the natural and intuitively accessible notion of pers over natural numbers. More concretely, we seek a category of pers over $\mathcal{N}$, where we can interpret basic datatypes like *nat*, *bool*, the type constructors of lifting $(\cdot)_\perp$, products $(\cdot)\times(\cdot)$, total function spaces $(\cdot)\to(\cdot)$, partial function spaces $(\cdot)\rightharpoonup(\cdot)$, fixed point operators at appropriate types, recursive types, polymorphism, and subtyping. To start off, we need to verify whether **Per** does not already give us such a category. We've already seen natural interpretations for the basic type constructors — so, the next thing to verify is the issue of fixed point operators.

We begin by showing that the full category **Per** is not suitable for intepreting fixed points. Our starting point might be the second recursion theorem of recursion theory [Rog67, Cut80] which states that for every $F\colon(N\rightharpoonup N)\to(N\rightharpoonup N)$, there is an $f\colon N\rightharpoonup N$ such that $(F\cdot f)\ (N\rightharpoonup N)\ f$, with f computable from F. We might therefore hope to generalize the statement to hold for all $F\colon(R\rightharpoonup S)\to(R\rightharpoonup S)$ for arbitrary pers R, S. However, as the following example shows, there may not even exist fixed points of such functions.

Example: [Failure of Fixed Points] Take R to be the per $\mathbf{1}$ and S to be the per of partial functions that are defined exactly on some finite prefix of the natural numbers. More formally, S is defined by $m\ S\ n$ iff $\exists u\in\mathcal{N}\ (\forall x<u\ (m\cdot x=n\cdot x)$ and $\forall x\geq u\ (m\cdot x\uparrow$ and $n\cdot x\uparrow))$. Let F be an index of the function, which on input $f\colon(R\rightharpoonup S)$, returns an index of the function, which on input $0\colon\mathbf{1}$ returns an index of the function given by the following algorithm:

$$\begin{aligned} F\cdot f\cdot 0\cdot n \ = \ & \text{compute } g=f\cdot 0 \\ & \text{if } n=0 \\ & \qquad \text{return } 0 \\ & \text{else return } g\cdot(n-1) \end{aligned}$$

It is easily verified that $F\colon(R\rightharpoonup S)\to(R\rightharpoonup S)$. We can now show that F has no fixed point. First note that $F\cdot f$ is always defined on $0\colon\mathbf{1}$ — thus, the function in $(R\rightharpoonup S)$ that is undefined on the unique element of R cannot be a fixed point. Otherwise, if $f\colon(R\rightharpoonup S)$ is defined on 0, with $f\cdot 0=g$, then by the assumption that $g\colon S$, we have that g is the index of a function with finite domain $\{0,\ldots,k\}$. In this case, $F\cdot f\cdot 0$ is the index of a function with larger domain $\{0,\ldots,k,k+1\}$. Therefore, no $f{:}\,\mathbf{1}\rightarrow S$ with $f\cdot 0\!\downarrow$ can be a fixed point of F either.

An important idea in the above example is that if we order the partial functions that are the elements of the per S by the pointwise ordering, as we would in domain theory, then S is not a cpo. Therefore, it seems natural to circumvent the problem illustrated by the per S by selecting a class of cpo-like pers. The first step is to associate an order with each per. To define an ordering on the elements of a per R, we focus attention on effective partial functions from R to $\mathbf{1}$. Since, $\mathbf{1}$ is a one-element per, the only significant behaviour of any $f{:}\,R\rightharpoonup\mathbf{1}$ on an $x{:}\,R$ is its convergence. We can thus think of $f{:}\,R\rightharpoonup\mathbf{1}$ as a partial decision algorithm computing with R-inputs or a *computable test* on R, with convergence announcing success. We then consider an $x{:}\,R$ to be distinguishable from $y{:}\,R$ if there is a computable test converging on x and diverging on y. Thus, an $x{:}\,R$ has computationally no more information than $y{:}\,R$ if x is not computationally distinguishable from y.

Definition 2.3.1 Given any per R, the *intrinsic preorder* $\leq_R$ on $x,y{:}\,R$ is given by

$$x\leq_R y \text{ iff } \forall f{:}\,R\rightharpoonup\mathbf{1}.\ f\cdot x\!\downarrow \text{ implies } f\cdot y\!\downarrow$$

For $[x]_R,[y]_R\in[R]$, we define $[x]_R\leq_{[R]}[y]_R$ if $x\leq_R y$.

Note that since $\leq_R$ trivially contains R, the relation $\leq_{[R]}$ is well-defined. It is also easy to verify that $\leq_{[R]}$ defines a preorder on $[R]$. This abstract definition of an intrinsic preorder specialises to natural orderings for concrete pers. For example, the ordering on the per N is discrete, *i.e.*, $x\leq_N y$ iff $x=y$, and the intrinsic preorder for $N\rightharpoonup N$ is the pointwise order, *i.e.*, $x\leq_{N\rightharpoonup N} y$ iff $\forall n.\ x\cdot n\!\downarrow\Rightarrow\ x\cdot n=y\cdot n$ (this is proved more generally as Proposition 2.4.4). More importantly, an immediate consequence of the definition of the intrinsic preorder is that all morphisms of the category **Per** are monotone with respect to this preorder:

Proposition 2.3.2 (Montonicity) *Let $R,S\subseteq\mathcal{N}\times\mathcal{N}$ be any pers and $f{:}\,R\rightharpoonup S$. Then f is monotonic, i.e., for all $x,y{:}\,R$, if $x\leq_R y$ then $f\cdot x\!\downarrow$ implies $f\cdot y\!\downarrow$ with*

$f \cdot x \leq_S f \cdot y$. *In particular, if* $f\colon [R] \to [S]$ *is a morphism of* **Per** *then for all* $a, b \in [R]$, *if* $a \leq_{[R]} b$ *then* $f(a) \leq_{[S]} f(b)$.

Proof Consider any $f\colon R \rightharpoonup S$ and suppose that $x \leq_R y$. Let F be the index of the function described as follows:

$$F \cdot n \quad = \quad \text{if } (f \cdot n)\downarrow \text{ then } 0$$

Since $F\colon R \rightharpoonup \mathbf{1}$ it follows that $F \cdot x\downarrow \Rightarrow F \cdot y\downarrow$ which implies that $f \cdot x\downarrow \Rightarrow f \cdot y\downarrow$. Now, consider any $g\colon S \rightharpoonup \mathbf{1}$. Consider the function $G\colon R \rightharpoonup \mathbf{1}$ defined by $G \cdot n = g \cdot (f \cdot n)$. Since $x \leq_R y$, we have that $G \cdot x\downarrow \Rightarrow G \cdot y\downarrow$ which implies that $g \cdot (f \cdot x)\downarrow \Rightarrow g \cdot (f \cdot y)\downarrow$. This gives us that $f \cdot x \leq_S f \cdot y$. ∎

In fact, $\leq_R$ can be proved to be the greatest preorder containing R for which we have the monotonicity property given by Proposition 2.3.2 — this gives us a more precise justification for our definition of the intrinsic preorder.

Once we have a preorder on each per, we can easily identify a class of cpo-like pers. Specifically, we take the pers where the intrinsic preorder is a complete partial order, with two qualifications. First, we insist on the existence of least upper bounds of only "effective" increasing chains; otherwise, we would exclude most pers of interest. For example, the per $N \to N$ only contains the recursive functions, but any total function on the natural numbers is expressible as a least upper bound of its finite approximations all of which are elements of $N \to N$. Second, we assert the existence of least upper bounds to be constructive, *i.e.*, that the least upper bound is computable as a function of the effective chain. Both these qualifications are merely a consequence of considering pers whose intrinsic orderings are complete partial orders in the internal language of the topos **Per**. This leads us to the definition of an "effective" cpo.

For any per R, define the per $Seq(R)$ of *effective increasing sequences* as the largest subset of the per $N \to R$ such that $s\colon Seq(R)$ iff for all $n \in \mathcal{N}$, $s \cdot n \leq_R s \cdot (n+1)$. We call a function (possibly partial) from $[Seq(R)]$ to $[R]$ a *supremum* function if it maps an element $[s]$ of $[Seq(R)]$ to a least upper bound of $\{[s \cdot n]_R \mid n \in \mathcal{N}\}$ with respect to the preorder $\leq_{[R]}$ if it exists and undefined otherwise. If the preorder $\leq_{[R]}$ on $[R]$ is antisymmetric, then there is a unique supremum function; in this case, we denote it by sup_R. We define a per R to be an *effective cpo* if it satisfies the following two conditions.

Antisymmetry For $a, b \in [R]$, if $a \leq_{[R]} b$ and $b \leq_{[R]} a$ then $a = b$.

Computable Completeness
The supremum function $sup_R\colon [Seq(R)] \to [R]$ is a morphism in **Per**, *i.e.*, it is total and effective.

Since every effective cpo is a per over the natural numbers, there are two natural morphisms from one effective cpo to another: the morphisms on the pers and the morphisms that are continuous with respect to the cpo structure. Remarkably, an important consequence of working with effective cpos is that all morphisms on the underlying pers are automatically continuous. To establish this connection, we first prove the following property of computable tests:

Lemma 2.3.3 *Let R be an effective cpo and let $f\colon R \rightharpoonup \mathbf{1}$ be any computable test. Then for any increasing sequence $s\colon Seq(R)$, and any $x \in sup_R([s]_{Seq(R)})$, $f \cdot x\downarrow$ iff there is some n with $f \cdot (s \cdot n)\downarrow$.*

Proof The implication to the left follows easily from the monotonicity property established in Proposition 2.3.2. For the implication to the right, suppose that $f \cdot x\downarrow$ and for the purpose of deriving a contradiction, suppose that $f \cdot (s \cdot n)\uparrow$ for all n. We will derive a contradiction by showing that $\bar{K} = \{z \mid z \cdot z\uparrow\}$ is r.e. For any $z \in \mathcal{N}$, let $t_z\colon Seq(R)$ be the index of the following algorithm:

$$\begin{aligned} t_z \cdot n = {} & \text{compute } z \cdot z \text{ for } n \text{ steps;} \\ & \text{if } z \cdot z \text{ halts in } i \leq n \text{ steps then return } s \cdot i \text{ else return } s \cdot n \end{aligned}$$

Note that $sup_R(t_z)$ is $[s \cdot n]_R$ for some n, if $z \in K$ and is $sup_R(s)$ otherwise. Let m be any realizer for sup_R. Then the function mapping z to $f \cdot (m \cdot t_z)$ is computable and is an r.e. test for membership in $\bar{K}$. ∎

Using Lemma 2.3.3 we can immediately prove the continuity of all computable functions between effective cpos:

Lemma 2.3.4 (Continuity) *Suppose that R, S are effective cpos. Then any effective $f\colon[R]\to[S]$ is continuous, i.e., if $s\colon Seq(R)$ then*

$$f(sup_R([s]_{Seq(R)})) \quad = \quad sup_S(f \circ [s]^{N,R})$$

By analogy with ordinary cpos, we say that an effective cpo R is *pointed* if $[R]$ contains a least element with respect to the intrinsic order $\leq_{[R]}$; we denote its least equivalence class by $\perp_R$. By the usual argument for obtaining least fixed points in cpos, and noting the effectivity of the operations involved, we have the following lemma:

Lemma 2.3.5 (Fixed Points) *Suppose that R is a pointed effective cpo. Every effective $f\colon[R]\to[R]$ has a least fixed point with respect to the ordering $\leq_{[R]}$. The function $fix_R\colon[R\to R]\to[R]$ mapping realizers for morphisms to their least fixed point is effective and is effectively realizable from sup_R and $\perp_R$.*

We have seen that the assumptions of antisymmetry and computability of least upper bounds were crucial in our arguments. In particular, computability of least upper bounds was important both for establishing the continuity of morphisms as well as the realizability of the fixed point operator. However, a natural question to ask is whether we have been overgenerous in the structure on pers that we have allowed ourselves. For example, it may well be that the existence of least upper bounds automatically guarantees their computability or that the intrinsic preorder is always antisymmetric. However, we can prove that this is not the case:

Lemma 2.3.6 (Independence of Axioms)

1. *There is a per R whose intrinsic order $\leq_{[R]}$ is not antisymmetric.*
2. *There is a pointed per S whose intrinsic order $\leq_{[S]}$ is antisymmetric, for which sup_S is total but not effective.*
3. *Both R and S cannot admit an effective fixed point operator.*

Proof Sketch

1. Consider the set $\mathit{Tot} = \{n \mid \varphi_n \text{ is a total function}\}$. Take R to be the per $\{\langle x, y\rangle \mid x \in \mathit{Tot} \text{ iff } y \in \mathit{Tot}\}$. Since Tot is neither r.e. nor co-r.e., we have that $[t]_R \leq_{[R]} [p]_R$ and $[p]_R \leq_{[R]} [t]_R$ where $t \in \mathit{Tot}, p \notin \mathit{Tot}$.

2. Take S to be the per defined by

$$\begin{aligned} f\ S\ g \quad \text{iff} \quad & \forall x \in \mathcal{N}\ f \cdot x \simeq g \cdot x, \text{ and} \\ & (\forall x \geq 1.\ f \cdot x\downarrow) \Leftrightarrow f \cdot 0 = 0 \Leftrightarrow f \cdot 0\downarrow \end{aligned}$$

   Intuitively, S consists of partial functions that are either not total and are undefined on 0 or total and map 0 to 0. One can show that $f \leq_S g$ iff $\forall x\ f \cdot x\downarrow \Rightarrow f \cdot x = g \cdot x$, from which one gets antisymmetry and the existence of all least upper bounds. Consider the sequence $s\colon \mathit{Seq}(S)$ defined by

$$s \cdot n \cdot x = \text{if } 0 < x \leq n \text{ then return } 0 \text{ else diverge}$$

   Then from the ordering on S, we have that $\mathit{sup}_S([s])$ is the equivalence class of the realizer of the function mapping all arguments to 0. Then $F\colon S \rightharpoonup \mathbf{1}$ defined by $F \cdot x = x \cdot 0$ contradicts Lemma 2.3.3 thus contradicting the computability of sup_S.

3. Trivially, R does not even have least fixed points (consider the identity function). For S, consider the index $t_z\colon (S \to S)$ of the following algorithm, for any $z \in \mathcal{N}$.

$$\begin{aligned} t_z \cdot f \cdot x = {} & \text{if } x = 0 \text{ then return } f \cdot 0 \\ & \quad \text{else compute } z \cdot z \text{ for } x \text{ steps;} \\ & \quad\quad \text{if } z \cdot z \text{ halts in } i \leq x \text{ steps then return } f \cdot x \text{ else return } 0 \end{aligned}$$

   Note that the least fixed point of t_z with respect to $\leq_S$ is the total function mapping every input to 0 if $z \in \bar{K}$ and is a partial function otherwise (and hence undefined on 0). If fix_S were computable then the function mapping z to $\mathit{fix}_S(t_z) \cdot 0$ is an r.e. test for $\bar{K}$.

■

2.4 Basic Types, Lifting, Products and Function Spaces

In Section 2.3, we have established the main properties that distinguish effective cpos from arbirary pers, namely, every morphism is continuous and the least fixed point operator exists and is effective. In this section, we show that basic types are effective cpos and the class of effective cpos is closed under the operations of cartesian products and function spaces. An important aspect of the closure of effective cpos under these operations is that it is constructive, *i.e.*, the supremum operator for the product or function space pers of R, S can be computed from the supremum operator for R and S. This property will be important for interpreting universal quantification as well as recursion over these types. We also indicate when the constructors yield pointed cpos; this will be important for interpreting recursive types.

Let $B \subseteq \mathcal{N}$. Then the intrinsic order on the per D_B (defined in Section 2.2) is discrete; for any $m, n \in B, m \neq n$, we have that $m \not\leq_{D_B} n$ since the index of the computable function

$$\lambda x \in \mathcal{N}.\ \text{if } x = m \text{ then return } 0 \text{ else diverge}$$

is an effective partial map from D_B to $\mathbf{1}$ and distinguishes m from n. It is straightforward to see that any per R with $\leq_R$ discrete is an effective cpo since it is trivially antisymmetric and the least upper bound of any computable increasing sequence s is just $s \cdot 0$. Thus, in particular, $N, \mathbf{1}$ are effective cpos. The per D_B is pointed iff $|B| = 1$.

The proof that the various constructors on pers that are effective cpos yield effective cpos relies on characterizing the intrinsic ordering.

Proposition 2.4.1 (Product Spaces) *If R, S are pers then $x \leq_{R\times S} y$ iff $(x)_1 \leq_R (y)_1$ and $(x)_2 \leq_S (y)_2$. Thus, if R and S are effective cpos then $R \times S$ is an effective cpo, with $sup_{R\times S}$ effectively realizable from sup_R and sup_S. If R, S are pointed then $R \times S$ is pointed with $\perp_{R\times S}$ effectively realizable from $\perp_R, \perp_S$.*

The proof of Proposition 2.4.1 is analogous to that given in [AP90]. While the intrinsic ordering being pointwise for products does not depend on the pers being effective cpos, there is something subtle about the pointwise intrinsic ordering provable for function spaces in that they crucially depend on the range being an effective cpo. To characterize the intrinsic ordering of function spaces, we first need the following lemma. The second part of the lemma will be useful in analyzing polymorphic types.

Lemma 2.4.2 *For any effective cpo R, there is an index $cnvg_R$ such that for all $z \in \mathcal{N}$ and $a \leq_R b$*

$$\begin{aligned} z \in \bar{K} \quad &implies \quad (cnvg_R \cdot z \cdot a \cdot b)\ R\ a \\ z \in K \quad &implies \quad (cnvg \cdot z \cdot a \cdot b)\ R\ b \end{aligned}$$

Moreover, $cnvg_R$ is effectively realizable from the supremum function sup_R.

Proof Given $z \in \mathcal{N}$ and $a \leq_R b$, let $s_{z,a,b}\colon Seq(R)$ be defined by $s_{z,a,b} \cdot n = b$ if $z \cdot z\downarrow$ in $\leq n$ steps and a otherwise. If c is a realizer for sup_R, then $cnvg_R$ returns $c \cdot s_{z,a,b}$. ■

Proposition 2.4.3 (Total Function Spaces) *If R, S are pers and S is an effective cpo, then $f \leq_{R \to S} g$ iff $\forall x\colon A. f \cdot x \leq_S g \cdot x$. Thus, if R and S are effective cpos then $R \to S$ is an effective cpo, with $sup_{R \to S}$ effectively realizable from sup_S. If S is pointed then $R \to S$ is pointed with $\perp_{R \to S}$ effectively realizable from $\perp_S$.*

Proof We first prove the characterization of the intrinsic ordering. The implication from left to right is obvious by the computability and hence monotonicity (by Proposition 2.3.2) of application to any x. For the reverse implication, suppose that $\forall x\colon R. f \cdot x \leq_S g \cdot x$ and $f \not\leq_{R \to S} g$. Then we have a computable test $h\colon (R \to S) \rightharpoonup \mathbf{1}$ with $h \cdot f\downarrow$ and $h \cdot g\uparrow$. For any $z \in \mathcal{N}$, we can define $k_z\colon R \to S$ by $k_z \cdot x = cnvg_S \cdot z \cdot (f \cdot x) \cdot (g \cdot x)$. Note that if $z \in \bar{K}$ then $k_z\,(R \to S)\,f$ and $k_z\,(R \to S)\,g$ otherwise. Thus the function mapping z to $h \cdot k_z$ is computable and is an r.e. test for membership in $\bar{K}$.

Using the intrinsic order, antisymmetry of $\leq_{R \to S}$ follows from the antisymmetry of $\leq_S$. For any sequence $s\colon Seq(R \to S)$, we can compute $sup_{R \to S}(s)$ "pointwise" by returning an index of the function mapping x to sup_S of the sequence $\lambda n \in \mathcal{N}.\, s \cdot n \cdot x$. The least element $\perp_{R \to S}$ is realized by the index of the function $\lambda x \in \mathcal{N}.n$ where n realizes $\perp_S$. ■

Analogously, we have the following proposition for partial function spaces.

Proposition 2.4.4 (Partial Function Spaces) *If R, S are pers and S is an effective cpo, then $f \leq_{R \rightharpoonup S} g$ iff $\forall x{:}\, A. f \cdot x \downarrow \Rightarrow f \cdot x \leq_S g \cdot x$ Thus, if R and S are effective cpos then $R \rightharpoonup S$ is an effective cpo, with $sup_{R \rightharpoonup S}$ computable from sup_S. The per $R \rightharpoonup S$ is always pointed with $\perp_{R \rightharpoonup S}$ realized by the index of the function "$\lambda x \in \mathcal{N}.diverge$" independently of R, S.*

Since lifting is defined using partial functions, it follows that effective cpos are closed under lifting as well.

2.5 Polymorphism

The main twist is that we do not interpret polymorphism as intersection; indeed, the intersection of even constructive families of effective cpos does not always yield an effective cpo. However, if we interpret $\forall X. \ \ldots$ as a formula with X quantified over effective cpos, then a realizer for this is a function from proofs of "X is an effective cpo". The proofs of this latter statement are just the realizers for its supremum operator — we can thus interpret polymorphic types as functions on supremum operators. Interpreting universal quantification in this way is not in contradiction to the usual interpretation as intersection; in such models, the supremum operator for all the pers over which the quantification is done has a single common realizer (*e.g.*, [AP90]) and the functional dependence on the supremum operator then becomes vacuous.

Let ECpo and PECpo be the set of all effective cpos and pointed effective cpos respectively. Define an *effective set* to be a set K equipped with an onto binary relation $\vdash_K \subseteq \mathcal{N} \times K$. For any function $\mathcal{C}{:}\, K \to \mathrm{ECpo}$, we can define the per $\forall^K(\mathcal{C})$ as follows:

$$f \,\forall^K(\mathcal{C})\, g \quad \text{iff} \quad \forall a \in K, x, y \in \mathcal{N} \quad x, y \vdash_K a \Rightarrow (f \cdot x)\, \mathcal{C}(a)\, (g \cdot y)$$

It is easy to see that $\forall^K(\mathcal{C})$ is symmetric and transitive, *i.e.*, that it is a per. We take ECpo to be an effective set with $n \vdash_{\mathrm{ECpo}} R$ iff $n \vdash_{Seq(R),R} sup_R$. And, we consider PECpo an effective set with $n \vdash_{\mathrm{PECpo}} R$ iff $(n)_1 \vdash_{\mathrm{ECpo}} R$ and $(n)_2 \vdash_R \perp_R$. Essentially the realizers for ECpo and PECpo are proofs, in the internal language of the effective topos, of the membership of a per in the particular subset. The following lemma shows that if $\mathcal{C}$ is an effective map (as defined in Section 2.2) from K to ECpo then $\forall^K(\mathcal{C})$ is an effective cpo. As usual, we also give the conditions under which it is pointed.

Lemma 2.5.1 (Higher-Order Polymorphism) *Let $\mathcal{C}: K \to \mathrm{ECpo}$ be an effective map. Then*

$$f \leq_{\forall^K(\mathcal{C})} g \textit{ iff } \forall a \in K, x \in \mathcal{N} \quad x \vdash_K a \Rightarrow (f \cdot x) \leq_{\mathcal{C}(a)} (g \cdot x)$$

Thus, $\forall^K(\mathcal{C})$ is an effective cpo with $sup_{\forall^K(\mathcal{C})}$ effectively realizable from $\mathcal{C}$. If $\mathcal{C}$ induces an effective map from K to PECpo *then $\forall^K(\mathcal{C})$ is pointed with $\perp_{\forall^K(\mathcal{C})}$ effectively realizable from the effective map $\mathcal{C}: K \to \mathrm{PECpo}$.*

Proof The proof is similar to that for Proposition 2.4.3; we only remark on the modifications. By the second part of Lemma 2.4.2, we have a realizer *cnvg* such that $cnvg \cdot y \vdash cnvg_S$, for any effective cpo S and natural number $y \vdash sup_S$. If c is a realizer for $\mathcal{C}$, we use $cnvg \cdot (c \cdot x)$ instead of sup_S in the definition of $k_z \cdot x$, proving the characterization of the intrinsic ordering. For any sequence $s: Seq(\forall^K(\mathcal{C}))$, we can compute $sup_{\forall^K(\mathcal{C})}(s)$ by returning an index of the function mapping x to $(c \cdot x) \cdot (\lambda n \in \mathcal{N}.\, s \cdot n \cdot x)$; it is thus computable from c. The statement about pointedness is easily obtained from the characterization of the intrinsic ordering. ∎

As particular instances of Lemma 2.5.1, we can define constructors $\forall^{\mathrm{ECpo}}$ which we denote $\forall$, and $\forall^{\mathrm{PECpo}}$ which we denote $\forall^p$, yielding effective cpos.

Interpreting polymorphism as intersection on pers has had great success in being able to prove various parametricity principles. Given this completely different interpretation of polymorphism, it is natural to ask whether such principles are validated in our model. The following proposition shows that in this semantics of polymorphism as well, the type $\forall t.t$ is empty and $\forall t.(t \rightarrow t)$ only contains the identity function. It thus suggests that many of the usual parametricity principles are validated even with this less "uniform" interpretation of polymorphism. Essentially, parametricity in our context arises from the fact that the realization relations $\vdash_{\mathrm{ECpo}}, \vdash_{\mathrm{PECpo}}$ are *not functions* — thus, any element of a polymorphic type must behave uniformly on all types that admit common realizers for their supremum functions.

Proposition 2.5.2 (Parametricity) *For $\mathcal{I}: ECpo \rightarrow ECpo$ defined by $\mathcal{I}(R) = R$, there is no n with $n{:}\forall(\mathcal{I})$. For $\mathcal{F}: ECpo \rightarrow ECpo$ defined by $\mathcal{F}(R) = R \rightarrow R$, if $n{:}\forall(\mathcal{F})$ then for any effective cpo R, natural number x, if $x \vdash sup_R$ then $n \cdot x \vdash_R id_R$.*

Turning next to the $\forall^p$ quantifier, consider the map $\mathcal{C}: \mathrm{PECpo} \rightarrow \mathrm{ECpo}$ given by

$$\mathcal{C}(R) = (R \rightarrow R) \rightarrow R$$

which is effective by Proposition 2.4.3. By Lemma 2.3.5 we have a realizer $\mathit{fix}{:}\forall^p(\mathcal{C})$, which computes the least fixed point at every pointed type. The following proposition shows that this is the *only* fixed point operator of this polymorphic type:

Proposition 2.5.3 (Polymorphic Fixed Point) *Consider the effective map $\mathcal{C}$ from* PECpo *to* ECpo *given by $\mathcal{C}(R) = (R \rightarrow R) \rightarrow R$. Then we have a realizer $\mathit{fix}{:}\forall^p(\mathcal{C})$ computing the least fixed point at every pointed type. Further, suppose that we have an $f{:}\forall^p(\mathcal{C})$ such that for any pointed effective cpo R, and $x \vdash sup_R, b \vdash \bot_R$ we have that $(f \cdot pr(x, b)) = n$ such that for any $g: R \rightarrow R$, $n \cdot g \; R \; g \cdot (n \cdot g)$, that is, f is a fixed point operator. Then $f \; \forall^p(\mathcal{C}) \; \mathit{fix}$.*

Just like Proposition 2.5.2, we can prove that the type $\forall^p t.t$ has exactly one element and the type $\forall^p t.(t \rightarrow t)$ has exactly two elements; the polymorphic operator $\forall^p$ thus supports the parametricity principles of system **F** extended with recursion on values.

2.6 Recursive Types

We are interested in solving recursive equations over effective cpos defined using the constructions on effective cpos that we have described so far. A standard way to solve such equations is given by [SP82] which asserts the existence of initial F-algebras for functors $F: \mathbf{C} \to \mathbf{C}$ under suitable conditions on the category $\mathbf{C}$ and functor F. Unfortunately, these results are not directly applicable to our setting since the category of effective cpos does not satisfy these conditions. For example, we do not have all $\boldsymbol{\omega}^{\mathrm{op}}$-limits; one can define the limit-per by the usual inverse-limit construction of domain theory, but this limit-per is not necessarily an effective cpo. The solution is to consider everything "effectively" and reformulate the framework and results of [SP82] for a suitable notion of effectivity. The main departure from earlier work in this regard (*e.g.*, [AP90]) is that effectiveness of a functor is a condition on the behaviour of a functor on objects in addition to arrows.

Let $\mathbf{C}$ be a category. We use $Obj(\mathbf{C})$ and $Morph(\mathbf{C})$ to denote the objects and morphisms respectively, and for any objects A, B we denote the homset by $\mathbf{C}(A, B)$. We assume that $Obj(\mathbf{C})$ and $Morph(\mathbf{C})$ are disjoint sets. The following is the definition of an effective category; note the difference between the conditions on the realization relation for morphisms and objects — in essence we require each homset to be a per but the class of objects does not necessarily have to be one; indeed, it doesn't even have to be a proper set.

Definition 2.6.1 An *effective* category $\mathbf{C}$ is a category equipped with a binary relation

$$\vdash_{\mathbf{C}} \subseteq \mathcal{N} \times (Obj(\mathbf{C}) \cup Morph(\mathbf{C}))$$

such that

1. $\vdash_{\mathbf{C}} \cap (\mathcal{N} \times Obj(\mathbf{C}))$ is an onto relation between $\mathcal{N}$ and $Obj(\mathbf{C})$
2. For any objects A, B the relation $\vdash_{\mathbf{C}} \cap (\mathcal{N} \times \mathbf{C}(A, B))$ is an onto function between $\mathcal{N}$ and $\mathbf{C}(A, B)$
3. For any object A the identity map id_A is effectively realizable from A.
4. For any morphisms $B \xrightarrow{f} C$, $A \xrightarrow{g} B$, the composition $f \circ g$ is effectively realizable from f, g.

Essentially, an effective category $\mathbf{C}$ is one equipped with a realization relation, $\vdash_{\mathbf{C}}$, on objects and morphisms that is *onto on objects* and an *onto function on each hom-set*, with identity morphisms effectively realizable from the objects and composition effective.

In the following, $\mathbf{C}$, $\mathbf{D}$ denote effective categories. Then, with the pairing function pr on natural numbers, $\mathbf{C} \times \mathbf{D}$ is an effective category if $\mathbf{C}$,$\mathbf{D}$ are effective categories. Define an effective functor $F\colon \mathbf{C} \to \mathbf{D}$ to be a functor such that for any object A, the object $F(A)$ is effectively realizable from A, and for any morphism f, the morphism $F(f)$ is effectively realizable from f; the realizer for an effective functor is the realizer for its object and morphism maps. For any functors $F, G\colon \mathbf{C} \to \mathbf{D}$, define an effective natural transformation $\eta\colon F \dot{\longrightarrow} G$ to be a natural transformation such that for any object A of $\mathbf{C}$, the morphism η_A in $\mathbf{D}$ is effectively realizable from A; the realizer for η is the realizer for the function mapping A to η_A. With realizers for effective functors and effective natural transformations, we can define the effective category $\mathbf{D}^{\mathbf{C}}$ of effective functors and effective natural transformations between effective functors. Define an effective $\boldsymbol{\omega}$-chain in $\mathbf{C}$ to be a diagram of the form

$$\Delta = A_0 \stackrel{f_0}{\to} A_1 \stackrel{f_1}{\to} \cdots$$

with A_i, f_i effective as functions of i (*i.e.*, an effective functor from $\boldsymbol{\omega}$ to $\mathbf{C}$); the realizer for Δ is the pair of realizers of the functions mapping i to A_i and f_i. Take the realizer for a cone $\nu\colon \Delta \to A$ to be the pair of the realizer for A and the realizer for the function mapping i to the morphism $A_i \stackrel{\nu_i}{\to} A$; a cone is effective if it is realizable. Define an effective colimit for an $\boldsymbol{\omega}$-chain Δ to be an effective cone ν over Δ such that for any other effective cone ν' there is a unique mediating morphism from the vertex of ν to the vertex of ν' with the mediating morphism effectively realizable from ν'. ; a realizer for an effective colimit is its realizer as an effective cone together with a realizer for the function mapping any effective cone to the mediating morphism. Analogously one can define effective $\boldsymbol{\omega}^{\mathrm{op}}$ chains and their effective limits. We say that the category $\mathbf{C}$ is *effectively* $\boldsymbol{\omega}^{\mathrm{op}}$*-complete* if every effective $\boldsymbol{\omega}^{\mathrm{op}}$ chain has an effective limit with the limit of every effective $\boldsymbol{\omega}^{\mathrm{op}}$ chain effectively realizable from the chain.

Note that for any effective category $\mathbf{C}$, we can define a per corresponding to each homset $\mathbf{C}(A, B)$, as in Section 2.2. We say that $\mathbf{C}$ is an **ECpo**-enriched category if the per corresponding to each homset $\mathbf{C}(A, B)$ is an effective cpo with its supremum function effectively realizable from the objects A, B. By Proposition 2.3.4, any effective functor between two **Ecpo**-enriched categories is automatically locally continuous in the sense of [SP82]. In an **ECpo**-enriched category $\mathbf{C}$, we define the pair $\langle e, p\rangle$ to be an embedding-projection pair from A to B, if $A \stackrel{e}{\to} B \stackrel{p}{\to} A$ are morphisms such that $p \circ e = id_A$ and $e \circ p \leq_{B,B} id_B$ where $\leq_{B,B}$ is the intrinsic ordering on the per corresponding to the homset $\mathbf{C}(B, B)$. Define the category $\mathbf{C}^{\mathrm{ep}}$ whose objects are those of $\mathbf{C}$ and whose morphisms are embedding-projection pairs. While the embeddings and projections determine each other uniquely, they may not necessarily do so effectively; it therefore does not suffice to consider the subcategory of only embeddings or projections. Clearly, $\mathbf{C}^{\mathrm{ep}}$ is an effective category with realizers inherited from $\mathbf{C}$. Call an effective category $\mathbf{C}$ a **PECpo**-enriched category if it is an **ECpo**-enriched category and the per corresponding to each homset $\mathbf{C}(A, B)$ is pointed with its least element, denoted by $\perp_{A,B}$, effectively realizable from objects A, B.

Lemma 2.6.2 (Recursive Solutions) *Suppose that* $\mathbf{C}$ *is an effectively* $\boldsymbol{\omega}^{\mathrm{op}}$*-complete,* **PECpo**-*enriched category with a terminal object* $\perp$ *and such that composition in* $\mathbf{C}$ *is left-strict,* i.e., *for any* $A \stackrel{f}{\to} B$ *we have* $\perp_{B,C} \circ f = \perp_{A,C}$. *Then for any effective functor* $F\colon \mathbf{C}^{\mathrm{ep}} \to \mathbf{C}^{\mathrm{ep}}$, *we have an object* $\mu(F)$ *in* $\mathbf{C}$ *and morphisms*

$$fold^F\colon F(\mu(F)) \to \mu(F), \;\; unfold^F\colon \mu(F) \to F(\mu(F))$$

with $fold^F, unfold^F$ *constituting an isomorphism pair in* $\mathbf{C}$. *Moreover, the object* $\mu(F)$ *and the morphisms* $fold^F, unfold^F$ *are effectively realizable from the effective functor* F.

Proof Sketch As in [SP82], the terminal object $\perp$ of $\mathbf{C}$ is an initial object of $\mathbf{C}^{\mathrm{ep}}$; however, the important point to note is that since $\mathbf{C}$ is **ECpo**-enriched, it is in fact effectively initial; the unique morphism to any object A which is the least element of the homset from the terminal object into A is effectively realizable from A. Taking a realizer for an $\mathbf{O}$-colimit (in the sense of [SP82]) of an $\boldsymbol{\omega}$-chain to be simply its realizer as a cone, one obtains *effective* correspondence between $\mathbf{O}$-colimits and effective $\boldsymbol{\omega}$-colimits of effective $\boldsymbol{\omega}$-chains in $\mathbf{C}^{\mathrm{ep}}$; thus the effective $\boldsymbol{\omega}^{\mathrm{op}}$-completeness of $\mathbf{C}$ implies that $\mathbf{C}^{\mathrm{ep}}$ is effectively $\boldsymbol{\omega}$-cocomplete. Since F is automatically $\mathbf{ECpo}$-enriched and thus $\boldsymbol{\omega}$-continous, the vertex $\mu(F)$ of the colimit of $\Delta_F = \perp \xrightarrow{!} F(\perp) \xrightarrow{F(!)} \cdots$ gives the initial F-algebra. Since the morphism from $F(\mu(F))$ to $\mu(F)$ of the initial F-algebra is an embedding-projection pair, they determine an isomorphism between $F(\mu(F))$ and $\mu(F)$. The colimit of Δ_F in $\mathbf{C}^{\mathrm{ep}}$ is effectively realizable from Δ_F (by effective $\boldsymbol{\omega}$-cocompletness of $\mathbf{C}^{\mathrm{ep}}$) which means that the object $\mu(F)$ is realizable from Δ_F. The morphism part of $\mu(F)$ as an initial F-algebra is a mediating morphism from $F(\mu(F))$ as a colimit of $F(\Delta_F)$. The cone of $F(\mu(F))$ is realizable from F and Δ_F which is thus its realizer as an $\mathbf{O}$-colimit from which one can effectively obtain its realizer as an effective-colimit giving effectively the mediating morphism. The $\boldsymbol{\omega}$-chain Δ_F in $\mathbf{C}^{\mathrm{ep}}$ in turn is realizable from F since the morphism ! is realizable from F by the effective initiality of $\perp$, thus giving us that $\mu(F)$ and the isomorphism is effectively realizable from F. ∎

To apply the lemmas to our particular setting, define $\mathbf{ECpo}$ to be the full subcategory of $\mathbf{Per}$. The realization relation on objects of $\mathbf{ECpo}$ is as in Section 2.5 for ECpo and on morphisms as in Section 2.2. In more detail, for an object R of $\mathbf{ECpo}$, *i.e.*, an effective cpo, $n \vdash_{\mathbf{ECpo}} R$ iff $n \vdash_{\mathrm{ECpo}} R$ (defined in Section 2.5). For morphisms $R \xrightarrow{f} S$, define $n \vdash_{\mathbf{ECpo}} f$ iff $n \vdash_{R,S} f$ ($\vdash_{R,S}$ as defined in Section 2.2). Clearly, $\mathbf{ECpo}$ is an effective category; by Proposition 2.4.3, $\mathbf{ECpo}$ is an $\mathbf{ECpo}$-enriched category. Define $\mathbf{PECpo}$ to be the full subcategory of $\mathbf{ECpo}$ of pointed effective cpos with the realization relation on objects as in Section 2.5 for PECpo. By the last part of Proposition 2.4.3, $\mathbf{PECpo}$ is a $\mathbf{PECpo}$-enriched category. Essentially then, the realizers for the subcategories $\mathbf{ECpo}$ and $\mathbf{PECpo}$ are proofs, in the internal language of the effective topos, of the membership of an object of $\mathbf{Per}$ in the particular subcategory.

Lemma 2.6.3 *The categories* $\mathbf{ECPo}$ *and* $\mathbf{PECpo}$ *are effectively* $\boldsymbol{\omega}^{\mathrm{op}}$*-complete.*

Proof The proof is essentially the "inverse-limit" construction; one just needs to be conscious of effectivity and intrinsic orderings. Let $\Delta = R_0 \xleftarrow{f_0} R_1 \xleftarrow{f_1} \cdots$ be an effective $\boldsymbol{\omega}^{\mathrm{op}}$-chain in **ECpo**. Define the per $\lim_{\leftarrow} \Delta$ as

$$f \lim_{\leftarrow} \Delta\, g \;\text{ iff }\; \forall i.\; f \cdot i\; R_i\; g \cdot i \text{ and } f_i([f \cdot (i+1)]_{R_{i+1}}) = [f \cdot i]_{R_i}$$

Analogously to Proposition 2.4.3, one can prove that $f \leq_{\lim_{\leftarrow} \Delta} g$ iff $\forall i.\ f \cdot i \leq_{R_i} g \cdot i$. The supremum operator for $\lim_{\leftarrow} \Delta$ is thus realizable by taking pointwise suprema using the realizer for Δ. Define the cone $\nu\colon \lim_{\leftarrow} \Delta \to \Delta$ as $\nu_i = [\lambda f \in \mathcal{N}.\ f \cdot i]^{\lim_{\leftarrow} R, R_i}$. Clearly, ν_i is effectively realizable as a function of i. Thus the cone ν is effectively realizable from Δ. For any other cone $\nu'\colon R \to \Delta$, define the morphism h from R to $\lim_{\leftarrow} \Delta$ by $h(a) = \lambda i.\ \nu'_i(a)$ which can be seen to be the unique mediating morphism from ν' to ν. Moreover, h is effectively realizable from ν', thus the ν as a colimit is effectively realizable from Δ. This shows that **ECPo** is effectively $\boldsymbol{\omega}^{\mathrm{op}}$-complete. To show a similar result for **PECpo** only needs to observe that the least element of $\lim_{\leftarrow} \Delta$ is $\lambda i.\ \perp_{R_i}$ which is effectively realizable from the least elements of R_i. ∎

The per **1** is a terminal object for **Per**; being an effective cpo and pointed, it is also a terminal object for **ECpo** and **PECpo**. One can also see that composition is left-strict in **ECpo** (for homsets that have least elements) and hence in **PECpo**. Hence, by Lemma 2.6.3, we can apply Lemma 2.6.2 to the category **PECpo**. Note however that this does not give us recursive solutions for any effective functor $F\colon \mathbf{ECpo}^{\mathrm{ep}} \to \mathbf{ECpo}^{\mathrm{ep}}$, but only for those that induce an effective functor from $\mathbf{PECpo}^{\mathrm{ep}}$ to $\mathbf{PECpo}^{\mathrm{ep}}$. Also note that because of different realization relations for **ECPo** and **PECpo**, for F to induce an effective functor from $\mathbf{PECpo}^{\mathrm{ep}}$, not only must $F(R)$ be a pointed effective cpo when R is one, but $\perp_{F(R)}$ must be effectively realizable from $\perp_R$. As the following example suggests, the condition on preservation of pointedness is necessary and not just an accident of our method of constructing recursive solutions.

Example: [Pointedness Preservation Essential] The per $R \to N$ is not necessarily pointed whenever R is, and we show that there can be no per A with A recursively isomorphic to $A \to N$ by showing that there is no recursive surjection from A to $A \to N$. Consider any morphism f from A to $A \to N$, and suppose that $n \vdash f$. Let g be the index of $\lambda x \in \mathcal{N}.((n \cdot x) \cdot x) + 1$; then clearly $g\colon (A \to N)$. By the usual diagonal argument, there is no $a \in [A]$ with $f(a) = [g]_{A \to N}$.

2.7 Subtyping

Suppose we have pers R, S with $R \subseteq S$ as relations. Since the index of $\lambda x \in \mathcal{N}.\, x$ realizes an effective map from R to S, we have by Proposition 2.3.2, that if $x \leq_R y$ then $x \leq_S y$. Thus, if $R \subseteq S$, then $Seq(R) \subseteq Seq(S)$. We define the relation that R is a subtype of S as follows:

Definition 2.7.1 Suppose that R, S are effective cpos. Then $R <: S$ iff

1. $R \subseteq S$,
2. For any $n\colon Seq(R)$, $sup_R([n]_{Seq(R)}) \subseteq sup_S([n]_{Seq(S)})$,
3. If R is pointed, then S is pointed with $\bot_R \subseteq \bot_S$.

Alternatively, condition 2 can be stated as that if $n \vdash sup_R$ and $s\colon Seq(R)$ then $n \cdot s \vdash sup_S(s)$. Condition 3 can be stated as that if R is pointed and $n \vdash_R \bot_R$ then S is pointed with $n \vdash_S \bot_S$. Essentially then, R is a subtype of S, if the proof of any "interesting property" of R is also a proof of S; the interesting properties in our context being membership in a per, equality in a per, and membership in the subcategories **ECpo**, **PECpo**. The following proposition asserts that with this notion of subtyping, one obtains all the expected properties of the subtyping relation:

Proposition 2.7.2 (Subtyping)

1. *If R is any effective cpo then $R <: R$. If R, S, T are effective cpos with $R <: S$ and $S <: T$ then $R <: T$.*
2. *If R, S, R', S' are effective cpos then*

$$\begin{array}{rcl}
R \times S <: R' \times S' & \textit{iff} & R <: R' \textit{ and } S <: S' \\
R \to S <: R' \to S' & \textit{iff} & R' <: R \textit{ and } S <: S' \\
R \rightharpoonup S <: R' \rightharpoonup S' & \textit{iff} & R' <: R \textit{ and } S <: S' \\
R_\bot <: R'_\bot & \textit{iff} & R <: R'
\end{array}$$

3. *If* $\mathcal{C}, \mathcal{C}' \colon K \to \mathrm{ECpo}$ *are effective maps on some effective set* K *then*

$$\forall^K(\mathcal{C}) <: \forall^K(\mathcal{C}') \textit{ iff for every } a \in K,\ \mathcal{C}(a) <: \mathcal{C}'(a)$$

4. *If* $F, F' \colon \mathbf{PECpo}^{\mathrm{ep}} \to \mathbf{PECpo}^{\mathrm{ep}}$ *are effective functors then*

$$\mu(F) <: \mu(F') \textit{ iff } F^n(\mathbf{1}) <: F'^n(\mathbf{1}),\ \textit{for every } n \in \mathcal{N}$$

In particular, if $R <: R' \Rightarrow F(R) <: F'(R')$ *then* $\mu(F) <: \mu(F')$.

Note, for example, that $R \to S$ is not a subtype $R \rightharpoonup S$ (because their least elements are different) which would be true if we were to take the subset interpretation. We can define the per $\mathit{Top} = \mathcal{N} \times \mathcal{N}$ as the supertype of all types. It is, of course, isomorphic to $\mathbf{1}$; however, as is typical, the notion of subtyping is not closed under isomorphism.

2.8 A Model of $F\omega$ with recursion and subtyping

We present our semantics as a categorical version of [BMM90] with additional structure for the distinction between types and pointed types, fixed points at pointed types, higher-order polymorphism and recursive solutions for maps from pointed kinds to pointed kinds. As a first approximation, we can take the category for kinds to be the category of all **ECpo**-enriched categories and effective functors, and take the categories **ECpo** and **PECpo** to interpret types and pointed types respectively. The problem with this is that type operators such as $\to$ cannot be expressed as covariant functors and are thus not in this category. Inspired by [AP90], we consider the class of doubles of categories and symmetric functors; our main technical contribution is that we show this to be cartesian closed, which allows us to give a categorical semantics of $\mathbf{F}\omega$ and recursive kinds.

2.8.1 Types

For any effective category $\mathbf{C}$, define the category $\mathbf{C}^D$ with the same objects as $\mathbf{C}$ and as morphisms $A \overset{\langle f,g\rangle}{\to} B$ where $A \overset{f}{\to} B \overset{g}{\to} A$ are morphisms in $\mathbf{C}$; clearly $\mathbf{C}^D$ inherits effectivity from $\mathbf{C}$. From Proposition 2.4.1, it follows that the intrinsic order on its homsets is $\langle f, g\rangle \leq \langle f', g'\rangle$ iff $f \leq f', g \leq g'$. For any morphism $\langle f, g\rangle$ in $\mathbf{C}^D$, set $\langle f, g\rangle^\dagger$ to be $\langle g, f\rangle$. We say that a functor from $\mathbf{C}^D$ to $\mathbf{D}^D$ is symmetric if $F(p^\dagger) = (F(p))^\dagger$; both covariant and contravariant functors on $\mathbf{C}$ can be expressed as (covariant) symmetric functors on $\mathbf{C}^D$. Clearly, $\mathbf{C}^{\mathrm{ep}}$ is a subcategory of $\mathbf{C}^D$. The main advantage of *symmetric* functors on $\mathbf{C}^D$ is that they preserve the embedding-projections pairs of $\mathbf{C}$ and hence any symmetric effective functor $F\colon \mathbf{C}^D \to \mathbf{C}^D$ induces an effective functor from the subcategory $\mathbf{C}^{\mathrm{ep}}$ to $\mathbf{C}^{\mathrm{ep}}$ and thus admits a recursive solution (if $\mathbf{C}$ has the properties stated in Lemma 2.6.2).

Let $\mathcal{K}$ be the category whose objects are $\mathbf{C}^D$ for $\mathbf{C}$ an effectively $\boldsymbol{\omega}^{\mathrm{op}}$-complete, **ECpo**-enriched category with terminal object and left-strict composition (which categories we henceforth abbreviate as "good" categories), and whose morphisms are effective symmetric functors. The category $\mathbf{1}$ with one object and just the identity map is trivially good and $\mathbf{1}^D$ is a terminal object of $\mathcal{K}$. For objects $\mathbf{C}^D, \mathbf{D}^D$ of $\mathcal{K}$, define the product object $\mathbf{C}^D \times^D \mathbf{D}^D$ to be $(\mathbf{C}\times\mathbf{D})^D$ where the $\times$ on the right is the product in **Cat**; it can be shown that $\mathbf{C}\times\mathbf{D}$ is good if $\mathbf{C}, \mathbf{D}$ are. The projection functors are symmetric and effective and $\mathbf{C}^D \times^{\mathcal{K}} \mathbf{D}^D$ can be shown to be the categorical product. For any effective category $\mathbf{C}$, define $\mathbf{Fst}\colon \mathbf{C}^D \to \mathbf{C}$ the forgetful functor mapping arrows to their first components. For any objects $\mathbf{C}^D, \mathbf{D}^D$, define $\mathbf{C}^D \Rightarrow \mathbf{D}^D$ to be the category whose objects are symmetric effective functors from $\mathbf{C}^D$ to $\mathbf{D}^D$ and morphisms from functor F to G are the effective natural transformations from $\mathbf{Fst}\circ F$ to $\mathbf{Fst}\circ G$. We can define the exponential object $\mathbf{C}^D {\Rightarrow}^D \mathbf{D}^D$ to be $(\mathbf{C} \Rightarrow \mathbf{D})^D$. The proof that this is an exponential object follows from the fact that for symmetric effective functors F, G if $\eta\colon F \stackrel{\cdot}{\longrightarrow} G$ and $\eta_A = \langle \eta^1_A, \eta^2_A\rangle$ then $\eta^1\colon \mathbf{Fst}\circ F \stackrel{\cdot}{\longrightarrow} \mathbf{Fst}\circ G$ and $\eta^2\colon \mathbf{Fst}\circ G \stackrel{\cdot}{\longrightarrow} \mathbf{Fst}\circ F$ are natural transformations; moreover any such pair of transformations is a natural transformation from G to F. Hence $\mathbf{C}^D {\Rightarrow}^D \mathbf{D}^D$ is isomorphic to the category of effective symmetric functors and effective natural transformations between them. It can be shown that $\mathbf{C}^D \Rightarrow \mathbf{D}^D$ is good, if $\mathbf{C}, \mathbf{D}$ are. In summary, we have the following theorem.

Theorem 2.8.1 *The category $\mathcal{K}$ is a cartesian closed category.*

We take types to be the object $\mathbf{ECpo}^D$ and pointed types to be $\mathbf{PECpo}^D$; we can show that the definitions of all the semantic constructions given in earlier sections on objects extend to symmetric effective functors. Their extension to the maps on arrows is standard; their effectivity on objects follows from Propositions 2.4.1, 2.4.3, 2.4.4, and Lemmas 2.5.1, 2.6.2. The following lemma describes the various constructions as type constructors as well as pointed type constructors.

Lemma 2.8.2

- *For any set $B \subseteq \mathcal{N}$, the per D_B is an effective cpo. It is pointed iff $|B| = 1$. Thus, we have effective symmetric functors*

$$N\colon \mathbf{1}^D \to \mathbf{ECpo}^D, \quad \mathbf{1}\colon \mathbf{1}^D \to \mathbf{ECpo}^D, \ \textit{with}\ \mathbf{1}\colon \mathbf{1}^D \to \mathbf{PECpo}^D$$

- *There is an effective symmetric product functor*

$$\times\colon \mathbf{ECpo}^D \overset{D}{\times} \mathbf{ECpo}^D \to \mathbf{ECpo}^D$$

that induces an effective symmetric functor

$$\times\colon \mathbf{PECpo}^D \overset{D}{\times} \mathbf{PECpo}^D \to \mathbf{PECpo}^D$$

- *There is an effective symmetric total function space functor*

$$\to\colon \mathbf{ECpo}^D \overset{D}{\times} \mathbf{ECpo}^D \to \mathbf{ECpo}^D$$

that induces an effective symmetric functor

$$\to\colon \mathbf{ECpo}^D \overset{D}{\times} \mathbf{PECpo}^D \to \mathbf{PECpo}^D$$

- *There is an effective symmetric partial function space functor*

$$\rightharpoonup\colon \mathbf{ECpo}^D \overset{D}{\times} \mathbf{ECpo}^D \to \mathbf{PECpo}^D$$

It thus trivially yields an effective symmetric functor

$$\rightharpoonup: \mathbf{ECpo}^D \overset{D}{\times} \mathbf{ECpo}^D \rightarrow \mathbf{ECpo}^D$$

We thus have an effective lifting functor $(\cdot)_\perp: \mathbf{ECpo}^D \rightarrow \mathbf{PECpo}^D$*.*

- *For any object* $\mathbf{C}^D$ *in* $\mathcal{K}$*, there is an effective symmetric functor*

  $$\forall^{\mathbf{C}}: (\mathbf{C}^D \Rightarrow^D \mathbf{ECpo}^D) \rightarrow \mathbf{ECpo}^D$$

  that induces an effective functor $\forall^{\mathbf{C}}: (\mathbf{C}^D \Rightarrow^D \mathbf{PECpo}^D) \rightarrow \mathbf{PECpo}^D$*. In particular, we have functors* $\forall^{\mathbf{ECpo}}$ *and* $\forall^{\mathbf{PECpo}}$*.*

- *For any object* $\mathbf{C}^D$ *in* $\mathcal{K}$*, if* $\mathbf{C}$ *is* $\mathbf{PECpo}$*-enriched, then the type recursion operator is an effective symmetric functor* $\mu^{\mathbf{C}}: (\mathbf{C}^D \Rightarrow^D \mathbf{C}^D) \rightarrow \mathbf{C}^D$*. In particular, we have a functor* $\mu^{\mathbf{PECpo}}: (\mathbf{PECpo}^D \Rightarrow^D \mathbf{PECpo}^D) \rightarrow \mathbf{PECpo}^D$ *that is trivially a functor into* $\mathbf{ECpo}^D$*.*

2.8.2 Terms

Having interpreted the type constructors as effective symmetric functors between doubles of categories, we *do not* interpret terms as natural transformations between them; syntactically, this corresponds to the fact that the meaning of every term of type B with free variable of type A is a map from A to B rather than a "double" map going both ways. For interpreting terms, we treat the type operators as functors ignoring the "double" on their domain and range; thus, we use functors

$$\begin{array}{ll}
\mathbf{1}: \mathbf{1} \rightarrow \mathbf{ECpo} & \times: \mathbf{ECpo} \times \mathbf{ECpo} \rightarrow \mathbf{ECpo} \\
\rightarrow: \mathbf{ECpo}^{\mathrm{op}} \times \mathbf{ECpo} \rightarrow \mathbf{ECpo} & \rightharpoonup: \mathbf{ECpo}^{\mathrm{op}} \times \mathbf{ECpo} \rightarrow \mathbf{ECpo} \\
(\cdot)_\perp: \mathbf{ECpo} \rightarrow \mathbf{ECpo} & \\
\forall^{\mathbf{C}}: (\mathbf{C}^D \Rightarrow \mathbf{ECpo}^D) \rightarrow \mathbf{ECpo} & \mu^{\mathbf{C}}: (\mathbf{C}^D \Rightarrow \mathbf{C}^D) \rightarrow \mathbf{C}
\end{array}$$

And, we interpret terms as effective natural transformations between functors into $\mathbf{ECpo}$; effectivity of the natural transformation corresponds to the fact that the meaning of every term will be effectively realizable from its type (kind) environment. The term constructors arise from the adjunctions expressing the cartesian closure of **ECpo**, the natural transformations asserting that the functor $(\cdot)_\perp$ is a monad, the natural isomorphism between $R \rightharpoonup S$ and $R \to S_\perp$, $\forall^{\mathbf{C}}$ as a right adjoint to the diagonal functor, and the interpretations of *fix*, *fold*, *unfold* as effective natural transformations. Let $\mathbf{C}$, $\mathbf{D}$ be effective categories and $F, G\colon \mathbf{C} \to \mathbf{D}$ be effective functors. We say that an effective natural transformation $\eta\colon F \stackrel{\cdot}{\longrightarrow} G$ is *uniform* if there is a natural number n with $n \vdash_{\mathbf{D}} \eta_A$ for all objects A of $\mathbf{C}$, *i.e.*, there is a uniform realizer. It turns out that except for the fixed point operator and the isomorphism pair between a recursive kind and its unfolding, every other term constructor arises from a *uniform* natural transformation.

Let $\mathbf{C}, \mathbf{D}$ be effective categories and suppose that $F\colon \mathbf{C} \to \mathbf{D}, G\colon \mathbf{D} \to \mathbf{C}$ are effective functors; we say that G is a *uniform* right adjoint to F if G is a right adjoint to F and the isomorphism $\mathbf{D}(F(X), A) \cong \mathbf{C}(X, G(A))$ natural in X, A is also uniform. The following lemma expresses the term constructors associated with product types, total function spaces and polymorphic types.

Lemma 2.8.3

- *Let* $!$ *be the unique functor from* **ECpo** *to* $\mathbf{1}$. *Then* $\mathbf{1}$ *is a uniform right adjoint to* $!$.

- *Let* $\Delta\colon \mathbf{ECpo} \to \mathbf{ECpo} \times \mathbf{ECpo}$ *be the diagonal functor defined by* $\Delta(A) = \langle A, A\rangle$. *Then* $\times$ *is a uniform right adjoint to* Δ.

- *For any effective cpo* R, $R \to (\cdot)$ *is a uniform right adjoint to the functor* $(\cdot) \times R$. *Moreover, the isomorphism* $\mathbf{ECpo}(A \times R, C) \cong \mathbf{ECpo}(A, R \to C)$ *is effectively natural in* R *and uniform.*

- *For any object* $\mathbf{C}^D$ *of* $\mathcal{K}$, *define the diagonal functor* $\Delta_{\mathbf{C}}\colon \mathbf{ECpo} \to (\mathbf{C}^D \Rightarrow \mathbf{ECpo}^D)$ *as* $\Delta(R)$ *is the functor that maps any object of* $\mathbf{C}^D$ *to* R *and any arrow to the identity on* R. *Then* $\forall^{\mathbf{C}}$ *is a uniform right adjoint to* $\Delta_{\mathbf{C}}$.

We define the uniform natural transformations η from the identity functor on **ECpo** to $(\cdot)_\perp$ and $\mu: ((\cdot)_\perp)_\perp \stackrel{\cdot}{\longrightarrow} (\cdot)_\perp$ as

$$\eta_R = [(\lambda x.\, \lambda y.\, x)]^{R,R_\perp} \quad \mu_R = [(\lambda f.\, f\ 0)]^{(R_\perp)_\perp,R_\perp}$$

for any object R in **ECpo**. The following lemma gives the terms associated with lifting and partial function spaces:

Lemma 2.8.4

- *The structure* $((\cdot)_\perp, \eta, \mu)$ *forms a monad.*
- *There is a uniform natural isomorphism between the functors*

  $$\rightharpoonup \quad \textit{and} \quad (\rightarrow \circ \langle \mathbf{ECpo}, (\cdot)_\perp \rangle)$$

  where **ECpo** *is used to also denote the identity functor on* **ECpo**.

Finally, by Lemma 2.3.5, the morphism fix^R is effectively realizable from R and by Lemma 2.6.2, the morphisms fold^F, unfold^F are effectively realizable from F; we can also prove that they are natural in R and F respectively. Hence, we get their intepretations as effective natural transformations.

Lemma 2.8.5

- *Let* $F: \mathbf{PECpo} \rightarrow \mathbf{ECpo}$ *be the functor defined by* $F(R) = R \rightarrow R$ *and* $\iota: \mathbf{PECpo} \rightarrow \mathbf{ECpo}$ *be the inclusion functor. Then, we have an effective natural transformation* $\mathit{fix}: F \stackrel{\cdot}{\longrightarrow} \iota$.
- *Let* $\mathbf{C}^D$ *be an object in* $\mathcal{K}$ *with* $\mathbf{C}$ *a* **PECpo**-*enriched category and* $\mathit{Unwind}: (C^D \Rightarrow C^D) \rightarrow C$ *be the functor defined on objects as* $\mathit{Unwind}(F) = F(\mu(F))$. *Then we have effective natural transformations* $\mathit{fold}: \mathit{Unwind} \stackrel{\cdot}{\longrightarrow} \mu$ *and* $\mathit{unfold}: \mu \stackrel{\cdot}{\longrightarrow} \mathit{Unwind}$ *with* $(\mathit{fold}, \mathit{unfold})$ *forming an isomorphism pair.*

We define the subtyping relation for objects of **ECpo** as in Section 2.7. Usually, coherence of a semantics (in the presence of subtyping) denotes that the meaning of a term does not depend on its typing derivation. Inspite of a fixed point operator whose interpretation depends on the type, we can prove a stronger form of coherence for our semantics. Informally, we can show that if for two terms M, N of type σ their erasures are the same then $[\![M\colon\sigma]\!] = [\![N\colon\sigma]\!]$, *i.e.*, their meanings are the same at type σ. The following lemma asserts this property for fixed points and the isomorphism pair between a recursive type and its unfolding, under the assumption that the subtypings given by Proposition 2.7.2 are the only subtypings.

Lemma 2.8.6 (Coherence)

1. *If* $n \vdash \mathit{fix}^R$, $n' \vdash \mathit{fix}^{R'}$ *and* $(R \to R) \to R, (R' \to R') \to R' \;<:\; S$ *then* $n \; S \; n'$.

2. *Suppose that* $F, F'\colon \mathbf{PECpo}^{\mathrm{ep}} \to \mathbf{PECpo}^{\mathrm{ep}}$ *are effective functors,* $n \vdash \mathit{fold}^F$, $n' \vdash \mathit{fold}^{F'}$, *and* $F(\mu(F)) \to F, F'(\mu(F')) \to F' \;<:\; S$ *then* $n \; S \; n'$. *Similarly, if* $n \vdash \mathit{unfold}^F, n' \vdash \mathit{unfold}^{F'}$, *and* $F(\mu(F)) \to \mu(F), F'(\mu(F')) \to \mu(F') \;<:\; S$ *then* $n \; S \; n'$.

Since the interpretation of all other term constructors arises from uniform natural transformations, our coherence theorem asserting the equality of the meaning of two terms with the same erasure at a particular type follows from Lemma 2.8.6.

2.9 Uniform Fixed Point Realizer

While we have defined a semantics for various programming language constructs including fixed point operators at pointed types, the algorithm for computing the fixed point uses the supremum operator which is dependent on the type. In this section, we explore the possibility of a single algorithm that computes the fixed point operator uniformly at all types. As explained in Section 2.1, one reason for our interest in a type-independent algorithm for the fixed point operator is the intuitive correspondence with the process of compilation. It is useful to note that it is not reasonable to expect a uniform realizer for fixed points at *all* pointed types, since the least elements are different. For example, suppose $R = \{\langle 0, 0\rangle\}, S = \{\langle 1, 1\rangle\}$. Then the index of $\lambda x \in \mathcal{N}.x$ is a common realizer for morphims $R \to R$ and $S \to S$ but without a common realizer for the fixed point in the two pers. We therefore focus on a uniform fixed point operator for pointed types with common realizers for their least elements.

2.9.1 Failure of Uniform Suprema

Using the definition of the fixed point operator in terms of the supremum operator, we could obtain a single fixed point realizer if there is a single realizer for the suprema operator for all effective cpos with a common realizer for their least elements. However, the following lemma shows that this is impossible even for $N_\perp, (N \rightharpoonup N)_\perp$.

Lemma 2.9.1 *There is no natural number f such that $f \vdash sup_N$ and $f \vdash sup_{N \rightharpoonup N}$,* i.e., *such that if $s\colon Seq(N)$ then $f \cdot s \vdash_N sup_N(s)$ and if $s\colon Seq(N \rightharpoonup N)$ then $f \cdot s \vdash_{N \rightharpoonup N}$ $sup_{N \rightharpoonup N}(s)$.*

Proof Suppose to the contrary that we have such an f. We will prove that this implies that $\bar{K} = \{z \mid z \cdot z\uparrow\}$ is recursive. Suppose that k is an index of the everywhere undefined function and let k' be the index of the function $\lambda x \in \mathcal{N}.\, 1$. By Proposition 2.4.4 note that $k \leq_{N \rightharpoonup N'} k'$ and $\neg(k(N \rightharpoonup N)k')$. Let s_z be the index of the algorithm which on input $n \in \mathcal{N}$ returns k if $z \cdot z$ does not converge in n steps and returns k' otherwise. Then $z \in Seq(N)$ if $z \in \bar{K}$ and $z \in Seq(N \rightharpoonup N)$ otherwise. Hence $f \cdot s_z = k$ iff $z \in \bar{K}$ obtaining that $\bar{K}$ is recursive. ∎

Corollary 2.9.2 *There is no natural number f such that $f \vdash sup_{N_\perp}$ and $f \vdash sup_{(N \rightharpoonup N)_\perp}$.*

Proof Suppose there were such an f. Then the index of the function

$$\lambda s.\, (f \cdot (\lambda n \in \mathcal{N}.\, \lambda x \in \mathcal{N}.\, s \cdot n)) \cdot 0$$

computes the suprema for both N and $N \rightharpoonup N$, contradicting Lemma 2.9.1. ∎

2.9.2 Failure of Uniform Fixed Points

While we have shown that the suprema operator for any class that includes $N_\perp$ and $(N \rightharpoonup N)_\perp$ cannot be uniform, this may not necessarily imply that the fixed point operator cannot be uniform since it is not clear that suprema can be defined in terms of least fixed points. As we have already seen, it is unreasonable to expect the fixed point operator on all pointed effective cpos to be uniform. As a first step then, we restrict attention to computing fixed points of $n\colon(R \rightharpoonup S) \to (R \rightharpoonup S)$ for effective cpos R, S.

A natural candidate for a uniform fixed point operator is the index, Y, given by the first recursion theorem (*e.g.*, see [Cut80]) which is also the index of the untyped call-by-value fixed point operator; we briefly recall its definition. Let d be a natural number such that $\varphi_{\varphi_d(x)}(y) = \varphi_{\varphi_x(x)}(y)$ where the right hand side is undefined if $\varphi_x(x)\uparrow$. Consider the index v with $\varphi_v(f) = f \circ d$. Then the index Y is defined by $Y \cdot f = \varphi_d(\varphi_v(f))$; clearly Y is computable and is also total (since v, d are indices of total functions). The important property of Y is that for any $e\colon N \to N$, $\varphi_{Y \cdot e} = \varphi_{\varphi_e(Y \cdot e)}$, or alternatively, for all n, $(Y \cdot e) \cdot n \simeq (e \cdot (Y \cdot e)) \cdot n$. Using Y, we can show that for any computable function $g(x, y)$ of two arguments, there is an e with $\varphi_e(y) \simeq g(e, y)$, *i.e.*, we can define computable functions by recursively using their indices. To see this consider the index $f\colon N \to N$ with $(f \cdot x) \cdot y = g(x, y)$ (this can be given by the s-m-n theorem), and take $e = Y \cdot f$. Then,

$$\varphi_e(y) \simeq \varphi_{f(e)}(y) \simeq g(e, y)$$

We collect these properties as a proposition since we will be making essential use of this:

Proposition 2.9.3 *The index Y, has the property that for any $m\colon N \to N$,*

$$Y \cdot m \ (N \rightharpoonup N) \ m \cdot (Y \cdot m)$$

If $g(x, y)$ is a computable function then there is an index e with $\varphi_e(y) \simeq g(e, y)$.

Of course, untyped lambda calculus abounds with other fixed point operators (*c.f.* [Bar84]), and in the world of recursion theory, the index of others may work as a realizer of fixed points on all effective cpos. Hence, more generally, we can consider the realizer for any untyped fixed point operator; we define this formally now:

Definition 2.9.4 A natural number f is an *untyped fixed point operator* if for any $n\colon N \to N$ we have $f \cdot n \ (N \rightharpoonup N) \ n \cdot (f \cdot n)$.

We can now show the following surprising result:

Lemma 2.9.5 *Consider any untyped fixed point operator f. Suppose that R, S are pers with $\neg(k(R \rightharpoonup S)k)$, for some $k \in \mathcal{N}$. Then there is an $e\colon (R \rightharpoonup S) \to (R \rightharpoonup S)$ with $\neg(f \cdot e(R \rightharpoonup S)f \cdot e)$. Thus, f does not even realize a morphism from $((R \rightharpoonup S) \to (R \rightharpoonup S))$ to $(R \rightharpoonup S)$ and hence cannot realize a fixed point operator for the per $R \rightharpoonup S$.*

Proof Suppose that $\neg(k(R \rightharpoonup S)k)$. Let p be an index of the everywhere divergent function. Consider the computable function $g(x, y)$ given by

$$g(x,y) \simeq \begin{cases} k & \text{if } y = f \cdot x \\ p & \text{otherwise} \end{cases}$$

By the second part of Proposition 2.9.3, we have an index e with

$$e \cdot y \simeq \begin{cases} k & \text{if } y = f \cdot e \\ p & \text{otherwise} \end{cases}$$

We first prove that $\neg(f \cdot e)(R \rightharpoonup S)(f \cdot e)$. Clearly $e\colon N \to N$; hence, by the definition of an untyped fixed point operator,

$$\begin{aligned} \forall n \in \mathcal{N}.\ (f \cdot e) \cdot n &= (e \cdot (f \cdot e)) \cdot n \\ &= k \cdot n \quad \text{by the definition of } e \end{aligned}$$

Thus, we have that $f \cdot e(N \rightharpoonup N)k$. Since $\neg(k(R \rightharpoonup S)k)$, we must thus have that $\neg((f \cdot e)(R \rightharpoonup S)(f \cdot e))$. Since $\neg((f \cdot e)(R \rightharpoonup S)(f \cdot e))$, for all $x\colon (R \rightharpoonup S)$, $e \cdot x = p$; thus trivially, $e\colon (R \rightharpoonup S) \to (R \rightharpoonup S)$. We have thus an $e\colon (R \rightharpoonup S) \to (R \rightharpoonup S)$ with $\neg((f \cdot e)(R \rightharpoonup S)(f \cdot e))$. ∎

In particular, by Lemma 2.9.5, the index Y cannot be a realizer for the fixed point operator on all $R \rightharpoonup S$. The proof of Lemma 2.9.5 uses the equality test $y = (f \cdot e)$ which essentially depends on the fact that realizers for functions (y in this case) are accessible as realizers of natural numbers and thus susceptible to equality tests. In Section 2.10 we show that this is the only reason for the failure of Y as a fixed point operator.

2.10 Effective Cpos over Lambda Terms

We consider an untyped lambda-calculus that corresponds closely to the partial combinatory algebra of turing machines that we have considered so far, except for introducing a distinction between natural numbers and codes of functions, by using λ-abstraction. The syntax of untyped terms is given by

$$M \;::=\; n \mid \mathsf{succ}\, M \mid \mathsf{pred}\, M \mid \mathsf{if0}(M, N, P) \mid \lambda x.\, M \mid M\ N$$

Intuitively, succ is the successor operation on natural numbers, pred is the predecessor operation, $\mathsf{if0}(M, N, P)$ evaluates N if M evaluates to 0 and evaluates P if N evaluates to a non-zero natural number and is otherwise divergent. We impose a call-by-value evaluation mechanism (corresponding to turing machines); values V are either numerals n or λ-abstractions and the application of a numeral to any expression diverges. Taking convergence to be the only observable behaviour, we can define an observational congruence relation $M \equiv N$ in the obvious way. We consider the partial combinatory algebra of values with application of a value V_1 to V_2 defined as the value that is the result of evaluating $(V_1\ V_2)$ and undefined if the evaluation diverges. We consider pers R of values with the condition that if $V \equiv V'$ and $V\colon R$ then $V\ R\ V'$; essentially, we are considering pers over observational congruence classes of values. We define *effective cpos* in the same way as for natural numbers, with realizers for suprema now being the values of this λ-calculus. With the above choice of operations on natural numbers, all computable functions are expressive; it follows that we have realizers for universal turing machines and running a turing machine with code n for

m steps etc.; we thus inherit all the results from previous sections about effective cpos over natural numbers.

Importantly, we can now show that the untyped term

$$Y = \lambda f.\, (\lambda x.\, f\ (\lambda z.\, x\ x\ z))\, (\lambda x.\, f\ (\lambda z.\, x\ x\ z))$$

is a realizer for the least fixed point operator $\mathit{fix}^{R \rightharpoonup S}$ for any effective cpos R, S; we briefly sketch its proof. Define the observational preorder $\sqsubseteq$ on values in the obvious way. This preorder $\sqsubseteq$ is not a cpo, but if for each $i \in \mathcal{N}$, V_i is a value such that $V_i \sqsubseteq V_{i+1}$ and their least upper bound exists, we denote the congruence class by $\bigsqcup\{V_i\}$. The application operation can be proved to be continuous with respect to $\sqsubseteq$. It is trivially true that if $M, N\colon R$ and $M \sqsubseteq R$ then $M \leq_R N$ in the intrinsic preorder of R. The important property of effective cpos is that if $s\colon \mathit{Seq}(R)$ is an increasing sequence such that $s \cdot n \sqsubseteq s \cdot (n+1)$ and $V \vdash \mathit{sup}_R$ then $V \cdot s \in \bigsqcup\{s \cdot i\}$, *i.e.*, if an increasing sequence is also an increasing sequence with respect to the preorder $\sqsubseteq$ then any realizer of the supremum operator of an effective cpo must return the least upper bound with respect to the global ordering. This is proved by considering sequences s_n that match s up to n and are divergent for arguments greater than n; then $s_n \sqsubseteq s_{n+1}$ and $\mathit{sup}_R(s_n)$ is $[s \cdot n]_R$ and $\bigsqcup\{s_n\}$ is s; since the supremum operator is realizable and application is continuous we have the result. Now, consider the λ-terms F_n defined by $F_0 = \lambda f.\, \Omega$, where Ω is any divergent term and $F_{n+1} = \lambda f.\, f\ (\lambda z.\, F_n\ f\ z)$. It can be shown that $F_n \sqsubseteq F_{n+1}$ and $Y \in \bigsqcup\{F_n\}$. We also have that $F_n\colon ((R \rightharpoonup S) \to (R \rightharpoonup S)) \to (R \rightharpoonup S)$ and that $\mathit{fix}^{R \rightharpoonup S} = \mathit{sup}_{((R \rightharpoonup S) \to (R \rightharpoonup S)) \to (R \rightharpoonup S)}(s)$ where s is the sequence with $s \cdot n = F_n$. From all this it follows that $Y \vdash \mathit{fix}^{R \rightharpoonup S}$.

2.11 Comparison with related work

From domain theory, we have several classes of domain models of polymorphism and recursion. The first are the universal domains. A general construction explored in [ABL86] interprets types as the finitary projections over a universal domain. Since types are domains, recursion is straightforward. An alternative model in essentially the same spirit uses the closures of $P\omega$ [Sco76, McC79, BMM90]. Two models that are not based on universal domains are the coherent spaces of Girard [Gir86] and the related use of dI-domains developed in [CGW89]. In each of these cases, we would interpret subtyping by selecting a class of continuous "subtype maps." However, since there is no general characterization of subtyping, this would have to be done on an *ad hoc* basis for each model. An example study identifying the linear maps between dI-domains as candidate subtype maps is [BTCGS91].

The alternatives to these domain models all use partial equivalence relations. The per models fall into two groups, one using pers over domains and the other pers over the natural numbers. Some constructions in the first group are explained in [AP90, Ama91, Car89, BM92]. The most sophisticated construction is [AP90], which uses an intrinsic order similar to the present paper, but in addition to being a cpo, the pers need to satisfy other conditions (*e.g.*, uniformity). In as much as the model presented in Section 2.10 works with only the simple conditions of being an effective cpo and is a semantics of polymorphism and recursion with uniform fixed points, these additional conditions on pers seem to be a direct consequence of working over specially constructed domains.

There are two previous models using pers over the natural numbers. The more accessible is the class of extensional pers [FRMS92]. Unfortunately, the property of being an extensional per is not categorical as evidenced by the fact that the class of extensional pers is not closed under recursive isomorphism. More importantly from our point of view, the natural interpretations of basic datatypes such as natural numbers and booleans also fail to be extensional (in the technical sense of "extensional pers"). The extensional pers are also not closed under the natural constructions of products, total function spaces and partial function spaces. Hence one uses variants of these operations that work but seem to have no natural explanation. In short, although the development of extensional pers is mathematically elegant and rich, these pers do not support our motivating intuition for preferring per models over the natural numbers.

The final class of models involve the development of domains within the effective topos [Hyl82, Hyl88]. The main idea, as explained briefly in [Pho90b] and elaborated in [Pho90a, Hyl90, HRR90] is to carry out the usual development of domain theory within a constructive set theory that may be interpreted over pers. This leads to a class of domain-like pers that are essentially the same as our effective cpos. However, the class of maps, interpretation of lifting, and fixed-point operators are technically different from ours. In particular, it does not seem that polymorphism is interpreted by considering supremum operators as the realizers of domains. There is also a model over untyped call-by-name lambda-terms in [Pho90a], using pers that have an order property defined using a specific per ω and its completion $\bar{\omega}$. The main difference is that our definition of effective cpo, which implies Phoa's condition on relations, makes sense for any partial combinatory algebra, while Phoa's condition appears specific to lambda terms.

Chapter 3

Full Abstraction for Sequential Call-by-Value Languages

3.1 Introduction

The problem of finding fully abstract models of various programming languages has remained a key but elusive goal of programming language semantics. A denotational model is *fully abstract* if denotational equivalence coincides with operational equivalence; two pieces of code are *operationally equivalent* if they yield the same outcomes when used in any program.

In this chapter, we describe how to build a fully abstract model of a sequential language with call-by-value reduction semantics. We consider two languages: a call-by-value version of PCF [Gun92, Plo77], called VPCF; and a call-by-value language with functions, sums, products, and recursive types, called FPC in [Gun92]. The models for these languages are constructed using the syntax of the particular language. Nevertheless, the models are *not* the straightforward term model constructions. Consider, for instance, the language VPCF which is defined formally in Section 3.5 below. One might construct a term model by first defining $M \sqsubseteq N$ iff for all contexts $C[\cdot]$, $C[M]$ halts implies that $C[N]$ halts, and $M \sim_{\mathrm{VPCF}} N$ iff $M \sqsubseteq N$ and $N \sqsubseteq M$. We can then take the term model to consist of equivalence classes of closed terms , *i.e.*, $[\![\sigma]\!] = \{[M] \mid M \text{ is closed of type } \sigma\}$, where $[M] = \{N \mid M \sim_{\mathrm{VPCF}} N\}$. The model is obviously fully abstract: terms that are distinguishable are distinguished. One may also prove that the elements in the functional types really are mathematical functions; this follows from a version of Milner's Context Lemma [Mil77]. Such term models, however, lack the necessary structure needed to interpret the recursion combinator as a least fixpoint operation. Consider, for instance, a sequence of terms M_i that compute the i^{th} approximation to some function $f\colon \mathbf{N} \to \mathbf{N}$ that is not partial recursive. That is, on arguments n less than i, the function M_i returns $f(n)$, and otherwise diverges. This sequence is increasing in the term model, but has no least upper bound because f is not programmable in the language.

The lack of all least upper bounds makes the usual term model inadequate for using certain proof principles, *e.g.*, those based on logical relations. Our fully abstract models—one for VPCF, the other for FPC—are based on complete partial orders (cpo's) and continuous functions (see Section 3.2 below). We base our constructions on the ideas of Milner [Mil77], extending Milner's work in two directions: first, we show how to build models for call-by-value languages as opposed to call-by-name (call-by-value languages are far more common than call-by-name languages); and second, we show how to interpret recursive types in the model for FPC. Changing from call-by-name to call-by-value evaluation does not cause any real technical problems, but adding recursive types does. Only Mulmuley [Mul87] seems to have built a fully abstract model for a sequential language with recursive types (Smith [Smi91] builds a model, using somewhat different techniques, for an untyped lazy language). While one difference between our work and Mulmuley's is that Mulmuley's language is call-by-name instead of call-by-value, our construction is also technically different (the sets are constructed directly instead of being retractions of some universal domain). Our methods also generalize more easily, since we construct a category suitable for defining the semantics of other sequential languages.

It would be better if the models were not based on the syntax of the language, but fully abstract models for sequential languages have been notoriously difficult to construct. The usual denotational semantics of a language based on Scott domains admits parallel operations that distinguish operationally equivalent terms (see, for instance, [BCL85, CF92, Cos89, Mul87, Plo77] for a history and discussion of the issues). There may very well be other means of constructing fully abstract models for the languages studied here: a model constructed using the recently announced game semantics [AJM93, HO93] might be possible and may provide a better tool for reasoning directly about programs.

Our reasons for building the models lie not in developing new reasoning techniques for sequential languages, but rather in proving meta-theoretic properties of languages. For these applications, the details of the models are almost irrelevant. Our primary goal was to generalize the coercion results of Riecke [Rie93a] to *sequential* programming languages. Riecke's paper addresses a common problem in functional languages: adding side effects to a purely functional language changes the notion of operational equivalence, *e.g.*, if we add state to VPCF, there are terms M, N such that $M \sim_{\mathrm{VPCF}} N$ but $M \not\sim_{\mathrm{VPCF+state}} N$. The theorems in [Rie93a] show how to add a coercion operation # to VPCF+state+pif—where pif is a "parallel if" construct—so that $M \sim_{\mathrm{VPCF+pif}} N$ iff $(\# M) \sim_{\mathrm{VPCF+state+pif}} (\# N)$. The main theorem relies on the construction of a fully abstract model of VPCF+pif and a model of VPCF+state+pif into which the model of VPCF+pif embeds. We can obtain a model of VPCF+state from a model of FPC using the recursive types to model a "state". Hence, by developing models of VPCF *without parallel if* and a model of FPC, we can carry out the program in [Rie93a] to sequential languages. The models must be continuous and extensional for the proof techniques to carry over, but the particular construction of the elements of the model is unimportant. One can also use the models to prove that translations between programming languages are fully abstract (*c.f.* [Rie93b]).

The development of the two models proceeds in stages. In Section 3.2, we briefly recall some of the basic definitions of domain theory and models of programming languages. This section is not meant as a tutorial; the reader should consult [Gun92] for more intuition. We then describe the model for VPCF. The construction of this model illustrates the techniques we use for call-by-value languages. Furthermore, it uses the domain-theoretic notion of an *inverse limit* which provides important clues to the construction of a model for FPC.

Table 3.1: Typing and Evaluation Rules for General Functional Languages.

Terms	(var)	$x^\sigma : \sigma$
	$(const)$	$\dfrac{\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma \rangle \in C \quad M_i : \sigma_i}{f(M_1, \ldots, M_n) : \sigma}$
	(abs)	$\dfrac{M : \tau}{(\lambda x^\sigma .\, M) : (\sigma \to \tau)}$
	(app)	$\dfrac{M : (\sigma \to \tau) \quad N : \sigma}{(M\ N) : \tau}$
Evaluation		$V \Downarrow V$
		$\dfrac{M \Downarrow (\lambda x.\, M') \quad N \Downarrow V' \quad [V'/x]M' \Downarrow V}{(M\ N) \Downarrow V}$

3.2 Preliminaries

3.2.1 Review of basic domain theory

A subset X of a partial order D is **directed** if every finite subset of X has an upper bound in X. A partial order is a **complete partial order** (cpo) if every directed subset X has a least upper bound, written $\sqcup X$. An element d of a cpo is **finite** if for any directed set X, if $d \sqsubseteq \sqcup X$ then there is an $x \in X$ such that $d \sqsubseteq x$. A cpo is **algebraic** if every element is the least upper bound of the finite elements below it.

A monotone function $f: D \to E$ (where D, E are cpo's) is **continuous** if for any directed $X \subseteq D$, $f(\sqcup X) = \sqcup_{x \in X}\{f(x)\}$. Given two continuous maps $f, g: D \to E$, we say that $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for every $x \in D$. Let id_D denote the identity map on D. An **embedding-projection pair** from cpo D to cpo E is a pair of continuous maps $f^E: D \to E$, $f^P: E \to D$ such that $f^P \circ f^E = \mathsf{id}_D$ and $f^E \circ f^P \sqsubseteq \mathsf{id}_E$.

Table 3.2: Additional Evaluation Rules for VPCF.

$$\frac{M \Downarrow \ulcorner n \urcorner}{\mathsf{succ}\, M \Downarrow \ulcorner n+1 \urcorner} \qquad \frac{M \Downarrow 0 \qquad P \Downarrow V}{\mathsf{if0}(M,P,Q) \Downarrow V}$$

$$\frac{M \Downarrow 0}{\mathsf{pred}\, M \Downarrow 0} \qquad \frac{M \Downarrow \ulcorner n+1 \urcorner \qquad Q \Downarrow V}{\mathsf{if0}(M,P,Q) \Downarrow Q}$$

$$\frac{M \Downarrow \ulcorner n+1 \urcorner}{\mathsf{pred}\, M \Downarrow \ulcorner n \urcorner} \qquad \frac{M \Downarrow Y \qquad N \Downarrow V' \qquad (V'\,(\lambda x.\, Y\, V'\, x)) \Downarrow V}{(M\ N) \Downarrow V}$$

3.2.2 Call-by-value functional languages

The set of types Σ of a general functional language includes a set B of base types and must be closed under $\rightarrow$, *viz.*, if $\sigma, \tau \in \Sigma$, then $(\sigma \rightarrow \tau) \in \Sigma$. A set of typing rules for functional languages appears in Table 3.1; all variables come tagged with types. The typing rules are parameterized by a set C of **term constructors**, which are tuples of the form $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma\rangle$ where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma$ are types. The two basic operational rules of call-by-value functional languages appear in Table 3.1. The rules are given in natural semantics style (*c.f.* [Gun92]): $M \Downarrow V$ is read as "the term M halts at value V", where any particular call-by-value language comes with its own set of values that must include λ-abstractions. We write $M \Downarrow$ to mean that there is a V such that $M \Downarrow V$.

3.2.3 Models of call-by-value languages

A **model** of a call-by-value language is a set of elements together with a means of interpreting each construct. The following definition is taken from [Rie91] and resembles definitions from [Abr90, Ong88a, Ong88b].

Definition 3.2.1 A **call-by-value type frame** $\mathcal{D}$ over a set of types Σ is a tuple

$$(\{D^\sigma \mid \sigma \in \Sigma\}, \{\uparrow^\sigma \mid \sigma \in \Sigma\}, \{App^{\sigma,\tau} \mid \sigma, \tau \in \Sigma\})$$

where each D^σ is a set, $\uparrow^\sigma \in D^\sigma$, and $App^{\sigma,\tau} : D^{\sigma\to\tau} \times D^\sigma \to D^\tau$ such that if either $d = \uparrow^{\sigma\to\tau}$ or $e = \uparrow^\sigma$ then $App^{\sigma,\tau}(d,e) = \uparrow^\tau$. We often omit the type superscripts on $\uparrow$ and App when they are obvious from the context. Also, we write $d\uparrow$ for $d = \uparrow$ and $d\downarrow$ for $d \neq \uparrow$.

Thus, in addition to application, a call-by-value type frame has an interpretation of divergence, and the interpretation of application must be faithful to the call-by-value parameter passing mechanism. A model is a type frame with enough elements to interpret every term:

Definition 3.2.2 Let $\mathcal{D}$ be a call-by-value type frame with $\leq_\sigma$ a partial order on each D^σ.

- A **$\mathcal{D}$-environment** ρ is a type-respecting map from variables to elements, *i.e.*, $\rho(x^\sigma) \in D^\sigma$.

- $\mathcal{D}$ is a **call-by-value model** if there is a meaning function $\mathcal{D}[\![\cdot]\!]$ satisfying

$$\begin{array}{rcl} \mathcal{D}[\![x^\sigma]\!]\rho & = & \rho(x^\sigma) \\ \mathcal{D}[\![M\ N]\!]\rho & = & App(\mathcal{D}[\![M]\!]\rho, \mathcal{D}[\![N]\!]\rho) \\ \mathcal{D}[\![\lambda x^\sigma.\, M]\!]\rho & = & f, \text{ where } f\downarrow \text{ and for any } d\downarrow, \\ & & \quad App(f,d) = \mathcal{D}[\![M]\!]\rho[x^\sigma \mapsto d] \end{array}$$

  and such that $\mathcal{D}[\![M]\!]\rho \in D^\sigma$ for any term $M{:}\,\sigma$ and $\mathcal{D}$-environment ρ.

- $\mathcal{D}$ is **extensional** if for all $f, g \in D^{\sigma\to\tau}$,

$$f \leq_{\sigma\to\tau} g \text{ iff } f\downarrow \text{ implies } g\downarrow, \text{ and for all } d \in D^\sigma,\ App(f,d) \leq_\tau App(g,d).$$

- $\mathcal{D}$ is **monotone** if for any $f\downarrow \in D^{\sigma\to\tau}$, if $x \leq_\sigma y$, then $App(f,x) \leq_\tau App(f,y)$.

- $\mathcal{D}$ is **continuous** if each D^σ is a cpo, and for any $f \in D^{\sigma\to\tau}$ and directed set $S \subseteq D^\sigma$, $App^{\sigma,\tau}(f, \bigsqcup S) = \bigsqcup\{App^{\sigma,\tau}(f,e) \mid e \in S\}$.

Our goal is to find extensional, continuous models for VPCF and FPC that are fully abstract.

Table 3.3: Partial Call-by-Value Languages.

Terms	$(div) \quad \Omega_\gamma : \gamma$
Results	$R \ ::= \ V \mid \Omega_\gamma$
Evaluation	$R \Downarrow R \qquad \dfrac{M \Downarrow \Omega}{(M\ N) \Downarrow \Omega} \qquad \dfrac{N \Downarrow \Omega}{(M\ N) \Downarrow V\Omega}$ $\dfrac{M \Downarrow (\lambda x.\, M') \quad N \Downarrow V' \quad [V'/x]M' \Downarrow V}{(M\ N) \Downarrow V}$

3.3 The language VPCF

VPCF is a rudimentary functional language with basic arithmetic and recursion. The types are

$$\sigma, \tau ::= \mathbf{num} \mid (\sigma \to \tau)$$

The term constructors of VPCF are $\langle \ulcorner n \urcorner; \mathbf{num} \rangle$ (where $\ulcorner n \urcorner$ denotes the syntactic representation of a natural number n), $\langle \mathsf{Y}_\sigma; (\sigma \to \sigma) \to \sigma \rangle$ for all types $\sigma \neq \mathbf{num}$, $\langle \mathsf{succ}; \mathbf{num} \Rightarrow \mathbf{num} \rangle$ and $\langle \mathsf{pred}; \mathbf{num} \Rightarrow \mathbf{num} \rangle$, and $\langle \mathsf{if0}^\sigma; \mathbf{num}, \sigma, \sigma \Rightarrow \sigma \rangle$ for each type σ; the values are $\ulcorner n \urcorner$, Y_σ, and λ-abstractions. The evaluation rules of VPCF are those in Table 3.1 together with those in Table 3.2. We say $M \sim_{\mathrm{VPCF}} N$ (M is **operationally equivalent** to N) if for any VPCF context $C[\cdot]$, $C[M] \Downarrow$ iff $C[N] \Downarrow$.

3.4 FVPCF: A Finite Approximation of VPCF

As in [Mil77], our fully abstract model of VPCF is constructed as the inverse limit of the term model of a suitable "partial" language. The crucial difference, however, lies in the choice of this language—to obtain full abstraction for call-by-value languages, we define the notion of a **partial call-by-value language**. In addition to the terms given by the typing rules in Table 3.1, a partial call-by-value language also contains explicit divergence constants Ω_γ at every type γ as given in Table 3.3. We use the variables β, γ, δ to range over its types. Table 3.3 contains the operational rules common to all such partial languages, where the "result" R of the evaluation of a term in such a language, in addition to being a value, can also be divergence.

Define the language FVPCF, which stands for *Finite* VPCF, to be the partial call-by-value language with base types $B = \{\mathbf{num}_i \mid i \geq 0\}$ and $\rightarrow$ type constructor, and term constructors

$$\begin{aligned} C \;=\; & \textstyle\bigcup_{i\geq 0} \{\langle \ulcorner n \urcorner_i ; \mathbf{num}_i \rangle \mid n \leq i\} \cup \\ & \textstyle\bigcup_{i\geq 0} \{\langle \mathsf{succ}_i ; \mathbf{num}_i \Rightarrow \mathbf{num}_i \rangle, \langle \mathsf{pred}_i ; \mathbf{num}_i \Rightarrow \mathbf{num}_i \rangle\} \cup \\ & \textstyle\bigcup_{i\geq 0} \{\langle \phi_{i,i+1} ; \mathbf{num}_i \Rightarrow \mathbf{num}_{i+1} \rangle, \langle \psi_{i,i+1} ; \mathbf{num}_{i+1} \Rightarrow \mathbf{num}_i \rangle\} \cup \\ & \{\langle \mathsf{if0}_i^{\gamma} ; \mathbf{num}_i, \gamma, \gamma \Rightarrow \gamma \rangle \mid i \geq 0, \gamma \text{ a type over } B.\} \end{aligned}$$

Intuitively, each type $\mathbf{num}_i$ is a finite approximation to the type $\mathbf{num}$ of VPCF and includes only the numerals up to i; the basic operations for manipulating these numerals must stay within these bounds. The coercions $\psi_{i,i+1}$ and $\phi_{i,i+1}$ convert between the various approximations to $\mathbf{num}$. A value V in FVPCF is either a constant $\ulcorner n \urcorner_i$ or a λ-abstraction. The operational rules for the additional term constructors in FVPCF are given in Table 3.4.

The term model of FVPCF is based on the following operational approximation relation:

Definition 3.4.1 Define the preorder $\lesssim$ as $M \lesssim N$ iff for all contexts $C[\cdot]$ such that $C[M]$ and $C[N]$ are closed terms, if $C[M] \Downarrow$ then $C[N] \Downarrow$. Also, define $M \approx N$ iff $M \lesssim N$ and $N \lesssim M$.

Denote the $\approx$ equivalence class of M by $[M]$, and define the model $\mathcal{L} = (\{\mathcal{L}^{\gamma}\}, \{\uparrow^{\gamma}\}, App)$ by $\mathcal{L}^{\gamma} = \{[M] \mid M{:}\,\gamma\}$, $\uparrow^{\gamma} = [\Omega_{\gamma}]$, and $App([M],[N]) = [M\,N]$. To make $\mathcal{L}$ a model, define the interpretation of the constructors $\langle f; \gamma_1, \ldots, \gamma_k \Rightarrow \gamma\rangle$ to be $\mathcal{L}[\![f]\!]\colon D^{\gamma_1} \times \cdots \times D^{\gamma_k} \rightarrow D^{\gamma}$ where $\mathcal{L}[\![f]\!]([M_1], \ldots, [M_k]) = [f(M_1, \ldots, M_k)]$. A substitution ϕ is said to represent an $\mathcal{L}$-environment ρ if $\phi(x^{\gamma}) \in \rho(x^{\gamma})$. Define the meaning function to be $\mathcal{L}[\![M]\!]\rho = [\phi\, M]$ where ϕ is a substitution representing ρ. Since $\approx$ is a congruence, the meaning function is independent of the choice of the substitution ϕ. This meaning function establishes $\mathcal{L}$ to be a model in the sense of

Table 3.4: Additional Evaluation Rules for FVPCF.

$$\frac{M \Downarrow \ulcorner n \urcorner_i}{\mathsf{succ}_i(M) \Downarrow \ulcorner n+1 \urcorner_i},\quad n < i \qquad \frac{M \Downarrow \ulcorner n \urcorner_i}{\mathsf{succ}_i(M) \Downarrow \Omega_{\mathbf{num}_i}},\quad n \geq i$$

$$\frac{M \Downarrow \Omega_{\mathbf{num}_i}}{\mathsf{succ}_i(M) \Downarrow \Omega_{\mathbf{num}_i}}$$

$$\frac{M \Downarrow 0_i}{\mathsf{pred}_i(M) \Downarrow 0_i} \qquad \frac{M \Downarrow \ulcorner n+1 \urcorner_i}{\mathsf{pred}_i(M) \Downarrow \ulcorner n \urcorner_i}$$

$$\frac{M \Downarrow \Omega_{\mathbf{num}_i}}{\mathsf{pred}_i(M) \Downarrow \Omega_{\mathbf{num}_i}}$$

$$\frac{M \Downarrow 0_i \qquad P \Downarrow V}{\mathsf{if0}_i(M, P, Q) \Downarrow V} \qquad \frac{M \Downarrow \ulcorner n+1 \urcorner_i \qquad Q \Downarrow V}{\mathsf{if0}_i(M, P, Q) \Downarrow Q}$$

$$\frac{M \Downarrow \Omega_{\mathbf{num}_i}}{\mathsf{if0}_i(M, P, Q) \Downarrow \Omega}$$

$$\frac{M \Downarrow \ulcorner n \urcorner_{i+1}}{\psi_{i,i+1}(M) \Downarrow \ulcorner n \urcorner_i},\quad n \leq i \qquad \frac{M \Downarrow \ulcorner n \urcorner_{i+1}}{\psi_{i,i+1}(M) \Downarrow \Omega_{\mathbf{num}_i}},\quad n > i$$

$$\frac{M \Downarrow \Omega_{\mathbf{num}_{i+1}}}{\psi_{i,i+1}(M) \Downarrow \Omega_{\mathbf{num}_i}}$$

$$\frac{M \Downarrow \ulcorner n \urcorner_i}{\phi_{i,i+1}(M) \Downarrow \ulcorner n \urcorner_{i+1}} \qquad \frac{M \Downarrow \Omega_{\mathbf{num}_i}}{\phi_{i,i+1}(M) \Downarrow \Omega_{\mathbf{num}_{i+1}}}$$

Section 3.2.3.

Lemma 3.4.2 *$\mathcal{L}[\![M]\!]\rho = [\phi\, M]$, where the substitution ϕ represents ρ, is a suitable meaning function.*

Corollary 3.4.3 *Every element of $\mathcal{L}$ is definable by a term.*

Proof For any $x \in \mathcal{L}^\sigma$, take any term $M \in x$. Then, $\mathcal{L}[\![M]\!] = x$. ∎

The model $\mathcal{L}$ is trivially monotone since if $N \sqsubsetapprox N'$ then for any term M, $M\,N \sqsubsetapprox M\,N'$. We now proceed to show that the model $\mathcal{L}$ is extensional. Consider the following call-by-value "extensional" ordering.

Definition 3.4.4 For closed terms $M, N{:}\,\gamma$ define $M \leq_E^\gamma N$ by induction on the structure of γ as follows:

- $M \leq_E^{\mathbf{num}_i} N$ iff $M \Downarrow \ulcorner n \urcorner_i$ implies $N \Downarrow \ulcorner n \urcorner_i$
- $M \leq_E^{\gamma\to\delta} N$ iff $M \Downarrow$ implies $N \Downarrow$, and for any closed $P{:}\,\gamma$, $(M\;P) \leq_E^\delta (N\;P)$.

To help establish that the notion of observational approximation $\sqsubsetapprox$ is equivalent to the extensional ordering on terms defined above, we define the following logical relation between closed terms.

Definition 3.4.5 For closed terms $M, N{:}\,\gamma$, define $M \sqsubseteq^\gamma N$ if

1. $M \Downarrow \Omega_\gamma$, or
2. $M \Downarrow V, N \Downarrow V'$ for some values V, V' and $V \sqsubseteq^\gamma V'$ where
   - $\ulcorner n \urcorner_i \sqsubseteq^{\mathbf{num}_i} \ulcorner n \urcorner_i$.
   - $(\lambda x{:}\,\gamma.\,P) \sqsubseteq^{\gamma\to\delta} (\lambda y{:}\,\gamma.\,P')$ if, for all values $U, U'{:}\,\gamma$ such that $U \sqsubseteq^\gamma U'$ we have that $(\lambda x{:}\,\gamma.\,P)\,U \sqsubseteq^\delta (\lambda y{:}\,\gamma.\,P')\,U'$.

We can now prove the following key technical lemma, which can be seen as similar to the fundamental theorem of logical relations.

Lemma 3.4.6 *Assume that $M \sqsubseteq^{\nu} N$. Let $C[\,]$ be a context whose free variables are among $x_1^{\gamma_1}, \ldots, x_n^{\gamma_n}$ such that $C[M]\colon \gamma$. If $P_i \sqsubseteq^{\gamma_i} Q_i$ for $i = 1, \ldots, n$, then*

$$[P_1, \ldots, P_n / x_1, \ldots, x_n]C[M] \sqsubseteq^{\gamma} [Q_1, \ldots, Q_n / x_1, \ldots, x_n]C[N]$$

Proof By induction on the structure of $C[\,]$. ∎

Lemma 3.4.7 *For closed terms $M, N\colon \gamma$, the following are equivalent:*

1. $M \leq_E^{\gamma} N$
2. $M \sqsubseteq^{\gamma} N$
3. $M \precsim N$

Proof

(1) $\Rightarrow$ (2) Assume that $M \leq_E^{\gamma} N$. We prove by induction on γ, that $M \sqsubseteq^{\gamma} N$.

$\gamma = \mathbf{num}_i$: Trivial

$\gamma = \alpha \to \beta$: By definition of $\leq_E^{\gamma}$ we have that if $M \Downarrow \lambda x\colon \alpha.\, P$ then $N \Downarrow \lambda y\colon \alpha.\, Q$ for some y, Q. Consider any $U \sqsubseteq^{\alpha} U'$. Now, $(M\,U) \leq_E^{\beta} (N\,U)$. Hence, by induction hypothesis, $(M\,U) \sqsubseteq^{\beta} (N\,U)$. Using Lemma 3.4.6 on $U \sqsubseteq^{\alpha} U'$ for the context $C[\;] = N[\;]$, we get that $(N\,U) \sqsubseteq^{\beta} (N\,U')$. Hence, $(M\,U) \sqsubseteq^{\beta} (N\,U')$ from which we can easily obtain that $M \sqsubseteq^{\alpha\to\beta} N$.

(2) $\Rightarrow$ (3) Assume that $M \sqsubseteq^{\gamma} N$. Consider any context $C[\;]$ such that $C[M]$ and $C[N]$ are closed terms. By Lemma 3.4.6, $C[M] \sqsubseteq C[N]$. By the definition of $\sqsubseteq$, this means that $C[M]\Downarrow$ implies that $C[N]\Downarrow$.

(3) $\Rightarrow$ (1) Immediate.

∎

Corollary 3.4.8 *The term model $\mathcal{L}$ is extensional.*

We thus obtain the following lemma.

Lemma 3.4.9 *$\mathcal{L}$ is an extensional, monotone model of FVPCF, with every element definable by an FVPCF term.*

Each of the sets $\mathcal{L}^\gamma$ is finite, and the monotone term model $\mathcal{L}$ is consequently also continuous.

3.5 Full Abstraction for VPCF

We are now ready to define the fully-abstract model $\mathcal{D}$ of VPCF. Define the partial order $\preceq$ over FVPCF and VPCF types by taking the reflexive, transitive closure of the relation

- $\mathbf{num}_i \preceq \mathbf{num}_{i+1} \preceq \mathbf{num}$ for all $i \geq 0$.
- If $\gamma \preceq \gamma' \preceq \sigma$ and $\delta \preceq \delta' \preceq \tau$ then $(\gamma \to \delta) \preceq (\gamma' \to \delta') \ \preceq (\sigma \to \tau)$.

For types $\gamma \preceq \gamma'$ we define $\phi_{\gamma,\gamma'} \in \mathcal{L}^{\gamma\to\gamma'}$, $\psi_{\gamma,\gamma'} \in \mathcal{L}^{\gamma'\to\gamma}$ by induction on the structure of γ. Abusing notation, we use $\phi_{\gamma,\gamma'}$ and $\psi_{\gamma,\gamma'}$ to denote $\mathcal{L}[\![\phi_{\gamma,\gamma'}]\!]$ and $\mathcal{L}[\![\psi_{\gamma,\gamma'}]\!]$.

$$\begin{array}{rcl}
\phi_{\mathbf{num}_i,\mathbf{num}_k} & = & \lambda x\colon \mathbf{num}_i.\, \phi_{k-1,k}(\phi_{k-2,k-1}(\ldots(\phi_{i,i+1}(x))))\\
\psi_{\mathbf{num}_i,\mathbf{num}_k} & = & \lambda x\colon \mathbf{num}_i.\, \psi_{k-1,k}(\psi_{k-2,k-1}(\ldots(\psi_{i,i+1}(x))))\\
\phi_{\gamma\to\delta,\gamma'\to\delta'} & = & \lambda f\colon \gamma \to \delta.\, \lambda x\colon \gamma'.\, \phi_{\delta,\delta'}(f(\psi_{\gamma,\gamma'}(x)))\\
\psi_{\gamma\to\delta,\gamma'\to\delta'} & = & \lambda f\colon \gamma' \to \delta'.\, \lambda x\colon \gamma.\, \psi_{\delta,\delta'}(f(\phi_{\gamma,\gamma'}(x)))
\end{array}$$

Each element f of $D^{\sigma\to\tau}$, for any type frame $\mathcal{D}$, defines a function from D^σ to D^τ which maps any $x \in D^\sigma$ to $App(f,x) \in D^\tau$. It is easy to see that the elements $\phi_{\gamma,\gamma'}, \psi_{\gamma,\gamma'}$ define an embedding-projection pair between $\mathcal{L}^\gamma$ and $\mathcal{L}^{\gamma'}$. For any VPCF type σ, take the poset D^σ to be the inverse limit of the posets $\mathcal{L}^\gamma, \gamma \preceq \sigma$, under the embedding-projections just defined, *i.e.*,

$$\begin{array}{rcl}
D^\sigma & = & \{x \in \prod_{\gamma\preceq\sigma} \mathcal{L}^\gamma \mid \forall\gamma \preceq \gamma'.(x)_\gamma = \psi_{\gamma,\gamma'}((x)_{\gamma'})\}\\
x \leq_\sigma y & \text{iff} & \forall\gamma \preceq \sigma.(x)_\gamma \leq (y)_\gamma
\end{array}$$

Each D^σ is a cpo by the pointwise ordering and the finiteness of each D^γ.

The divergent elements $\uparrow^\sigma$ are defined as $(\uparrow^\sigma)_\gamma = \uparrow^\gamma$, for all $\gamma \preceq \sigma$. To define application in the model $\mathcal{D}$, we first define embedding-projection pairs $\Phi_{\gamma,\sigma}\colon \mathcal{L}^\gamma \to D^\sigma, \Psi_{\gamma,\sigma}\colon D^\sigma \to \mathcal{L}^\gamma$. First note that $\{\gamma \mid \gamma \preceq \sigma\}$ is a lattice and hence there is a pairwise lub operation $\vee$. Then,

$$\begin{aligned}(\Phi_{\gamma,\sigma}(d))_{\gamma'} &= \psi_{\gamma',\gamma\vee\gamma'}(\phi_{\gamma,\gamma\vee\gamma'}(d)), \quad \text{for all } \gamma' \preceq \sigma \\ \Psi_{\gamma,\sigma}(x) &= (x)_\gamma\end{aligned}$$

One can easily verify that $\Phi_{\gamma,\sigma}, \Psi_{\gamma,\sigma}$ are continuous. Since each D^σ is a cpo, define application by

$$App^{\sigma,\tau}(x,y) = \bigsqcup\{\Phi_{\delta,\tau}(App^{\gamma,\delta}((x)_\gamma,(y)_\delta)) \mid \gamma \preceq \sigma, \delta \preceq \tau\},$$

where $App^{\gamma,\delta}$ is defined in the model $\mathcal{L}$. The term constructors of VPCF are interpreted by

$$\begin{aligned}(\mathcal{D}[\![\ulcorner n \urcorner]\!])_{\mathbf{num}_i} &= \begin{cases}\Omega_{\mathbf{num}_i} & \text{if } i < n \\ \mathcal{L}[\![\ulcorner n \urcorner_i]\!] & \text{otherwise}\end{cases} \\ \mathcal{D}[\![\mathsf{succ}]\!](x) &= \bigsqcup\{\Phi_{\mathbf{num}_i,\mathbf{num}}(\mathcal{L}[\![\mathsf{succ}_i]\!]((x)_{\mathbf{num}_i})) \mid i \geq 0\} \\ \mathcal{D}[\![\mathsf{pred}]\!](x) &= \bigsqcup\{\Phi_{\mathbf{num}_i,\mathbf{num}}(\mathcal{L}[\![\mathsf{pred}_i]\!]((x)_{\mathbf{num}_i})) \mid i \geq 0\} \\ \mathcal{D}[\![\mathsf{if0}^\sigma]\!](x,y,z) &= \bigsqcup\{\Phi_{\gamma,\sigma}(\mathcal{L}[\![\mathsf{if0}_i^\gamma]\!]((x)_{\mathbf{num}_i},(y)_\gamma,(z)_\gamma)) \mid i \geq 0, \gamma \preceq \sigma\} \\ \mathcal{D}[\![Y_\sigma]\!] &= \bigsqcup\{F^n \mid n \geq 0\}, \text{ where } F^0 = \uparrow^{(\sigma\to\sigma)\to\sigma} \\ &\qquad\qquad F^{m+1} = \mathcal{D}[\![\lambda f\colon \sigma \to \sigma.\, f\,(\lambda x\colon \tau.\, F^m\, f\, x)]\!]\end{aligned}$$

3.5.1 Extensionality and Continuity

Firstly note that by the definition of $\psi_{\gamma\to\delta,\gamma'\to\delta'}$, for any $x \in \mathcal{L}^{\gamma'\to\delta'}$, $\psi_{\gamma\to\delta,\gamma'\to\delta'}(x)\downarrow$ iff $x\downarrow$. Hence, for any $f \in D^{\sigma\to\tau}$,

$$f\downarrow \text{ iff } (f)_{\gamma\to\delta}\downarrow \text{ for any } \gamma \preceq \sigma, \delta \preceq \tau$$

Next, we can easily verify that for any $x \in D^{\sigma\to\tau}, y \in D^\sigma, \gamma \preceq \sigma, \delta \preceq \tau$,

$$(App^{\sigma,\tau}(x,\Phi_{\gamma,\sigma}(y)))_\delta = App^{\gamma,\delta}((x)_{\gamma\to\delta},(y)_\gamma)$$

Now, consider any $f, g \in D^{\sigma\to\tau}$. Suppose $f\downarrow$ implies that $g\downarrow$ and for all $x \downarrow\in D^{\sigma}$, $App(f,x) \leq_{\tau} App(f,x)$. Consider any $\gamma \preceq \sigma, \delta \preceq \tau$. Then $(f)_{\gamma\to\delta}\downarrow$ iff $f\downarrow$ and similarly for g. Hence $(f)_{\gamma\to\delta}\downarrow$ implies that $(g)_{\gamma\to\delta}\downarrow$. Consider any $d \in \mathcal{L}^{\gamma}$. Then for $\Phi_{\gamma,\sigma}(d) \in D^{\sigma}$, we have that $App(f, \Phi_{\gamma,\sigma}(d)) \leq_{\tau} App(g, \Phi_{\gamma,\sigma}(d))$. Hence $(App(f, \Phi_{\gamma,\sigma}(d))_{\delta} \leq_{\delta} (App(g, \Phi_{\gamma,\sigma}(d))_{\delta}$ which means that $App((f)_{\gamma\to\delta}, d) \leq_{\delta} App((g)_{\gamma\to\delta}, d)$. By extensionality of the model $\mathcal{L}$, this means that $(f)_{\gamma\to\delta} \leq_{\gamma\to\delta} (g)_{\gamma\to\delta}$ and from the pointwise ordering of $D^{\sigma\to\tau}$, this means that $f \leq_{\sigma\to\tau} g$. Similarly, we can prove continuity of the model $\mathcal{D}$ from the continuity of the model $\mathcal{L}$. We thus obtain the following proposition.

Proposition 3.5.1 *D is an extensional, monotone, continuous call-by-value frame.*

3.5.2 Model

To show that there are enough elements in $\mathcal{D}$ to interpret all VPCF terms, we use FVPCF terms to approximate the meanings of VPCF terms in the model $\mathcal{D}$. For any VPCF type σ, define the FVPCF type $(\sigma)_i$ to be the result of replacing all occurrences of **num** in σ with $\mathbf{num}_i$. More formally, for any VPCF type σ, and any $i \geq 0$, define the FVPCF type $(\sigma)_i$ as follows :

$$\begin{array}{rcl} (\mathbf{num})_i & = & \mathbf{num}_i \\ (\sigma \to \tau)_i & = & (\sigma)_i \to (\tau)_i \end{array}$$

Now let $M{:}\,\sigma$ be any VPCF term. For any $i \geq 0$, we define the FVPCF-term

$(M)_i \colon (\sigma)_i$ as follows:

$$\begin{array}{rcl}
(x^\sigma)_i & = & x^{(\sigma)_i}, \quad \text{for any variable } x^\sigma \\
(\ulcorner n \urcorner)_i & = & \begin{cases} \ulcorner n \urcorner_i & \text{if } n \leq i \\ \Omega_{\mathbf{num}_i} & \text{otherwise} \end{cases} \\
(\mathsf{succ}\, M)_i & = & \mathsf{succ}_i\, (M)_i \\
(\mathsf{pred}, M)_i & = & \mathsf{pred}_i\, (M)_i \\
(\mathsf{if0}^\sigma\; M\; N\; P)_i & = & \mathsf{if0}_i^{(\sigma)_i}\, (M)_i\, (N)_i\, (P)_i \\
(\lambda x \colon \sigma.\, M)_i & = & \lambda x \colon (\sigma)_i.\, (M)_i \\
(M\; N)_i & = & (M)_i\, (N)_i
\end{array}$$

To define $(\mathsf{Y}_\sigma)_i$, define the FVPCF-terms $F_i^n \colon ((\sigma \to \sigma) \to \sigma)_i$ for all $n \geq 0$ as follows:

$$\begin{array}{rcl}
F_i^0 & = & \Omega_{((\sigma \to \sigma) \to \sigma)_i} \\
F_i^{m+1} & = & \lambda g \colon (\sigma \to \sigma)_i.\, g\, (\lambda x \colon (\tau)_i.\, F_i^m\, g\, x)
\end{array}$$

where $\sigma = \tau \to \nu$. Then since each type is finite in $\mathcal{L}$, $\bigsqcup \{\mathcal{L}[\![F_i^n]\!] \mid n \geq 0\} = \mathcal{L}[\![F_i^m]\!]$ for some m. Take $(\mathsf{Y}_\sigma)_i$ to be F_i^m.

For any $\mathcal{D}$-environment ρ, define the $\mathcal{L}$-environment $(\rho)_i$ by

$$(\rho)_i(x \colon (\sigma)_i) \;=\; \Psi_{(\sigma)_i, \sigma}(\rho(x \colon \sigma))$$

Then, the following lemma establishes that $\mathcal{D}$ is a model of VPCF.

Lemma 3.5.2 *For any VPCF term M of type σ and any $\mathcal{D}$-environment ρ,*

$$\mathcal{D}[\![M]\!]\rho = \bigsqcup_{i \geq 0} \Phi_{(\sigma)_i, \sigma}(\mathcal{L}[\![(M)_i]\!](\rho)_i)$$

Proof By induction on the structure of M. ∎

It can also be easily verified that the interpretation of the constants of VPCF in $\mathcal{D}$ are sound. Hence, it follows that $\mathcal{D}$ is a sound model of VPCF.

Adequacy

Definition 3.5.3 Define a family of relations $\sqsubseteq_{\gamma,\sigma}$ for any $\gamma \preceq \sigma$, between closed L-terms $M{:}\,\gamma$ on the left and closed VPCF terms $N{:}\,\sigma$ on the right as follows. For any $M{:}\,\gamma$ and $N{:}\,\sigma$, define $M \sqsubseteq_{\gamma,\sigma} N$ if

1. $M \Downarrow \Omega_\gamma$, or
2. $M \Downarrow V$ and $N \Downarrow V'$ for some values V, V' such that $V \sqsubseteq_{\gamma,\sigma} V'$ where
   - $\ulcorner n \urcorner_i \sqsubseteq_{\mathbf{num}_i,\mathbf{num}} \ulcorner n \urcorner$
   - $\lambda x{:}\,\beta.\, M' \sqsubseteq_{\beta\to\delta,\rho\to\tau} \lambda y{:}\,\rho.\, N'$ if, for all closed terms $P{:}\,\beta, Q{:}\,\rho$, $P \sqsubseteq_{\beta,\rho} Q$ implies $(\lambda x{:}\,\beta.\, M')\, P \sqsubseteq_{\delta,\tau} (\lambda y{:}\,\rho.\, N')\, Q$

Lemma 3.5.4 *Consider any VPCF term $M{:}\,\sigma$ with free variables in $\{x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}\}$. For any $i \geq 0$, if $P_k{:}\,(\sigma_k)_i, Q_k{:}\,\sigma_k$ are closed terms such that $P_k \sqsubseteq_{(\sigma_k)_i,\sigma_k} Q_k$ for $k = 1, \ldots, n$ then*

$$[P_1, \ldots, P_n/x_1, \ldots, x_n](M)_i \sqsubseteq_{(\sigma)_i,\sigma} [Q_1, \ldots, Q_n/x_1, \ldots, x_n]M$$

Proof By induction on the structure of M. ∎

Theorem 3.5.5 (Adequacy) *If M is a closed term of type* **num**, *then $\mathcal{D}[\![M]\!] = \mathcal{D}[\![\ulcorner n \urcorner]\!]$ iff $M \Downarrow \ulcorner n \urcorner$.*

Proof The if direction follows by soundness of the model $\mathcal{D}$. For the only if direction, assume that $\mathcal{D}[\![M]\!] = \mathcal{D}[\![\ulcorner n \urcorner]\!]$. Then, by Lemma 3.5.2, this means that for some $i \geq 0$, $\mathcal{L}[\![(M)_i]\!] = \mathcal{L}[\![\ulcorner n \urcorner_i]\!]$. By the adequacy of the model $\mathcal{L}$, this means that $(M)_i \Downarrow \ulcorner n \urcorner_i$. By Lemma 3.5.4, $(M)_i \sqsubseteq_{\mathbf{num}_i,\mathbf{num}} M$ which by the definition of $\sqsubseteq_{\mathbf{num}_i,\mathbf{num}}$ implies that $M \Downarrow \ulcorner n \urcorner$. ∎

Full Abstraction

For types $\gamma \preceq \sigma$, we define VPCF terms $PS^{\sigma}_{\gamma}\colon \sigma \to \sigma$ as follows :

$$\begin{array}{rcl}
\Omega_{\mathbf{num}} & = & \mathsf{Y}_{\mathbf{num}}\,(\lambda x\colon \mathbf{num} \to \mathbf{num}.\, x)\, 0 \\
PS^{\,\mathbf{num}}_{\,\mathbf{num}_i} & = & \lambda x\colon \mathbf{num}.\, \mathsf{if0}^{\,\mathbf{num}}\, \underbrace{(\mathsf{pred}\ \ldots \mathsf{pred}\, x \ldots)}_{k \text{ copies}}\, x\, \Omega_{\mathbf{num}} \\
PS^{\rho\to\tau}_{\beta\to\delta} & = & \lambda f\colon \rho \to \tau.\, \lambda x\colon \rho.\, PS^{\tau}_{\delta}(f(PS^{\beta}_{\rho}(x)))
\end{array}$$

Lemma 3.5.6 *For any $x \in D^{\sigma}$,*

$$\mathcal{D}[\![PS^{\sigma}_{\gamma}]\!](x) = \Phi_{\gamma,\sigma}(\Psi_{\sigma,\gamma}(x))$$

Proof By induction on σ. ∎

Note that for any type γ in L, there is a unique type $\gamma^* = \sigma$ of VPCF with $\gamma \preceq \sigma$. For any term $M\colon \gamma$ in FVPCF, we define the term $(M)^*\colon \gamma^*$ of VPCF, by induction on the structure of M as follows :

$$\begin{array}{rcl}
(x^{\gamma})^* & = & PS^{\gamma^*}_{\gamma}\, x^{\gamma^*} \\
(\Omega_{\gamma})^* & = & \mathsf{Y}_{\gamma^*}\,(\lambda x\colon \mathbf{num} \to \gamma^*.\, x)\, 0 \\
(\ulcorner n \urcorner_i)^* & = & \ulcorner n \urcorner \\
(\mathsf{succ}_i(M))^* & = & PS^{\,\mathbf{num}}_{\,\mathbf{num}_i}(\mathsf{succ}\,(M)^*) \\
(\mathsf{pred}_i(M))^* & = & PS^{\,\mathbf{num}}_{\,\mathbf{num}_i}(\mathsf{pred}\,(M)^*) \\
(\mathsf{if0}^{\gamma}_i\ M\ N\ P)^* & = & \mathsf{if0}^{\sigma}\,(M)^*\,(N)^*\,(P)^* \\
(\phi_{i,i+1}(M))^* & = & (M)^* \\
(\psi_{i,i+1}(M))^* & = & PS^{\,\mathbf{num}}_{\,\mathbf{num}_i}((M)^*) \\
(\lambda x\colon \beta.\, M)^* & = & \lambda x\colon \beta^*.\, (M)^* \\
(M\, N)^* & = & (M)^*\,(N)^*
\end{array}$$

Lemma 3.5.7 *Let $M\colon \gamma$ be an FVPCF term. Then for any $\mathcal{D}$-environment ρ,*

$$\mathcal{D}[\![M^*]\!]\rho = \Phi_{\gamma,\sigma}(\mathcal{L}[\![M]\!]\rho')$$

where $\rho'(x^{\delta}) = \rho(x^{\delta^*})$.

Table 3.5: Additional Evaluation Rules for FPC.

$$\frac{M \Downarrow V \quad N \Downarrow V'}{\mathbf{pair}(M,N) \Downarrow \mathbf{pair}(V,V')}$$

$$\frac{M \Downarrow \mathbf{pair}(V,V')}{\mathbf{fst}(M) \Downarrow V} \qquad \frac{M \Downarrow \mathbf{pair}(V,V')}{\mathbf{snd}(M) \Downarrow V'}$$

$$\frac{M \Downarrow V}{\mathbf{inl}(M) \Downarrow \mathbf{inl}(V)} \qquad \frac{M \Downarrow V}{\mathbf{inr}(M) \Downarrow \mathbf{inr}(V)}$$

$$\frac{M \Downarrow \mathbf{inl}(V') \quad (N\ V') \Downarrow V}{\mathbf{case}(M,N,P) \Downarrow V} \qquad \frac{M \Downarrow \mathbf{inr}(V') \quad (P\ V') \Downarrow V}{\mathbf{case}(M,N,P) \Downarrow V}$$

$$\frac{M \Downarrow V}{\mathbf{intro}(M) \Downarrow \mathbf{intro}(V)} \qquad \frac{M \Downarrow \mathbf{intro}(V)}{\mathbf{elim}(M) \Downarrow V}$$

Proof By induction on M using Lemma 3.5.6. ∎

Corollary 3.5.8 *For any VPCF-type σ, every finite element in D^σ is definable in VPCF.*

Proof Let $x \in D^\sigma$ be a finite element. Because of the inverse limit construction of D^σ, x must be $\Phi_{\gamma,\sigma}(d)$ for some $\gamma \preceq \sigma, d \in \mathcal{L}^\gamma$. But then d is definable by some closed term M in FVPCF, and by Lemma 3.5.7, the term M^* defines x. ∎

Theorem 3.5.9 (Full Abstraction) *For any PCF terms M and N, $M \sim_{\mathrm{VPCF}} N$ iff for all environments ρ such that $\rho(x)\downarrow$ for all x, $\mathcal{D}[\![M]\!]\rho = \mathcal{D}[\![N]\!]\rho$.*

Proof The model $\mathcal{D}$ consists of algebraic cpo's and therefore full abstraction follows from definability of finite elements, by standard arguments [Gun92, Plo77]. ∎

3.6 Full Abstraction for FPC

3.6.1 The language and its operational semantics

FPC, like PCF, fits the general form of a call-by-value functional language. The grammar

$$\sigma \ ::= \ t \mid \mathbf{unit} \mid (\sigma \times \sigma) \mid (\sigma + \sigma) \mid (\sigma \to \sigma) \mid \mu t.\, \sigma$$

generates the set of FPC types, where t ranges over a set of type variables. The only base type of FPC, **unit**, contains two elements, a divergent element and an element that halts; it is thus like the **unit** type of Standard ML [MTH90]. FPC has term constructors

$$\begin{array}{lll} \langle \mathbf{fst}; (\sigma \times \tau) \Rightarrow \sigma \rangle & \langle \mathbf{snd}; (\sigma \times \tau) \Rightarrow \sigma \rangle & \langle \mathbf{pair}; \sigma, \tau \Rightarrow (\sigma \times \tau) \rangle \\ \langle \mathbf{inl}; \sigma \Rightarrow (\sigma + \tau) \rangle & \langle \mathbf{inr}; \tau \Rightarrow (\sigma + \tau) \rangle & \langle \mathbf{case}; (\sigma + \tau), (\sigma \to \rho), (\tau \to \rho) \Rightarrow \rho \rangle \\ \langle \mathbf{unity}; \mathbf{unit} \rangle & \langle \mathbf{intro}; [\mu t.\, \sigma / t]\sigma \Rightarrow \mu t.\, \sigma \rangle & \langle \mathbf{elim}; \mu t.\, \sigma \Rightarrow [\mu t.\, \sigma / t]\sigma \rangle. \end{array}$$

The values in FPC are generated by the grammar

$$V ::= \mathbf{unity} \mid (\lambda x.\, M) \mid \mathbf{pair}(V, V) \mid \mathbf{inl}(V) \mid \mathbf{inr}(V) \mid \mathbf{intro}(V).$$

and the rules in Tables 3.1 and 3.5 completely define the operational semantics of FPC. The operational equivalence relation $M \sim_{\mathrm{FPC}} N$ is defined exactly as it is in VPCF.

At first glance FPC looks like a modest language, but it is very expressive and many useful programming languages can be encoded in FPC. PCF, for example, can be represented in FPC by taking $\mathbf{num} = (\mu t.\, \mathbf{unit} + t)$, a shorthand for the unwinding $(\mathbf{unit} + \mathbf{unit} + \mathbf{unit} + \ldots)$, which builds a set of elements that has one divergent element below all others. The numeral 0 may be represented by $\mathbf{intro}(\mathbf{inl}(\mathbf{unity}))$; the other numerals have similar representations, as do the other term constructors of PCF (see [Gun92] for a more complete account). Lists of type σ can also be represented in FPC by the type $(\mu t.\, \mathbf{unit} + (\sigma \times t))$. Indeed, languages with references and assignments, control operations, and other features can also be represented within

Table 3.6: Additional Evaluation Rules for FFPC.

$$\frac{M \Downarrow \mathbf{pair}(V, V')}{\mathbf{fst}(M) \Downarrow V} \qquad \frac{M \Downarrow \Omega}{\mathbf{fst}(M) \Downarrow \Omega}$$

$$\frac{M \Downarrow \mathbf{pair}(V, V')}{\mathbf{snd}(M) \Downarrow V'} \qquad \frac{M \Downarrow \Omega}{\mathbf{snd}(M) \Downarrow \Omega}$$

$$\frac{M \Downarrow V}{\mathbf{inl}(M) \Downarrow \mathbf{inl}(V)} \qquad \frac{M \Downarrow \Omega}{\mathbf{inl}(M) \Downarrow \Omega}$$

$$\frac{M \Downarrow V}{\mathbf{inr}(M) \Downarrow \mathbf{inr}(V)} \qquad \frac{M \Downarrow \Omega}{\mathbf{inr}(M) \Downarrow \Omega}$$

$$\frac{M \Downarrow V \quad N \Downarrow V'}{\mathbf{pair}(M, N) \Downarrow \mathbf{pair}(V, V')}$$

$$\frac{M \Downarrow \Omega}{\mathbf{pair}(M, N) \Downarrow \Omega} \qquad \frac{N \Downarrow \Omega}{\mathbf{pair}(M, N) \Downarrow \Omega}$$

$$\frac{M \Downarrow \mathbf{inl}(V') \quad (N\ V') \Downarrow R}{\mathbf{case}(M, N, P) \Downarrow R} \qquad \frac{M \Downarrow \mathbf{inr}(V') \quad (P\ V') \Downarrow R}{\mathbf{case}(M, N, P) \Downarrow R}$$

$$\frac{M \Downarrow \Omega}{\mathbf{case}(M, N, P) \Downarrow \Omega}$$

FPC. This makes a fully abstract model of FPC particularly important, since it provides the structure necessary to give a good denotational semantics for all languages that can be translated into FPC [RV95].

3.6.2 Partial types

The construction of a fully abstract model for FPC follows along the same lines as the construction for VPCF. We first define a "finite" language, construct a term model for the language, and then build the model of FPC using this term model.

The "finite" language for FPC, called FFPC, essentially consists of FPC stripped of its recursive types. Define FFPC to be the partial call-by-value language with types given by the grammar

$$\beta, \gamma \ ::= \mathbf{void} \mid \mathbf{unit} \mid (\beta \times \gamma) \mid (\beta + \gamma) \mid (\beta \to \gamma)$$

and containing all the term constructors of FPC except **intro** and **elim**. Intuitively, the base type **void** represents the set with only one element, namely the divergent one. The values of FFPC are generated by the same grammar as for FPC, but without the **intro** case. The operational rules for the term constructors of FFPC appear in Table 3.6. As for FVPCF, let $\mathcal{L}$ be the term model for FFPC; this model is extensional and monotone in which every type is finite, and every element is definable.

3.6.3 A Category of Finite Type Approximations

We use the model $\mathcal{L}$ of FFPC to construct the model of FPC. As for VPCF, each type is constructed from a set of FFPC types that serve as its approximations. In domain theory, the notion of embedding-projection pairs is successfully used to solve recursive domain equations — we need an anologous notion for objects that are characterized by a set of FFPC type approximations. We hence arrive at the following category to interperet recursive types.

Definition 3.6.1 Define the category $\mathbf{C}^{EP}$ as follows:

- The objects are tuples $\mathcal{T} = \left\langle \{\mathcal{T}_n\}_{n<\omega}, \{\delta_n^{\mathcal{T}}\}_{n<\omega} \right\rangle$ where $\mathcal{T}_n$ is a type in FFPC and $\delta_n^{\mathcal{T}}$ is an embedding-projection pair between $\mathcal{L}[\![\mathcal{T}_n]\!]$ and $\mathcal{L}[\![\mathcal{T}_{n+1}]\!]$ that is definable in FFPC.
- A morphism $\mathcal{F}$ from an object $\mathcal{S}$ to an object $\mathcal{T}$ is a family of maps $\{\mathcal{F}_n\}_{n<\omega}$ such that $\mathcal{F}_n$ is an embedding-projection pair between $\mathcal{L}[\![\mathcal{S}_n]\!]$ and $\mathcal{L}[\![\mathcal{T}_n]\!]$ that is

definable in FFPC, where the following diagram commutes for all n:

$$\begin{tikzcd}
\mathcal{L}[\![\mathcal{S}_{n+1}]\!] \arrow[r, "\mathcal{F}^E_{n+1}"] & \mathcal{L}[\![\mathcal{T}_{n+1}]\!] \\
\mathcal{L}[\![\mathcal{S}_n]\!] \arrow[r, "\mathcal{F}^E_n"] \arrow[u, "(\delta^{\mathcal{S}}_n)^E"] & \mathcal{L}[\![\mathcal{T}_n]\!] \arrow[u, "(\delta^{\mathcal{T}}_n)^E"']
\end{tikzcd}$$

It is convenient to have a few notational conventions. For any object $\mathcal{T}$ of $\mathbf{C}^{EP}$, define $\mathsf{id}_{\mathcal{T}}$ as $(\mathsf{id}_{\mathcal{T}})^E_n = (\mathsf{id}_{\mathcal{T}})^P_n = (\lambda x{:}\,\mathcal{T}_n.\, x)$. For any ep-pairs $f_1{:}\,\beta_1 \to \gamma_1$ and $f_2{:}\,\beta_2 \to \gamma_2$ definable in FFPC, define the ep-pairs $f_1 \odot f_2{:}\,(\beta_1 \odot \beta_2) \to (\gamma_1 \odot \gamma_2)$, where $\odot = +, \times$, or $\to$, by

$$\begin{aligned}
(f_1 + f_2)^E &= \lambda h{:}\,\beta_1 + \beta_2.\,\mathbf{case}(h, (\lambda x.\,\mathbf{inl}(f_1^E\ x)), (\lambda x.\,\mathbf{inr}(f_2^E\ x))) \\
(f_1 + f_2)^P &= \lambda h{:}\,\gamma_1 + \gamma_2.\,\mathbf{case}(h, (\lambda x.\,\mathbf{inl}(f_1^P\ x)), (\lambda x.\,\mathbf{inr}(f_2^P\ x))) \\
(f_1 \times f_2)^E &= \lambda h{:}\,\beta_1 \times \beta_2.\,\mathbf{pair}(f_1^E\ (\mathbf{fst}\ h), f_2^E\ (\mathbf{snd}\ h)) \\
(f_1 \times f_2)^P &= \lambda h{:}\,\gamma_1 \times \gamma_2.\,\mathbf{pair}(f_1^P\ (\mathbf{fst}\ h), f_2^P\ (\mathbf{snd}\ h)) \\
(f_1 \to f_2)^E &= \lambda h{:}\,\beta_1 \to \beta_2.\,\lambda x{:}\,\gamma_1.\,f_2^E\ (h\ (f_1^P\ x)) \\
(f_1 \to f_2)^P &= \lambda h{:}\,\gamma_1 \to \gamma_2.\,\lambda x{:}\,\beta_1.\,f_2^P\ (h\ (f_1^E\ x))
\end{aligned}$$

Then for any objects $\mathcal{S}$ and $\mathcal{T}$, define the object $\mathcal{S} \odot \mathcal{T}$ by $(\mathcal{S} \odot \mathcal{T})_n = \mathcal{S}_n \odot \mathcal{T}_n$ and $\delta^{\mathcal{S}\odot\mathcal{T}}_n = \delta^{\mathcal{S}}_n \odot \delta^{\mathcal{T}}_n$, and if $\mathcal{S}_1 \xrightarrow{\mathcal{F}_1} \mathcal{T}_1$ and $\mathcal{S}_2 \xrightarrow{\mathcal{F}_2} \mathcal{T}_2$ are morphisms in $\mathbf{C}^{EP}$, define the morphism $\mathcal{F}_1 \odot \mathcal{F}_2$ from $(\mathcal{S}_1 \odot \mathcal{S}_2)$ to $(\mathcal{T}_1 \odot \mathcal{T}_2)$ as $(\mathcal{F}_1 \odot \mathcal{F}_2)_n = (\mathcal{F}_1)_n \odot (\mathcal{F}_2)_n$. Finally, if γ is a type in FFPC, let $\overline{\gamma}$ denote the object defined by $(\overline{\gamma})_n = \gamma$ and $(\delta^{\overline{\gamma}}_n)^E = (\delta^{\overline{\gamma}}_n)^P = (\lambda x{:}\,\gamma.\,x)$.

We now give a preliminary meaning of types in this category; the meaning of types in FPC will be defined as inverse limits of these objects. A **type environment** η is a map from type variables to objects of $\mathbf{C}^{EP}$. Similarly, an **EP-environment** Δ is a map from type variables to ep-pairs. Let Δ_{dom} (Δ_{cod}) denote the type environment mapping t to the domain (codomain, or range) of the ep-pair $\Delta(t)$. For any type environment η, the EP-environment id_η is defined as $\mathsf{id}_\eta(t) = \mathsf{id}_{\eta(t)}$. For any FPC type σ, we define the object $\{\![\sigma]\!\}\,\eta$ which intuitively corresponds to its finite type approximations. However, defining the approximations of a recursive type requires us to define simultaneously the morphism $\{\![\sigma]\!\}\,\Delta\colon \{\![\sigma]\!\}\,\Delta_{dom} \to \{\![\sigma]\!\}\,\Delta_{cod}$.

- For type variables, $\{\![t]\!\}\,\eta = \eta(t)$ and $\{\![t]\!\}\,\Delta = \Delta(t)$.
- For **unit**, $\{\![\mathbf{unit}]\!\}\,\eta = \overline{\mathbf{unit}}$ and $\{\![\mathbf{unit}]\!\}\,\Delta = \mathsf{id}_{\overline{\mathbf{unit}}}$.
- For $\odot = +, \times$, or $\to$, $\{\![\sigma_1 \odot \sigma_2]\!\}\,\eta = \{\![\sigma_1]\!\}\,\eta \odot \{\![\sigma_2]\!\}\,\eta$ and $\{\![\sigma_1 \odot \sigma_2]\!\}\,\Delta = \{\![\sigma_1]\!\}\,\Delta \odot \{\![\sigma_2]\!\}\,\Delta$.
- To define $\{\![\mu t.\,\sigma]\!\}\,\eta$, first construct the "unwinding" objects $U^i_{\sigma,\eta}$ and morphisms $F^i_{\sigma,\eta}\colon U^i_{\sigma,\eta} \to U^{i+1}_{\sigma,\eta}$ for $i \geq 0$:

$$
\begin{array}{rcl}
U^0_{\sigma,\eta} & = & \overline{\mathbf{void}} \\
U^{i+1}_{\sigma,\eta} & = & \{\![\sigma]\!\}\,\eta[t \mapsto U^i_{\sigma,\eta}] \\[1ex]
(F^0_{\sigma,\eta})^E_j & = & \lambda x\colon \mathbf{void}.\, \Omega_{(U^1_{\sigma,\eta})_j} \\
(F^0_{\sigma,\eta})^P_j & = & \lambda x\colon (U^1_{\sigma,\eta})_j.\, \Omega_{\mathbf{void}} \\[1ex]
F^{i+1}_{\sigma,\eta} & = & \{\![\sigma]\!\}\,\mathsf{id}_\eta[t \mapsto F^i_{\sigma,\eta}]
\end{array}
$$

  Then set $(\{\![\mu t.\,\sigma]\!\}\,\eta)_n = (U^n_{\sigma,\eta})_n$ and $\delta_n^{\{\![\mu t.\,\sigma]\!\}\,\eta} = \delta_n^{U^{n+1}_{\sigma,\eta}} \circ (F^n_{\sigma,\eta})_n$. This definition can be viewed graphically as:

$$
\begin{tikzcd}
\vdots & & \vdots & & \vdots \\
(U^1_{\sigma,\eta})_0 \arrow[rr, "\delta_0^{U^1_{\sigma,\eta}}"] & & (U^1_{\sigma,\eta})_1 \arrow[rr, "\delta_1^{U^1_{\sigma,\eta}}"] & & (U^1_{\sigma,\eta})_2 \;\cdots\cdots \\
\mathbf{void} \arrow[u, "(F^0_{\sigma,\eta})_0"] \arrow[rr, "\mathsf{id}"'] & & \mathbf{void} \arrow[u, "(F^0_{\sigma,\eta})_1"] \arrow[rr, "\mathsf{id}"'] & & \mathbf{void} \arrow[u, "(F^0_{\sigma,\eta})_2"] \;\cdots\cdots
\end{tikzcd}
$$

  The commuting diagram condition on morphisms of $\mathbf{C}^{EP}$ ensures that any path between two points on this grid yields the same embedding-projection pair.

To define $\{\![\mu t.\, \sigma]\!\}\,\Delta$, define morphisms $G^i_{\sigma,\Delta}\colon U^i_{\sigma,\Delta_{dom}} \to U^i_{\sigma,\Delta_{cod}}$ by $G^0_{\sigma,\Delta} = \mathsf{id}_{\overline{\mathbf{void}}}$ and $G^{i+1}_{\sigma,\Delta} = \{\![\sigma]\!\}\,\Delta[t \mapsto G^i_{\sigma,\Delta}]$. Then, $\{\![\mu t.\, \sigma]\!\}\,\Delta = (G^n_{\sigma,\Delta})_n$.

We can prove that with the above definitions $\{\![\sigma]\!\}\,\eta$ is indeed an object of $\mathbf{C}^{EP}_{\mathbf{L}}$. This is done by induction on σ, but for the proof to work it is important to simultaneously also establish that $\{\![\sigma]\!\}\,\Delta$ is a "functor" on $\mathbf{C}^{EP}_{\mathbf{L}}$, *i.e.*, it preserves composition. This is proved in the following lemma, where for EP-environments Δ_1 and Δ_2, the EP-environment $\Delta_1 \circ \Delta_2$ is defined as

$$(\Delta_1 \circ \Delta_2)(t) = \Delta_1(t) \circ \Delta_2(t)$$

For ease of readability, we postpone the proofs of the lemmas of this section to Section 3.6.5.

Lemma 3.6.2 *For any type σ, type-environment η, and EP-environment Δ*

1. $\{\![\sigma]\!\}\,\eta$ *is an object of* $\mathbf{C}^{EP}_{\mathbf{L}}$.
2. $\{\![\sigma]\!\}\,\Delta\colon \{\![\sigma]\!\}\,\Delta_{dom} \to \{\![\sigma]\!\}\,\Delta_{cod}$ *is a morphism in* $\mathbf{C}^{EP}_{\mathbf{L}}$.
3. $\{\![\sigma]\!\}\,(\Delta_1 \circ \Delta_2) = (\{\![\sigma]\!\}\,\Delta_1) \circ (\{\![\sigma]\!\}\,\Delta_2)$.

The following lemma characterizes the type approximations of an unwinding of a recursive type, and is very useful in establishing various properties of the model.

Lemma 3.6.3 *For any FPC type σ and type environment η,*

$$\begin{array}{rcl} (\{\![[\mu t.\, \sigma/t]\sigma]\!\}\,\eta)_n & = & (U^{n+1}_{\sigma,\eta})_n \\ \delta^{\{\![[\mu t.\, \sigma/t]\sigma]\!\}\,\eta}_n & = & \delta^{U^{n+2}_{\sigma,\eta}}_n \circ (F^{n+1}_{\sigma,\eta})_n \end{array}$$

3.6.4 Fully Abstract Model for FPC

If $\mathcal{T}$ is an object of $\mathbf{C}^{EP}$, define the cpo $\lim_{\leftarrow} \mathcal{T}$ as the the set

$$\{x \in \prod_{n<\omega} \mathcal{L}[\![\mathcal{T}_n]\!] \mid \forall i < \omega.(x)_i = (\delta^{\mathcal{T}}_i)^P (x)_{i+1}\}$$

ordered pointwise, *i.e.*,

$$x \leq y \text{ iff } \forall n < \omega.(x)_n \leq (y)_n$$

Define the embedding-projection pairs $\Phi_i^{\mathcal{T}}, \Psi_i^{\mathcal{T}}$ between the cpo's $\mathcal{L}[\![\mathcal{T}_i]\!]$ and $\lim_{\leftarrow} \mathcal{T}$ as follows:

$$\begin{array}{rcl} (\Phi_i^{\mathcal{T}}(d))_j & = & \begin{cases} d & \text{if } j = i \\ (\delta_j^{\mathcal{T}})^P((\delta_{j+1}^{\mathcal{T}})^P(\ldots(\delta_{i-1}^{\mathcal{T}})^P(d))\ldots) & \text{if } 0 \leq j < i \\ (\delta_{j-1}^{\mathcal{T}})^E((\delta_{j-2}^{\mathcal{T}})^E(\ldots(\delta_i^{\mathcal{T}})^E(d))\ldots) & \text{if } j > i \end{cases} \\ \Psi_i^{\mathcal{T}}(x) & = & (x)_i \end{array}$$

We can now define the call-by-value model $\mathcal{A}$ for FPC. Let η be a type environment as before, *i.e.*, a map from type variables to objects of $\mathbf{C}^{EP}$. Then for any FPC type σ,

$$\mathcal{A}[\![\sigma]\!]\,\eta = \lim_{\leftarrow} \{\![\sigma]\!\}\,\eta$$

Divergence and application are interpreted as in $\mathcal{D}$, *i.e.*, for any $x \in \mathcal{A}[\![\sigma]\!]\,\eta$, $x\!\uparrow$ iff $\forall i < \omega.(x)_i\!\uparrow$; and for any $x \in \mathcal{A}[\![\sigma_1 \to \sigma_2]\!]\,\eta$ and $y \in \mathcal{A}[\![\sigma_1]\!]\eta$

$$App(x, y) = \bigsqcup \{\Phi_i^{\{\![\sigma_2]\!\}\,\eta}(App((x)_i, (y)_i)) \mid i < \omega\}$$

For any term constructor $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma\rangle$ except **intro** and **elim**,

$$\mathcal{A}[\![f]\!](x_1, \ldots, x_k) = \bigsqcup \{\Phi_i^{\{\![\sigma]\!\}\,\eta}(\mathcal{L}[\![f]\!]((x_1)_i, \ldots, (x_k)_i)) \mid i < \omega\}$$

Finally, if $x \in \mathcal{A}[\![[\mu t.\, \sigma/t]\sigma]\!]\,\eta, y \in \mathcal{A}[\![\mu t.\, \sigma]\!]\,\eta$,

$$\begin{array}{rcl} \mathcal{A}[\![\mathbf{intro}]\!](x) & = & \bigsqcup \{\Phi_i^{\{\![\mu t.\, \sigma]\!\}\,\eta}((F_{\sigma,\eta}^i)_i^P\ (x)_i) \mid i < \omega\} \\ \mathcal{A}[\![\mathbf{elim}]\!](y) & = & \bigsqcup \{\Phi_i^{\{\![[\mu t.\, \sigma/t]\sigma]\!\}\,\eta}((F_{\sigma,\eta}^i)_i^E\ (y)_i) \mid i < \omega\} \end{array}$$

where the two definitions make sense by Lemma 3.6.3.

To show that the model $\mathcal{A}$ has enough elements to interpret all terms of FPC, let M be an FPC term of type σ whose free variables are among $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$. For any type environment η and $i < \omega$, define the FFPC term M_i^η of type $(\{\![\sigma]\!\}\,\eta)_i$ whose free variables are among $x_1^{(\{\![\sigma_1]\!\}\,\eta)_i}, \ldots, x_n^{(\{\![\sigma_n]\!\}\,\eta)_n}$, by induction on the structure of M:

$$
\begin{array}{rclrcl}
(x^\sigma)_i^\eta & = & x^{(\{\![\sigma]\!\}\,\eta)_i} & (f(M_1,\ldots,M_k))_i^\eta & = & f((M_1)_i^\eta,\ldots,(M_k)_i^\eta) \\
(\lambda x{:}\,\sigma_1.\, M)_i^\eta & = & \lambda x{:}\,(\{\![\sigma_1]\!\}\,\eta)_i.\, M_i^\eta & (M\ N)_i^\eta & = & M_i^\eta\ N_i^\eta \\
(\mathbf{intro}(M))_i^\eta & = & (F_{\sigma,\eta}^i)_i^P\ M_i^\eta & (\mathbf{elim}(M))_i^\eta & = & (F_{\sigma,\eta}^i)_i^E\ M_i^\eta
\end{array}
$$

where f is any term constructor of FPC except **intro** and **elim**, and the cases for **intro**, **elim** are well-defined by virtue of the fact that every element of $\mathcal{L}$, including $(F_{\sigma,\eta}^i)_i^E, (F_{\sigma,\eta}^i)_i^P$ is definable by a term in $\mathcal{L}$. Let η be a type environment. An η-**compatible value environment** ρ is a map on variables such that $\rho(x^\sigma) \in \mathcal{A}[\![\sigma]\!]\,\eta$. For any value environment ρ compatible with a type environment η, define the $\mathcal{L}$ environment ρ_i^η as $\rho_i^\eta(x^{(\{\![\sigma]\!\}\,\eta)_i}) = \Psi_i^{\{\![\sigma]\!\}\,\eta}(\rho(x^\sigma))$. The following lemma shows that $\mathcal{A}$ is a model of FPC.

Lemma 3.6.4 *For any term $M{:}\,\sigma$, type environment η, and η-compatible value environment ρ,*

$$\mathcal{A}[\![M]\!]\eta\rho = \bigsqcup\{\Phi_i^{\{\![\sigma]\!\}\,\eta}(\mathcal{L}[\![M_i^\eta]\!]\,\rho_i^\eta) \mid i < \omega\}.$$

Proof By induction on the structure of M. ∎

Finally, we can establish that the model is adequate and fully abstract whose proofs are exactly as for VPCF.

Theorem 3.6.5 (Adequacy) *Suppose M is a closed FPC term of type* **unit**. *Then $\mathcal{A}[\![M]\!] = \mathcal{A}[\![$*__unity__*$]\!]$ iff $M \Downarrow$* **unity**.

Theorem 3.6.6 (Full Abstraction) *For any FPC terms M and N of closed type, $M \sim_{\mathrm{FPC}} N$ iff for all environments ρ such that $\rho(x){\downarrow}$ for all x, $\mathcal{A}[\![M]\!]\rho = \mathcal{A}[\![N]\!]\rho$.*

3.6.5 Proofs

Proof of Lemma 3.6.2 By induction on the structure of σ. We just show few representative cases.

- $\sigma \equiv t$. Trivially follows from the fact that η and Δ map t to an object and morphism of $\mathbf{C}_{\mathbf{L}}^{EP}$ respectively, and from the definition of composition of EP-environments.

- $\sigma \equiv \sigma_1 \to \sigma_2$.

  1. Since $(\{\![\sigma_1]\!\}\,\eta)_i$ and $(\{\![\sigma_2]\!\}\,\eta)_i$ are types of FFPC, so is $(\{\![\sigma]\!\}\,\eta)_i$. And, since $\delta_i^{\{\![\sigma_1]\!\}\,\eta}, \delta_i^{\{\![\sigma_2]\!\}\,\eta}$ are embedding-projections, so is $\delta_n^{\{\![\sigma]\!\}\,\eta}$.

  2. Clearly, using the induction hypothesis, each $(\{\![\sigma]\!\}\,\Delta)_i$ is an embedding-projection pair. We need to prove that the following diagram commutes:

$$\begin{array}{ccc}
(\{\![\sigma]\!\}\,\Delta_{dom})_{n+1} & \xrightarrow{(\{\![\sigma]\!\}\,\Delta)^E_{n+1}} & (\{\![\sigma]\!\}\,\Delta_{cod})_{n+1} \\
(\delta_n^{\{\![\sigma]\!\}\,\Delta_{dom}})^E \Big\uparrow & & \Big\uparrow (\delta_n^{\{\![\sigma]\!\}\,\Delta_{cod}})^E \\
(\{\![\sigma]\!\}\,\Delta_{dom})_n & \xrightarrow{(\{\![\sigma]\!\}\,\Delta)^E_n} & (\{\![\sigma]\!\}\,\Delta_{cod})_n
\end{array}$$

  Consider any $h\colon (\{\![\sigma]\!\}\,\Delta_{dom})_n$. If $h = \Omega$, then clearly both compositions will yield Ω , since all elements of $\mathcal{L}$ are strict. Otherwise,

$$\begin{aligned}
& ((\delta_n^{\{\![\sigma]\!\}\,\Delta_{cod}})^E \circ (\{\![\sigma]\!\}\,\Delta)^E_n)\ h \\
= \ & (\delta_n^{\{\![\sigma]\!\}\,\Delta_{cod}})^E\ (\lambda x\colon (\{\![\sigma_1]\!\}\,\Delta_{cod})_n .\ (\{\![\sigma_2]\!\}\,\Delta)^E_n\ (h\ ((\{\![\sigma_1]\!\}\,\Delta)^P_n\ x))) \\
= \ & \lambda x\colon (\{\![\sigma_1]\!\}\,\Delta_{cod})_{n+1} . \\
& \quad (\delta_n^{\{\![\sigma_1]\!\}\,\Delta_{cod}})^E\ ((\{\![\sigma_2]\!\}\,\Delta)^E_n\ (h\ ((\{\![\sigma_1]\!\}\,\Delta)^P_n\ ((\delta_n^{\{\![\sigma_1]\!\}\,\Delta_{cod}})^P\ x)))) \\
= \ & \lambda x\colon (\{\![\sigma_1]\!\}\,\Delta_{cod})_{n+1} . \\
& \quad (\{\![\sigma_2]\!\}\,\Delta)^E_{n+1}\ ((\delta_n^{\{\![\sigma_2]\!\}\,\Delta_{dom}})^E\ (h\ ((\delta_n^{\{\![\sigma_1]\!\}\,\Delta_{dom}})^P\ ((\{\![\sigma_1]\!\}\,\Delta)^P_{n+1}\ x)))) \\
& \ \text{(by induction hypothesis about } \sigma_1 \text{ and } \sigma_2) \\
= \ & ((\{\![\sigma]\!\}\,\Delta)^E_{n+1} \circ (\delta_n^{\{\![\sigma]\!\}\,\Delta_{dom}})^E)\ h
\end{aligned}$$

3.

$$
\begin{array}{cl}
 & (\{\![\sigma]\!\}\,(\Delta_1 \circ \Delta_2))^E_n \\
= & ((\{\![\sigma_1]\!\}\,(\Delta_1 \circ \Delta_2))_n \to (\{\![\sigma_2]\!\}\,(\Delta_1 \circ \Delta_2))_n)^E \\
= & (((\{\![\sigma_1]\!\}\,\Delta_1)_n \circ (\{\![\sigma_1]\!\}\,\Delta_2)_n) \to ((\{\![\sigma_2]\!\}\,\Delta_1)_n \circ (\{\![\sigma_2]\!\}\,\Delta_2)_n))^E \\
 & \quad \text{(by induction hypothesis about } \sigma_1 \text{ and } \sigma_2) \\
= & \lambda h.\, \lambda x.\, (\{\![\sigma_2]\!\}\,\Delta_1)^E\, ((\{\![\sigma_2]\!\}\,\Delta_2)^E_n\, (h\, ((\{\![\sigma_1]\!\}\,\Delta_2)^P_n\, ((\{\![\sigma_1]\!\}\,\Delta_1)^P_n\, x)))) \\
= & ((\{\![\sigma_1]\!\}\,\Delta_1)_n \to (\{\![\sigma_2]\!\}\,\Delta_1)_n)^E \circ ((\{\![\sigma_1]\!\}\,\Delta_2)_n \to (\{\![\sigma_2]\!\}\,\Delta_2)_n)^E \\
= & (\{\![\sigma]\!\}\,\Delta_1)^E_n \circ (\{\![\sigma]\!\}\,\Delta_2)^E_n
\end{array}
$$

- $\sigma \equiv \mu t.\, \sigma_1$.

  1. By induction on j, we can prove that $U^j_{\sigma_1,\eta}$ is an object of $\mathbf{C}^{EP}_{\mathbf{L}}$, using 1 of the induction hypothesis about σ_1. Hence $(\mu t.\, \sigma_1)_n$ is a type of FFPC, and $\delta_n^{U^{n+1}_{\sigma_1,\eta}}$ is an embedding-projection in $\mathcal{L}$. Using 2 of the induction hypothesis about σ_1, we can now establish by induction on j that $F^j_{\sigma_1,\eta}$ is a morphism of $\mathbf{C}^{EP}_{\mathbf{L}}$. Hence $(F^n_{\sigma_1,\eta})_n$ is an emedding-projection and since the composition of two embedding-projections is an embedding-projection so is $\delta_n^{\{\![\mu t.\, \sigma_1]\!\}\,\eta}$.

  2. By induction on j, and using 2 of the induction hypothesis about σ_1, we can prove that $G^j_{\sigma_1,\Delta}$ is a morphism in $\mathbf{C}^{EP}_{\mathbf{L}}$. Hence, $(G^n_{\sigma_1,\eta})_n$ is an embedding-projection in $\mathcal{L}$. We now prove that the following diagram commutes :

$$
\begin{tikzcd}
(\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{dom})_{n+1} \arrow[r, "(\{\![\mu t.\, \sigma_1]\!\}\,\Delta)^E_{n+1}"] & (\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{cod})_{n+1} \\
(\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{dom})_n \arrow[u, "(\delta_n^{\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{dom}})^E"] \arrow[r, "(\{\![\mu t.\, \sigma_1]\!\}\,\Delta)^E_n"] & (\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{cod})_n \arrow[u, "(\delta_n^{\{\![\mu t.\, \sigma_1]\!\}\,\Delta_{cod}})^E"']
\end{tikzcd}
$$

  We first state and prove the following proposition which makes crucial use of 3 of the induction hypothesis about σ_1.

Proposition 3.6.7 *For all $n \geq 0$, the following diagram commutes :*

$$\begin{tikzcd}
(U^{n+1}_{\sigma_1,\Delta_{dom}})_n \arrow[r, "(G^{n+1}_{\sigma_1,\Delta})^E_n"] & (U^{n+1}_{\sigma_1,\Delta_{cod}})_n \\
(U^{n}_{\sigma_1,\Delta_{dom}})_n \arrow[u, "(F^{n}_{\sigma_1,\Delta_{dom}})^E_n"] \arrow[r, "(G^{n}_{\sigma_1,\Delta})^E_n"] & (U^{n}_{\sigma_1,\Delta_{cod}})_n \arrow[u, "(F^{n}_{\sigma_1,\Delta_{cod}})^E_n"']
\end{tikzcd}$$

Proof We prove by induction on n that

$$F^n_{\sigma_1,\Delta_{cod}} \circ G^n_{\sigma_1,\Delta} = G^{n+1}_{\sigma_1,\Delta} \circ F^n_{\sigma_1,\Delta_{dom}}$$

which implies the required statement.

– $n = 0$. The following diagram commutes for all k,

$$\begin{tikzcd}
(U^{1}_{\sigma_1,\Delta_{dom}})_k \arrow[r, "(G^{1}_{\sigma_1,\Delta})^E_k"] & (U^{1}_{\sigma_1,\Delta_{cod}})_k \\
\mathbf{void} \arrow[u, "(F^{0}_{\sigma_1,\Delta_{dom}})^E_k"] \arrow[ur, "!"] \arrow[r, "(G^{0}_{\sigma_1,\Delta})^E_k"] & \mathbf{void} \arrow[u, "(F^{0}_{\sigma_1,\Delta_{cod}})^E_k"']
\end{tikzcd}$$

since **void** is the initial object in $\mathbf{C}^{EP}_{\mathbf{L}}$ and hence ! is a unique arrow from **void** to $(U^1_{\sigma_1,\Delta_{cod}})_0$.

– $n = j + 1$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{F^{j+1}_{\sigma_1,\Delta_{cod}} \circ G^{j+1}_{\sigma_1,\Delta}} \\
= & (\{\![\sigma_1]\!\}\, \mathsf{id}_{\Delta_{cod}}[t \mapsto F^{j}_{\sigma_1,\Delta_{cod}}]) \circ (\{\![\sigma_1]\!\}\, \Delta[t \mapsto G^{j}_{\sigma_1,\Delta}]) \\
= & \{\![\sigma_1]\!\}\, \Delta[t \mapsto F^{j}_{\sigma_1,\Delta_{cod}} \circ G^{j}_{\sigma_1,\Delta}] \\
 & \text{(by 3 of induction hypothesis about } \sigma_1) \\
= & \{\![\sigma_1]\!\}\, \Delta[t \mapsto G^{j+1}_{\sigma_1,\Delta} \circ F^{j}_{\sigma_1,\Delta_{cod}}] \\
 & \text{(by induction hypothesis for } n = j) \\
= & (\{\![\sigma_1]\!\}\, \Delta[t \mapsto G^{j+1}_{\sigma_1,\Delta}]) \circ (\{\![\sigma_1]\!\}\, \mathsf{id}_{\Delta_{dom}}[t \mapsto F^{j}_{\sigma,\Delta_{dom}}]) \\
= & G^{j+2}_{\sigma_1,\Delta} \circ F^{j+1}_{\sigma_1,\Delta_{dom}}
\end{array}
$$

■

Moreover, it was already estalished that for any j, $G^{j}_{\sigma_1,\Delta}$ is a morphism in $\mathbf{C}^{EP}_{\mathbf{L}}$. In particular, $G^{n+1}_{\sigma_1,\Delta}$ is a morhism, and hence the following diagram commutes.

$$
\begin{tikzcd}
(U^{n+1}_{\sigma_1,\Delta_{dom}})_{n+1} \arrow[r, "(G^{n+1}_{\sigma_1,\Delta})^E_{n+1}"] & (U^{n+1}_{\sigma_1,\Delta_{cod}})_{n+1} \\
(U^{n+1}_{\sigma_1,\Delta_{dom}})_{n} \arrow[u, "(\delta_n^{U^{n+1}_{\sigma_1,\Delta_{dom}}})^E"] \arrow[r, "(G^{n+1}_{\sigma_1,\Delta})^E_{n}"] & (U^{n+1}_{\sigma_1,\Delta_{cod}})_{n} \arrow[u, "(\delta_n^{U^{n+1}_{\sigma_1,\Delta_{cod}}})^E"']
\end{tikzcd}
$$

Together with Proposition 3.6.7, this establishes that $\{\![\mu t.\, \sigma_1]\!\}\, \Delta$ is a morphism in $\mathbf{C}^{EP}_{\mathbf{L}}$, as required.

3. We first establish by induction on n that

$$G^{n}_{\sigma_1,\Delta_1 \circ \Delta_2} = G^{n}_{\sigma_1,\Delta_1} \circ G^{n}_{\sigma_1,\Delta_2}$$

– $n = 0$. Trivial.

– $n = j+1$.

$$\begin{array}{rcl} G^{j+1}_{\sigma_1,\Delta_1\circ\Delta_2} & = & \{\![\sigma_1]\!\}\,(\Delta_1\circ\Delta_2)[t\mapsto G^{j}_{\sigma_1,\Delta_1\circ\Delta_2}] \\ & = & \{\![\sigma_1]\!\}\,(\Delta_1\circ\Delta_2)[t\mapsto G^{j}_{\sigma_1,\Delta_1}\circ G^{j}_{\sigma_1,\Delta_2}] \\ & = & (\{\![\sigma_1]\!\}\,\Delta_1[t\mapsto G^{j}_{\sigma_1,\Delta_1}])\circ(\{\![\sigma_1]\!\}\,\Delta_2[t\mapsto G^{j}_{\sigma_1,\Delta_2}]) \\ & = & G^{j+1}_{\sigma_1,\Delta_1}\circ G^{j+1}_{\sigma_1,\Delta_2} \end{array}$$

Using the above, we now have

$$\begin{array}{rcl} (\{\![\mu t.\,\sigma_1]\!\}\,(\Delta_1\circ\Delta_2))_n & = & (G^{n}_{\sigma_1,(\Delta_1\circ\Delta_2)})_n \\ & = & (G^{n}_{\sigma_1,\Delta_1}\circ G^{n}_{\sigma_1,\Delta_2})_n \\ & = & (G^{n}_{\sigma_1,\Delta_1})_n\circ(G^{n}_{\sigma_1,\Delta_2})_n \\ & = & (\{\![\mu t.\,\sigma_1]\!\}\,\Delta_1)_n\circ(\{\![\mu t.\,\sigma_1]\!\}\,\Delta_2)_n \end{array}$$

■

We now establish two technical lemmas that assert that the meaning of a type depends only on its free type variables and that the meaning function is well-behaved with respect to substitution.

Lemma 3.6.8 *For any type σ,*

1. *For any type environments η, η', if $\eta(t) = \eta'(t)$ for all type variables t free in σ, then $\{\![\sigma]\!\}\,\eta = \{\![\sigma]\!\}\,\eta'$.*
2. *For any EP environments Δ, Δ', if $\Delta(t) = \Delta'(t)$ for all type variables t free in σ, then $\{\![\sigma]\!\}\,\Delta = \{\![\sigma]\!\}\,\Delta'$.*

Proof A trivial induction on σ. ■

Lemma 3.6.9

1. $\{\![[\sigma'/t]\sigma]\!\}\,\eta = \{\![\sigma]\!\}\,\eta[t\mapsto\{\![\sigma']\!\}\,\eta]$
2. $\{\![[\sigma'/t]\sigma]\!\}\,\Delta = \{\![\sigma]\!\}\,\Delta[t\mapsto\{\![\sigma']\!\}\,\Delta]$

Proof By induction on σ. We sketch the non-trivial case when σ is a recursive type. Suppose $\sigma\equiv\mu s.\,\sigma_1$.

- $s = t$. Then $[\sigma'/t]\sigma = \sigma$, and since t is not free in σ, by Lemma 3.6.8, $\{[\sigma]\}\,\eta[t \mapsto \{[\sigma']\}\,\eta] = \{[\sigma]\}\,\eta$. Hence, it is trivially established. Similarly for Δ.

- $s \neq t$. Since recursive types are identified up to renaming of bound variables, we can assume that s is not free in σ'. Then $[\sigma'/t]\sigma = \mu s.\,[\sigma'/t]\sigma_1$.

  1. We first establish by induction on i that

$$U^i_{[\sigma'/t]\sigma_1,\eta} = U^i_{\sigma_1,\eta[t\mapsto\{[\sigma']\}\,\eta]} \tag{3.1}$$
$$F^i_{[\sigma'/t]\sigma_1,\eta} = F^i_{\sigma_1,\eta[t\mapsto\{[\sigma']\}\,\eta]} \tag{3.2}$$

  From Equation 3.1 and Equation 3.2, it follows that

$$\{[[\sigma'/t]\sigma]\}\,\eta = \{[\sigma]\}\,\eta[t \mapsto \{[\sigma']\}\,\eta]$$

  2. We prove by induction on i that

$$G^i_{[\sigma'/t]\sigma_1,\Delta} = G^i_{\sigma_1,\Delta[t\mapsto\{[\sigma']\}\,\Delta]}$$

  from which we have that

$$\begin{aligned}(\{[[\sigma'/t]\sigma]\}\,\Delta)_n &= (G^n_{[\sigma'/t]\sigma_1,\Delta})_n \\ &= (G^n_{\sigma_1,\Delta[t\mapsto\{[\sigma']\}\,\Delta]})_n \\ &= (\{[\sigma]\}\,\Delta[t \mapsto \{[\sigma']\}\,\Delta])_n\end{aligned}$$

∎

The following technical lemma characterizes the dependence of the approximations of each FPC type σ and is important to the proof of Lemma 3.6.3.

Lemma 3.6.10 *For any FPC type σ, type-environments η, η', EP-environments Δ, Δ', and any $i < \omega$,*

1. *If for all variables t free in σ, $(\eta(t))_i = (\eta'(t))_i$, then $(\{[\sigma]\}\,\eta)_i = (\{[\sigma]\}\,\eta')_i$.*

2. *If for all variables t free in σ, $(\Delta(t))_i = (\Delta'(t))_i$, then $(\{\![\sigma]\!\}\,\Delta)_i = (\{\![\sigma]\!\}\,\Delta')_i$.*

3. *If for all variables t free in σ, $\delta_i^{\eta(t)} = \delta_i^{\eta'(t)} \circ (\Delta(t))_i$,* i.e.

$$\begin{tikzcd}
 & (\eta(t))_{i+1} = (\eta'(t))_{i+1} \\
(\eta(t))_i \arrow[ur, "\delta_i^{\eta(t)}"] \arrow[r, "(\Delta(t))_i"'] & (\eta'(t))_i \arrow[u, "\delta_i^{\eta'(t)}"']
\end{tikzcd}$$

then

$$\begin{tikzcd}
 & (\{\![\sigma]\!\}\,\eta)_{i+1} = (\{\![\sigma]\!\}\,\eta')_{i+1} \\
(\{\![\sigma]\!\}\,\eta)_i \arrow[ur, "\delta_i^{\{\![\sigma]\!\}\,\eta}"] \arrow[r, "(\{\![\sigma]\!\}\,\Delta)_i"'] & (\{\![\sigma]\!\}\,\eta')_i \arrow[u, "\delta_i^{\{\![\sigma]\!\}\,\eta'}"']
\end{tikzcd}$$

i.e. $\delta_i^{\{\![\sigma]\!\}\,\eta} = \delta_i^{\{\![\sigma]\!\}\,\eta'} \circ (\{\![\sigma]\!\}\,\Delta)_i$.

Proof

1. By induction on σ:

   - $\sigma \equiv t, \mathbf{unit}, \sigma_1 \odot \sigma_2$. Trivial.
   - $\sigma \equiv \mu t.\, \sigma_1$. We can prove by induction on j that

$$(U^j_{\sigma_1,\eta})_i = (U^j_{\sigma_1,\eta'})_i \tag{3.3}$$

   from which $(\{\![\sigma]\!\}\,\eta)_i = (U^i_{\sigma_1,\eta})_i = (U^i_{\sigma_1,\eta'})_i = (\{\![\sigma]\!\}\,\eta')_i$.

2. By induction on σ:

   - $\sigma \equiv t, \mathbf{unit}, \sigma_1 \odot \sigma_2$. Trivial.
   - $\sigma \equiv \mu t.\, \sigma_1$. We can prove by induction on j that

$$(G^j_{\sigma_1,\Delta})_i = (G^j_{\sigma_1,\Delta'})_i \tag{3.4}$$

Hence $(\{\![\sigma]\!\}\,\Delta)_i = (G^i_{\sigma_1,\Delta})_i = (G^i_{\sigma_1,\Delta'})_i = (\{\![\sigma]\!\}\,\Delta')_i$.

3. By induction on σ:

- $\sigma \equiv t, \mathbf{unit}, \sigma_1 \odot \sigma_2$. Trivial.
- $\sigma \equiv \mu t.\, \sigma_1$. We prove by induction on j that

$$
\begin{aligned}
\delta_i^{U^j_{\sigma_1,\eta}} &= \delta_i^{U^j_{\sigma_1,\eta'}} \circ (G^j_{\sigma_1,\Delta})_i &\quad (3.5)\\
(F^j_{\sigma_1,\eta})_{i+1} &= (F^j_{\sigma_1,\eta'})_{i+1} &\quad (3.6)
\end{aligned}
$$

Base Case:

$$
\begin{aligned}
\delta_i^{U^0_{\sigma_1,\eta}} &= \mathsf{id}_{\mathbf{void}}\\
&= \delta_i^{U^0_{\sigma_1,\eta'}} \circ (G^0_{\sigma_1,\Delta})_n\\
(F^0_{\sigma_1,\eta})^E_{i+1} &= \lambda x\colon \mathbf{void}.\, \Omega_{(U^1_{\sigma_1,\eta})_{i+1}}\\
&= \lambda x\colon \mathbf{void}.\, \Omega_{(U^1_{\sigma_1,\eta'})_{i+1}}\\
&\quad ((\eta(s))_{i+1} = (\eta'(s))_{i+1},\ \text{1 of this Lemma })\\
&= (F^0_{\sigma_1,\eta'})^E_{i+1}
\end{aligned}
$$

Induction Step:

$$
\begin{aligned}
\delta_i^{U^{j+1}_{\sigma_1,\eta}} &= \delta_i^{\{\![\sigma_1]\!\}\,\eta[t\mapsto U^j_{\sigma_1,\eta}]}\\
&= \delta_i^{\{\![\sigma_1]\!\}\,\eta'[t\mapsto U^j_{\sigma_1,\eta'}]} \circ (\{\![\sigma_1]\!\}\,\Delta[t\mapsto G^j_{\sigma_1,\Delta}])_i\\
&\quad (\text{Ind. Hyp. for } j \text{ and } \sigma_1)\\
&= \delta_i^{U^{j+1}_{\sigma_1,\eta'}} \circ (G^{j+1}_{\sigma_1,\Delta})_i\\
(F^{j+1}_{\sigma_1,\eta})_{i+1} &= (\{\![\sigma_1]\!\}\,\mathsf{id}_\eta[t\mapsto F^j_{\sigma_1,\eta}])_{i+1}\\
&= (\{\![\sigma_1]\!\}\,\mathsf{id}_{\eta'}[t\mapsto F^j_{\sigma_1,\eta'}])_{i+1}\\
&\quad (\text{Ind. Hyp., 2 of Lemma})\\
&= (F^{j+1}_{\sigma_1,\eta'})_{i+1}
\end{aligned}
$$

Finally,

$$
\begin{array}{rcll}
\delta_i^{\{\![\sigma]\!\}\,\eta} & = & \delta_i^{U^{i+1}_{\sigma_1,\eta}} \circ (F^i_{\sigma_1,\eta})_i & \\
& = & (F^i_{\sigma_1,\eta})_{i+1} \circ \delta_i^{U^i_{\sigma_1,\eta}} & (F^i_{\sigma_1,\eta} \text{ is a morphism in } \mathbf{C}^{EP}_{\mathbf{L}}) \\
& = & (F^i_{\sigma_1,\eta'})_{i+1} \circ \delta_i^{U^i_{\sigma_1,\eta'}} \circ (G^i_{\sigma_1,\Delta})_i & (\text{Equation 3.5, 3.6}) \\
& = & \delta_i^{U^{i+1}_{\sigma_1,\eta'}} \circ (F^i_{\sigma_1,\eta'})_i \circ (G^i_{\sigma_1,\Delta})_i & (F^i_{\sigma_1,\eta'} \text{ is a morphism in } \mathbf{C}^{EP}_{\mathbf{L}}) \\
& = & \delta_i^{\{\![\sigma]\!\}\,\eta'} \circ (\{\![\sigma]\!\}\,\Delta)_i &
\end{array}
$$

∎

Finally, using the above lemmas, we can establish Lemma 3.6.3.

Proof of Lemma 3.6.3

$$
\begin{array}{rcll}
(\{\![[\mu t.\,\sigma/t]\sigma]\!\}\,\eta)_n & = & (\{\![\sigma]\!\}\,\eta[t \mapsto \{\![\mu t.\,\sigma]\!\}\,\eta])_n & (1 \text{ of Lemma 3.6.9}) \\
& = & (\{\![\sigma]\!\}\,\eta[t \mapsto S^n_{\sigma,\eta}])_n & (1 \text{ of Lemma 3.6.10}) \\
& = & (S^{n+1}_{\sigma,\eta})_n &
\end{array}
$$

$$
\begin{array}{rcll}
\delta_n^{\{\![[\mu t.\,\sigma/t]\sigma]\!\}\,\eta} & = & \delta_n^{\{\![\sigma]\!\}\,\eta[t\mapsto\{\![\mu t.\,\sigma]\!\}\,\eta]} & (1 \text{ of Lemma 3.6.9}) \\
& = & \delta_n^{\{\![\sigma]\!\}\,\eta[t\mapsto S^{n+1}_{\sigma,\eta}]} \circ (\{\![\sigma]\!\}\,\mathsf{id}_\eta[t \mapsto F^n_{\sigma,\eta}])_n & (3 \text{ of Lemma 3.6.10}) \\
& = & \delta_n^{S^{n+2}_{\sigma,\eta}} \circ (F^{n+1}_{\sigma,\eta})_n &
\end{array}
$$

∎

3.7 Conclusion

We have shown how the ideas in [Mil77] can be generalized to obtain fully abstract models of call-by-value languages and of recursive types. The former is done by having explicit divergence constants at each type; the latter relies on the definition of an appropriate category from the partial language. We use both ideas in constructing the model for FPC. Using the second idea, we can easily obtain a fully abstract model for call-by-name FPC as well. A technical novelty is that in the proofs of the required properties of our model of FPC, we are able to avoid having to use any recursively defined inclusive relations—a technique that is typically necessary in the presence of recursive types.

It would be interesting to see if every sequential functional language can be given a fully abstract, extensional, continuous model using these methods, *e.g.*, one might try to construct a model for a language with the limited form of parametric polymorphism found in ML [Mil78, MTH90]. Indeed, such models may exist within the category $\mathbf{C}^{EP}$. Milner also proves that his model of sequential, call-by-name PCF is the unique extensional, monotone, and continuous model [Mil77]; an important problem is to see if the models of sequential VPCF and FPC are the unique extensional, monotone, continuous models.

Chapter 4

Isolating Side Effects in Sequential Languages

4.1 Introduction

Two pieces of code are "operationally equivalent" if in any program, one can be replaced by the other without affecting the observable outcome of the program. A good understanding of operational equivalence can facilitate modular development of programs and make it possible to prove the soundness of various optimizations. It is therefore an advantage of a *purely* functional language, such as Haskell [HPW$^+$92], that it supports some well-understood principles for reasoning about operational equivalence. Purely functional languages, however, lack features that make it possible to write more efficient programs. For this reason, many functional languages, *e.g.*, Standard ML [MTH90] and Scheme [CR91], include control operations and ways of manipulating memory. Such side-effecting features do not come without cost: simple reasoning principles may fail to be sound, as two purely functional pieces of code may be equivalent in purely functional contexts but not in contexts with side effects.

To illustrate the problem, consider a call-by-value version of PCF—here called VPCF (*c.f.* [Gun92, Rie93a, RV93])—which is formally defined in Section 4.3.1. VPCF is a purely functional language, but we can add operations $\mathsf{ref}, !, :=$ for manipulating the state to obtain another language called VPCF+S. Briefly, $\mathsf{ref}\ x$ allocates a new cell initialized to x, $(r := x)$ updates the contents of the cell r to x, and $!r$ returns the current contents of the cell r. (The typing rules and formal operational semantics for VPCF+S appear in Section 4.3.3.) Two terms may be operationally equivalent in VPCF but not in the extended language. For example, consider the expressions

$$\begin{aligned} M_1 &= \lambda x^{\mathbf{num}}.\, \lambda f^{\mathbf{num}\to\mathbf{num}}.\, \lambda g^{\mathbf{num}\to\mathbf{num}}.\, (f\ x); (g\ x) \\ M_2 &= \lambda x^{\mathbf{num}}.\, \lambda f^{\mathbf{num}\to\mathbf{num}}.\, \lambda g^{\mathbf{num}\to\mathbf{num}}.\, (f\ x); (f\ x); (g\ x) \end{aligned}$$

where ; is the usual sequencing operator (definable in VPCF as $(M; N) = ((\lambda d.\ N)\ M)$ for $d \notin FV(N)$). M_1 and M_2 are operationally indistinguishable in VPCF. However, in VPCF+S, they can be distinguished by the context

$$C[\cdot] = (\lambda r.\ [\cdot]\ 0\ (\lambda d.\ r := \mathsf{succ}\,(!r))\ (\lambda d.\ !r))\ (\mathsf{ref}\ 0).$$

$C[M_1]$ returns 1 whereas $C[M_2]$ returns 2.

This example, which shows how adding assignments fundamentally changes the character of VPCF, runs counter to a vague intuition that the VPCF is somehow "embedded" in VPCF+S, and that reasoning principles on the purely functional core extend easily to the language with state. In this chapter we show how to recover that intuition. More precisely, we show that VPCF is embedded in an extension with side effects, *but mediated by the application of an additional operation* encap. The encap operation "encapsulates" side effects: the application of encap to some part of a program prevents its side effects from being visible to the rest of the program, and conversely, encap prevents any potential side effects of the surrounding program from affecting the behaviour of the code under encap. However, it does not change the "functional" behaviour of a term; formally, if M is a purely functional piece of code, then for any purely functional context $C[\cdot]$, the programs $C[M]$ and $C[\mathsf{encap}\,M]$ yield the same observable behaviour. As a consequence, the encap operation restores old operational equivalences, *i.e.*, two terms M and N are operationally equivalent in VPCF iff $(\mathsf{encap}\ M)$ and $(\mathsf{encap}\ N)$ are operationally equivalent in VPCF+S+encap.

Finally, the operation encap forces non-functional code to behave purely functionally; we give a theorem illustrating this in Section 4.5.

We can now give some intuition for the behaviour of encap when applied to a term M of VPCF+S+encap. In executing (encap M), the current state of the memory is saved, say on disk, and M is executed with the contents of all memory locations flushed. If M is of base type, any resulting value is returned. If M is of function type, the resulting closure is returned, with its argument and body coerced by encap. In either case, memory is then restored to the saved state. Thus the execution of (encap M) does not change the state of memory and its result does not depend on the state in which it is executed.

The uses of encap are twofold. First, one may use encap to enforce certain portions of code to be oblivious to side effects caused by other modules; this would allow a programmer to aggressively optimize the code under an encap later since functional reasoning principles remain valid for purely functional code under an encap. Second, one may use encap to establish facts about code. If one proves that adding encap to a subterm M does not change the meaning of the whole term, then any side effects of M are purely local and so the proof of correctness of the whole term can be carried out using functional reasoning principles. Section 4.6 below outlines an example of proving program correctness using encap.

The idea of using coercion operations to encapsulate side effects is not new; in [Rie93a], Riecke adds a similar operation for functional languages extended by imperative features. In broad outline, the design and proof of correctness of encap in [Rie93a] relies on a connection between a model of the underlying functional language and a model of its extension with a side-effecting operation. This connection between the two models is easiest to establish in a general semantic framework of "strict monads" over a category, which are monads with additional structure (see [Mog91, Wad92] for a discussion of monads). The proofs in [Rie93a] then require that the model of the extension must be definable from a strict monad. Moreover, the proofs rely crucially on two properties: the model of the functional language must be fully abstract, *i.e.*, operational equivalence must match denotational equivalence, and the model of the extension must be adequate (*c.f.* [Gun92]). In order to use the most natural semantic models, the full abstraction requirement forces the languages to include a "parallel if" (*c.f.* [Gun92, Plo77]). This operation, found in no practical programming language, is the main obstacle to applying the results.

The main contribution of this work is the design and proof of correctness of `encap` for *purely sequential* programming languages. Our operational semantics for `encap` is also simpler than that of [Rie93a], making reasoning with `encap` and implementing it much easier. As applications, we describe the operational rules for `encap` when added to a language with state and to one with continuations. The operational rules for isolating other side effects, such as exceptions and non-determinism, can also be immediately obtained from our general framework.

Adapting the proofs of [Rie93a] for the sequential setting is not straightforward. There are three key insights. First, since general categories for sequential languages are not well understood, it is essential to work over a *call-by-value model* instead of a category. Section 4.2 develops an appropriate definition of a "call-by-value language" and "call-by-value model". Second, the naive extension of "strict monads" to a call-by-value model does not work because a call-by-value model itself arises from a lifting monad over a category C, and the natural monads for side effects defined over C fail to be monads when extended to the lifted category. We develop the definition of a *call-by-value monad* in Section 4.4; these call-by-value monads are weak enough to express all the side effects of interest; yet, have enough structure to entail the desired semantic connections. Third, a close examination of the proofs in [Rie93a] reveals that the fully abstract model for the base language VPCF need only be constructed out of continuous, extensional functions; other mathematically attractive properties of the model are simply not needed. In the previous chapter, we showed how to construct—using techniques of Milner [Mil77]—fully abstract models of VPCF and FPC (a sequential language with recursive types [Gun92], in which the semantics of VPCF+S can be encoded), called $\mathcal{V}$ and $\mathcal{S}$. While our proofs only require the model $\mathcal{V}$ to be fully abstract, the intimate connection between $\mathcal{V}$ and $\mathcal{S}$ requires that the model $\mathcal{S}$ also not contain the "parallel" functions which are missing from the fully abstract model $\mathcal{V}$ (*c.f.* [Plo77]). The existence of continuous, extensional, fully abstract models of VPCF and FPC is thus crucial. However, because our framework

Table 4.1: Typing and Evaluation Rules for Call-by-Value Functional Languages.

Terms	(var)	$x^\sigma : \sigma$
	$(const)$	$\dfrac{\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma \rangle \in C \quad M_i : \sigma_i}{f(M_1, \ldots, M_n) : \sigma}$
	(abs)	$\dfrac{M : \tau}{(\lambda x^\sigma .\, M) : (\sigma \to \tau)}$
	(app)	$\dfrac{M : (\sigma \to \tau) \quad N : \sigma}{(M\ N) : \tau}$
Evaluation		$(\lambda x.\, M)\ V \quad \leadsto \quad [V/x]M$ $E \ ::= \ [\cdot] \mid (E\ M) \mid (V\ E) \mid \ \ldots$

is applicable to any call-by-value model, the details of the construction in Chapter 3 play no role in the proofs of this chapter.

4.2 Call-by-value Languages

4.2.1 Syntax and Operational Semantics

The type signature of a general functional language is a set Σ that is closed under $\to$, *viz.*, if $\sigma, \tau \in \Sigma$, then $(\sigma \to \tau) \in \Sigma$. A set of typing rules for functional languages appears in Table 4.1; all variables come tagged with types. The typing rules are parameterized by a set C of **term constructors**, which are tuples of the form $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma \rangle$ where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma$. We chose term constructors, as opposed to term constants, to allow the possibility of non-strict operators in the call-by-value language.

We use a parameterized rewriting semantics, with two basic relations, to specify operational behaviour. The first relation, $\leadsto$, reduces redexes with $M \leadsto M'$ indicating that M is a redex reducing directly to M'. The second relation, $\to$, is the basic one-step evaluation relation for the particular language, which is defined using the notion of an **evaluation context** to impose an order on the evaluation of subexpressions [Fel88a]. Table 4.1 specifies a general rewriting framework for call-by-value languages. The only redex rule common to all call-by-value languages is the β-reduction rule; here, V ranges over a set of **values**—expressions that cannot be further evaluated—which varies among languages but always includes λ-abstractions. The evaluation contexts specify the next redex using a left-to-right order of evaluation for applications. To give the semantics of a particular language, one must add rules for $\rightsquigarrow$ to specify the behaviour of term constructors, and add new productions to the grammar of evaluation contexts to reflect the order of evaluation.

4.2.2 Call-by-value Models

A denotational model for a call-by-value language is based on a collection of sets, indexed by type, suitable for giving meaning to terms. The following definition is taken from [Rie91] and resembles definitions from [Abr90, Ong88b].

Definition 4.2.1 A **call-by-value type frame** over a type signature Σ is a tuple

$$\mathcal{D} = (\{D^\sigma \mid \sigma \in \Sigma\}, \{\uparrow^\sigma \mid \sigma \in \Sigma\}, \{App^{\tau,\nu} \mid \tau, \nu \in \Sigma\})$$

where each D^σ is a poset, $\uparrow^\sigma \in D^\sigma$, and $App^{\tau,\nu}\colon D^{\tau\to\nu} \times D^\tau \to D^\nu$ such that

1. For all $d \in D^\sigma$, $\uparrow^\sigma \leq d$.

2. If either $d = \uparrow^{\tau\to\nu}$ or $e = \uparrow^\tau$, then $App^{\tau,\nu}(d, e) = \uparrow^\nu$.

3. $f \leq_{D^{\tau\to\nu}} g$ iff ($f \neq \uparrow^{\tau\to\nu}$ implies $g \neq \uparrow^{\tau\to\nu}$) and for all $d \leq_{D^\tau} e$, $App^{\tau,\nu}(f, d) \leq_{D^\nu} App^{\tau,\nu}(g, e)$.

We use $\Sigma(\mathcal{D})$ to denote the type signature over which the type frame $\mathcal{D}$ is defined. As notational conventions, we omit the type superscripts on $\uparrow$ and App when obvious from the context, and write $(f\ \ x)$ for $App(f, x)$ when clear . Also, we write $d\uparrow$ whenever $d = \uparrow$ and $d\downarrow$ whenever $d \neq \uparrow$.

- The type frame $\mathcal{D}$ is **flat** at type $\sigma \in \Sigma(\mathcal{D})$ if for any $d, e \in D^\sigma$, $d\!\downarrow$ and $d \leq e$ implies $d = e$.
- The type frame $\mathcal{D}$ is **continuous** if each D^σ is a cpo, and for any $f \in D^{\tau\to\nu}$ and directed set $S \subseteq D^\tau$, $App(f, \bigsqcup S) = \bigsqcup\{App(f,e) \mid e \in S\}$.

The element $\uparrow$ is the semantic representation of a divergent computation. The conditions on a call-by-value frame capture the call-by-value parameter passing mechanism and the fact that non-divergent elements of a function type behave monotonically and extensionally.

Let $\mathcal{D}$ be a type frame over Σ. A **$\mathcal{D}$-environment** is a type-respecting map from variables to elements of $\mathcal{D}$, *i.e.*, $\rho(x^\sigma) \in D^\sigma$. A **call-by-value model** for a language with types Σ and term constructors C is then a type frame $\mathcal{D}$ together with functions $\mathcal{D}[\![f]\!]\colon D^{\sigma_1} \times \cdots \times D^{\sigma_n} \to D^\sigma$ for each $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma\rangle \in C$, with enough elements to interpret every term of the language, *i.e.*, there is a meaning function $\mathcal{D}[\![\cdot]\!]$ satisfying

$$\begin{array}{rcl}
\mathcal{D}[\![x^\sigma]\!]\rho & = & \rho(x^\sigma) \\
\mathcal{D}[\![f M_1 \ldots M_k]\!]\rho & = & \mathcal{D}[\![f]\!](\mathcal{D}[\![M_1]\!]\rho, \ldots, \mathcal{D}[\![M_k]\!]\rho) \\
\mathcal{D}[\![M\ N]\!]\rho & = & App(\mathcal{D}[\![M]\!]\rho, \mathcal{D}[\![N]\!]\rho) \\
\mathcal{D}[\![\lambda x^\sigma.\, M]\!]\rho & = & g,\ \text{where } g\!\downarrow \text{ and for any } d\!\downarrow, \\
 & & \quad App(g,d) = \mathcal{D}[\![M]\!]\rho[x^\sigma \mapsto d]
\end{array}$$

and such that $\mathcal{D}[\![M]\!]\rho \in D^\sigma$ for any term M of type σ and any $\mathcal{D}$-environment ρ.

4.3 VPCF and its Extensions

In this section, we define our basic functional language, VPCF, and two extensions. The operational semantics of each language specifies an interpreter function $Eval_L$ which is a partial function from programs, *i.e.*, closed terms of base type **num** to numerals. The evaluation function determines the operational equivalence relation of the language.

Definition 4.3.1 Two terms M and N in L are **operationally equivalent**, written $M \equiv_L N$, if in any context $C[\cdot]$ such that $C[M]$ and $C[N]$ are programs,

Table 4.2: Operational Semantics of VPCF, VPCF+K, and VPCF+S

VPCF	$\mathsf{succ}(n) \leadsto n+1$ $\mathsf{if0}(0, M, N) \leadsto M$ $\mathsf{pred}(0) \leadsto 0$ $\mathsf{if0}(n+1, M, N) \leadsto N$ $\mathsf{pred}(n+1) \leadsto n$ $(\mathsf{Y}\ V) \leadsto (V\ (\lambda x.\, \mathsf{Y}\ V\ x))$ $E \ ::= \ \ldots \mid \mathsf{succ}(E) \mid \mathsf{pred}(E) \mid \mathsf{if0}(E, M, N) \mid (\mathsf{Y}\ E) \mid \ldots$
VPCF+K	$E[M] \to_c E[M']$ if $M \leadsto M'$ $E[\mathsf{callcc}(M)] \to_c E[M\ (\lambda x.\, \mathsf{abort}(E[x]))]$ $E[\mathsf{abort}(n)] \to_c n$ $E \ ::= \ \ldots \mid \mathsf{abort}(E)$
VPCF+S	$(E[M], s) \to_s (E[M'], s)$ if $M \leadsto M'$ $(E[\mathsf{ref}\ V], s) \to_s (E[l], s[l \mapsto V])$ if $l \notin dom(s)$ $(E[!l], s) \to_s (E[s(l)], s)$ $(E[l := V], s) \to_s (E[V], s[l \mapsto V])$ $E \ ::= \ \ldots \mid \mathsf{ref}\ (E) \mid !(E) \mid (E := M) \mid (V := E)$

$Eval_L(C[M]) = n$ iff $Eval_L(C[N]) = n$.

4.3.1 VPCF

VPCF is a purely functional language with basic arithmetic and a mechanism for recursive function definitions. Its types are given by the grammar

$$\sigma \ ::= \ \mathbf{num} \mid (\sigma \to \sigma)$$

and the term constructors are

$$\begin{array}{l}
\langle n; \mathbf{num}\rangle \quad \text{for all } n \geq 0 \\
\langle \mathsf{succ}; \mathbf{num} \Rightarrow \mathbf{num}\rangle \\
\langle \mathsf{pred}; \mathbf{num} \Rightarrow \mathbf{num}\rangle \\
\langle \mathsf{if0}^{\sigma}; \mathbf{num}, \sigma, \sigma \Rightarrow \sigma\rangle \\
\langle \mathsf{Y}_{\sigma}; (\sigma \to \sigma) \to \sigma\rangle \quad \text{where } \sigma \neq \mathbf{num}.
\end{array}$$

The syntax of VPCF is now completely defined by Table 4.1.

In addition to the redex rule in Table 4.1, VPCF has the redexes given in Table 4.2 which reflect the behaviour of its term constructors. The additional productions for evaluation contexts (beyond those of Table 4.1) are also given in Table 4.2, where a value V is a numeral n, a λ-abstraction, or Y_{σ}. One can verify that every closed non-value can be written uniquely as $E[R]$ with R a redex. The basic one-step evaluation relation $\to_v$ for closed VPCF terms is defined by

$$E[M] \ \to_v \ E[M'], \ \text{ if } M \leadsto M'.$$

Define $Eval_v(M) = n$ iff $M \twoheadrightarrow_v n$, where $\twoheadrightarrow_v$ is the reflexive, transitive closure of $\to_v$.

4.3.2 VPCF+K

Control operations can be added to VPCF by adding the term constructors

$$\langle \mathsf{abort}; \mathbf{num} \Rightarrow \sigma\rangle \quad \langle \mathsf{callcc}; ((\sigma \to \mathbf{num}) \to \sigma) \Rightarrow \sigma\rangle$$

for terminating computations and capturing continuations, respectively. The resulting language is called VPCF+K (for "control").

Table 4.2 specifies the one-step evaluation relation $\to_c$ and set of evaluation contexts for VPCF+K. The last two rules for $\to_c$ give the formal semantics of the new operations: callcc captures the current evaluation context, makes it into a λ-abstraction, and passes it to the argument of callcc; abort discards the evaluation context. The interpreter function $Eval_k$ is defined as for VPCF.

4.3.3 VPCF+S

One may add Standard ML-style references and assignments to VPCF (*c.f.* [MTH90]). The enhanced language, called VPCF+S (for "state"), has types generated by the grammar

$$\sigma \ ::= \ \mathbf{num} \mid (\sigma \to \sigma) \mid (\mathsf{ref}\ \sigma)$$

where $(\mathsf{ref}\ \sigma)$ is the type of locations holding values of type σ. In addition to the VPCF term constructors, VPCF+S has term constructors $\langle l^\sigma; (\mathsf{ref}\ \sigma)\rangle$, where l is drawn from an infinite set of location constants; $\langle \mathsf{ref}\ ; \sigma \Rightarrow (\mathsf{ref}\ \sigma)\rangle$ for creating references; $\langle !; (\mathsf{ref}\ \sigma) \Rightarrow \sigma\rangle$ for dereferencing; and $\langle :=; (\mathsf{ref}\ \sigma), \sigma \Rightarrow \sigma\rangle$ for assignment, written in infix notation. The location constants l^σ do not appear in user programs (this would be tantamount to allowing the user to pick addresses), but help in defining the operational semantics.

The evaluation relation $\to_s$ and the additional evaluation contexts of VPCF+S appear in Table 4.2. Values in VPCF+S include locations l in addition to the values of VPCF. The operational rules manipulate a state s in addition to terms; a **state** is simply a finite map from a set of locations to values. The operational rules for the assignments and dereferencing are standard (*c.f.* [Gun92]), and the evaluation function $Eval_s$ is defined as $Eval_s(M) = n$ iff $(M, \emptyset) \twoheadrightarrow_s (n, s')$ for some state s', where $\emptyset$ is the empty map denoting the initial store.

4.4 Call-by-value Monads and Coercions

We design and prove the correctness of a coercion operator for an extension of VPCF in two steps. First, we show that a continuous model $\mathcal{D}$ for VPCF can be regarded as a retract of a model $\mathcal{T}$ for an extension of VPCF. The retractions act as semantic coercion operators, *viz.*, they coerce any element of $\mathcal{T}$ into one with the same functional behaviour but which has no side effects. Second, we use the semantic coercions to design the operational rules for the coercion operators encap. The operational facts are proven through model-theoretic facts: if the model $\mathcal{D}$ is fully abstract for VPCF, and the model $\mathcal{T}$ is adequate for the extension of VPCF together with the encap operator, then the properties of the retractions imply the desired operational correctness properties of encap.

In this section we define a general framework in which the semantics of extensions of VPCF can be described using a "call-by-value monad" and the model that it defines, and prove that any call-by-value model $\mathcal{D}$ of VPCF is a retraction of the model determined by a call-by-value monad. We defer the application of the results to Section 4.5.

4.4.1 Call-by-value Monads

Ideally, monads are built over categories with some abstract structure, *e.g.*, the notion of a "strict monad" over **DCPO** defined in [Rie93a]. Call-by-value monads are, however, defined over *call-by-value models*, not over categories. There is a good reason for this: constructing categories suitable for *sequential* languages like VPCF (see, *e.g.*, [BCL85, CF92, Cur92]) has been open for many years (although recent announcements [AJM93, HO93, OR95] suggest that a categorical framework can be constructed). Moreover, by defining call-by-value monads over models, we can state and prove our results independent of the details of the construction of any particular model.

Our definition of a call-by-value monad is inspired by the notion of a "strict monad" defined in [Rie93a] ("strict monads" are monads [Mog91] with some additional structure).

Definition 4.4.1 Let $\mathcal{D}$ be a **least fixpoint model**, *i.e.*, a continuous call-by-value model with $\mathcal{D}[\![\mathsf{Y}_\sigma]\!] = \bigsqcup_n F^n_\sigma$, where

$$\begin{aligned} F^0_\sigma &= \uparrow^{((\sigma\to\sigma)\to\sigma)} \\ F^{n+1}_\sigma &= \mathcal{D}[\![\lambda f^{(\sigma\to\sigma)}.\, f(\lambda x.\, G\ f\ x)]\!][G \mapsto F^n_\sigma], \end{aligned}$$

and where $\mathcal{D}$ is flat at a type **num** $\in \Sigma(\mathcal{D})$. A **call-by-value monad** over $\mathcal{D}$ is a tuple $\langle T, \eta, \mu, \beta\rangle$ with $T\colon \Sigma(\mathcal{D}) \to \Sigma(\mathcal{D})$ and a family of functions $T_{\tau,\nu}\colon D^{\tau\to\nu} \to D^{T(\tau)\to T(\nu)}$, $\eta_\sigma\colon D^\sigma \to D^{T(\sigma)}$, and $\mu_\sigma\colon D^{T(T(\sigma))} \to D^{T(\sigma)}$ for each type $\sigma, \tau, \nu \in \Sigma(\mathcal{D})$ satisfying the following properties:

1. The functions $T_{\tau,\nu}, \eta_\sigma, \mu_\sigma$ are **interpretable** in $\mathcal{D}$. That is, when the meaning function of $\mathcal{D}$ is extended to interpret new constructors

$$\begin{aligned}&\langle T_{\tau,\nu}; (\tau \to \nu) \Rightarrow (T(\tau) \to T(\nu))\rangle,\\ &\langle \eta_\sigma; \sigma \Rightarrow T(\sigma)\rangle, \text{ and}\\ &\langle \mu_\sigma; T(T(\sigma)) \Rightarrow T(\sigma)\rangle\end{aligned}$$

   using the functions, the meaning of any term over the new constructors still exists in the model.

2. For any $x \in D^\sigma$ and $f \in D^{\sigma\to\tau}$, if $f\downarrow$, then $\eta_\tau(f\ x) = T_{\sigma,\tau}(f)\ (\eta_\sigma\ x)$.

3. For any $x \in D^{T(\sigma)}$,

   - If $x\downarrow$, then $\mu_\sigma(\eta_{T(\sigma)}\ x) = x$.
   - If $x\uparrow$, then $\mu_\sigma(\eta_{T(\sigma)}\ x) = \eta_\sigma(\uparrow^\sigma)$.

4. For any $x, y \in D^\sigma$, if $\eta_\sigma(x) \leq \eta_\sigma(y)$, then $x \leq y$.

5. For any $x \in D^{\mathbf{num}}$, $\beta(\eta_{\mathbf{num}}(x)) = x$.

If one takes the elements of a call-by-value model as defining the morphisms of a category, then a call-by-value monad differs from a monad over this category in two respects. Firstly, we do not require the maps η_σ and μ_σ to be elements of the model. In particular, they do not have to be strict functions. Secondly, the equations that they need to satisfy are weaker forms of the naturality and unit laws of a monad. Indeed, all the side effects of interest would fail to satisfy the stronger monad laws over a call-by-value model. For example, in the monads given in Section 4.5, η is not strict and they only satisfy the modified monad laws expressed in this definition. The weaker properties of call-by-value monads will, however, be strong enough to prove the semantic connections established in Section 4.4.3.

Monads capture the notion of "computations" that consist of results and side effects [Mog91]. However, unlike categorical monads, the notion of divergence is already

Table 4.3: Semantics of VPCF Over a Call-by-value Monad.

$$
\begin{array}{rcl}
\mathcal{T}[\![x^\sigma]\!]\rho_{\mathcal{T}} &=& \eta\ (\rho_{\mathcal{T}}(x^\sigma)) \\
\mathcal{T}[\![(\lambda x^\sigma.\, M)]\!]\rho_{\mathcal{T}} &=& \eta\ (\underline{\lambda} d \in D^{\sigma'}.\, \mathcal{T}[\![M]\!]\rho_{\mathcal{T}}[x^\sigma \mapsto d]) \\
\mathcal{T}[\![(M\ N)]\!]\rho_{\mathcal{T}} &=& \mathsf{Ap}\ \mathcal{T}[\![M]\!]\rho_{\mathcal{T}}\ \mathcal{T}[\![N]\!]\rho_{\mathcal{T}} \\
\mathcal{T}[\![n]\!]\rho_{\mathcal{T}} &=& \eta\ (\mathcal{D}[\![n]\!]) \\
\mathcal{T}[\![\mathsf{succ}\,(M)]\!]\rho_{\mathcal{T}} &=& T\ (\underline{\lambda} x \in D^{\mathbf{num}}.\, \mathsf{succ}\,(x))\ \mathcal{T}[\![M]\!]\rho_{\mathcal{T}} \\
\mathcal{T}[\![\mathsf{pred}\,(M)]\!]\rho_{\mathcal{T}} &=& T\ (\underline{\lambda} x \in D^{\mathbf{num}}.\, \mathsf{pred}\,(x))\ \mathcal{T}[\![M]\!]\rho_{\mathcal{T}} \\
\mathcal{T}[\![\mathsf{if0}(M,N,P)]\!]\rho_{\mathcal{T}} &=& \mu[T\ (\underline{\lambda} b \in D^{\mathbf{num}}.\, \mathsf{if0}(b, \mathcal{T}[\![N]\!]\rho_{\mathcal{T}}, \mathcal{T}[\![P]\!]\rho_{\mathcal{T}}))\ \mathcal{T}[\![M]\!]\rho_{\mathcal{T}}] \\
\mathcal{T}[\![\mathsf{Y}^\sigma]\!]\rho_{\mathcal{T}} &=& \eta\ (\bigsqcup_n F^n_\sigma),\ \text{where} \\
&& \quad F^0_\sigma \ = \ \uparrow^{((\sigma\to\sigma)\to\sigma)'} \\
&& \quad F^{n+1}_\sigma \ = \ \underline{\lambda} f \in D^{(\sigma\to\sigma)'}.\, f\ (\underline{\lambda} x \in D^{\tau'}.\, \mathsf{Ap}\ (F^n_\sigma\ f)\ (\eta_{\tau'}\ x))
\end{array}
$$

represented in results; this is the reason for the departure from the definition of a categorical monad. Thus computations incorporate only the possibility of side effects. A call-by-value monad provides basic operations for manipulating computations: the type $T(\sigma)$ represents the set of computations whose results are of type σ; the function $T_{\tau,\nu}(f)$ transforms the result of a computation by f without changing its side effect; η maps a result d to a computation with no side effect; μ composes two computations to give the cumulative computation; and β extracts the result from a computation of type **num**. The conditions in the definition of a call-by-value monad simply express some natural and general properties of side effects.

4.4.2 Monadic Semantics

A call-by-value monad over a model of VPCF generates a new interpretation of application and the VPCF constructs—one which passes along side effects. The meaning of a VPCF term thus never generates any side effects itself. The type constructor T of the call-by-value monad is used to promote the types. More precisely, for any VPCF type σ, let $\sigma', \sigma^* \in \Sigma(\mathcal{D})$ be defined by induction on the structure of σ as follows.

$$\mathbf{num}' \ = \ \mathbf{num}$$

Table 4.4: Embedding $\mathcal{D}$ into $\mathcal{T}$.

$$
\begin{array}{rcl}
\alpha_r^{\mathbf{num}} & = & \underline{\lambda} x \in D^{\mathbf{num}}.\, x \\
\alpha_r^{\sigma\to\tau} & = & \underline{\lambda} x \in D^{\sigma\to\tau}.\, \underline{\lambda} y \in D^{\sigma'}.\, \alpha_c^{\tau}\ (x\ (\beta_r^{\sigma}\ y)) \\
\alpha_c^{\sigma}(x) & = & \eta\ (\alpha_r^{\sigma}\ x)
\end{array}
$$

$$
\begin{array}{rcl}
\beta_r^{\mathbf{num}} & = & \underline{\lambda} x \in D^{\mathbf{num}}.\, x \\
\beta_r^{\sigma\to\tau} & = & \underline{\lambda} x \in D^{(\sigma\to\tau)'}.\, \underline{\lambda} y \in D^{\sigma}.\, \beta_c^{\tau}\ (x\ (\alpha_r^{\sigma}\ y)) \\
\beta_c^{\mathbf{num}} & = & \beta \\
\beta_c^{\sigma\to\tau}(x) & = & \begin{cases} \uparrow^{\sigma\to\tau} & \text{if } \beta\ (\mathsf{Ap}\ (\eta\ (\underline{\lambda} z \in D^{(\sigma\to\tau)'}.\, \eta\ 0))\ x)\uparrow \\ \beta_r^{\sigma\to\tau}\ (\underline{\lambda} y \in D^{\sigma'}.\, \mathsf{Ap}\ x\ (\eta\ y)) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
(\sigma \to \tau)' & = & (\sigma' \to \tau^*) \\
\sigma^* & = & T(\sigma')
\end{array}
$$

Let the domains of results and computations in $\mathcal{T}$ be given by $\mathcal{R}_T[\![\sigma]\!] = D^{\sigma'}$ and $\mathcal{C}_T[\![\sigma]\!] = D^{\sigma^*}$.

We can now give the meaning of VPCF terms. For any VPCF term M of type σ, $\mathcal{T}[\![M]\!]\rho_{\mathcal{T}}$ is an element of $\mathcal{C}_T[\![\sigma]\!]$, where $\rho_{\mathcal{T}}$ is a $\mathcal{T}$**-environment**, *i.e.*, a map from variables to $\mathcal{D}$ such that $\rho_{\mathcal{T}}(x^{\sigma}) \in \mathcal{R}_T[\![\sigma]\!]$. The clauses for interpreting the VPCF constructs are given in Table 4.3, which use the functions of the monad and the semantic elements $\mathsf{Ap}^{\sigma,\tau}$ defined as

$$
\begin{array}{rcl}
\mathsf{Ap}^{\sigma,\tau} & = & \underline{\lambda} f \in D^{(\sigma\to\tau)^*}.\, \underline{\lambda} x \in D^{\sigma^*}. \\
& & \mu\ [T\ (\underline{\lambda} g \in D^{(\sigma\to\tau)'}.\, \mu\ ((T\ g)\ x))\ f]
\end{array}
$$

and where $(\underline{\lambda} x.\, e)$ is a meta-notation to denote elements of the model. Requirement (1), that the call-by-value monad operations be "interpretable", ensures that these definitions are good definitions.

4.4.3 Semantic Coercions

We are now ready to establish the key semantic result—the model $\mathcal{D}$ is a retract of the model $\mathcal{T}$. Let $\langle T, \eta, \mu, \beta \rangle$ be a call-by-value monad over the model $\mathcal{D}$. For each VPCF type σ, Table 4.4 defines elements $\alpha_r^\sigma \in D^{\sigma \to \sigma'}, \beta_r^\sigma \in D^{\sigma' \to \sigma}$, and functions $\alpha_c^\sigma : D^\sigma \to \mathcal{C}_T[\![\sigma]\!], \beta_c^\sigma : \mathcal{C}_T[\![\sigma]\!] \to D^\sigma$. (Note: The elements $\alpha_r^\sigma, \beta_r^\sigma$ are thus strict functions.) Intuitively, the function α_c^σ embeds elements of D^σ into computations in $\mathcal{C}_T[\![\sigma]\!]$ that have identical functional behaviour but no side effects; the function β_c^σ extracts the result of the computation and maps it to the element that has the same functional behaviour. One can show that $(\beta_c^\sigma \circ \alpha_c^\sigma)$ is the identity function, *i.e.*, D^σ is a **retract** of $\mathcal{C}_T[\![\sigma]\!]$. We thus define the retraction function $\delta_c^\sigma = (\alpha_c^\sigma \circ \beta_c^\sigma)$; the retractions δ_c^σ coerce a computation into one with identical functional behaviour but no side effects. Add the term constructors $\langle \mathsf{encap}_\sigma ; \sigma \Rightarrow \sigma \rangle$ for each VPCF type σ to VPCF; $\mathcal{T}$ is a model of this extended language with $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]\rho_{\mathcal{T}} = \delta_c^\sigma(\mathcal{T}[\![M]\!]\rho_{\mathcal{T}})$. The following theorem then captures the behaviour of the retractions as semantic coercions.

Theorem 4.4.2 *Suppose M, N are closed VPCF terms and $\mathcal{T}$ is a call-by-value monad over $\mathcal{D}$. Then*

1. *Adequacy:* $\mathcal{D}[\![M]\!] = \beta(\mathcal{T}[\![\mathsf{encap}_{\mathbf{num}}\ M]\!])$.
2. *Preservation: If $C[\cdot]$ is a closed VPCF context of type* **num**, *then* $\mathcal{T}[\![C[M]]\!] = \mathcal{T}[\![C[\mathsf{encap}_\sigma\ M]]\!]$.
3. *Full abstraction:* $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!] = \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!]$ *if and only if* $\mathcal{D}[\![M]\!] = \mathcal{D}[\![N]\!]$.

We use logical relations to prove Theorem 4.4.2. First, define the following family of binary relations that are used to relate elements of D^σ with computations in $\mathcal{C}_T[\![\sigma]\!]$ whose results have the same functional behaviour.

Definition 4.4.3 Define the relations $R_r^\sigma : D^\sigma \times \mathcal{R}_T[\![\sigma]\!]$ and $R_c^\sigma : D^\sigma \times \mathcal{C}_T[\![\sigma]\!]$ by

- $d\ R_r^{\mathbf{num}}\ e$ iff $d = e$.
- $d\ R_r^{\sigma \to \tau}\ e$ iff ($d\!\uparrow$ and $e\!\uparrow$), or
  ($d\!\downarrow$ and $e\!\downarrow$) and
  $\forall d'\!\downarrow, e'\!\downarrow .\ d'\ R_r^\sigma\ e' \supset (d\ d')\ R_c^\tau\ (e\ e')$

- $d\ R_c^\sigma\ e$ iff there is an e' such that $e = (\eta\ e')$ and $d\ R_r^\sigma\ e'$.

The relations R_c^σ can be shown to be directed complete, which together with the fact that the relations R_c^σ are "logical" helps establish the following lemma, that relates the meaning of a term in the two models.

Lemma 4.4.4 *Suppose that $\rho_{\mathcal{D}}$ is a $\mathcal{D}$-environment and $\rho_{\mathcal{T}}$ is a $\mathcal{T}$-environment such that $\rho_{\mathcal{D}}(x^\sigma)\ R_r^\sigma\ \rho_{\mathcal{T}}(x^\sigma)$ for all x^σ. Then for any PCF-term $M{:}\,\sigma$, $\mathcal{D}[\![M]\!]\rho_{\mathcal{D}}\ R_c^\sigma\ \mathcal{T}[\![M]\!]\rho_{\mathcal{T}}$.*

To prove Theorem 4.4.2, we need a few more simple facts:

Lemma 4.4.5

1. *For any $x \in D^\sigma$, $\beta_c^\sigma\ (\alpha_c^\sigma\ x) = x$.*
2. *If $d\ R_c^\sigma\ e$ then $d\ R_c^\sigma\ (\delta_c^\sigma\ e)$.*
3. *If $d\ R_c^\sigma\ (\delta_c^\sigma\ e)$ then $\alpha_c^\sigma\ d = \delta_c^\sigma\ e$.*

The proof, which we omit, uses a simple induction on the structure of types. Part 1 of Lemma 4.4.5 states the previously mentioned property that $\beta_c \circ \alpha_c$ is the identity function. An important consequence of this property is that the functions α_c^σ are injective. It therefore follows from the last two statements in Lemma 4.4.5 that the inverse of the logical relations are injective on the range of the coercions δ_c^σ.

Proof of Theorem 4.4.2

1. By Lemma 4.4.4, $\mathcal{D}[\![M]\!]\ R_c^{\mathbf{num}}\ \mathcal{T}[\![M]\!]$. Hence, by the second part of Lemma 4.4.5,
$$\mathcal{D}[\![M]\!]\ R_c^{\mathbf{num}}\ \mathcal{T}[\![\mathsf{encap}_{\mathbf{num}}\ M]\!].$$
   Using the definition of $R_c^{\mathbf{num}}$, this means that
$$\mathcal{T}[\![\mathsf{encap}_{\mathbf{num}}\ M]\!] = \eta_{\mathbf{num}}(\mathcal{D}[\![M]\!])$$
   By Property 5 in the definition of a call-by-value monad, the adequacy statement now follows.

2. By Lemma 4.4.4,
$$\mathcal{D}[\![M]\!]\ R_c^\sigma\ \mathcal{T}[\![M]\!] \tag{4.1}$$
Using the definition of R_c^σ, this means that
$$\mathcal{T}[\![M]\!] = (\eta\ e) \text{ and } \mathcal{D}[\![M]\!]\ R_r^\sigma\ e.$$
Thus, if x^σ is a variable that does not appear in the context $C[\cdot]$, then the environments $\rho_{\mathcal{D}}$ and $\rho_{\mathcal{T}}$ such that $\rho_{\mathcal{D}}(x^\sigma) = \mathcal{D}[\![M]\!]$ and $\rho_{\mathcal{T}}(x^\sigma) = e$ satisfy the conditions of Lemma 4.4.4. It therefore follows from Lemma 4.4.4 that $\mathcal{D}[\![C[x^\sigma]]\!]\rho_{\mathcal{D}}\ R_c^{\mathbf{num}}\ \mathcal{T}[\![C[x^\sigma]]\!]\rho_{\mathcal{T}}$ and so
$$\mathcal{D}[\![C[M]]\!]\ R_c^{\mathbf{num}}\ \mathcal{T}[\![C[M]]\!].$$
Now, using the definition of $R_c^{\mathbf{num}}$,
$$\mathcal{T}[\![C[M]]\!] = \eta(\mathcal{D}[\![C[M]]\!]) \tag{4.2}$$
Using the second part of Lemma 4.4.5 on (4.1),
$$\mathcal{D}[\![M]\!]\ R_c^\sigma\ \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]$$
Then by a similar argument as before,
$$\mathcal{T}[\![C[\mathsf{encap}_\sigma\ M]]\!] = \eta(\mathcal{D}[\![C[M]]\!]) \tag{4.3}$$
Hence we get the required statement from (4.2) and (4.3).

3. By Lemma 4.4.4 and the second part of Lemma 4.4.5, we have
$$\begin{array}{ccc} \mathcal{D}[\![M]\!] & R_c^\sigma & \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!] \\ \mathcal{D}[\![N]\!] & R_c^\sigma & \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!] \end{array}$$

Table 4.5: Operational Rules for encap.

General monad	$(\mathsf{encap}_{\mathbf{num}}\ M) \ \rightsquigarrow\ n, \quad \text{if } \beta(\mathcal{T}[\![M]\!]) = \mathcal{D}[\![n]\!]$ $(\mathsf{encap}_{\sigma\to\tau}\ M) \ \rightsquigarrow\ \mathsf{encap}_{\mathbf{num}}(M;0)\ ;\ \lambda y^{\sigma}.\,\mathsf{encap}_{\tau}\ (M\ (\mathsf{encap}_{\sigma}\ y))$
VPCF+K	$(\mathsf{encap}_{\mathbf{num}}\ M) \ \rightsquigarrow\ n, \quad \text{if } M \twoheadrightarrow_c n$
VPCF+S	$(\mathsf{encap}_{\mathbf{num}}\ M) \ \rightsquigarrow\ n, \quad \text{if } (M,\emptyset) \twoheadrightarrow_s (n,s')$

Using the third part of Lemma 4.4.5, we have

$$\begin{aligned} \alpha_c^{\sigma}\ (\mathcal{D}[\![M]\!]) &= \mathcal{T}[\![\mathsf{encap}_{\sigma}\ M]\!] \\ \alpha_c^{\sigma}\ (\mathcal{D}[\![N]\!]) &= \mathcal{T}[\![\mathsf{encap}_{\sigma}\ N]\!] \end{aligned}$$

It follows that

$$\begin{aligned} \mathcal{D}[\![M]\!] = \mathcal{D}[\![N]\!] \quad &\text{iff} \quad \alpha_c^{\sigma}\ (\mathcal{D}[\![M]\!]) = \alpha_c^{\sigma}\ (\mathcal{D}[\![N]\!]) \\ &\text{iff} \quad \mathcal{T}[\![\mathsf{encap}_{\sigma}\ M]\!] = \mathcal{T}[\![\mathsf{encap}_{\sigma}\ N]\!] \end{aligned}$$

where the first equivalence follows from the injectivity of α_c^{σ}.

This completes the proof of the theorem. ■

4.5 Syntactic Coercion Operators

We now use the framework of call-by-value monads to interpret extensions of VPCF with side effects. We first show how to add coercion operation encap to extensions of VPCF, with the operational rules for the coercion operation determined by the monad, and state a correctness theorem for the resulting coercions given some properties of the monadic semantics of the extension. We then apply the theorem to our two example extensions, VPCF+K and VPCF+S.

4.5.1 General extensions

Let L be an extension of VPCF and $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over a model $\mathcal{D}$ of VPCF. As can be seen from the examples in Section 4.3, a general way to specify the syntax of terms in L is to possibly add type and term constructors to VPCF. A denotational semantics $\mathcal{T}[\![\cdot]\!]$, over the monad $\mathcal{T}$, for L is given by extending the definition of σ' to the additional types σ of L and specifying functions $\mathcal{T}[\![f]\!]: \mathcal{C}_{\mathcal{T}}[\![\sigma_1]\!] \times \ldots \times \mathcal{C}_{\mathcal{T}}[\![\sigma_n]\!] \to \mathcal{C}_{\mathcal{T}}[\![\sigma]\!]$ for each new term constructor $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma\rangle$ in L, such that $\mathcal{T}[\![f]\!]$ is interpretable in $\mathcal{D}$. Together with Table 4.3, the clause $\mathcal{T}[\![f(M_1, \ldots, M_n)]\!]\rho_{\mathcal{T}} = \mathcal{T}[\![f]\!](\mathcal{T}[\![M_1]\!]\rho_{\mathcal{T}}, \ldots, \mathcal{T}[\![M_n]\!]\rho_{\mathcal{T}})$ defines the meaning of all terms of L. We add term constructors $\langle \mathsf{encap}_\sigma; \sigma \Rightarrow \sigma\rangle$ for each VPCF type σ to L; call the resulting language $L + \mathsf{encap}$. The meaning function $\mathcal{T}[\![\cdot]\!]$ can be extended to $L + \mathsf{encap}$ with $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]\rho_{\mathcal{T}} = \delta_c^\sigma(\mathcal{T}[\![M]\!]\rho_{\mathcal{T}})$. The following lemma, proved by expanding out the definitions of the retractions δ_c, gives us operational rules for the encap operators.

Lemma 4.5.1 *Let L be an extension of call-by-value PCF whose model $\mathcal{T}[\![\cdot]\!]$ is defined over a call-by-value monad. Then, for any L+encap term M and $\mathcal{T}$-environment ρ,*

- $\mathcal{T}[\![\mathsf{encap}_{\mathbf{num}}\ M]\!]\rho = \eta\ (\beta\ (\mathcal{T}[\![M]\!]\rho))$.
- *If y^σ is not free in M, then*

$$\mathcal{T}[\![\mathsf{encap}_{\sigma\to\tau}\ M]\!]\rho = \mathcal{T}[\![\mathsf{encap}_{\mathbf{num}}(M; 0); \lambda y^\sigma.\, \mathsf{encap}_\tau(M(\mathsf{encap}_\sigma y))]\!]\rho$$

The rewriting operational semantics framework makes it particularly easy to describe the operational rules for encap. The operational semantics of $L + \mathsf{encap}$ is obtained by adding the redexes in Table 4.5 to those of VPCF. Note that the second redex is independent of the language extension or the particular monad. Call the denotational semantics $\mathcal{T}[\![\cdot]\!]$ of L, over the monad T, **adequate**, if for all programs, *i.e.*, closed terms M of type **num**, $Eval_L(M) = n$ iff $\beta(\mathcal{T}[\![M]\!]) = \mathcal{T}[\![n]\!]$, and similarly for $L +$ encap. Using Lemma 4.5.1, we can prove the following lemma which shows that the operational rules for encap are faithful to its denotational semantics.

Lemma 4.5.2 *Let $\langle T, \eta, \mu, \beta \rangle$ be a call-by-value monad over a call-by-value model $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Then $\mathcal{T}[\![\cdot]\!]$ is also adequate for $L + \mathsf{encap}$.*

Using Lemma 4.5.2 and Theorem 4.4.2, we can now prove that under certain conditions encap accurately recovers the operational behaviour of VPCF:

Theorem 4.5.3 *Let $\mathcal{D}$ be a fully abstract, least fixpoint model for VPCF.*
Let $\langle T, \eta, \mu, \beta \rangle$ be a call-by-value monad over $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Then, for any closed VPCF terms M, N of the same type,

1. *Adequacy: $Eval_{L+\mathsf{encap}}(\mathsf{encap}_{\mathbf{num}} M) = n$ if and only if $Eval_v(M) = n$.*

2. *Preservation: Let $C[\cdot]$ be any closed VPCF context of type* **num**. *Then*

$$Eval_{L+\mathsf{encap}}(C[M]) = n \textit{ iff } Eval_{L+\mathsf{encap}}(C[\mathsf{encap}_\sigma\ M]) = n$$

3. *Full abstraction: $(\mathsf{encap}_\sigma\ M) \equiv_{L+\mathsf{encap}} (\mathsf{encap}_\sigma\ N)$ if and only if $M \equiv_{\mathrm{VPCF}} N$.*

While the above theorem describes the behaviour of encap on purely functional terms, one can also use the semantic theorems to describe the behaviour of encap on arbitrary terms. For instance, one can prove that encap produces observationally congruent terms from "functionally" equivalent terms. Define a closed term M of arbitrary type σ to **halt** iff $Eval(M;0)$ is defined. Then we can define two closed terms M and N of type σ to be functionally equivalent, $M \equiv^{\sigma}_{pure} N$, as follows:

- $M \equiv^{\mathbf{num}}_{pure} N$ iff $Eval(M) \simeq Eval(N)$
- $M \equiv^{\tau \to \nu}_{pure} N$ if M halts iff N halts and for all VPCF values V of type τ, $M\ V \equiv^{\nu}_{pure} N\ V$.

Inuitively, M and N are functionally equivalent if they have the same input-output behaviour. Then, the following theorem shows that functionally equivalent pieces of code are interchangeable within the scope of encap.

Theorem 4.5.4 *Let $\mathcal{D}$ be a fully abstract, least fixpoint model for VPCF. Let $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Suppose M, N are closed $L+$encap terms of PCF type such that $M \equiv_{pure} N$. Then*

$$(\mathsf{encap}\ M) \equiv_{L+\mathsf{encap}} (\mathsf{encap}\ N).$$

Since the logical relations establish a bijective correspondence between the call-by-value model of VPCF and the range of the retractions, we conjecture that the theory of the call-by-value model (when the logic is suitably defined) will be the same as for coerced expressions.

4.5.2 Adding encap to VPCF+K

Let $\mathcal{V}$ be any least fixpoint model of VPCF. We can define the continuation monad $\mathcal{K} = \left\langle K, \eta^{\mathcal{K}}, \mu^{\mathcal{K}}, \beta^{\mathcal{K}}\right\rangle$ over $\mathcal{V}$ by

$$\begin{aligned}
K(\sigma) &= (\sigma \rightarrow \mathbf{num}) \rightarrow \mathbf{num} \\
K_{\tau,\nu} &= \underline{\lambda} f \in V^{\tau\rightarrow\nu}.\ \underline{\lambda} x \in V^{K(\tau)}. \\
&\qquad \underline{\lambda}\kappa \in V^{\nu\rightarrow\mathbf{num}}.\ x\ (\underline{\lambda} v \in V^{\tau}.\ \kappa\ (f\ v)) \\
\eta^{\mathcal{K}}_{\sigma}(x) &= \underline{\lambda}\kappa \in V^{\sigma\rightarrow\mathbf{num}}.\ \kappa\ x \\
\mu^{\mathcal{K}}_{\sigma} &= \underline{\lambda} x \in V^{K(K(\sigma))}. \\
&\qquad \underline{\lambda}\kappa \in V^{\sigma\rightarrow\mathbf{num}}.\ x\ (\underline{\lambda} m \in V^{K(\sigma)}.\ m\ \kappa) \\
\beta^{\mathcal{K}} &= \underline{\lambda} x \in V^{K(\mathbf{num})}.\ x\ (\underline{\lambda} y \in V^{\mathbf{num}}.\ y)
\end{aligned}$$

One may verify that this definition gives a call-by-value monad. We can obtain a semantics $\mathcal{K}$ for VPCF+K by

$$\begin{aligned}
\mathcal{K}[\![\mathsf{callcc}]\!](f) &= \underline{\lambda}\kappa.\ f\ (\underline{\lambda} m.\ m\ \kappa\ \kappa) \\
\mathcal{K}[\![\mathsf{abort}]\!](f) &= \underline{\lambda}\kappa \in V^{\mathbf{num}\rightarrow\mathbf{num}}.\ f\ (\underline{\lambda} x \in V^{\mathbf{num}}.\ x)
\end{aligned}$$

Since $\beta^{\mathcal{K}}$ corresponds to evaluating in the empty context, we thus get the redex defined in Table 4.5 for VPCF+K. Note that the semantics of $\mathsf{encap}_{\mathbf{num}}$ coincides with that of first-class prompts [Fel88b]. The semantics $\mathcal{K}[\![\cdot]\!]$ corresponds directly to usual cps (continuation-passing style) transforms [Plo75, SF90]. It is thus easy to check that if $\mathcal{V}$ is an adequate model for VPCF, then the semantics $\mathcal{K}[\![\cdot]\!]$ is adequate for VPCF+K. Since any fully abstract model is also adequate, we can apply Theorem 4.5.3 to the case of VPCF+K, if the model $\mathcal{V}$ is fully abstract. Thus, to establish the correctness of encap for VPCF+K, all that remains is the construction of such a model, *i.e.*, prove the following theorem:

Theorem 4.5.5 *There exists a fully abstract, least fixpoint model $\mathcal{V}$ of VPCF.*

One obvious way to construct a fully abstract model of $\mathcal{V}$ is to take the term model consisting of equivalence classes of all closed terms. Unfortunately the term model for VPCF, while trivially fully abstract, is *not continuous*. However, by a suitable "context lemma" (*c.f.* [Mil77]), the term model for a purely functional call-by-value language is always monotone. Thus, if all types in the term model are finite, then it is also *continuous*. We rely upon this insight to prove Theorem 4.5.5. Instead of VPCF, we consider a language FVPCF in which all types are finite and which will serve as an approximation to VPCF. FVPCF has base types $\mathbf{num}_i$, whose elements are $\{0, \ldots, i\}$, that serve as finite approximations of the infinite type $\mathbf{num}$. FVPCF does not have fixed point operators, but includes suitable "finite" approximations of the other term constructors of VPCF modified to operate on the $\mathbf{num}_i$'s. More importantly, FVPCF needs to include constants representing divergence explicitly in its syntax to reflect the call-by-value operational equivalence. The continuous term model of FVPCF contains a lattice of "partial types" related by embedding-projection maps approximating each type of VPCF. We obtain a least fixed point model $\mathcal{V}$ for VPCF by taking the meaning of each type to be the (inverse) limit of its "partial type" approximations. In fact, each type is an algebraic cpo with its finite elements corresponding to the definable terms of FVPCF; definability of the finite elements is enough to prove full abstraction. The precise details of the construction appear in Chapter 3.

4.5.3 Adding encap to VPCF+S

To give a denotational semantics for VPCF+S, we need to model states. Since states may be circular, the appropriate representation of states has to be recursive. A suitable framework for building states is the language FPC defined in [Gun92], which includes a one-element type **unit**, product types, sum types, and recursive types. We can thus define a store call-by-value monad over $\mathcal{F}$ as follows. Using the recursive types of FPC, we first define types $\mathsf{Loc} = (\mu t.\, \mathbf{unit} + t)$ and Store to be the appropriate recursive type given by the equations

$$\begin{array}{rcl} \mathsf{Store} & = & \mathsf{Loc} \to (\mathsf{Res} + \mathbf{unit}) \\ \mathsf{Res} & = & \mathbf{num} + \mathsf{Loc} + (\mathsf{Res} \to \mathsf{Comp}) \\ \mathsf{Comp} & = & \mathsf{Store} \to (\mathsf{Res} \times \mathsf{Store}) \end{array}$$

(*c.f.* [Rie93a]). Let init be the initial store mapping every location to the right portion of the sum, *i.e.*, declaring each cell to be "unused". Let the call-by-value monad $\mathcal{S} = \left\langle S, \eta^{\mathcal{S}}, \mu^{\mathcal{S}}, \beta^{\mathcal{S}} \right\rangle$ be defined by

$$\begin{array}{rcl} S(\sigma) & = & \mathsf{Store} \to (\sigma \times \mathsf{Store}) \\ S_{\tau,\nu} & = & \underline{\lambda} f \in F^{\tau \to \nu}.\, \underline{\lambda} x \in F^{S(\tau)}. \\ & & \quad \underline{\lambda} s.\, \mathbf{pair}(f(\mathbf{fst}(x\ s)), \mathbf{snd}(x\ s)) \\ \eta^{\mathcal{S}}_{\sigma}(x) & = & \underline{\lambda} s.\, \mathbf{pair}(x, s) \\ \mu^{\mathcal{S}}_{\sigma} & = & \underline{\lambda} x \in F^{S(S(\sigma))}.\, \underline{\lambda} s.\, \mathbf{fst}(x\ s)\ \mathbf{snd}(x\ s) \\ \beta^{\mathcal{S}} & = & \underline{\lambda} x \in F^{S(\mathbf{num})}.\, \mathbf{fst}(x\ \mathsf{init}) \end{array}$$

The semantics $\mathcal{S}$ of types is determined by setting $(\mathsf{ref}\ \sigma)' = \mathsf{Loc}$; the semantics of the additional term constructors in VPCF+S can be written as meanings of the appropriate terms of FPC. Since $\beta^{\mathcal{S}}$ corresponds operationally to evaluating in the initial store, the operational rule in Table 4.5 is an implementation of $\beta^{\mathcal{S}}$. This breaks the usual "single-threaded" property of state; see below for more discussion.

One can show that if $\mathcal{F}$ is an adequate model for FPC then the resulting semantics

$\mathcal{S}[\![\cdot]\!]$ of VPCF+S is also adequate. Moreover, if the model $\mathcal{F}$ of FPC is fully abstract, then the submodel of $\mathcal{F}$ for VPCF obtained by taking $\mathbf{num} = (\mu t.\, \mathbf{unit} + t)$ is also fully abstract. Thus, by Theorem 4.5.3, we can obtain the correctness result for VPCF+S+encap if we can prove

Theorem 4.5.6 *There exists a continuous, fully abstract model $\mathcal{F}$ of FPC.*

We follow the construction for VPCF; the details also appear in Chapter 3. We define a finite language FFPC that includes an "empty" base type **void**, the one-element type **unit**, products and sums, but not recursive types. The main difficulty in constructing the model of FPC from the term model of FFPC is in defining the types of FFPC that serve as approximations to the recursive types of FPC and suitable embedding-projection maps between them. Intuitively, we think of a recursive type $\mu t.\, \sigma$ as the limit of its finite unwindings $\mu^n t.\, \sigma$, where $\mu^0 t.\, \sigma = \mathbf{void}, \mu^{i+1} t.\, \sigma = [\mu^i t.\, \sigma/t]\sigma$. The partial types approximating a recursive type $\mu t.\, \sigma$ are those approximating each unwinding $\mu^n t.\, \sigma$.

4.6 Example of encap with State

Let us consider an extended example of how to use encap for reasoning in a language with state. We use the syntax of Standard ML [MTH90] instead of VPCF in this example, partly because we will need a richer set of data types, and partly to improve readability.

The example is unification (also considered as a test case in [SRI91] for the Imperative Lambda Calculus). Ordinary (first-order) unification admits two elegant solutions: the first, Robinson's algorithm using substitutions, can be implemented in a purely functional manner; the second, using directed acyclic graphs (dags), is much more efficient but requires state. The following is an encoding of the latter algorithm; we leave it to the reader to find the purely functional encoding.

```
datatype term = Num | Typevar of string
                    | Fnsp of term*term
```

```
local
  datatype tag = Unbound | Instance of dagterm
  and dagterm  = DNum | DTypevar of string*(ref tag)
                      | DFnsp of dagterm*dagterm

  fun find (x,[]) = None
    | find (x,(y,z)::tl) =
      if x=y then Some z else find (x,tl)

  fun convertToDags (t1,t2) =
    let val table = ref []
        fun convert Num = DNum
          | convert (Typevar x) =
            (case find (x,!table)
               of Some r => DTypevar (x,r)
                | _ => let val p = (x,ref Unbound)
       in  table := p::(!table);
   DTypevar p
       end)
          | convert (Fnsp(t,u)) =
            DFnsp(convert t,convert u)
    in (convert t1, convert t2)
    end

  fun occur x DNum = false
    | occur x (DTypevar (y,r)) =
      (case !r
         of Unbound => (x=y)
          | Instance t => occur x t)
    | occur x (DFnsp (t1,t2)) =
      (occur x t1) orelse (occur x t2)
```

```
  fun unifyDags (DNum, DNum) = true
    | unifyDags (DTypevar v1, DTypevar v2) =
      if (v1=v2) then true else unifyVs(v1, v2)
    | unifyDags (DTypevar v1, t2) = unifyVT (v1, t2)
    | unifyDags (t1, DTypevar v2) = unifyVT (v2, t1)
    | unifyDags (DFnsp(t1, t2),DFnsp(t3, t4)) =
      (unifyDags (t1, t3)) andalso
      (unifyDags (t2, t4))
    | unifyDags _ = false

  and unifyVs (v1 as (x,r), v2 as (x',r')) =
      (case (!r, !r')
         of (Instance ty1, _) => unifyVT(v2, ty1)
          | (_, Instance ty2) => unifyVT(v1, ty2)
          | (Unbound,Unbound) =>
    (r := (Instance (DTypevar v2));true))

  and unifyVT (v1 as (x,r), t) =
      (case !r of Unbound =>
          if occur x t
             then false
             else (r := Instance t; true)
        | (Instance t') => unifyDags(t,t'))

in fun unify p = unifyDags (convertToDags p)
end
```

Terms are encoded by the datatype `term` and the function `unify` returns `true` if the terms are unifiable and `false` otherwise. The datatype `dagterm` changes only the representation of variables—variables can now point to other terms as an "instance", which serves to make unification of terms with variables a simple matter. Two auxiliary functions in this code use assignments. The function `convertToDags`, which converts elements of the datatype `term` into dags, uses a reference cell to store type variables that have already been assigned references; the function `unifyVT` also uses assignment to link a variable to a new instance.

We can use the two encodings of `unify` to illustrate a general programming methodology of modular refinement where a programmer first implements a module purely functionally and then refines it using state to produce a more efficient version. In this context, one may hope to use the imperative version of `unify` in place of the functional version in any program since they have the same input-output behaviour. However, as the example in the introduction showed, in the presence of side-effects, such a refinement may not always preserve the behaviour of the whole program. If the language includes `encap`, we can guarantee the correctness of this process of modular refinement by wrapping the code of `unify` with `encap`. By Theorem 4.5.4, having the same input-output behaviour is then enough to guarantee the correctness of this refinement, since the refinement is within the scope of `encap`.

If we do not wish to include `encap` in the language, we can still use `encap` as a useful reasoning tool to establish the correctness of replacing the functional version with the imperative version directly (*i.e.*, without wrapping it with `encap`). First, observe that the code accesses no reference cell that is not allocated during a call to `unify`, and that none of the reference cells can be accessed after a call to `unify`. Also observe that the declaration yields a value, and that the arguments to `unify` cannot themselves contain any reference cells (due to the limited structure of the datatype `term`). From these facts and the denotational or operational semantics of `encap`, one may prove that wrapping `encap` around the entire expression yields a term that is operationally equivalent to the expression itself. Similarly, since the functional version of `unify` also cannot access any reference cells from outside, one may show that wrapping `encap` around it gives an operationally equivalent expression. Since Theorem 4.5.4 gives us the operational equivalence of the two pieces of code wrapped with `encap`, it follows that the two pieces of code are equivalent even without `encap` around them.

This example and the example in [Rie93a] illustrate a common technique for proofs of correctness. If one can show that state is already encapsulated (by showing that applying `encap` results in an operationally equivalent term), one can then apply Theorem 4.5.4 to deduce that functional reasoning principles are sufficient. This is an important reasoning principle and well-supported by programmer's intuition.

4.7 Related Work

This approach in this paper is certainly not the first attempt to control side effects in functional languages; there is a long tradition of language designs that attempt to control side effects. Such efforts have mainly focused on simplifying the reasoning principles required for proving properties of programs and on automatically determining opportunities for parallel evaluation. Broadly speaking, two distinct and seemingly unrelated camps have emerged, divided along the traditional call-by-name versus call-by-value line.

Most of the work has followed the call-by-name tradition, beginning with Algol 60. Reynolds' more contemporary work on syntactic control of interference [Rey78, Rey89], Idealized Algol [Rey81], and Forsythe [Rey89] have carried forth this tradition. Each of these languages incorporates restrictions on both types and terms so as to force assignments and references to be well-behaved. For example, in Idealized Algol, the basis for the other two programming language designs, terms that write to the state are grouped into the type of *commands*; the other two types of basic terms, locations and expressions, may read the state but not modify the state. This greatly simplifies reasoning about programs: with the call-by-name parameter passing, programs of command type may be unwound into a simple sequence of imperative statements—in other words, user-defined functions can be completely removed from the program by partial evaluation [Rey89]. The imperative lambda calculus (ILC) of [SRI91] has a similar design philosophy in separating the terms that can modify the state from those that may not. The lazy functional programming community has also been active in searching for new, clean ways of adding state to programming languages; just a few of these efforts are dialogue-based I/O in Haskell [HPW+92], monads [PW93, Wad92], continuation-based mutable abstract datatypes [Hud92], and lazy functional state threads [LP94] (whose type-theoretic encapsulation of state is remarkably similar to the work in semantics of [OT93]). In each case, the design preserves some functional character of the language—the equation (β) is sound for proving operational equivalences, for example, so terms of functional type do not yield side effects when applied.

Another language in the call-by-name tradition, and possibly the closest design to VPCF+S+encap, is the language λ_{var} of [ORH93]. Syntactically, λ_{var} is an extension of the untyped λ-calculus with assignments and some additional operations for encapsulating state. λ_{var} enjoys a property similar to our main theorem—two pieces of code M, N without assignments are operationally equivalent in the full language iff they M, N are operationally equivalent in the language without assignments; in contrast to our main theorem, no operations are required to coerce M and N. λ_{var} enforces this purity of assignments via two new term constructors, pure and return. Using the syntax of this paper, the operational semantics for the special operations of λ_{var} are

$$\begin{aligned}(\mathsf{pure}\,(\mathsf{return}\; n)) &\leadsto n\\ (\mathsf{pure}\,(\mathsf{return}\,(\lambda x.\, M))) &\leadsto \lambda x.\, \mathsf{pure}\,(\mathsf{return}\; M)\end{aligned}$$

As with the semantics of encap, pure applied to functional abstractions forces the value returned by the function to be pure; the argument, on the other hand, is not forced to be pure (probably because of the call-by-name reduction semantics). Unlike encap, the operation return is needed in the language to mark which subexpressions may be marked as side-effect free. Those expressions that are not returned, *e.g.*, (pure 0), are "stuck" expressions that cannot be reduced further; the counterpart in VPCF+S+encap is an expression like (encap l) for some location l. The advantage of λ_{var} over VPCF+S+encap is that it can be implemented with a single-threaded store; the disadvantage is that it is manifestly based on call-by-name semantics (as with ILC and Idealized Algol), and that the syntax of assignments, pure, and return are somewhat restrictive.

Far less attention has been paid to controlling side effects in the call-by-value realm. Some of the most well-developed approaches are the type and effect systems for assignments [JG91, Luc87, LG88, Tal93, TJ92a], continuations [JG89], and exceptions [GS94]. In each of these languages, a static system assigns both types and possible effects to expressions; the type and effect can then be used in the generation of code for parallel architectures and other applications. For example, in the type and effect system for state, one envisages memory as divided into a set of *regions*, or blocks of locations. The type and effect system calculates a conservative approximation of dividing the cells into regions. Consider a simple example, the "gensym" function that returns a new integer each time it is called:

$$(\lambda x^{\mathsf{ref}\ \mathbf{num}}.\, \lambda d^{\mathbf{num}}.\, (x{:=}\ \mathsf{succ}\ (!x)); (!x))\ (\mathsf{ref}\ 0)$$

In the type and effect system, the type is more complicated than $(\mathbf{num} \rightarrow \mathbf{num})$ (the type in VPCF+S). This expression creates a new cell, so the effect is $\mathsf{init}(\rho)$ for some region ρ, and when given an argument of type **num** the expression returns a **num** and reads in writes in region ρ, so the type is

$$(\mathbf{num} \overset{\mathsf{read}(\rho),\mathsf{write}(\rho)}{\rightarrow} \mathbf{num}).$$

The effects above the $\rightarrow$ is called a *latent* effect, since they are effects poised to fire during a function call but do not occur during the evaluation of the expression.

Automatic parallelization of code with side effects is one of the primary uses of effect systems (see, *e.g.*, [Hue93]). In the case of state, for example, if the manifest effects of two subterms of a term occur in different regions, the two subterms may be safely reduced in parallel. The operations encap may also be used for this purpose, albeit at a much less refined level: for example, the subterms $(\mathsf{encap}\ M)$ and $(\mathsf{encap}\ N)$ may be reduced in parallel in evaluating $((\mathsf{encap}\ M)\ (\mathsf{encap}\ N))$; the correctness of this parallelization follows from Theorem 4.5.3 and a suitable standardization theorem for VPCF (*c.f.* [Plo75]).

The type and effect system has at least two advantages over the encap operation in VPCF+S+encap. First, using ideas from type inference, the compiler can determine the type and effect of an expression automatically [TJ92a]; the programmer need not worry about the type and effects of expressions initially. Second, the language can be implemented using a single-threaded state: type and effect annotations give hints to where parallelization can happen safely without write-after-read or write-after-write conflicts, *i.e.*, where parallelization can occur without violating the sequential, single-threaded semantics; encap breaks the single-threaded semantics of state. The automatic character of type and effect systems also has drawbacks, primarily because it requires a kind of conservativity. For instance, the expression

$$(\mathsf{if0}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ x\!:= 0; 8)$$

has a type **num** and effect $\mathsf{write}(\rho)$ if x is bound to a location in the region ρ, even though the expression has no effect. Indeed, there may be a way to meld the two approaches together by redefining the semantics of VPCF+S to use computations of the form

$$\mathsf{Comp} \quad = \quad (\mathsf{Region} \to \mathsf{Store}) \to (\mathsf{Res} \times (\mathsf{Region} \to \mathsf{Store}))$$

so that, in effect, the store is divided into regions. Defining β in this setting is the only difficult part, along with tying it into a general theorem like Theorem 4.5.3.

4.8 Conclusion

We have shown how to construct encapsulation operations that force portions of a program to behave purely functionally. The coercion operations rely on a few deep denotational theorems. The framework developed here, in contrast to the framework of [Rie93a], encompasses sequential programming languages, and hence is applicable to a wider variety of practical languages.

It is worth reexamining the stated design goals of encap from the introduction. The first design goal, to allow the programmer to protect pieces of code, requires an efficient implementation of encap. This seems to be very unclear in the case of state. Morrisett [Mor93] has shown how one can implement first-class stores in Standard

ML, with operations for naming the current store and returning back to a previously named store. It is not hard to implement `encap` at base type: one could name the initial store, and when entering a `encap`, name the current store and "snap-back" to the initial store. The higher-type `encap` could be implemented by evaluating the term inside the `encap` in the initial store, then using the closure if the evaluation converges to a λ-abstraction. In the case of polymorphic languages, some variant of the `typeseq` operation of [HM95] is essential in implementing `encap`. Without single-threading of state, however, the efficiency of a language with state and `encap` may discourage anyone from using it. There may be similar efficiency issues with `encap` in the language with control.

On the other hand, we feel that the second design goal—to encapsulate state for proving properties of programs—is met. We feel that the example of Section 4.6 is quite convincing, and it would be worth pursuing making a formal logic to reflect the reasoning of the example.

The techniques developed in this chapter are useful beyond the context of the particular problem studied here. The fully abstract models of VPCF and FPC can be used in establishing other general results about sequential languages. For example, we can use them to prove that translations between programming languages are fully abstract (*c.f.* [Rie93b]). More importantly, this work brought out the shortcoming of monads as a framework for expressing the semantics of side effects when applied to models arising from other monads (such as other side effects). Since the definition of a call-by-value monad formulated here had to bypass precisely this problem, it may prove to be more appropriate, for example, in analyzing languages with more than one side effecting feature.

Chapter 5

Standard ML-NJ weak polymorphism and imperative constructs

5.1 Introduction

In the absence of side effects and so-called control effects (continuations and exceptions), the type inference methods developed by Curry, Hindley and Milner [CF58, Hin69, Mil78] provide an effective and useful method for detecting type errors at compile time. For any typable expression, the type inference algorithm computes a single *most general type,* from which all other types of the expression may be derived easily. Although the worst-case complexity is high [KMM91, KTU90], the algorithm has been found efficient in practice. More importantly, experience suggests that it is possible for a programmer to predict, with relative ease, the type that the algorithm will produce. In addition, type inference allows a form of error detection that is unfamiliar to explicitly-typed programming. Even when a declaration is typable, the programmer may detect an error by noticing that the type inferred by the compiler differs from the type expected by the programmer (*c.f.* [Koe92]).

Unfortunately, the straightforward, predictable polymorphic typing algorithm fails for programs with side-effects or control effects. To illustrate the problems involved, we use a lambda calculus extended with let-expressions (for polymorphism) and imperative operators `ref`, !, and $:=$. Briefly, $\texttt{ref}\, x$ allocates a new cell, initialized to x, $r := x$ updates the contents of reference r to the value of x, and $!r$ returns the current contents of reference r. The incorrectness of the naive polymorphic typing of side-effects is illustrated by the program

$$\texttt{let}\ \ r = \texttt{ref}\,(\lambda x.\, x)\ \ \texttt{in}\ \ r := \ \lambda x.\, x + 1\ ;\ !r\ \mathit{true}$$

where ; is the usual sequencing operator (definable by lambda abstraction under call-by-value). If we apply the Milner typing algorithm, we give r type $\forall\alpha.[(\alpha \rightarrow \alpha)\ \mathit{ref}]$, where $\tau\ \mathit{ref}$ is the type of references to values of type τ. Intuitively, this means that for any type σ, α can be instantiated to σ to give r the type $(\sigma \rightarrow \sigma)\ \mathit{ref}$. Both expressions in the body of the declaration are well-typed, under this assumption, since we can take the instances $(\mathit{int} \rightarrow \mathit{int})\ \mathit{ref}$ and $(\mathit{bool} \rightarrow \mathit{bool})\ \mathit{ref}$ for the two occurrences of r, respectively. However, since the assignment changes the value of r from a polymorphic function to an integer function, the application of $!r$ to true results in a run-time type error. Thus, a naive polymorphic treatment of side-effects does not prevent run-time type errors in the presence of assignment.

We can view solutions to this problematic interaction of polymorphism and side-effects from a general perspective, by understanding the way polymorphic type inference works. In typing an expression of the form $\texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2$, the most general type τ of e_1 is calculated. Certain type variables in τ are determined to be "generic" or "polymorphic". If $\alpha_1, \ldots, \alpha_n$ is a list of all the generic variables in τ, we write the type of e_1 as $\forall\alpha_1 \ldots \forall\alpha_n.\tau$. In typing e_2, all the generic/polymorphic type variables may then be replaced by types, independently chosen for each occurrence of x. For example, if e_1 has type $\forall\alpha.\alpha \rightarrow \mathit{int}$, and e_2 is the expression $x\ x$, then e_2 may be typed by replacing α by $\mathit{int} \rightarrow \mathit{int}$ in the first occurrence of x and int in the second occurrence. If a type variable is determined to be non-polymorphic, it can be replaced only by the same type in all occurrences of the let-bound variable x. Thus a critical aspect of polymorphic type inference is the method for identifying the type variables that may be safely be treated as polymorphic. In the absence of side-effects or control effects, there is a simple condition depending on the types of free variables of e_1. The main topic of this chapter, and previous studies of polymorphism and imperative constructs, is the safe and flexible extension of the condition for pure terms to terms with side-effects. Over the past decade, a number of restrictions on the polymorphic uses of references have been proposed. Some of these have been implemented and others have been analyzed and proved correct, as described below. Our main contribution is a precise definition of a type system that appears to subsume the algorithm used in SML-NJ and a correctness proof for it. Although similar problems occur with continuations and exceptions, as illustrated in [HL91, WF92], we restrict our attention to side effects, in this chapter.

In Standard ML of New Jersey [AM87, AM91], so-called weak type variables are used to characterize functions that may create reference cells after some number of applications. Since the type of a polymorphic function may become more specialized with each application (as, for example, when a polymorphic function is applied to a monomorphic argument), many functions that create references could create polymorphic references for some actual parameters, and monomorphic references for others. For example, the function

$$\lambda f.\, \lambda x.\, \mathtt{ref}(fx)$$

creates a reference to the result of applying the first actual parameter to the second. This function is unproblematic for creating references to integers or booleans, but we cannot allow the result of application to two arguments to be used polymorphically because of the phenomenon described above. The most restrictive solution to the problem of typing references would be to require some fixed, non-polymorphic type for this function. A more flexible approach might be to force every argument to this function to be non-polymorphic. However, this too is overly restrictive since the reference cell is not created until the second actual parameter is supplied. For example, we may safely obtain a polymorphic function by applying to the polymorphic identity, provided any eventual second argument is non-polymorphic. The main innovation of SML-NJ weak typing, over previous approaches, is to use numeric indices on each type variable to count the number of function applications required to produce a reference cell of that type. Although this system has seen widespread and largely successful use in recent years, the SML-NJ algorithm has never been presented in a form other than relatively complicated source code, and consequently has never been analyzed or proved sound.

We present our type system as a set of typing rules, using a mild extension of the weak type variables of SML-NJ. These rules were derived from experiments with the algorithm implemented in SML-NJ, Version 75 (November 11, 1991) and perusal of the source code. Although our original intention was to formalize the implemented algorithm exactly, we found it easier and more informative to define a more general type system with simpler typing rules and fewer apparent special cases. One insight came as a result of examining the way that the type of an expression depends on the types of its non-polymorphic free variables. In SML-NJ, every weak type variable appears throughout the type of an expression with the same strength. For example, the expression `ref` has the type $\alpha^1 \rightarrow \alpha^1\ ref$. Such a restriction forces the numeric superscripts in the type of a free variable to characterize two sources of storage allocation. They must simultaneously give a lower bound on the number of applications of the variable itself and a lower bound on the number of applications of the expression containing the variable that may result in a reference cell being created. Thus, in typing an expression, the type assumption of a free variable needs to be modified according to the subexpression that one is typing. We avoid this by allowing both "ordinary" and "weak" occurrences of a type variable in a type. This leads to the formulation of a stronger typing rule for application than the compiler output indicates. Consequently, our typing rules appear to subsume the implemented algorithm, particularly for applications of purely functional expressions to potentially imperative arguments. We show that our typing rules are sound, using an operational semantics of ML with references, implying the semantic soundness of any algorithm that decides a conservative approximation to these rules.

More expressive type systems have been developed by Leroy and Weis [LW91], and, independently, by Talpin, Jouvelot and others [TJ92b, Wri92], partly based on the framework of Gifford's FX system [JG91]. In contrast to the SML-NJ algorithm and preceding ML compilers, which keep track of the types of all values in the store, these approaches approximately identify the values in the store that are accessible in a particular context. This additional accessibility information leads to type systems that are capable of giving functions which use references only locally the same types as their corresponding applicative versions. However, to be able to maintain this information, more complicated type expressions are required. While the more complex systems are promising, it is more difficult to predict the types of certain functions. Since we believe that the success of ML typing lies not only in its correctness, but the ease with which it may be understood in programming, any added complication to the type system deserves careful scrutiny. Consequently, it seems worthwhile to explore systems using only weak type variables and determine the limits of this approach.

5.2 Weak Polymorphism

In the approach we refer to as *weak polymorphism,* each type variable has an associated *degree of strength,* or *degree* for short. In the SML-NJ implementation, a degree may be a nonnegative integer, or ∞. Variables of infinite degree are printed without any degree designation. Intuitively, the degree of a type variable α, occurring in the typing of some expression e, indicates how many applications of e are required to produce a reference to a value whose type contains α. Functions that do not produce references at all are typed using variables of infinite degree. For example, the type of `ref` itself is

$$\texttt{ref} : \alpha^1 \rightarrow \alpha^1 \; \mathit{ref}$$

where the superscript indicates that α^1 is a weak type variable of degree 1. This type implies that when the function `ref` is applied, α may appear in the type of a stored value, and should not be treated polymorphically. However, the α in the type of `ref` is polymorphic, as is any type variable of non-zero degree in the type of a closed expression. This means that `ref` may be applied to an argument of any type, but the result cannot be used polymorphically. Another example is the list-producing function $\lambda x.\, \texttt{ref}[x]$, which has type $\alpha^1 \rightarrow \alpha^1 \; \mathit{list} \; \mathit{ref}$ since it takes one application of this function to produce a reference to a value whose type contains

α, or some type substituted for α. Since each application weakens the type of a function, each surrounding lambda abstraction strengthens every type variable in the type. As another interesting example, the expresion $\lambda f.\, \lambda x.\, \texttt{ref}(f x)$ has the type $(\alpha \rightarrow \beta^2) \rightarrow (\alpha \rightarrow \beta^2\ ref)$, where the reader may note that it is only the result of the function f which is constrained to be weak and of strength 2.

We may put the SML-NJ type system in perspective by reviewing the history of polymorphic type systems for imperative constructs. Early implementations of ML only allowed references to values of monomorphic, statically-known types. This approach guarantees soundness, but makes it impossible to write polymorphic functions that create or use references. In his thesis, Damas introduced the first type system to allow polymorphic use of functions that create references [Dam85]. Unfortunately, although his system has not been shown unsound, his claimed soundness proof is incorrect. Tofte was the first to give a type discipline for polymorphic references together with a correct soundness proof [Tof88, Tof90]. The central idea of Tofte's solution is to partition the types into "imperative" and "applicative" types, requiring all type variables that occur in an imperative type to be imperative. Values of imperative type can be stored, but only applicative type variables are treated polymorphically, successfully avoiding the creation of references to polymorphic values. MacQueen's system, implemented in the Standard ML of New Jersey ML compiler, is largely derived from Tofte's, using the numeric "degree" of a weak type variable to count how many function applications it takes for a type variable to become "imperative."

The power of weak polymorphism may be understood by comparison with Tofte's system. In Tofte's system, any type variable that appears in the type of a value that could potentially be stored has to be imperative and hence cannot be used polymorphically, even if the value does not actually appear in the store yet. Technically, imperative type variables are treated non-polymorphically only in the types of so-called "expansive" expressions, which include `let` expressions. Therefore, we illustrate the limitations of Tofte's system using the following expression, *rid*, which is a version of the identity function using side-effects.

$$rid \quad \equiv \quad \texttt{let}\ z = dummy\ \texttt{in}\ \lambda x.\, !(\texttt{ref}\ x)$$

Since x is stored, its type must be imperative. Consequently, in Tofte's system, the expression *rid* is given a type $u \rightarrow u$, where u is an imperative type variable that cannot be generalized. With weak polymorphism, however, we can take into account the fact that *rid* must be applied once before the reference to x is created. This allows us to give *rid* the type $\alpha^1 \rightarrow \alpha^1$, where the type variable α^1 can now be generalized. Since *rid* may be used polymorphically using weak polymorphism, but not in Tofte's system, the expression

$$\texttt{let}\ f = rid\ \texttt{in}\ f\,1\ ;\ f\,true$$

is typable in SML-NJ but not in Tofte's system.

In our typing rules, we generalize the SML-NJ type expressions slightly to associate numeric indices with ordinary type variables. In addition, a type variable can have both "ordinary" and "weak" occurrences, as well as occur with different strengths in a type. Aside from these relatively minor changes, which allow us to express the relationship between degrees of various type variables more clearly, our type expressions are identical to those used in Standard ML of New Jersey. It seems that significantly more complicated typing rules are needed if only the SML-NJ type expressions are used. In fact, we were not able to devise a sound system with SML-NJ type expressions using ordinary proof rules without excessively complicated side conditions.

5.3 Language

5.3.1 Syntax

In describing our typing rules and proving them correct, we use an untyped call-by-value language with ML-style imperative features. In writing the expressions of our language we allow arbitrary term constructors of specified arity. More formally, the language is parameterized by a set C of pairs, $\langle f, n\rangle$, where $n \geq 0$ gives the arity of the term constructor f. Having term constructors, as opposed to term constants, allows one to represent non-strict operators in the call-by-value language. The expressions are given by the grammar

$$\begin{array}{rcl} e & ::= & x \mid f(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \lambda x.\, e \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \\ & & l \mid \mathtt{ref} \mid !e \mid e_1 := e_2 \end{array}$$

where $\langle f, n\rangle \in C$, $x \in Var$, $l \in Loc$, Var is a denumerable set of variables, and Loc is a denumerable set of location constants. The location constants do not appear in user programs, but help in defining the operational semantics. We thus define programs to be closed expressions not containing any location constants.

In defining the operational semantics, it is useful to distinguish a subset of closed expressions that cannot be further evaluated. These are called *values*. Besides some expressions that depend on the particular choice of term constructors, the set of values always includes λ-abstractions, location constants, and the constant `ref`.

$$v \ ::= \ \cdots \mid \lambda x.\, e \mid l \mid \mathtt{ref}$$

The additional values that depend on the term constructors are specified in a set V. Each element of V is of the form $f(d_1, \ldots, d_n)$ where $\langle f, n\rangle \in C$ and each d_i is an expression or a value.

5.3.2 Operational Semantics

We use a rewriting semantics to specify operational behavior. In defining the operational semantics, we use a syntactic form `wrong` to represent run-time errors.

We begin by defining a relation, $\leadsto$, with $e \leadsto r$ indicating that e is a purely functional redex reducing directly to r, *i.e.*, without any side-effects. The result, r, of the reduction is either another expression or `wrong`. The behavior of the functional term constructors is given by a set F of pairs of the form $\langle e, r\rangle$, indicating that e reduces to r, where e is an expression of the form $f(e_1, \ldots, e_n)$ and r is another expression or `wrong`. The purely *functional* reduction relation, $\leadsto$, is defined by

$$
\begin{array}{rcll}
(\lambda x.\, e)\; v & \leadsto & [v/x]e & \\
\texttt{let}\ x = v\ \texttt{in}\ e & \leadsto & [v/x]e & \\
v_1\; v_2 & \leadsto & \texttt{wrong} & \text{if } v_1 \text{ not a } \lambda\text{-abstraction, } \texttt{ref} \\
e & \leadsto & r & \text{if } \langle e, r\rangle \in F
\end{array}
$$

To impose an order on the evaluation of subexpressions, we use the notion of an *evaluation context* [WF92]. The evaluation contexts $E[\,]$ are defined by the following grammar

$$E[\,] \;::=\; [\,] \mid E\; e \mid v\; E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \cdots \mid \texttt{ref}\ E \mid\, !E \mid E := e \mid v := E$$

where the unspecified productions are used to reflect the order of evaluation of the arguments of term constructors and are of the form $f(d_1, \ldots, d_n)$, where exactly one of the d_i's is an evaluation context E and the others are values v or expressions e; these are specified in a set EV. Intuitively, the hole in an evaluation context specifies the next subexpression to be evaluated.

To handle imperative features, our operational rules will manipulate "stores" in addition to terms, where a store is a finite partial map from location constants to values. The basic one-step evaluation relation $\longmapsto$ is defined in Table 5.1, with $(e, s) \longmapsto (r, s')$ indicating that the expression e in store s evaluates in one step to the result r resulting in new store s'. The result r is either another expression or the run-time error `wrong`. The reduction relation is parameterized by the choice of V, F, and EV, which we collect in a signature $\langle V, F, EV\rangle$.

The result of execution of any program, obtained by evaluating it in the initial empty store, is captured by a partial function *Eval*. Writing $\longmapsto\!\!\!\!\rightarrow$ for the reflexive and transitive closure of $\longmapsto$, $Eval(e) = a$ iff $(e, \mathbf{init}) \longmapsto\!\!\!\!\rightarrow (a, s')$ for some store s', where the answer a is either a value or `wrong` and **init** is the empty store that is undefined on all locations. A program e is said to *diverge* if there is an infinite reduction sequence from e, *i.e.*, for every expression e' such that $e \longmapsto\!\!\!\!\rightarrow e'$, there is an expression e'' such that $e' \longmapsto e''$. If a program e diverges, $Eval(e)$ is undefined. We say that a program

$$\begin{array}{rcll}
(E[e], s) & \longmapsto & (E[e'], s) & \text{if } e \leadsto e', e' \neq \texttt{wrong} \\
(E[\texttt{ref } v], s) & \longmapsto & (E[l], s[l \mapsto v]) & \text{where } l \notin dom(s) \\
(E[!l], s) & \longmapsto & (E[s(l)], s) & \\
(E[l := v], s) & \longmapsto & (E[v], s[l \mapsto v]) & \\
(E[e], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } e \leadsto \texttt{wrong} \\
(E[!v], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } v \notin Loc \\
(E[v_1 := v_2], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } v_1 \notin Loc
\end{array}$$

Table 5.1: One-step evaluation relation.

e does not lead to a run-time error if $Eval(e) \neq \texttt{wrong}$.

In the description of the previous paragraph, we have made two implicit assumptions: (i) if $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (a, s)$ and $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (a', s')$, where a and a' are values or `wrong` then a and a' are identical, and (ii) it is impossible that $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (e', s)$ and e' is not a value or `wrong` and there is no e'', s' such that $(e', s) \longmapsto (e'', s')$. The first property is needed for *Eval* to define a *function* and the second property is needed to ensure that all run-time errors are captured accurately by reduction to the expression `wrong` in our system. It is worth mentioning however that our proof of soundness does not rely on any of these properties. Both these properties depend on the choice of V, F, and EV used to express the behavior of the term constructors. In the following, we identify one sufficient condition on them that ensures that we have properties (i) and (ii).

Definition 5.3.1 Let $e \equiv f(e_1, \ldots, e_n)$ be any closed term where $\langle f, n \rangle \in C$. Consider the following three possiblities for the expression e:

(i) The expression e is a value, *i.e.*, $e \in V$.

(ii) There is an $f(d_1, \ldots, d_n) \in EV$ and integer i such that d_i is an evaluation context E, $e_i \notin V$ and $e \equiv f(d_1, \ldots, e_i, \ldots, d_n)$.

(iii) For some r, we have that $\langle e, r \rangle \in F$.

We say that the signature $\langle V, F, EV \rangle$ is

- **deterministic**, if for every closed expression $e \equiv f(e_1, \ldots, e_n)$ at most one of (i), (ii), (iii) is true with the choice of $i, f(d_1, \ldots, d_n)$ unique in case (ii), and the choice of r unique in case (iii).

- **complete**, if for every closed expression $e \equiv f(e_1, \ldots, e_n)$ at least one of (i), (ii), or (iii) is true of e.

If the signature is *deterministic* then we can show that property (i) holds, *i.e.*, that the result of evaluation is unique. And if it is *complete* then we can show that property (ii) holds, *i.e.*, that evaluation never results in "stuck" expressions.

Lemma 5.3.2 *Assume that a signature $\langle V, F, EV\rangle$ is deterministic and complete. Then, for every closed expression e, either e diverges, or $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (v, s)$ for a unique value v, or $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (\mathtt{wrong}, s)$.*

5.4 Type System

5.4.1 Type Expressions and Instances

Our type expressions are built from type variables using the constructor $\rightarrow$ for function types, and *ref* for reference cells, in addition to type constructors for other datatypes. To incorporate weak polymorphism, every "weak" type variable will have an index which is its degree of strength. However, unlike ML, in our type system an ordinary type variable will also have an associated index in our type system.

We start with a set B of type constructors for datatypes such as integers, lists etc. Each element of B is a pair of the form $\langle b, n\rangle$, where $n \geq 0$ gives the arity of the type constructor b. Let TVars be an infinite set of type variables whose elements will be indicated by metavariables $\alpha, \alpha_1, \ldots, \beta, \beta_1, \ldots$. A "weak" occurrence of a type variable will be signified by underlining it. As mentioned before, every "weak" and "ordinary" occurrence of a type variable has an associated index, which will be a natural number. We thus arrive at the following grammar for generating pre-types, some of which will be deemed to be well-formed types:

$$\rho \;::=\; \alpha^j \mid \underline{\alpha}^j \mid \rho_1 \rightarrow \rho_2 \mid \rho_1\ ref \mid b(\rho_1, \ldots, \rho_n)$$

In this grammar we assume that $\langle b, n\rangle \in B$ and $j \in \mathcal{N}$. Intuitively, a pre-type of the form $\rho_1 \rightarrow \rho_2$ can be used as the type of some expression only if any type variable occurring "imperatively" in ρ_1 does not occur "applicatively" in ρ_2 and vice-versa. We now elaborate the precise condition distinguishing types from pre-types.

For any pre-type ρ, we define its free type variables, $\mathrm{FTV}(\rho)$, to be the set of variable names appearing in it, without regard to their index or whether they are underlined. More precisely,

$$\begin{array}{rcl}
\mathrm{FTV}(\alpha^j) & = & \{\alpha\} \\
\mathrm{FTV}(\underline{\alpha}^j) & = & \{\alpha\} \\
\mathrm{FTV}(\rho_1 \rightarrow \rho_2) & = & \mathrm{FTV}(\rho_1) \cup \mathrm{FTV}(\rho_2) \\
\mathrm{FTV}(\rho_1\ ref) & = & \mathrm{FTV}(\rho_1) \\
\mathrm{FTV}(b(\rho_1, \ldots, \rho_n)) & = & \mathrm{FTV}(\rho_1) \cup \cdots \cup \mathrm{FTV}(\rho_n)
\end{array}$$

Intuitively, a weak occurrence of a type variable α indicates that α appears in the type of a value to which a reference may potentially be created. The index j in a weak occurrence $\underline{\alpha}^j$ indicates that at least j applications of the relevant expression are needed before the reference to the type containing α is *actually* created. Thus, only $\underline{\alpha}^0$ is deemed to be an *imperative occurrence* and indicates that α is imperative, *i.e.*, appears in the type of a stored value. On the other hand ordinary type variables never appear in the type of a stored value. Thus the *applicative* variables, $\mathrm{ATV}(\rho)$, *i.e.*, those variables that do not appear in the store typing can be defined by

$$\begin{array}{rcl}
\mathrm{ATV}(\alpha^j) & = & \{\alpha\} \\
\mathrm{ATV}(\underline{\alpha}^j) & = & \{\alpha\} \text{ if } j > 0 \\
\mathrm{ATV}(\underline{\alpha}^0) & = & \phi \\
\mathrm{ATV}(\rho_1 \rightarrow \rho_2) & = & \mathrm{ATV}(\rho_1) \cup \mathrm{ATV}(\rho_2) \\
\mathrm{ATV}(\rho_1\ \mathit{ref}) & = & \mathrm{ATV}(\rho_1) \\
\mathrm{ATV}(b(\rho_1, \ldots, \rho_n)) & = & \mathrm{ATV}(\rho_1) \cup \cdots \cup \mathrm{ATV}(\rho_n)
\end{array}$$

The imperative type variables are the free type variables that do not occur applicatively:

$$\mathrm{ITV}(\rho) = \mathrm{FTV}(\rho) \setminus \mathrm{ATV}(\rho)$$

We say that a pre-type ρ is imperative if all its free variables occur imperatively, *i.e.*, $\mathrm{ITV}(\rho) = \mathrm{FTV}(\rho)$.

Since the index in a weak occurrence indicates the number of applications needed before it becomes imperative, we would intuitively expect every application to decrement the strength in every occurrence by 1. We will also decrement the index in ordinary occurrences after an application. All this is formally captured by the *weakening map* W defined on indexed variables as

$$\begin{array}{rcl}
W(\alpha^j) & = & \begin{cases} \alpha^{j-1} & \text{if } j > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
W(\underline{\alpha}^j) & = & \begin{cases} \underline{\alpha}^{j-1} & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{cases}
\end{array}$$

and extended to all pre-types in the obvious way using the type constructors, $\rightarrow$, *ref*, b. If ordinary occurrences of type variables were allowed to have negative indices then we can avoid having $W(\alpha^0)$ undefined. However, we prefer the present framework as it allows us to define substitutions more elegantly.

We can now give some intuition for the indices in ordinary occurrences of a type variable. These are used to keep track of any applications and abstractions occurring in the body of an expression. The weakening map W is used in typing an application to decrease the index in every ordinary occurrence by 1. Analogously, in typing an abstraction, we will increase the indices in ordinary occurrences by 1. Thus if an there is an ordinary occurrence of a type variable α in the type ρ of an expression e, its index relative to the indices of other occurrences of α in ρ gives us information about any applications and abstractions within the body of e. This allows a more uniform treatment when a type containing weak occurrences of type variables is substituted for an ordinary occurrence of a type variable. This arises, for example, when a purely functional expression is applied to an expression using imperative features, as is further discussed in Section 5.5.

The main property guaranteed by the type system is that when a type variable α appears in the type of a stored value after evaluation of an expression e, all occurrences of α in the type of e must be imperative. This requires some restrictions on the pre-types that can be used as types of expressions. The well-formed monotypes, indicated by meta variables $\tau, \tau', \tau'', \ldots, \tau_1, \tau_2, \ldots$ are defined as follows:

- For any $\alpha \in \mathrm{TVars}, j \in \mathcal{N}$, $\alpha^j, \underline{\alpha}^j$ are monotypes.
- If τ_1, τ_2 are well-formed monotypes then $\tau_1 \rightarrow \tau_2$ is a well-formed monotype, provided
  - (i) For all $\alpha \in \mathrm{FTV}(\tau_1) \cap \mathrm{FTV}(\tau_2)$, $\alpha \in \mathrm{ITV}(\tau_1)$ iff $\alpha \in \mathrm{ITV}(\tau_2)$.
  - (ii) $W(\tau_2)$ is defined and is a well-formed monotype.
- τ_1 *ref* is a well-formed monotype, if τ_1 is a well-formed monotype.
- $b(\tau_1, \ldots, \tau_n)$ is a well-formed monotype, if $\tau_1, \ldots, \tau_n$ are well-formed monotypes.

Condition (i) on a type $\tau_1 \rightarrow \tau_2$ forces any imperative type variable of τ_1 to also occur only imperatively in τ_2 and vice-versa. When an expression of type $\tau_1 \rightarrow \tau_2$ is applied to an expression of type τ_1, the type of the resulting expression is $W(\tau_2)$, which explains Condition (ii). An example of a pre-type not satisfying condition (i) is $\underline{\alpha}^0 \rightarrow \alpha^2$, and one not satisfying condition (ii) is $\alpha^1 \rightarrow \alpha^0$. However, $\underline{\alpha}^1 \rightarrow \alpha^2, \underline{\alpha}^0 \rightarrow \beta^1, \alpha^2 \rightarrow \alpha^1$ are well-formed monotypes.

The polymorphic types, also referred to as polytypes, indicated by $\sigma, \sigma_1, \ldots$, are given by the grammar

$$\sigma \ ::= \ \tau \mid \forall\alpha.\sigma$$

where τ is a monotype and $\alpha \in$ TVars. The variables $\alpha_1, \ldots, \alpha_n$ are bound in the polymorphic type $\forall\alpha_1 \ldots \alpha_n.\ \tau$. We consider polymorphic types which differ only by a consistent renaming of bound variables equivalent. We define the set of free type variables, FTV(σ), of a polytype as FTV$(\forall\alpha_1 \ldots \alpha_n.\ \tau) =$ FTV$(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$ and similarly ITV(σ) as the set of free variables in σ that occur imperatively.

The instances of a polytype are defined using substitutions. A substitution S with domain $T \subseteq$ TVars is a map from T to the well-formed monotypes. We often use $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ to denote the substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ defined by $S(\alpha_i) = \tau_i$, for $i = 1, \ldots, n$. Intuitively, if $S(\alpha) = \tau$, then in applying the substitution S to a type, we replace all ordinary occurrences of α with index 0 (α^0) with τ, and replace other occurrences of α ($\underline{\alpha}^j$; α^i, for $i > 0$) by types determined systematically from τ. To state the types to use for other occurrences of a type variable, we will define maps $Set_i, i \geq 0$ and Inc on monotypes. We will specify the behavior of these maps on indexed variables, with the obvious extension to monotypes. The map Set_i sets all occurrences of a type variable, except those that are imperative, to a weak occurrence with strength i.

$$\begin{aligned} Set_i(\alpha^j) &= \underline{\alpha}^i \\ Set_i(\underline{\alpha}^j) &= \begin{cases} \underline{\alpha}^i & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{cases} \end{aligned}$$

The following map, Inc, increments the indices in all occurrences of type variables, except the imperative ones.

$$\begin{aligned} Inc(\alpha^j) &= \alpha^{j+1} \\ Inc(\underline{\alpha}^j) &= \begin{cases} \underline{\alpha}^{j+1} & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{cases} \end{aligned}$$

The effect of a substitution S with domain T on an indexed type variable is given by $S(\alpha^j) = Inc^j(S(\alpha))$ and $S(\underline{\alpha}^j) = Set_j(S(\alpha))$, for $\alpha \in T$, where Inc^n denotes the n-fold composition of Inc. If S has domain T, then S leaves any variable not in T unchanged. The type $S(\tau)$ and $S(\sigma)$ for monotype τ and polytype σ can then be defined in the usual way, with possible renaming of bound variables. In more detail,

$$\begin{array}{rcll}
S(\alpha^0) & = & S(\alpha) & \text{if } \alpha \in T \\
S(\alpha^j) & = & Inc^j(S(\alpha)) & \text{if } \alpha \in T, j > 0 \\
S(\underline{\alpha}^j) & = & Set_j(S(\alpha)) & \text{if } \alpha \in T \\
S(\alpha^j) & = & \alpha^j & \text{if } \alpha \notin T \\
S(\underline{\alpha}^j) & = & \underline{\alpha}^j & \text{if } \alpha \notin T \\
S(\tau_1 \to \tau_2) & = & S(\tau_1) \to S(\tau_2) & \\
S(\tau\ ref) & = & S(\tau)\ ref & \\
S(b(\tau_1, \ldots, \tau_n)) & = & b(S(\tau_1), \ldots, S(\tau_n)) & \\
S(\forall\alpha.\, \sigma) & = & \forall\beta.\, S([\beta^0/\alpha](\sigma)) & \text{where } \beta \notin T \cup \{FTV(S(\alpha)) \mid \alpha \in T\}
\end{array}$$

Note that for a monotype τ, $S(\tau)$ may not necessarily be a well-formed monotype.

We say that a well-formed monotype τ' is an instance of a polymorphic type $\sigma = \forall\alpha_1 \ldots \alpha_n.\, \tau$, written $\forall\alpha_1 \ldots \alpha_n.\, \tau \ \succeq\ \tau'$, if there exists a finite substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S\ \tau\ =\ \tau'$.

5.4.2 Typing Rules

The type system is given by a set of axioms and inference rules. The formulas that are derivable in this proof system are judgements of the form $\Gamma, \mathcal{R} \triangleright e\colon \tau$, where Γ is a variable context, $\mathcal{R}$ is a location context, e and expression, and τ a monotype. The variable context Γ states assumptions about the free variables of e and is a set of pairs $x\colon \sigma$, where σ is a polytype, and no variable x appears twice in Γ. The location context $\mathcal{R}$ gives the types of values stored in the location constants that may appear in e. It is a set of pairs $l\colon \tau$, where τ is an imperative monotype, with no location constant l appearing twice. A provable judgement $\Gamma, \mathcal{R} \triangleright e\colon \tau$ can thus be read as "the expression e has type τ in context $\Gamma, \mathcal{R}$". Since expressions that appear in ML prorams never contain location constants, we never need assumptions about location constants in typing them. However, since expressions containing location constants could be the result of an evaluation step, and our proof of soundness proceeds by showing that evaluation preserves typability, we include location contexts, $\mathcal{R}$, in our judgements. A type system for ML program expressions can be obtained by letting $\mathcal{R}$ be empty in all the axioms and inference rules.

Just as for pre-types, we will place some restrictions on valid typing judgements for expressions. Define the imperative type variables of a context to be the set of type variables that are imperative in any type appearing in the context, *i.e.*,

$$\mathrm{ITV}(\Gamma, \mathcal{R}) = \bigcup_{x:\sigma \in \Gamma} \mathrm{ITV}(\sigma) \cup \bigcup_{l:\tau \in \mathcal{R}} \mathrm{ITV}(\tau)$$

To understand the restrictions that we will place on typing judgements, consider a store s with domain $\{l_1, \ldots, l_m\}$ such that the value, $s(l_i)$, stored in location l_i has type τ_i. Let $e_1, \ldots, e_n$ be closed expressions with e_i having type σ_i. Informally, a judgement, $\{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}, \{l_1\colon \tau_1, \ldots, l_m\colon \tau_m\} \triangleright e\colon \tau$, asserts that if, starting in store s, we evaluate expressions $e_1, \ldots, e_n$ and substitute the resulting values for $x_1, \ldots, x_n$ in e, the resulting expression evaluates to a value of type τ. In this case, we want any type variable free in the type of a value in the final store to be imperative in τ. Any variable that is imperative in one of the σ_i's or τ_i's may be free in the type of a stored value after evaluation of the e_i's or in the initial store, and will therefore be free in the type of a stored value in the final store obtained after evaluation of e. We therefore say that a judgement $\Gamma, \mathcal{R} \triangleright e\colon \tau$ is a *well-formed typing judgement* iff

- τ is well-formed monotype, and
- $\mathrm{ITV}(\Gamma, \mathcal{R}) \cap \mathrm{FTV}(\tau) \subseteq \mathrm{ITV}(\tau)$.

The type system depends on a set of typings for term constructors. These are given by a function *TypeOf* defined on all term constructors f, where if $\langle f, n\rangle \in C$, then $\mathit{TypeOf}(f)$ is an expression of the form $(\tau_1, \ldots, \tau_n \Rightarrow \tau)$. We use the following notational conventions in the presentation of the type system. We write $\Gamma(x)$ (and similarly $\mathcal{R}(l)$) for the unique σ with $x\colon \sigma$ in Γ, if it exists, and $\Gamma, x\colon \sigma$ for $\Gamma \cup \{x\colon \sigma\}$, assuming $x \notin \Gamma$. For any set of type variables, T, we define the substitution Imp_T

with domain T by $Imp_T(\alpha) = \underline{\alpha}^0$. The *closure of* τ *with respect to* Γ, written $Clos_\Gamma \tau$ is the polytype $\forall \alpha_1, \ldots, \alpha_n . \tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau) \setminus \mathrm{FTV}(\Gamma)$ and $\mathrm{FTV}(\Gamma)$ is the set of all free type variables in the types appearing in Γ. The restriction on only binding applicative type variables keeps variables that might appear in the types of stored values from being used polymorphically.

$$(var) \qquad \Gamma, \mathcal{R} \triangleright x \colon Imp_T(\tau) \qquad \text{if } \Gamma(x) \succeq \tau$$

$$(loc) \qquad \Gamma, \mathcal{R} \triangleright l \colon \mathcal{R}(l)\ ref$$

$$(const) \qquad \frac{\Gamma, \mathcal{R} \triangleright e_1 \colon S\tau_1 \quad \cdots \quad \Gamma, \mathcal{R} \triangleright e_n \colon S\tau_n}{\Gamma, \mathcal{R} \triangleright f(e_1, \ldots, e_n) \colon S\tau}$$

$$\text{if} \quad TypeOf(f) = (\tau_1, \ldots, \tau_n \Rightarrow \tau) \text{ and } \mathrm{ITV}(S\tau_i) \cap \mathrm{ATV}(S\tau) = \phi$$

$$(abs) \qquad \frac{\Gamma, x \colon \tau_1, \mathcal{R} \triangleright e \colon W(\tau_2)}{\Gamma, \mathcal{R} \triangleright \lambda x.\, e \colon \tau_1 \rightarrow \tau_2}$$

$$(app) \qquad \frac{\Gamma, \mathcal{R} \triangleright e_1 \colon \tau_1 \rightarrow \tau_2 \qquad \Gamma, \mathcal{R} \triangleright e_2 \colon \tau_1}{\Gamma, \mathcal{R} \triangleright e_1\, e_2 \colon W(\tau_2)}$$

$$(let) \qquad \frac{\Gamma, \mathcal{R} \triangleright e_1 \colon \tau_1 \qquad \Gamma, x \colon Clos_\Gamma \tau_1, \mathcal{R} \triangleright e_2 \colon \tau}{\Gamma, \mathcal{R} \triangleright \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \colon \tau}$$

$$(ref) \qquad \Gamma, \mathcal{R} \triangleright \mathtt{ref} \colon \tau \rightarrow Set_1(\tau)\ ref$$

(*deref*)
$$\frac{\Gamma, \mathcal{R} \triangleright e\colon \tau\ \mathit{ref}}{\Gamma, \mathcal{R} \triangleright !e\colon \tau}$$

(*assign*)
$$\frac{\Gamma, \mathcal{R} \triangleright e_1\colon \tau\ \mathit{ref} \qquad \Gamma, \mathcal{R} \triangleright e_2\colon \tau}{\Gamma, \mathcal{R} \triangleright e_1 := e_2\colon \tau}$$

Note that in the rule (*var*), T can be any arbitrary set of type variables. An implicit side-condition on all the axioms and typing rules is that the type judgement in the consequent is a well-formed typing judgement. This condition is non-trivial only for (*var*), (*const*), (*ref*), and (*abs*). In other words, for all the remaining rules, if the premises are well-formed typing judgements, so is the consequent.

The typing rule for `ref` enforces the restriction that any value placed in the store must have a type in which all type variables are imperative. Thus, while `ref` can be applied to an expression of unrestricted type τ, the resulting reference cell would have an imperative type, since $W(Set_1(\tau))$ is an imperative type for any type τ. Since applicative variables do not appear in the type of a stored value, the (*let*) rule generalizes all applicative type variables that are not free in the context. Further, by our restriction on well-formed typing judgements, these applicative variables also cannot occur free in $\mathcal{R}$, since all the free variables of $\mathcal{R}$ are imperative.

To understand the (*abs*) rule, we can give an alternative but equivalent formulation. For any set T of type variables, we define the map $Up_{\setminus T}$ on monotypes, increasing the strength of all variables except imperative occurrences of variables from T, by

$$\begin{aligned} Up_{\setminus T}(\underline{\alpha}^0) &= \underline{\alpha}^0 && \text{if } \alpha \in T \\ Up_{\setminus T}(\underline{\beta}^i) &= \underline{\beta}^{i+1} && \text{if } \beta \notin T \text{ or } i > 0 \\ Up_{\setminus T}(\alpha^i) &= \alpha^{i+1} \end{aligned}$$

Then the (*abs*) rule, together with the side-condition that the resulting judgement is well-formed, is equivalent to the rule

$$\frac{\Gamma, x\colon \tau_1, \mathcal{R} \triangleright e\colon \tau_2}{\Gamma, \mathcal{R} \triangleright \lambda x.\, e\colon \tau_1 \to Up_{\setminus T}(\tau_2)}$$

with $T \supseteq \mathrm{ITV}(\Gamma, x\colon \tau_1, \mathcal{R})$, $T \subseteq \mathrm{ATV}(\tau_1)$ and for any $\alpha \in \mathrm{ITV}(\Gamma, \mathcal{R}) \cap \mathrm{FTV}(\tau_1), \alpha \in \mathrm{ITV}(\tau_1)$. Focusing only on the condition that $T \supseteq \mathrm{ITV}(\Gamma, x\colon \tau_1, \mathcal{R})$, this rule has the effect of permitting any of the imperative variables of τ_2, except those occurring imperatively in $\Gamma, x\colon \tau_1, \mathcal{R}$, to be made applicative. The variables that are imperative in τ_2 but not in $\Gamma, x\colon \tau_1, \mathcal{R}$ are those that appear in a stored value as a result of the evaluation of e. Since the evaluation of $\lambda x.\, e$ does not evaluate e, the references that e creates would not result from the evaluation of $\lambda x.\, e$ and hence can be safely made applicative.

Since stores are defined using values, which are syntactic expressions, we can use the type system to define store typings for stores. We say that a store s has store typing $\mathcal{R}$, written $s\colon \mathcal{R}$, if $\mathrm{Dom}(s) = \mathrm{Dom}(\mathcal{R})$ and for all $l \in \mathrm{Dom}(s)$ we can derive the judgement $\phi, \mathcal{R} \triangleright s(l)\colon \mathcal{R}(l)$.

5.5 Relation to SML-NJ

In the absence of a simple presentation of the algorithm used by the New Jersey implementation of Standard ML, it is difficult to prove that all the expressions typable in SML-NJ are typable in our type system. Such a result coupled with our soundness proof would establish the soundness of SML-NJ. However, by allowing mixed "ordinary" and "weak" occurrences of a type variable and associating indices with "ordinary" occurrences as well, and a more careful accounting of the strengths during application and abstraction we are able to type-check many expressions which SML-NJ rejects. In this section, we give examples of such expressions. Along with demonstrating the expressive power of our type system, these examples help understand how the type system works.

In [LW91], Leroy and Weis give examples of various benchmark expressions to type-check. Their expressions fall into two categories. One class of expressions is those which use references only locally and which are given the same types in their type system as the corresponding applicative expressions. The other class of expressions are those which exercise the compatibility between functions that create mutable data and higher-order functions, all of which are rejected by SML-NJ. Leroy and Weis take this to be strong evidence that SML-NJ does not handle full functionality correctly. However, our type system accepts such examples of expressions given in [LW91, TJ92b]. One example is the expression $(\lambda x.\, x)\,\texttt{ref}$. With our abstraction and application rules it can be seen that $\phi \triangleright \lambda x.\, x\colon (\alpha^0 \rightarrow \underline{\alpha}^1\ \mathit{ref}) \rightarrow (\alpha^1 \rightarrow \underline{\alpha}^2\ \mathit{ref})$ and hence $\phi \triangleright (\lambda x.\, x)\,\texttt{ref}\colon \alpha^0 \rightarrow \underline{\alpha}^1\ \mathit{ref}$ which is the same type as $\texttt{ref}$. However, SML-NJ rejects this expression. A more complicated example in the same vein is $\mathit{map}\,\texttt{ref}$, where map is the familiar functional that when applied to a function f returns a function that maps any list l to the list resulting from applying f to each element of the list l. While $\mathit{map}\,\texttt{ref}$ is rejected by SML-NJ, in our type system it can be given the type $\mathit{list}(\alpha^0) \rightarrow \mathit{list}(\underline{\alpha}^1\ \mathit{ref})$, where list is a unary type-constructor representing the type of lists. Thus our type system demonstrates that weak polymorphism, if formulated carefully, can indeed work well when applicative and imperative functions are mingled.

A concept in our system that is different from SML-NJ is the association of indices with ordinary type variables, which allows us to infer any applications in the body of expressions, including purely functional ones. Thus $\lambda x.\, x$ has the type $\alpha^0 \rightarrow \alpha^1$, while $\lambda x.\, x\,\texttt{nil}$ has the type $(\mathit{list}(\alpha^0) \rightarrow \beta^1) \rightarrow \beta^1$. The increase in the index of α in the result type of $\lambda x.\, x$ relative to its index in the argument type indicates the lack of any applications, while the index of β remaining the same in both argument and result types of $\lambda x.\, x\,\texttt{nil}$ indicates the application inside its body. (Since the occurrences of β are all ordinary, we also have $\lambda x.\, x\,\texttt{nil}\colon (\mathit{list}(\alpha^0) \rightarrow \beta^6) \rightarrow \beta^6$, for example.) On the other hand SML-NJ gives these expressions the types $\alpha \rightarrow \alpha$ and $(\alpha\ \mathit{list} \rightarrow \beta) \rightarrow \beta$, with α and β implicitly considered to be of strength ∞. However, since such a type gives no information about applications inside the body, SML-NJ is forced to be conservative when such expressions are applied to say, $\texttt{ref}$. Hence, SML-NJ rejects $(\lambda x.\, x\,\texttt{nil})\,\texttt{ref}$ correctly, but is also forced to reject $(\lambda x.\, x)\,\texttt{ref}$. Using indices on ordinary type variables, we correctly deal with both these expressions as shown below:

$$(\lambda x.\, x\, \mathtt{nil}) : \forall\alpha\beta.(list(\alpha^0) \to \beta^1) \to \beta^1 \;\succeq\; (list(\alpha^0) \to list(\underline{\alpha}^1)\, ref) \to list(\underline{\alpha}^1)\, ref$$

Hence, $(\lambda x.\, x\, \mathtt{nil})\, \mathtt{ref} : list(\underline{\alpha}^0)\, ref$.
On the other hand,

$$(\lambda x.\, x) : \forall\alpha.\alpha^0 \to \alpha^1 \;\succeq\; (\alpha^0 \to \underline{\alpha}^1\, ref) \to (\alpha^1 \to \underline{\alpha}^2\, ref)$$

and thus, $(\lambda x.\, x)\, \mathtt{ref} : \alpha^0 \to \underline{\alpha}^1\, ref$.

5.6 Soundness

In this section, we prove the soundness of the type system. The main lemma is a weaker form of the subject reduction property which shows the preservation of typability, for closed expressions, under evaluation. Since `wrong`, the run-time error, is not typable and only closed expressions are compiled, this is sufficient to prove the correctness of our type system. Since store typings were defined in Section 5.4.2 using derivability in the type system, rather than as maximal fixed points, we do not have to use co-induction in our proofs. This makes our soundness proof simpler than the one given in [Tof90].

We begin by proving some syntactic properties of the type system. In the following we use $\vdash \Gamma, \mathcal{R} \triangleright e\colon \tau$ to indicate that the judgement $\Gamma, \mathcal{R} \triangleright e\colon \tau$ is derivable. Our first lemma is a generalization of the standard property of closure of typing derivations under substitution. It shows that by substituting for variables in a provable typing judgement, adding assumptions to the context, and making any number of variables in the type of the expression imperative, we still obtain provable typing judgements. To make the lemma easier to state, we first define the notion of an "instance" of a typing judgement.

Definition 5.6.1 Let $\Gamma, \mathcal{R} \triangleright e\colon \tau$ and $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ be well-formed typing judgements.

We say that $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ is an **instance** of $\Gamma, \mathcal{R} \triangleright e\colon \tau$ if there is a substitution S such that $\forall x \in \mathrm{Dom}(\Gamma)$, $\Gamma'(x) = Imp_{T_x}(S\,\Gamma(x))$ for some set T_x, $\forall l \in \mathrm{Dom}(\mathcal{R})$, $\mathcal{R}'(l) = S\,\mathcal{R}(l)$, and $\tau' = Imp_T(S\,\tau)$ for some set T.

Lemma 5.6.2 *If* $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ *is an instance of* $\Gamma, \mathcal{R} \triangleright e\colon \tau$ *and* $\vdash \Gamma, \mathcal{R} \triangleright e\colon \tau$*, then* $\vdash \Gamma', \mathcal{R}' \triangleright e\colon \tau'$*.*

Proof The proof proceeds by induction on the proof of $\Gamma, \mathcal{R} \triangleright e\colon \tau$ and case analysis on the last axiom or proof rule used. We describe the representative cases of (*var*) and (*abs*). The latter case is particularly illustrative since it uses the restrictions on well-formed types and well-formed typing judgements. In each case, assume that $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ is an instance of $\Gamma, \mathcal{R} \triangleright e\colon \tau$ via substitution S and sets T_x, for each $x \in \mathrm{Dom}(\Gamma)$, and set T.

(*var*) The axiom has the form

$$\Gamma, \mathcal{R} \triangleright x\colon Imp_{T_1}(\tau_1)$$

with $\tau = Imp_{T_1}(\tau_1)$. Let $\Gamma(x) = \forall\alpha_1 \ldots \alpha_n.\ \rho$ and let S_1 be the substitution with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $\tau_1 = S_1\,\rho$. We need to find substitution S' and set T', so that τ' is the result of making the variables of T' imperative in an instance of $\Gamma'(x)$ via substitution S'. Let $\Gamma'(x) = \forall\beta_1 \ldots \beta_n.\ \rho'$. Consider the substitution S' with domain $\{\beta_1, \ldots, \beta_n\}$ given by $S'(\beta_i) = S(S_1(\alpha_i))$. Then, $\tau' = Imp_{\mathrm{ITV}(\tau')}(S'\,\Gamma'(x))$ and hence $\Gamma', \mathcal{R} \triangleright x\colon \tau'$ is a valid application of the (*var*) axiom.

(*abs*) The last step is of the form

$$\frac{\Gamma, x\colon \tau_1, \mathcal{R} \triangleright e'\colon W(\tau_2)}{\Gamma, \mathcal{R} \triangleright \lambda x.\, e'\colon \tau_1 \to \tau_2}$$

with $\tau = \tau_1 \to \tau_2$. Consider $\Gamma'' = \Gamma', x\colon Imp_T(S\,\tau_1), \tau'' = W(Imp_T(S\,\tau_2))$. We prove that $\vdash \Gamma'', \mathcal{R}' \triangleright e'\colon \tau''$, from which it follows using the (*abs*) rule that $\vdash \Gamma', \mathcal{R}' \triangleright \lambda x.\, e'\colon \tau'$. We first see that $\Gamma'', \mathcal{R}' \triangleright e'\colon \tau''$ is a well-formed judgement. Since, $\tau' = Imp_T(S\,\tau_1) \to Imp_T(S\,\tau_2)$ is a well-formed type expression, we must have that τ'' is a well-formed type expression. Now, suppose that $\alpha \in \mathrm{ITV}(\Gamma'', \mathcal{R}') \cap \mathrm{FTV}(\tau'')$. If $\alpha \in \mathrm{ITV}(\Gamma', \mathcal{R}')$ then since $\Gamma', \mathcal{R}' \triangleright e' : \tau'$ is a well-formed typing judgment we have that $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_2))$ and hence $\alpha \in \mathrm{ITV}(\tau'')$. And, if $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_1))$ then since τ' is a well-formed type expression we get that $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_2))$ and hence $\alpha \in \mathrm{ITV}(\tau'')$. We next observe that $\tau'' = Imp_T(S(W(\tau_2)))$, since for any substitution S, $S(W(\tau)) = W(S(\tau))$ if $W(\tau)$ is defined. Hence using substitution S, $T_x = T$, we can obtain $\Gamma'', \mathcal{R}' \triangleright e' : \tau''$ as an instance of $\Gamma, x{:}\,\tau_1, \mathcal{R} \triangleright e' : W(\tau_1)$ which is therefore derivable by the induction hypothesis.

■

As corollaries of Lemma 5.6.2, we obtain statements describing the behavior of derivable type judgements under substitutions, and adding variables and locations to the context. In our type system adding assumptions to the context may have the effect of forcing some additional variables in the type of the expression to be imperative.

Corollary 5.6.3 (Type Substitution) *Suppose* $\vdash \Gamma, \mathcal{R} \triangleright e : \tau$. *Let S be any substitution and T any set of type variables such that* $S\Gamma, S\mathcal{R} \triangleright e : Imp_T(S\,\tau)$ *is a well-formed typing judgement. Then,* $\vdash S\Gamma, S\mathcal{R} \triangleright e : Imp_T(S\,\tau)$.

Corollary 5.6.4 (Adding Variables and Locations) *Suppose* $\vdash \Gamma, \mathcal{R} \triangleright e : \tau$. *Let* $\Gamma' \supseteq \Gamma, \mathcal{R}' \supseteq \mathcal{R}$. *Then* $\vdash \Gamma', \mathcal{R}' \triangleright e : Imp_{ITV(\Gamma', \mathcal{R}')}\,\tau$.

Proof For any well-formed monotype τ and any set of type variables T, $Imp_T(\tau)$ is still a well-formed monotype. ■

We now establish a term-substitution property which is crucial to proving the subject reduction property for β-reduction and the reduction of `let`-expressions. The property we are interested in is Corollary 5.6.6 which is not provable directly by induction. We therefore prove a stronger statement in the following lemma. The statement of the lemma also provides formal justification for allowing type variables to be made imperative in the (*var*) axiom.

Lemma 5.6.5 *Assume that* $\vdash \Gamma, x{:}\, \forall \alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \rhd e{:}\, \tau'$ *and*

$$\{\alpha_1, \ldots, \alpha_n\} \cap FTV(\Gamma, \mathcal{R}) = \emptyset$$

If $\vdash \Gamma, \mathcal{R} \rhd v{:}\, Imp_{ITV(\Gamma,\mathcal{R})}\, \tau$ *then* $\vdash \Gamma, \mathcal{R} \rhd [v/x]e{:}\, \tau'$*.*

Proof By induction on the structure of e. We show the cases when e is a variable, λ-abstraction, and `let`-expression — the last two cases illustrate our choice of induction hypothesis.

Case $e = x'$. If $x' \neq x$ then the lemma is trivially true. Otherwise, there is a substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $\tau' = Imp_T(S\tau)$. By the definition of well-formed typing judgements, we can assume that $T \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$. Since $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma, \mathcal{R}) = \emptyset$ we have that $S\Gamma, S\mathcal{R} = \Gamma, \mathcal{R}$. Thus, applying Lemma 5.6.2 to $\Gamma, \mathcal{R} \rhd v{:}\, Imp_{\mathrm{ITV}(\Gamma,\mathcal{R})}\, \tau$, we can derive the judgment

$$\Gamma, \mathcal{R} \rhd v{:}\, Imp_T(S(Imp_{\mathrm{ITV}(\Gamma,\mathcal{R})}\, \tau))$$

And, since $T \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$, $Imp_T(S(Imp_{\mathrm{ITV}(\Gamma,\mathcal{R})}\, \tau)) = \tau'$.
Hence $\vdash \Gamma, \mathcal{R} \rhd [v/x]x{:}\, \tau'$.

Case $e = \lambda x'.\, e_1$. Then,

$$\frac{\Gamma, x{:}\, \forall \alpha_1 \ldots \alpha_n.\tau, x'{:}\, \tau_1, \mathcal{R} \rhd e_1{:}\, W(\tau_2)}{\Gamma, x{:}\, \forall \alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \rhd \lambda x'.\, e_1{:}\, \tau_1 \to \tau_2}$$

Let $\{\beta_1, \ldots, \beta_n\}$ be distinct variables such that $\beta_i \notin \mathrm{FTV}(\Gamma, \mathcal{R}) \cup \{\alpha_1, \ldots, \alpha_n\}$. Let S be the substitution defined by $S(\alpha_i) = \beta_i^0$, $\Gamma' = \Gamma, x'{:}\, S\tau_1$. By the definition of well-formed typing judgements and the nature of S, all imperative variables of $\Gamma', x{:}\, \forall \alpha_1 \ldots, \alpha_n.\tau, \mathcal{R}$ can only appear imperatively in $S(W(\tau_2))$. Hence, using Lemma 5.6.2, and substitution S on the premise of the *(abs)* rule, we can derive

$$\Gamma', x{:}\, \forall \alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \rhd e_1{:}\, S(W(\tau_2))$$

since $S\Gamma, S\mathcal{R} = \Gamma, \mathcal{R}$. By the choice of S, $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma', \mathcal{R}) = \emptyset$. We have that $\vdash \Gamma, \mathcal{R} \triangleright v\colon \tau''$, where $\tau'' = \mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau$. Using Corollary 5.6.4, $\vdash \Gamma', \mathcal{R} \triangleright v\colon \mathit{Imp}_{\mathrm{ITV}(\Gamma',\mathcal{R})}\,\tau''$. Since $\mathrm{ITV}(\Gamma', \mathcal{R}) \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$, we thus have that $\vdash \Gamma', \mathcal{R} \triangleright v\colon \mathit{Imp}_{\mathrm{ITV}(\Gamma',\mathcal{R})}\,\tau$. Hence, by induction hypothesis, we can derive

$$\Gamma', \mathcal{R} \triangleright [v/x]e_1\colon S(W(\tau_2))$$

Let $S'(\beta_i) = \alpha_i^0$. Using S' with Corollary 5.6.4, we can derive

$$\Gamma, x'\colon \tau_1, \mathcal{R} \triangleright [v/x]e_1\colon W(\tau_2)$$

Now, using the (*abs*) rule, we can get the desired judgement.

Case $e = \texttt{let}\ \ x' = e_1\ \ \texttt{in}\ \ e_2$. Then,

$$\frac{\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e_1\colon \tau_1 \quad \Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, x'\colon \mathit{Clos}_{\Gamma, x:\forall\alpha_1 \ldots \alpha_n.\tau}\tau_1, \mathcal{R} \triangleright e_2\colon \tau'}{\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright \texttt{let}\ \ x' = e_1\ \ \texttt{in}\ \ e_2\colon \tau'}$$

Let $\{\alpha_{i_1}, \ldots, \alpha_{i_m}\} = \mathrm{ITV}(\tau_1) \cap \{\alpha_1, \ldots, \alpha_n\}$, with $m \geq 0, i_1 < \cdots < i_m$. Let $\beta_1, \ldots, \beta_m$ be distinct variables such that $\beta_i \notin \mathrm{FTV}(\Gamma, \mathcal{R}) \cup \mathrm{FTV}(\tau_1) \cup \{\alpha_1, \ldots, \alpha_n\}$ and let S be the substitution defined by $S(\alpha_{i_j}) = \beta_j^0$. Using Lemma 5.6.2, and substitution S on the left premise of the (*let*) rule, we can derive $\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e_1\colon S(\tau_1)$, since $S\Gamma, S\mathcal{R} = \Gamma, \mathcal{R}$. Thus, by induction hypothesis applied to e_1, we have that

$$\vdash \Gamma, \mathcal{R} \triangleright [v/x]e_1\colon S(\tau_1) \tag{5.1}$$

Since $\beta_i \notin \mathrm{FTV}(\tau_1)$ and since the α_{i_j}'s occur imperatively in τ_1, it follows that

$$S(\mathit{Clos}_{\Gamma, x:\forall\alpha_1 \ldots \alpha_n.\tau}\tau_1) = \mathit{Clos}_{\Gamma, x:\forall\alpha_1 \ldots \alpha_n.\tau}(S\,\tau_1)$$

Thus, applying substitution S to the second premise of the (*let*), we can derive $\Gamma', x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e_2\colon S\tau'$, where $\Gamma' = \Gamma, x'\colon \mathit{Clos}_{\Gamma, x:\forall\alpha_1 \ldots \alpha_n.\tau}(S\tau_1)$. We have that $\vdash \Gamma, \mathcal{R} \triangleright v{:}\, \tau''$, where $\tau'' = \mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\tau$. By an argument, similar to the λ-abstraction case, we can show that $\vdash \Gamma', \mathcal{R} \triangleright v{:}\, \mathit{Imp}_{\mathrm{ITV}(\Gamma',\mathcal{R})}\tau$. Thus, by induction hypothesis applied to e_2, we can derive $\Gamma', \mathcal{R} \triangleright [v/x]e_2{:}\, S\tau'$. By Proposition 5.6.7, we have that

$$\vdash \Gamma, x'{:}\, \mathit{Clos}_{\Gamma} S\tau_1, \mathcal{R} \triangleright [v/x]e_2{:}\, S\tau' \qquad (5.2)$$

From (5.1), (5.2), using the (*let*) rule, we get

$$\vdash \Gamma, \mathcal{R} \triangleright [v/x]e{:}\, S\tau'$$

Now applying the substitution $S'(\beta_j) = \alpha^0_{i_j}$, we get the desired judgement.

■

Corollary 5.6.6 (Term Substitution) *If* $\vdash \Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e{:}\, \tau'$ *and* $\vdash \Gamma, \mathcal{R} \triangleright v{:}\, \tau$ *with* $\{\alpha_1, \ldots, \alpha_n\} \cap FTV(\Gamma) = \emptyset$ *then* $\vdash \Gamma, \mathcal{R} \triangleright [v/x]e{:}\, \tau'$.

Proof We have that $\Gamma, \mathcal{R} \triangleright v{:}\, \tau$ is derivable. By the restriction on well-formed typing judgements, $\mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\, \tau = \tau$. Hence, by Lemma 5.6.5, we have the desired result.
■

Proposition 5.6.7 *If* $\vdash \Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e{:}\, \tau'$ *then* $\vdash \Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n \ldots \alpha_{n+m}.\tau, \mathcal{R} \triangleright e{:}\, \tau'$, *for any* $m \geq 0$ *and variables* $\alpha_{n+1}, \ldots, \alpha_{n+m}$.

Proof A straightforward induction on the structure of e. The only important cases are when e is the variable x or a `let`-expression, which we show below.

Case $e = x$. Then $\tau' = \mathit{Imp}_T(S\tau)$ for some substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$. Consider the substitution S' with $S'(\alpha_{n+i}) = \alpha^0_{n+i}$ for $i \leq m$ and $S'(\alpha_i) = S(\alpha_i)$ for $i \leq n$. Clearly, $S'\tau = S\tau$ and we get the desired conclusion using (*var*) with substitution S' and set T.

Case $e = \texttt{let}\ \ x' = e_1\ \ \texttt{in}\ \ e_2$. Then, we have

$$\frac{\Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e_1{:}\, \tau_1 \quad \Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n.\tau, x'{:}\, \mathit{Clos}_{\Gamma, x{:}\forall\alpha_1\ldots\alpha_n.\tau}\tau_1, \mathcal{R} \triangleright e_2{:}\, \tau'}{\Gamma, x{:}\, \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright \texttt{let}\ \ x' = e_1\ \ \texttt{in}\ \ e_2{:}\, \tau'}$$

By induction hypothesis applied to the left premise, we get

$$\vdash \Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n.\ldots \alpha_{n+m}.\tau, \mathcal{R} \triangleright e_1{:}\,\tau_1 \tag{5.3}$$

By induction hypothesis applied to the second premise, we get

$$\Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n \ldots \alpha_{n+m}.\tau, x'{:}\,Clos_{\Gamma, x:\forall\alpha_1\ldots\alpha_n.\tau}\tau_1, \mathcal{R} \triangleright e_2{:}\,\tau'$$

Since $\mathrm{FTV}(\Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n \ldots \alpha_{n+m}.\tau) \subseteq \mathrm{FTV}(\Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n.\tau)$, we can apply the induction hypothesis to e_2 again to obtain

$$\vdash \Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n.\tau, x'{:}\,Clos_{\Gamma, x:\forall\alpha_1\ldots\alpha_n\ldots\alpha_{n+m}.\tau}\tau_1, \mathcal{R} \triangleright e_2{:}\,\tau' \tag{5.4}$$

Applying the (*let*) rule to (5.3), (5.4) we get the desired conclusion.

■

Using the above lemmas, we are now ready to connect the type system with the operational semantics. To establish the correspondence with the semantics, we need to assume some properties of the typings for the term constructors, since our proof system depends on them. The following definition states what we require of the typings for the term constructors.

Definition 5.6.8 The typing for term-constructors given by *TypeOf* is **admissible** if the resulting proof system for deriving typing judgments has the following properties:

(i) For all values $v \in V$ (*i.e.*, other than λ-abstractions, `ref`, and location constants), if $\vdash \phi, \mathcal{R} \triangleright v{:}\,\tau$ for any store typing $\mathcal{R}$ and type τ, then τ is of the form $b(\tau_1, \ldots, \tau_n)$ for some type constructor b.

(ii) Let $\langle e_1, e_2\rangle \in F$. For any $\mathcal{R}, \tau$, if $\vdash \phi, \mathcal{R} \triangleright e_1{:}\,\tau$ then $\vdash \phi, \mathcal{R} \triangleright e_2{:}\,\tau$.

It is a consequence of (i) that the only functional values are λ-abstractions and `ref`, and the only values that are reference cells are location constants. (Indeed, we could have eliminated `ref` from being a value as well, by having the redex `ref` $\leadsto$ $\lambda x.$ `ref` x). Condition (ii) requires the typings for term constructors to be consistent with their operational properties as given by their functional reductions. We assume in the following lemmas that the given typings for term-constructors is admissible.

Our operational semantics was described by defining the notions of redexes, evaluation contexts, and evaluation steps, in that order. The presentation of the lemmas for establishing the soundness of our type system follows the same order. First, we prove subject reduction for the purely functional reductions.

Lemma 5.6.9 (Subject Reduction for Functional Reductions) *If* $\vdash \phi, \mathcal{R} \triangleright e_1 \colon \tau$ *and* $e_1 \leadsto e_2$ *then* $\vdash \phi, \mathcal{R} \triangleright e_2 \colon \tau$.

Proof By case analysis on $e_1 \leadsto e_2$.

Case $(\lambda x.\, e)\ v \leadsto [v/x]e$. Assume $\vdash \phi, \mathcal{R} \triangleright (\lambda x.\, e)\ v \colon \tau$. Then $\vdash \phi, \mathcal{R} \triangleright \lambda x.\, e \colon \tau_1 \rightarrow \tau_2, \vdash \phi, \mathcal{R} \triangleright v \colon \tau_1$ and $\tau = W(\tau_2)$. From $\vdash \phi, \mathcal{R} \triangleright \lambda x.\, e \colon \tau_1 \rightarrow \tau_2$, we have $\vdash x \colon \tau_1, \mathcal{R} \triangleright e \colon W(\tau_2)$. Using Corollary 5.6.6 we get $\vdash \phi, \mathcal{R} \triangleright [v/x]e \colon \tau$.

Case $(\texttt{let}\ \ x = v\ \ \texttt{in}\ \ e) \leadsto [v/x]e$. Assume $\vdash \phi, \mathcal{R} \triangleright \texttt{let}\ \ x = v\ \ \texttt{in}\ \ e \colon \tau$. Then

$$\vdash \phi, \mathcal{R} \triangleright v \colon \tau_1, \vdash x \colon \mathit{Clos}_{\phi}\tau_1, \mathcal{R} \triangleright e \colon \tau$$

Since $\phi, \mathcal{R} \triangleright v \colon \tau_1$ is a well-formed typing judgement, if $\alpha \in \mathrm{ATV}(\tau_1)$ then $\alpha \notin \mathrm{FTV}(\mathcal{R})$. Hence, using Corollary 5.6.6 we get $\vdash \phi, \mathcal{R} \triangleright [v/x]e \colon \tau$.

Case $v_1\ v_2 \leadsto \texttt{wrong}$. Since v_1 is not a λ-abstraction or `ref`, it must be a location constant or a value in V. The former can only have a type of the form $\tau'\ \mathit{ref}$ and the latter can only have a type of the form $b(\tau_1, \ldots, \tau_n)$ by condition (i) of admissibility. Hence, $\not\vdash \phi, \mathcal{R} \triangleright v_1 \colon \tau_1 \rightarrow \tau_2$ for any $\mathcal{R}, \tau_1, \tau_2$. Hence $\not\vdash \phi, \mathcal{R} \triangleright v_1\ v_2 \colon \tau$ for any $\mathcal{R}, \tau$ and the statement is true vacuously.

Otherwise $e_1 \leadsto e_2$ is a redex involving term-constructors and the statement is true by condition (ii) of admissibility. ■

We might expect a proof of soundness to proceed by extending Lemma 5.6.9 to establish subject reduction for evaluation. In more detail, suppose that $(e, s) \longmapsto (e', s')$. We would like to establish that if e has type τ, then so does e'. Note that because of the possibility of creation of reference cells, e' may mention location constants that do not appear in e. For this reason, the typing of e' must involve assumptions about additional location constants. By our restriction on typing judgements, e' can then have type τ only if the type variables in the types of the new location constants appearing in e' all appear imperatively in τ. Unfortunately this condition on type variables may not be true, since there is nothing in our type system to force "new" type variables to be used whenever possible. As a result, an applicative type variable in τ could "accidentally" be the same as one in the type of a new reference cell created. However, as (iii) of the following lemma shows, such applicative variables in τ can be suitably renamed so that e' can be given the renamed type. The form of the renaming is the crucial formal justification for allowing all applicative variables to be treated polymorphically in the (let) rule. It is also in the proof of (iii) that we see the formal origin of our restrictions on the syntactic form of types and typing judgements. Parts (i) and (ii) express obvious properties, but have been stated explicitly merely for ease of reference in the proof of Lemma 5.6.11.

Lemma 5.6.10 *Let $E[\,]$ be any evaluation context and suppose that $\vdash \phi, \mathcal{R} \triangleright E[e]\colon \tau$. Then,*

(i) $\vdash \phi, \mathcal{R} \triangleright e\colon \tau_1$ *for some type* τ_1.

(ii) *If* $\vdash \phi, \mathcal{R} \triangleright e'\colon \tau_1$, *then* $\vdash \phi, \mathcal{R} \triangleright E[e']\colon \tau$.

(iii) *Suppose that for some $\mathcal{R}' \supseteq \mathcal{R}$ and some expression e', we have that $\vdash \phi, \mathcal{R}' \triangleright e'\colon \tau_1$. Let $\{\alpha_1, \ldots, \alpha_n\} = ATV(\tau)$. Then for some distinct variables $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin FTV(\mathcal{R}')$ we have that $\vdash \phi, \mathcal{R}' \triangleright E[e']\colon S_E\tau$ where $S_E = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$.*

Proof By induction on the structure of $E[\,]$. Statements (i) and (ii) can be proved trivially, so we show details only for (iii). Note that for any distinct $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}')$ we have that $\phi, \mathcal{R} \triangleright E[e']\colon [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]\tau$ is a well-formed judgement since $\mathrm{ATV}([\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]\tau) = \{\beta_1, \ldots, \beta_n\}$.

- $E[\,] \equiv [\,]$. Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau_1)$. Choose any $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}')$. Since $\phi, \mathcal{R}' \triangleright e'\colon \tau_1$ is a well-formed judgement and $\mathrm{FTV}(\mathcal{R}') =$

ITV($\mathcal{R}'$) we have that $\alpha_i \notin$ FTV($\mathcal{R}'$). Hence for $S_{[\,]} = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$, $S_{[\,]}\mathcal{R}' = \mathcal{R}'$. Hence using Lemma 5.6.2, $\vdash \phi, \mathcal{R}' \triangleright e' \colon S_{[\,]}\tau_1$.

- $E[\,] \equiv E_1\, e_2$. Then we can derive

$$\phi, \mathcal{R} \triangleright E_1[e] \colon \tau' \rightarrow \tau_2 \tag{5.5}$$

$$\phi, \mathcal{R} \triangleright e_2 \colon \tau' \tag{5.6}$$

and $\tau = W(\tau_2)$. Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau)$. Clearly $\mathrm{ITV}(\tau_2) \subseteq \mathrm{ITV}(\tau)$. Hence $\mathrm{ATV}(\tau) \subseteq \mathrm{ATV}(\tau_2) \subseteq \mathrm{ATV}(\tau' \rightarrow \tau_2)$. Thus,

$$\mathrm{ATV}(\tau' \rightarrow \tau_2) = \{\alpha_1, \ldots, \alpha_n, \alpha_{n+1}, \ldots, \alpha_p\}$$

for some variables $\alpha_{n+1}, \ldots, \alpha_p$. By inductive hypothesis about $E_1[\,]$,

$$\vdash \phi, \mathcal{R}' \triangleright E_1[e'] \colon S_{E_1}(\tau') \rightarrow S_{E_1}(\tau_2)$$

where $S_{E_1}(\alpha_i) = \beta_i^0, 1 \leq i \leq p$ for some variables $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin$ FTV($\mathcal{R}'$). Since (5.5) is a well-formed judgement, $\mathrm{Dom}(S_{E_1}) \cap \mathrm{FTV}(\mathcal{R}') = \phi$. Hence $S_{E_1}\mathcal{R} = \mathcal{R}$. Note that $\phi, \mathcal{R}' \triangleright e_2 \colon S_{E_1}\tau'$ is a well-formed judgement since $\mathrm{ATV}(S_{E_1}\tau') \subseteq \{\beta_1, \ldots, \beta_p\}$ and $\{\beta_1, \ldots, \beta_p\} \cap \mathrm{FTV}(\mathcal{R}') = \phi$. Hence using Lemma 5.6.2 on (5.6) with substitution S_{E_1}, we get that $\vdash \phi, \mathcal{R}' \triangleright e_2 \colon S_{E_1}\tau'$. Thus using (*app*), $\vdash \phi, \mathcal{R}' \triangleright E[e'] \colon S_{E_1}(W(\tau_2))$, *i.e.*,

$$\vdash \phi, \mathcal{R}' \triangleright E[e'] \colon S_{E_1}(W(\tau_2)) \tag{5.7}$$

since $W(S_{E_1}\tau_2) = S_{E_1}(W(\tau_2))$, as $W\tau_2$ is defined.
Let $S' = [\alpha_{n+1}^0/\beta_{n+1}, \ldots, \alpha_p^0/\beta_p]$. Take $S_E = S' \circ S_{E_1}$. Since $\mathrm{Dom}(S') \cap \mathcal{R}' = \phi$, using substitution S' and Corollary 5.6.3 on (5.7) we get $\vdash \phi, \mathcal{R}' \triangleright E[e'] \colon S_E(\tau)$.

- $E[\,] \equiv v\ E_1$. Then we can derive

$$\phi, \mathcal{R} \triangleright v\colon \tau' \rightarrow \tau_2$$
$$\phi, \mathcal{R} \triangleright E_1[e]\colon \tau'$$

and $\tau = W(\tau_2)$. Let $\mathrm{ATV}(\tau') = \{\alpha_1, \ldots, \alpha_n\}$. By inductive hypothesis about E_1, we have variables $\beta_1, \ldots, \beta_n \notin \mathrm{FTV}(\mathcal{R}')$ such that

$$\phi, \mathcal{R}' \triangleright E_1[e']\colon S_{E_1}(\tau')$$

with $S_{E_1} = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$. Since $\mathrm{ATV}(\tau') \subseteq \mathrm{ATV}(\tau' \rightarrow \tau_2)$, we have

$$\mathrm{ATV}(\tau' \rightarrow \tau_2) = \{\alpha_1, \ldots, \alpha_n, \alpha_{n+1}, \ldots, \alpha_p\}$$

for some variables $\alpha_{n+1}, \ldots, \alpha_p$. Choose any distinct variables $\beta_{n+1}, \ldots, \beta_p$ such that $\beta_j \notin \mathrm{FTV}(\mathcal{R}'), n+1 \leq j \leq p$. Consider $S' = [\beta_1^0/\alpha_1, \ldots, \beta_p^0/\alpha_p]$.

We claim that $S'\tau' = S_{E_1}\tau'$. For consider any $\alpha \in \mathrm{FTV}(\tau')$. If $\alpha \in \mathrm{ATV}(\tau')$ then clearly $S'(\alpha) = S_{E_1}(\alpha)$. Otherwise $\alpha \in \mathrm{ITV}(\tau')$ whence $\alpha \in \mathrm{ITV}(\tau_2)$ by our restriction on the well-formed type $\tau' \rightarrow \tau_2$. *This is the formal origin of our restriction on well-formed types.*
Hence $\alpha \in \mathrm{Dom}(S')$ and $S'(\alpha) = S_{E_1}(\alpha)$. Then we use similar reasoning as in the previous case.

- $E[\,] \equiv$ `let` $x = E_1$ `in` e_2. Then we can derive

$$\phi, \mathcal{R} \triangleright E_1[e]\colon \tau' \tag{5.8}$$
$$x\colon Clos_\phi \tau', \mathcal{R} \triangleright e_2\colon \tau \tag{5.9}$$

By induction hypothesis,

$$\vdash \phi, \mathcal{R}' \triangleright E_1[e']\colon S_{E_1}\tau' \tag{5.10}$$

for some suitable S_{E_1}. By the form of substitution S_{E_1}, $Clos_\phi \tau_1 = Clos_\phi S_{E_1}(\tau_1)$, *which was precisely the purpose of the inductive statement in (iii).*

Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau)$. Choose any distinct $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}') \cup \mathrm{ITV}(\tau')$. Let $S_E = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$. Then clearly

$$x\colon \mathit{Clos}_\phi \tau_1, \mathcal{R}' \triangleright e_2\colon S_E\tau$$

is a well-formed judgement. Also since $\mathrm{FTV}(\mathit{Clos}_\phi\tau_1) = \mathrm{ITV}(\mathit{Clos}_\phi\tau_1)$ and since (5.9) is a well-formed judgement, $\alpha_i \notin \mathrm{FTV}(\mathit{Clos}_\phi\tau_1)$. Thus using Lemma 5.6.2 on (5.9) with substitution S_E we have

$$\vdash x\colon \mathit{Clos}_\phi(S_{E_1}\tau'), \mathcal{R}' \triangleright e_2\colon S_E\tau$$

Combining with (5.10),

$$\vdash \phi, \mathcal{R}' \triangleright E[e']\colon S_E\tau$$

Other cases of evaluation contexts are similar. ∎

Under suitable assumptions about the store in which an expression is evaluated, we can now establish that evaluation of typable expressions leads to typable expressions.

Lemma 5.6.11 (Preservation of Typability) *Suppose* $(e, s) \longmapsto (e', s')$ *and* $\mathcal{R}$ *is a store typing such that* $s\colon \mathcal{R}$ *and* $\vdash \phi, \mathcal{R} \triangleright e\colon \tau$. *Then there is a store typing* $\mathcal{R}'$ *such that* $s'\colon \mathcal{R}'$ *and for some type* τ', $\vdash \phi, \mathcal{R}' \triangleright e'\colon \tau'$.

Proof By case analysis on $\longmapsto$.

- $(E[e], s) \longmapsto (E[e'], s)$ with $e \leadsto e'$. Assume that $\vdash \phi, \mathcal{R} \triangleright E[e]\colon \tau$. From Lemma 5.6.10(i), we have that $\vdash \phi, \mathcal{R} \triangleright e\colon \tau_1$ for some type τ_1. By Lemma 5.6.9, $\vdash \phi, \mathcal{R} \triangleright e'\colon \tau_1$. Now, using Lemma 5.6.10(ii), $\vdash \phi, \mathcal{R} \triangleright E[e']\colon \tau$. And, by assumption $s\colon \mathcal{R}$.

- $(E[\texttt{ref}\ v], s) \longmapsto (E[l], s[l \mapsto v])$.
  By Lemma 5.6.10(i), $\vdash \phi, \mathcal{R} \triangleright \texttt{ref}\ v\colon \tau_1$ for some type τ_1. Hence $\vdash \phi, \mathcal{R} \triangleright v\colon \tau_1'$ and $\tau_1 = W(\mathit{Set}_1(\tau_1'))\ \mathit{ref}$.

Consider $\mathcal{R}' = \mathcal{R}, l{:}\, Set_0(\tau_1')$. For $l' \neq l$, since $\vdash \phi, \mathcal{R} \triangleright s(l'){:}\, \mathcal{R}(l')$ and $\mathcal{R}(l')$ is imperative, we have by Corollary 5.6.4, $\vdash \phi, \mathcal{R}' \triangleright s(l'){:}\, \mathcal{R}(l')$, *i.e.*, $\vdash \phi, \mathcal{R}' \triangleright s'(l'){:}\, \mathcal{R}'(l')$ where $s' = s[l \mapsto v]$. And, since $\vdash \phi, \mathcal{R} \triangleright v{:}\, \tau_1'$, by Corollary 5.6.4, $\vdash \phi, \mathcal{R}' \triangleright v{:}\, Imp_{\mathrm{ITV}(\mathcal{R}')}\tau_1'$. Clearly, $Imp_{\mathrm{ITV}(\mathcal{R}')}\tau_1' = Set_0(\tau_1')$. Hence, $\vdash \phi, \mathcal{R}' \triangleright s'(l){:}\, \mathcal{R}'(l)$ and thus $s'{:}\, \mathcal{R}'$. Trivially, $\vdash \phi, \mathcal{R}' \triangleright l{:}\, \tau_1$, since $W(Set_1(\tau_1)') = Set_0(\tau_1')$. Thus, by Lemma 5.6.10(iii), $\vdash \phi, \mathcal{R}' \triangleright E[l]{:}\, \tau'$ for $\tau' = S_E(\tau)$.

- $(E[!l], s) \longmapsto (E[s(l)], s)$. By Lemma 5.6.10(i), $\vdash \phi, \mathcal{R} \triangleright !l{:}\, \tau_1$. Hence, $\vdash \phi, \mathcal{R} \triangleright l{:}\, \tau_1\ ref$, *i.e.*, $\mathcal{R}(l) = \tau_1$. Since $s{:}\, \mathcal{R}$, we have that $\vdash \phi, \mathcal{R} \triangleright s(l){:}\, \mathcal{R}(l)$, *i.e.*, $\vdash \phi, \mathcal{R} \triangleright s(l){:}\, \tau_1$. Now, using Lemma 5.6.10(ii), we get that $\vdash \phi, \mathcal{R} \triangleright E[s(l)]{:}\, \tau$.

- $(E[l := v], s) \longmapsto (E[v], s[l \mapsto v])$. Then $\vdash \phi, \mathcal{R} \triangleright l{:}\, \tau_1\ ref, \vdash \phi, \mathcal{R} \triangleright v{:}\, \tau_1$. Hence $s[l \mapsto v]{:}\, \mathcal{R}$. And since $\vdash \phi, \mathcal{R} \triangleright v{:}\, \tau_1$, by Lemma 5.6.10(ii), $\vdash \phi, \mathcal{R} \triangleright E[v]{:}\, \tau_1$.

- $(E[e], s) \longmapsto (\mathtt{wrong}, s)$, where $e \rightsquigarrow \mathtt{wrong}$. By Lemma 5.6.9, since `wrong` is not typable, there is no $\mathcal{R}, \tau_1$ such that $\vdash \phi, \mathcal{R} \triangleright e{:}\, \tau_1$. By Lemma 5.6.10(i), this implies that $\not\exists \mathcal{R}, \tau$ such that $\vdash \phi, \mathcal{R} \triangleright E[e]{:}\, \tau$ and the Lemma is thus true vacuously.

- $(E[!v], s) \mapsto (\mathtt{wrong}, s)$, where $v \notin Loc$. By admissibility condition (i) and the *(abs)* rule, $\not\vdash \phi, \mathcal{R} \triangleright v{:}\, \tau_1\ ref$ for any $\mathcal{R}, \tau_1$. Hence $\not\vdash \triangleright \triangleright \phi{:}\, , \mathcal{R}{:}\, !v\tau_1$ for any $\mathcal{R}, \tau_1$. Then this is similar to the previous case.

- $(E[v_1 := v_2], s) \longmapsto (\mathtt{wrong}, s)$, where $v_1 \notin Loc$. This is similar to the previous case.

∎

As a corollary of Lemma 5.6.11, we can now establish that starting in the initial store, evaluation of any typable user program leads to typable expressions.

Corollary 5.6.12 *Let e be a program,* i.e., *a closed expression with no location constants. If e is typable,* i.e., $\vdash \phi, \phi \triangleright e{:}\, \tau$ *for some type τ then for any e', s' such that* $(e, \mathbf{init}) \longmapsto\!\!\!\!\rightarrow (e', s')$ *we have that e' is typable.*

Proof The store **init** has the empty store typing, *i.e.*, $\mathbf{init}{:}\, \phi$. Now the statement follows from Lemma 5.6.11. ∎

Finally, we can establish our main theorem, that well-typed programs do not lead to run-time errors. And, by Lemma 5.3.2, under suitable assumptions about the specification of the operational behavior of the term constructors this implies that every typable program either diverges or results in a value.

Theorem 5.6.13 *Let e be a typable program. Then $Eval(e) \neq$* `wrong`, i.e., *e does not lead to a run-time error.*

Proof There is no $\mathcal{R}', \tau'$ such that $\vdash \phi, \mathcal{R}' \triangleright \texttt{wrong}\colon \tau'$. Hence, by Corollary 5.6.12 we cannot have $(e, \mathbf{init}) \longmapsto\!\!\!\!\rightarrow (\texttt{wrong}, s')$, for any store s'. ∎

5.7 Conclusion

We present a type system, in the form of typing rules, using weak type variables. Intuitively, a weak type variable is a variable with a numeric index, indicating the number of function applications needed to produce a reference cell containing a value of this type. Although the Standard ML of New Jersey source code is sufficiently complex to make direct proof difficult, extensive experiments with the compiler and examination of the source code suggest that our type system strictly subsumes the implemented type system. Moreover, as noted in Section 5.5, our type system is sufficiently flexible to accept all the benchmark examples listed in [LW91] without the more complicated type expressions used in [LW91, TJ92b, Wri92]. In light of the comments on the limitations of SML-NJ in [LW91], our type system shows that simple indexed type variables allow more flexible typing of imperative programs than previously believed. To the extent that our type system subsumes the implemented algorithm, Theorem 5.6.13 shows not only the semantic soundness of our typing rules, but also the soundness of the SML-NJ algorithm that has been used for several years. Since our type system is more flexible than the one used in the SML-NJ, but requires very little change in displayed type expressions, we believe it is a good candidate for later implementations of ML and other polymorphic programming languages that have imperative or control constructs.

Bibliography

[ABL86] R.M. Amadio, K. Bruce, and G. Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 122–130, 1986.

[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[AJM93] Samson Abramsky, Radha Jagadeesan, and Pascale Malacaria. Games and full abstraction for PCF: preliminary announcement. Unpublished, 1993.

[AM87] A.W. Appel and D.B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer LNCS 274, 1987.

[AM91] A.W. Appel and D.B. MacQueen. Standard ML of New Jersey. In *Int'l Symp. Prog. Lang. Implementation and Logic*, pages 1–13. Springer-Verlag, 1991.

[Ama91] R.M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–86, 1991.

[AP90] M Abadi and G.D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 355–365, 1990.

[Bar75] J. Barwise. *Admissible sets and structures*. Springer-Verlag, Berlin, 1975.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984. Second edition.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 89–132. Cambridge Univ. Press, 1985.

[BM92] K. Bruce and J.C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 316–327, January 1992.

[BMM90] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990. Reprinted in *Logical Foundations of Functional Programming,* ed. G. Huet, Addison-Wesley (1990) 213–273.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as explicit coercion. *Information and Computation*, 93(1):172–221, 1991. Reprinted in [GM94].

[Car89] F. Cardone. Relational semantics for recursive types and bounded quantification. In *ICALP*, pages 164–178, Berlin, 1989. Springer LNCS 372.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic I.* North-Holland, Amsterdam, 1958.

[CF92] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 328–342. ACM, 1992.

[CGW89] T. Coquand, C.A. Gunter, and G. Winskel. Domain-theoretic models of polymorphism. *Information and Computation*, 81(2):123–167, 1989.

[Cos89] Stavros S. Cosmadakis. Computing with recursive types. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 24–38. IEEE, 1989.

[CR91] W. Clinger and J. Rees. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4:1–55, 1991.

[Cur92] P.-L. Curien. Observable sequential algorithms on concrete data structures. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 432–443, 1992.

[Cut80] N.J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge Univ. Press, Cambridge, 1980.

[Dam85] L. Damas. *Type assignment in programming languages.* PhD thesis, Edinburgh University CST-33-85, 1985.

[Fel88a] Matthias Felleisen. λ-V-CS: An extended λ-calculus for Scheme. In *Proc. of Conf. LISP and Functional Programming*, pages 72–85. ACM, July 1988.

[Fel88b] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[FRMS92] P. Freyd, G. Rosolini, P. Mulry, and D.S. Scott. Extensional PER's. *Information and Computation*, 98(2):211–227, 1992. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* IEEE, 1990, 346–354.

[Gir86] J.-Y. Girard. The system F of variable types, fifteen years later. *Theor. Comp. Sci.*, 45(2):159–192, 1986.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical aspects of object-oriented programming.* MIT Press, Cambridge, MA, 1994.

[GS90] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 633–674. North-Holland, Amsterdam, 1990.

[GS94] J.C. Guzmán and A. Suárez. A type system for exceptions. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, 1994. Available as INRIA Technical Report 2265.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[HL91] R. Harper and M. Lillibridge. A bug in SML polymorphic typing for continuations. Email to `types@theory.lcs.mit.edu`, 1991.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, 1995.

[HO93] Martin Hyland and Luke Ong. Dialogue games and innocent strategies: An approach to intensional full abstraction to PCF (preliminary announcement). Unpublished, 1993.

[HPW+92] Paul Hudak, Simon L. Peyton Jones, P.L. Wadler, B. Boutel Arvind, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R.S. Nikhil, W. Partain, and J. Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[HRR90] J.M.E. Hyland, E.P. Robinson, and G. Rosolini. The discrete objects in the effective topos. *Proc. London Math. Soc.*, 60:1–36, 1990.

[Hud92] Paul Hudak. Continuation-based mutable abstract datatypes, or how to have your state and munge it too. Technical Report Technical Report Research Report YALEU/DCS/RR-914, Yale University, 1992.

[Hue93] L.F. Huelsbergen. *Dynamic Language Parallelization*. PhD thesis, University of Wisconsin, 1993. Available as Computer Sciences Department Technical Report Number 1178.

[Hyl82] J.M.E. Hyland. The effective topos. In *The L.E.J. Brouwer Centenary Symposium*, pages 165–216. North-Holland, Amsterdam, 1982.

[Hyl88] J.M.E. Hyland. A small complete category. *Ann. Pure and Applied Logic*, 40, 1988. Lecture delivered at the conference Church's Thesis: Fifty Years Later, Zeiss(NL), June 1986.

[Hyl90] J.M.E. Hyland. First steps in synthetic domain theory. In A. Carboni *et al.*, editor, *Category Theory, Proc. Como 1990*, pages 131–156. Springer LNM 1488, 1990.

[JG89] P. Jouvelot and D.K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[JG91] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *18th ACM Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[KMM91] P.C. Kanellakis, H.G. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[Koe92] A. Koenig. An anecdote about ML type inference. Unpublished Manuscript, 1992.

[KTU90] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is Dexptime-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990. To appear in *J. Assoc. Comput. Machinery* under title "An Analysis of ML Typability".

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[LP94] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

[Luc87] J.M. Lucassen. *Type-checking Fluent Languages.* PhD thesis, Massachusetts Institute of Technology, 1987. Available as technical report MIT/LCS/TR-408 (MIT Laboratory for Computer Science).

[LW91] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *18th ACM Symposium on Principles of Programming Languages*, pages 291–302, 1991.

[McC79] N. McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse Univ., 1979.

[Mil77] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[Mor93] G. Morrisett. Generalizing first-class stores. In *ACM SIGLPLAN Workshop on State in Programming Languages*, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Mul87] K. Mulmuley. *Full abstraction and semantic equivalence.* The MIT Press, 1987.

[Ong88a] Chih-Hao Luke Ong. Fully abstract models of the lazy lambda calculus. In *29th Annual Symposium on Foundations of Computer Science*, pages 368–376. IEEE, 1988.

[Ong88b] Chih-Hao Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Imperial College, University of London, 1988.

[OR95] P.W. O'Hearn and J.G. Riecke. Kripke relations and PCF. *Information and Computation*, To appear, 1995.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 43–56. ACM, 1993.

[OT93] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 171–184. ACM, 1993.

[Pho90a] W. Phoa. *Domain theory in realizability toposes.* PhD thesis, Cambridge, 1990. Available as University of Edinburgh Dept. of Computer Science report CST-82-91 and ECS-LFCS-91-171.

[Pho90b] W. Phoa. Effective domains and intrinsic structure. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 366–377, 1990.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PW93] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. Twentieth ACM POPL Conf.*, pages 71–84, 1993.

[Rey78] J.C. Reynolds. Syntactic control of interference. In *Proc. 5th ACM POPL Conf.*, pages 39–46, New York, 1978.

[Rey81] J.C. Reynolds. The essence of Algol. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. IFIP, North-Holland.

[Rey89] J.C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S.R.D. Rocca, editors, *Automata, Languages and Programming: 16th International Colloquium*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer-Verlag, 1989.

[Rie91] Jon G. Riecke. *The Logic and Expressibility of Simply-Typed Call-by-Value and Lazy Languages.* PhD thesis, Massachusetts Institute of Technology, 1991. Available as technical report MIT/LCS/TR-523 (MIT Laboratory for Computer Science).

[Rie93a] Jon G. Riecke. Delimiting the scope of effects. In *Proceedings of the 1993 Conference on Functional Programming and Computer Architecture*, pages 146–158. ACM, 1993.

[Rie93b] Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 1993. To appear.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[RV93] Jon G. Riecke and Ramesh Viswanathan. Full abstraction for call-by-value sequential languages. Unpublished manuscript, 1993.

[RV95] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1995.

[Sco76] D. Scott. Data types as lattices. *Siam J. Computing*, 5(3):522–587, 1976.

[SF90] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175. ACM, 1990.

[Smi91] Scott F. Smith. From operational to denotational semantics. In *Mathematical Foundations of Programming Language Semantics*, 1991. To appear.

[SP82] M. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.

[SRI91] V. Swarup, U. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Conference Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 192–214. Springer-Verlag, 1991.

[Tal93] J.-P. Talpin. *Theoretical and Practical Aspects of Type and Effect Inference.* PhD thesis, University of Paris VI, 1993.

[TJ92a] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, pages 2:245–271, 1992.

[TJ92b] J.-P. Talpin and P. Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Edinburgh University, 1988. Available as Edinburgh University

Laboratory for Foundations of Computer Science Technical Report ECS–LFCS–88–54.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.

[WF92] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, 1992.

[Wri92] A.K. Wright. Typing references by effect inference. In *Proceedings of the European Symposium on Programming*, 1992.