

Two-Level Languages for Program Optimization

Cristiano Calcagno *

DISI, Univ. di Genova, Genova, Italy
calcagno@disi.unige.it

Abstract. Two-level languages incorporate binding time information inside types, that is, whether a piece of code is completely known at compile-time, or needs some more inputs and can be evaluated only at run-time. We consider the use of 2-level languages in the framework of partial evaluation, and use a 2-level version of the simply typed lambda calculus with recursion. We give an operational semantics, an equational theory and a denotational semantics, that give an account of the distinction between compilation and execution phases. An adequacy theorem is given to relate the two semantics, showing in particular how they agree on non-termination at compile time. We finally give a more refined model using functor categories.

1 Introduction

Partial evaluation is an attempt to fill the gap between interpreting and compiling. In the first case we obtain an easy-to-prove correctness and good flexibility to modifications. Unfortunately, we usually get also a poor run-time behaviour, often an order of magnitude slower than the non-interpretative counterpart. On the other hand, compiled code is comparatively hard to understand, and prove correct.

The aim of partial evaluation is to take a program as input and produce a new program that gives the same output as the original one. But constant evaluations are performed just once, during the program generation process. This new program will incorporate all the data that remains constant, and is called a *specialized* version of the old one. The following picture illustrates the process:

$$\text{Program} \xrightarrow{\text{compile-time}} \text{Residual} \xrightarrow{\text{run-time}} \text{Value}$$

In this view, it is essential to distinguish between the computations that can be performed at compile-time, called *static*, and the computations that need some more data to be executed, that are called *dynamic*: the process of making this distinction is called *binding time analysis*.

A classical example is the function *power*, that takes two integers x and y , and computes y^x , the x -th power of y . This function could be defined as follows

$$f \ x \ y = \text{if } x = 0 \text{ then } 1 \text{ else } y * f \ (x - 1) \ y$$

* Research partially supported by MURST

Suppose that we know at compile-time that x is the number 3. Then x is a static variable and y dynamic, and we can produce a residual program

$$y * y * y$$

This program is typically more efficient than $f\ 3\ y$.

There are various ways to perform a binding time analysis, but a promising technique is to use a 2-level language: a language that incorporates binding time information inside its types. Usually these languages have two versions of each data type constructor, one for static and one for dynamic types. 2-level languages were originally introduced in [12], and have been studied extensively in [18], but their use for partial evaluation is more recent. Some examples can be found in [4] and [11].

In this paper we study the semantics of a 2-level language, and give an operational semantics and a denotational model. The basic idea of the semantics is to evaluate all the static components of a program during compilation, obtaining a *residual* program, that is a specialized version of the original one, and execute it at run-time.

In the second section we introduce a 2-level language, essentially a 2-level variant of the call-by-name language PCF [19], and give an operational semantics that gives an account of the distinction between the compilation and the execution phases. This distinction is made clear on showing that all the dynamic operations can be postponed until after the static ones have been performed. We study equivalences between terms, and show how an unrestricted β -rule is not valid for dynamic terms, although both levels of the language are purely call-by-name if taken in isolation. This makes it more difficult to give an abstract model, because two kinds of undefinedness are required, one for each level, and it is not clear at first how to generalize it to higher types.

In the third section we relate the operational semantics and the denotational model given, and show an adequacy theorem asserting that the model not only gives the same results, but also preserves the phase distinction from the operational semantics.

The last section shows that, although adequate, the model contains some junk, that is, some basic elements that are not definable within the language. We suggest a better solution using a slightly more complex machinery.

The purpose of this work is to provide some insight in how to prove correctness of partial evaluation, that is to prove the equivalence between the semantics of the original program and the semantics of the specialized one. Both [11] and [4] give a partial evaluator for a 2-level language but leave a correctness proof for further work, while [18] gives a dynamic semantics parametrised on a fixed static semantics, but does not consider partial evaluation. A model is given in [18] but no connection to operational notions is made. In our view, this connection is crucial for understanding the link between the semantics and implementation issues, especially for partial evaluation.

Proving correctness of partial evaluators is difficult, and often error prone, as shown in [8], where partial evaluation for an untyped language is considered.

An alternative solution is given in [16], and the correctness of the original model is proved in [7], which also provides a correctness proof of the partial evaluator of [4].

Existing models ([8, 7, 16]) give a syntactic interpretation of dynamic types, namely as collections of open terms of the language, while our approach is more abstract and similar to a standard treatment of PCF. The novelty of our approach is the interpretation of both the compilation and execution phases in the same framework, allowing to study their interaction with respect to computational effects like non-termination.

2 A simple 2-level language

In this section we introduce a simple language, 2-level PCF, obtained from PCF by augmenting each construct with a dynamic counterpart. First we explain the syntax of the language and the type formation rules; then we give an operational semantics, an equational theory and a denotational model.

2.1 2-level PCF

2-level PCF is a language obtained from PCF by adding dynamic types and dynamic terms. The intended use of the language is to evaluate the static components at compile-time, and execute the remaining part - the *residual program* - at run-time. The base types are the usual natural numbers, *nat*, and the dynamic numbers, *nat*. The higher types are constructed with static arrows, \rightarrow , and dynamic ones, $\underline{\rightarrow}$. There are well-formedness rules on dynamic types, introduced originally in [18]: a dynamic arrow must have dynamic types on either side. These rules are motivated by the consideration that dynamic functions should not depend on static computations, because at run-time the static computations have already been performed. For the base case, there is no reason to allow dynamic functions to take static naturals as inputs, because there is a construct - **lift** - that embeds terms of type *nat* in dynamic terms. Moreover, these constraints give interesting properties that allow to postpone all the dynamic evaluations after the static ones in the operational semantics.

The types of 2-level PCF are given by the following grammar:

$$\begin{aligned} s &::= \text{nat} \mid t \rightarrow t \\ d &::= \underline{\text{nat}} \mid d \underline{\rightarrow} d \\ t &::= s \mid d \end{aligned}$$

where *s* are the static types and *d* are the dynamic ones. We will use t, t_1, t_2, \dots as generic type variables, and d, d_1, d_2, \dots for dynamic type variables.

The terms and type rules are structured in a similar way: essentially two copies of every construct.

Terms:

$$M ::= x \mid n \mid \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{ifz} M \mathbf{then} M \mathbf{else} M \mid \lambda x.M \mid M M \mid \\ \mathbf{fix} x.M \mid \mathbf{lift}(M) \mid \underline{\mathbf{succ}}(M) \mid \underline{\mathbf{pred}}(M) \mid \underline{\mathbf{ifz}} M \underline{\mathbf{then}} M \underline{\mathbf{else}} M \mid \\ \lambda x.M \mid M @ M \mid \underline{\mathbf{fix}} x.M$$

Here **succ** is the successor function over numbers, **ifz** is the conditional whose guard checks if the first argument is zero, **fix** is the recursion operator, and **lift** embeds a static expression of type *nat* in a dynamic expression: note that this is possible only for the base type. The purpose of **lift** is to insert a static value into the residual program.

A *type assignment* Γ , written $x_1 : t_1, \dots, x_n : t_n$, is a finite set of variable/type pairs, where all the x_i are distinct. $\Gamma, x : t$ indicates the extension of Γ with the pair (x, t) .

Type formation rules:

$$\begin{array}{c} \Gamma, x : t \vdash x : t \quad \Gamma \vdash n : \mathit{nat} \\ \\ \frac{\Gamma \vdash M : \mathit{nat} \quad \Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t}{\Gamma \vdash \mathbf{ifz} M \mathbf{then} M_1 \mathbf{else} M_2 : t} \\ \\ \frac{\Gamma \vdash M : \mathit{nat}}{\Gamma \vdash \mathbf{succ} M : \mathit{nat}} \quad \frac{\Gamma \vdash M : \mathit{nat}}{\Gamma \vdash \mathbf{pred} M : \mathit{nat}} \quad \frac{\Gamma, x : t_1 \vdash M : t_2}{\Gamma \vdash \lambda x.M : t_1 \rightarrow t_2} \\ \\ \frac{\Gamma \vdash M : t_1 \rightarrow t_2 \quad \Gamma \vdash N : t_1}{\Gamma \vdash M N : t_2} \quad \frac{\Gamma, x : t \vdash M : t}{\Gamma \vdash \mathbf{fix} x.M : t} \\ \\ \frac{\Gamma \vdash M : \mathit{nat}}{\Gamma \vdash \mathbf{lift}(M) : \underline{\mathit{nat}}} \quad \frac{\Gamma \vdash M : \underline{\mathit{nat}}}{\Gamma \vdash \underline{\mathbf{succ}} M : \underline{\mathit{nat}}} \quad \frac{\Gamma \vdash M : \underline{\mathit{nat}}}{\Gamma \vdash \underline{\mathbf{pred}} M : \underline{\mathit{nat}}} \\ \\ \frac{\Gamma \vdash M : \underline{\mathit{nat}} \quad \Gamma \vdash M_1 : d \quad \Gamma \vdash M_2 : d}{\Gamma \vdash \underline{\mathbf{ifz}} M \underline{\mathbf{then}} M_1 \underline{\mathbf{else}} M_2 : d} \\ \\ \frac{\Gamma, x : d_1 \vdash M : d_2}{\Gamma \vdash \lambda x.M : d_1 \rightarrow d_2} \quad \frac{\Gamma, x : d \vdash M : d}{\Gamma \vdash \underline{\mathbf{fix}} x.M : d} \\ \\ \frac{\Gamma \vdash M : d_1 \rightarrow d_2 \quad \Gamma \vdash N : d_1}{\Gamma \vdash M @ N : d_2} \end{array}$$

Note that the dynamic if-then-else requires all the arguments to be dynamic; this is to avoid, for example, static terms like

$$\mathbf{ifz} D \mathbf{then} 2 \mathbf{else} 3 : \mathit{nat}$$

whose evaluation would depend on the evaluation of the dynamic term D . This would be a problem especially if we consider the possibility of non-termination: if D does not terminate at run-time, this cannot influence the value of the whole term at compile-time.

2.2 Operational Semantics

In this section we give an operational semantics for 2-level PCF. The basic idea is to give two reduction relations: \rightarrow_s for static reduction and \rightarrow_d for dynamic reduction. The intended constraint is that static reduction must always be performed before the dynamic one.

Operational semantics:

$$\begin{array}{l}
\mathbf{succ} \, n \rightarrow_s n + 1 \\
\mathbf{pred} \, n \rightarrow_s n -_0 1 \\
\lambda x.M \, N \rightarrow_s M[N/x] \\
\mathbf{fix} \, x.M \rightarrow_s M[\mathbf{fix} \, x.M/x] \\
\mathbf{ifz} \, 0 \mathbf{then} \, M_1 \mathbf{else} \, M_2 \rightarrow_s M_1 \\
\mathbf{ifz} \, n + 1 \mathbf{then} \, M_1 \mathbf{else} \, M_2 \rightarrow_s M_2 \\
\frac{M \rightarrow_s M'}{C_s[M] \rightarrow_s C_s[M']} \\
\mathbf{succ} \, \mathbf{lift}(n) \rightarrow_d \mathbf{lift}(n + 1) \\
\mathbf{pred} \, \mathbf{lift}(n) \rightarrow_d \mathbf{lift}(n -_0 1) \\
\lambda x.M \, @ \, N \rightarrow_d M[N/x] \\
\mathbf{fix} \, x.M \rightarrow_d M[\mathbf{fix} \, x.M/x] \\
\mathbf{ifz} \, \mathbf{lift}(0) \mathbf{then} \, M_1 \mathbf{else} \, M_2 \rightarrow_d M_1 \\
\mathbf{ifz} \, \mathbf{lift}(n + 1) \mathbf{then} \, M_1 \mathbf{else} \, M_2 \rightarrow_d M_2 \\
\frac{M \rightarrow_d M'}{C_d[M] \rightarrow_d C_d[M']}
\end{array}$$

where static contexts C_s and dynamic contexts C_d are defined as follows:

$$C_s ::= \mathbf{succ} \, [] \mid \mathbf{pred} \, [] \mid [] \, M \mid \mathbf{ifz} \, [] \mathbf{then} \, M_1 \mathbf{else} \, M_2 \mid \mathbf{succ} \, [] \mid \mathbf{pred} \, [] \mid \lambda x.[] \mid [] \, @ \, [] \mid \mathbf{fix} \, x.[] \mid \mathbf{ifz} \, [] \mathbf{then} \, [] \mathbf{else} \, []$$

$$C_d ::= \mathbf{succ} \, [] \mid \mathbf{pred} \, [] \mid [] \, @ \, M \mid \mathbf{ifz} \, [] \mathbf{then} \, M_1 \mathbf{else} \, M_2$$

The notation $n -_0 m$ indicates the subtraction of natural numbers extended with 0 when $m > n$. Each relation is defined in an analogous way to the usual reductions for PCF, where $M[N/x]$ indicates the capture-free substitution of N for x in the term M . In particular \rightarrow_s is *exactly* the one of PCF, while \rightarrow_d is the obvious dynamic counterpart. Note the use of **lift** to form basic constants of type *nat*: they are in fact the canonical forms of dynamic numbers, i.e. the form that a terminating computation is expected to reach.

Consider the static term *sund* defined as follows.

$$sund \equiv \mathbf{fix} \, x.x : \mathit{nat}$$

We will use *sund* as a canonical term to represent static undefinedness at base type. Clearly its evaluation will fail to terminate already during the compilation phase. On the other hand, consider the dynamic term *dund*:

$$dund \equiv \mathbf{fix} \ x.x : \underline{nat}$$

This new term is evaluated to itself during compilation. In fact, it cannot be reduced statically, but an attempt to execute it at run-time will lead to non-termination.

The need for the constraints on the interleaving of static and dynamic reductions can be explained with a few examples. Consider the following term:

$$M \equiv (\underline{\lambda}y.\mathbf{lift}(0)) \ @ \ \mathbf{lift}(sund)$$

In principle it is possible to use dynamic reduction to yield a result: $M \rightarrow_d \mathbf{lift}(0) : \underline{nat}$. But a careful analysis of M reveals the presence of a sub-term, *sund*, that does not terminate at compile-time, and because we are in the setting of partial evaluation we should try to reduce it, causing non-termination during the compilation of M . Conversely, the term

$$M' \equiv (\lambda x.0) \ sund$$

can be statically reduced to 0, although it has an infinite reduction sequence.

The discussion above suggests that a term can be reduced dynamically only if it can be compiled; in other words terms that do not have a terminating static reduction should not be reduced dynamically. The simplest way to ensure that a term can be run safely is to require it to be in Static Normal Form (SNF).

Definition 1 *A term M is in SNF if there is no M' such that $M \rightarrow_s M'$.*

We can now put the two relations together.

Definition 2 *Given a well-typed term M , $M \rightarrow_{sd} M'$ if*

1. $M \rightarrow_s M'$, or
2. M is in SNF and $M \rightarrow_d M'$.

The following lemma is crucial, in the sense that it shows how all the static reductions can be performed before the dynamic ones, obtaining a complete separation between the two phases.

Lemma 3 Postponent

Given a well-typed term M , $M \rightarrow_{sd}^ M'$ if and only if there is a term M'' such that $M \rightarrow_s^* M''$ and $M'' \rightarrow_d^* M'$.*

Proof. We show that if M is in SNF and $M \rightarrow_d M'$, then M' is in SNF. The only interesting part is the rule

$$(\underline{\lambda}x.M_1) \ @ \ M_2 \rightarrow_d M_1[M_2/x]$$

Clearly M_1 and M_2 are in SNF, hence the only problem is if M_1 contains a sub-term of the form $(x N)$ for some N , but this is not possible because x must have dynamic type due to the type formation rules. \square

If we relax the condition that M must be well-typed, the previous lemma is no longer valid. Consider the ill-typed term

$$M \equiv (\lambda y. (y \ 3)) \underline{\@} (\lambda x. x)$$

Clearly M is in SNF, but $M \rightarrow_d ((\lambda x. x) \ 3) \rightarrow_s 3$.

We conclude this section with a complete example of partial evaluation: the power function discussed in the introduction. Let $POW : nat \rightarrow \underline{nat} \rightarrow \underline{nat}$ be the term

$$\mathbf{fix} \ f. \ \lambda n. \lambda x. \mathbf{ifz} \ n \ \mathbf{then} \ \mathbf{lift}(1) \ \mathbf{else} \ x \ \underline{*} \ (f \ (n - 1) \ x)$$

where $-$ is the static subtraction, and $\underline{*}$ is the dynamic product, both definable in the obvious way. The reduction has the effect of compiling the program yielding an optimized version:

$$(POW \ 2) \rightarrow_{sd}^* \lambda x. \ x \ \underline{*} \ x \ \underline{*} \ \mathbf{lift}(1)$$

If all the arguments are given to POW , also the execution phase is performed:

$$(POW \ 2 \ \mathbf{lift}(7)) \rightarrow_{sd}^* \mathbf{lift}(49)$$

From the example, it should be clear that the operational semantics gives information about both compilation and execution of programs.

2.3 Equational Theory

In this section we consider the β and η rules familiar from type theory, and state which versions of them are valid in this setting. Two versions of each rule can be expressed in the language, and only one of the four combinations is invalid.

The following rules are valid

$$\begin{aligned} \beta \ (\lambda x. M) \ N &= M[N/x] \\ \eta \ \lambda x. (M \ x) &= M \\ \underline{\eta} \ \underline{\lambda} x. (M \ \underline{\@} \ x) &= M \end{aligned}$$

The β and η rules are valid because the static fragment of the language is purely call-by-name. The $\underline{\eta}$ rule does not affect the static behaviour of the term, and purely dynamic terms behave in a call-by-name fashion.

The $\underline{\beta}$ rule is not valid:

$$\underline{\beta} \ (\underline{\lambda} x. M) \ \underline{\@} \ N = M[N/x]$$

In fact, consider the term $(\lambda x.\mathbf{lift}(0)) @ \mathbf{lift}(sund)$. It cannot be equal to $\mathbf{lift}(0)$, because the term $sund$ does not terminate at compile-time, thus the code $\mathbf{lift}(0)$ is never produced. Borrowing the ideas from operational semantics, we can give a special rule

$$\underline{\beta}' (\lambda x.M) @ N = M[N/x] \text{ if } N \text{ is in SNF}$$

The remarks in this section will be verified after introducing the denotational model.

2.4 Denotational Semantics

A CPO (complete partial order) is a partially ordered set with a least element and all least upper bounds of ω -chains, and a function between CPOs is said to be continuous if it preserves least upper bounds of ω -chains (hence is monotone). In this section we give a model of the language interpreting types as CPOs and terms as continuous functions between CPOs. We write N_\perp for the flat CPO of natural numbers (with $\perp \leq n$ for each number n), and in general X_\perp for the CPO obtained from the CPO X by adding a new least element. $X \rightarrow Y$ indicates the CPO of continuous functions between X and Y , and \circ is for function composition.

Interpretation of types:

$$\begin{aligned} \llbracket nat \rrbracket &\triangleq N_\perp \\ \llbracket \underline{nat} \rrbracket &\triangleq N_{\perp\perp} \\ \llbracket t_1 \rightarrow t_2 \rrbracket &\triangleq \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket \\ \llbracket d_1 \rightrightarrows d_2 \rrbracket &\triangleq (X \rightarrow Y)_\perp, \quad \text{where } X_\perp = \llbracket d_1 \rrbracket \text{ and } Y_\perp = \llbracket d_2 \rrbracket. \end{aligned}$$

The interpretation of static types is the usual one from call-by-name PCF, while the dynamic types deserve an explanation. We have seen that there are two kinds of undefinedness: at compile-time and at run-time. In this view, the type \underline{nat} contains two elements apart from natural numbers: \perp is intended to interpret terms like $\mathbf{lift}(sund)$ and indicate failure during the compilation phase, while $\underline{\perp}$ is intended to interpret terms like $dund$ that do not terminate when executed. Normally $\llbracket \underline{nat} \rrbracket$ would be written $(N_\perp)_\perp$, but we use $\underline{\perp}$ as a notational convenience to denote the inner \perp .

For the dynamic arrow, the idea is that a dynamic function can either not terminate at compile-time, or behave in a usual call-by-name fashion at run-time. Thus the semantics of dynamic types turns out to be the lifting of the semantics of the corresponding static types obtained removing underlines.

From the semantics of types one might have expected $\underline{\beta}$ to be valid and $\underline{\eta}$ invalid, because $\llbracket - \rightrightarrows - \rrbracket$ looks like in lazy λ -calculus. But we will see that just the reverse is true.

For each CPO X we consider a function $up : X \rightarrow X_\perp$ that embeds $x \in X$ in X_\perp ,

and $down : X_{\perp} \rightarrow X$ that is the identity on X extended by sending \perp to the least element of X . The two following functions will be essential to give the semantics of dynamic terms:

$$\begin{aligned}
 dyn : \quad & (X_{\perp} \rightarrow Y_{\perp}) \rightarrow (X \rightarrow Y)_{\perp} \\
 & f \mapsto \begin{array}{l} up(down \circ f \circ up) \quad \text{if } f(X) \subseteq Y \\ \perp \quad \text{otherwise.} \end{array} \\
 \\
 dapp : (X \rightarrow Y)_{\perp} \times X_{\perp} \rightarrow Y_{\perp} \\
 & \langle f, x \rangle \mapsto \begin{array}{l} \perp \quad \text{if } f = \perp \text{ or } x = \perp \\ up(down(f)(down(x))) \quad \text{otherwise.} \end{array}
 \end{aligned}$$

The function dyn is used to transform a static arrow between dynamic types into a dynamic arrow, and corresponds to the type formation rule

$$\frac{\Gamma, x : d_1 \vdash M : d_2}{\Gamma \vdash \underline{\lambda}x.M : d_1 \rightarrow d_2}$$

The idea behind the definition of dyn is that a static function between dynamic types is either undefined or is defined for each defined argument. That is, the failure to compile the function, once applied, does not depend on the dynamic argument passed to it, unless the argument itself fails to be compiled.

The $dapp$ operator is used to interpret dynamic application, and has the purpose of propagating the undefinedness of terms in a call-by-value style. Note that both levels are essentially call-by-name if taken in isolation, but the dynamic level behaves like call-by-value with respect to static undefinedness.

Given a type assignment $\Gamma \equiv x_1 : t_1, \dots, x_n : t_n$, an *environment* ρ is a function $\{x_1, \dots, x_n\} \rightarrow \bigcup_{i=1}^n \llbracket t_i \rrbracket$ such that $\rho(x_i) \in \llbracket t_i \rrbracket$. Given a judgement $\Gamma \vdash M : t$, we write $\llbracket M \rrbracket \rho$ for the interpretation of M in the environment ρ . Figure 1 defines the interpretation of the terms of the language.

The static components of the language are interpreted as in call-by-name PCF. The use of the functions dyn and $dapp$ has already been explained, and the other constructs are interpreted in a very natural way: propagating static undefinedness in case of non-termination, using a standard interpretation otherwise.

Now we reconsider the rules introduced in the equational theory section, and show that they are valid in the model. To see that $\underline{\beta}$ is not valid, consider the terms

$$\begin{aligned}
 M &\equiv \underline{\lambda}x.\mathbf{lift}(0) \\
 N &\equiv \mathbf{lift}(sund)
 \end{aligned}$$

From the denotational rules we have

$$\begin{aligned} \llbracket M \rrbracket &= up(x \mapsto up(0)) \in (N_{\perp} \rightarrow N_{\perp})_{\perp} \\ \llbracket N \rrbracket &= \perp \in N_{\perp\perp} \\ \llbracket M \ @ \ N \rrbracket &= \perp \in N_{\perp\perp} \end{aligned}$$

For the η rule, we have

$$\llbracket \lambda x. M \ @ \ x \rrbracket \rho = dyn(c \mapsto \llbracket M \ @ \ x \rrbracket \rho[x \mapsto c]) = dyn(c \mapsto dapp(\llbracket M \rrbracket \rho[x \mapsto c], c))$$

First note that $\llbracket M \rrbracket \rho = \llbracket M \rrbracket \rho[x \mapsto c]$ since x is not free in M by assumption.

If $\llbracket M \rrbracket \rho = \perp$, then $\llbracket \lambda x. M \ @ \ x \rrbracket \rho$, since $dyn(c \mapsto \perp) = \perp$. Otherwise, there exists an f such that $\llbracket M \rrbracket \rho = up(f)$, and $dyn(c \mapsto dapp(up(f), c)) = up(f)$.

The η rule is trivial, and in order to prove the other rules we need a fundamental lemma.

Lemma 4 Substitution Lemma

If $\Gamma, x: t_1 \vdash M: t_2$ and $\Gamma \vdash N: t_1$, then for each ρ

$$\llbracket M[N/x] \rrbracket \rho = \llbracket M \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho].$$

Proof. The proof is by induction on M . We indicate with ρ' the environment $\rho[x \mapsto \llbracket N \rrbracket \rho]$, and will consider only two important cases.

$$\begin{aligned} \text{Case } M &\equiv M_1 \ @ \ M_2, \\ \llbracket (M_1 \ @ \ M_2)[N/x] \rrbracket \rho &= \\ \llbracket M_1[N/x] \ @ \ M_2[N/x] \rrbracket \rho &= \\ dapp(\llbracket M_1[N/x] \rrbracket \rho, \llbracket M_2[N/x] \rrbracket \rho) &= \\ dapp(\llbracket M_1 \rrbracket \rho', \llbracket M_2 \rrbracket \rho') &= \\ \llbracket M_1 \ @ \ M_2 \rrbracket \rho'. \end{aligned}$$

$$\begin{aligned} \text{Case } M &\equiv \lambda y. M', \\ \llbracket (\lambda y. M')[N/x] \rrbracket \rho &= \\ \llbracket \lambda y. M'[N/x] \rrbracket \rho &= \\ dyn(c \mapsto \llbracket M'[N/x] \rrbracket \rho[y \mapsto c]) &= \\ dyn(c \mapsto \llbracket M' \rrbracket \rho'[y \mapsto c]) &= \\ \llbracket \lambda y. M' \rrbracket \rho'. \end{aligned} \quad \square$$

Exploiting this result, it is easy to see that β rule is valid. Note that the failure of the β rule is not due to a fail in the Substitution Lemma, but to the fact that in general

$$\llbracket \lambda x. M \ @ \ N \rrbracket \rho \neq \llbracket M \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho]$$

The validity of the β' rule follows easily from a result of the next section: if N is in static normal form, then $\llbracket N \rrbracket \rho \neq \perp$.

3 Adequacy

So far we have seen an operational semantics and a denotational model of 2-level PCF. Now it is time to relate the two semantics and prove an adequacy result. This is achieved using two different logical relations defined by induction on the structure of types. The first one is used to obtain a static adequacy result, and gives a semantic counterpart to the syntactic result of phase distinction between static and dynamic evaluation. The second is easier and shows how the purely dynamic fragment behaves exactly like call-by-name PCF, because there is no way to introduce static undefinedness.

3.1 Adequacy Theorems

We state below two adequacy theorems, leaving the proofs to the end of each section.

Theorem 5 Static Adequacy

Let M be a closed term of type nat , then $M \rightarrow_s^ n$ iff $\llbracket M \rrbracket = n$.*

The Dynamic Adequacy Theorem states that a term of type nat can be compiled if and only if its denotation is not \perp ; moreover, if it can be compiled then it terminates at run-time if and only if its denotation is not \perp .

This is a crucial result, because it shows that the operational distinction between static and dynamic evaluation is respected in the model; in particular, the way how the two relations \rightarrow_s and \rightarrow_d are combined is reflected.

Theorem 6 Dynamic Adequacy

Let M be a closed term of type nat , then

$\llbracket M \rrbracket \neq \perp$ if and only if $M \rightarrow_s^ D$ for some dynamic value D .
Moreover, $\llbracket M \rrbracket = n \in N$ iff $D \rightarrow_d^* \mathit{lift}(n)$.*

3.2 Proof of Static Adequacy

In this section we prove the Static Adequacy Theorem. A logical relation is given to relate the operational semantics and the denotational model in a way that ignores the denotational semantics of dynamic terms.

The following lemma states the independence of static reduction from dynamic terms.

Lemma 7 *Let $\Gamma, x: d \vdash M: t$, and $N: u$.*

If $M[N/x] \rightarrow_s^ M'$ for some term M' , then $M \rightarrow_s^* M''$ for some term M'' .*

Proof. It is enough to observe that a static rewriting can only discard or duplicate a dynamic sub-term, but does not depend on its shape, that is if $M[N/x] \rightarrow_s M'$ then $M \rightarrow_s M''$ and $M' = M''[N/x]$. \square

We now introduce the notion of value: a shape that a term can reach through static reduction if the compilation succeeds. The values V are divided into static (V_s) and dynamic (V_d):

$$\begin{aligned}
V &::= V_s \mid V_d \\
V_s &::= n \mid \lambda x.M \\
V_d &::= x \mid \mathbf{lift}(n) \mid \mathbf{succ} V_d \mid \mathbf{pred} V_d \mid \mathbf{ifz} V_d \mathbf{then} V_d \mathbf{else} V_d \mid \\
&\quad \underline{\lambda}x.V_d \mid V_d \underline{\textcircled{}} V_d \mid \underline{\mathbf{fix}} x.V_d
\end{aligned}$$

The following two lemmas give a characterization of values and a confluence property.

Lemma 8 *A term M is a value if and only if either it is an abstraction or it is in SNF.*

Lemma 9 *If $M \rightarrow_s^* V$ and $M \rightarrow_s^* V'$ then $V = V'$, where equality up to α -equivalence is considered.*

Definition 10 Logical Relation \lesssim_t .

Define a family of relations \lesssim_t between elements of $\llbracket t \rrbracket$ and closed terms of type t . For any $d \in \llbracket t \rrbracket$ and closed $M : t$, define $d \lesssim_t M$ if

- 1 $d = \perp$, or
- 2 exists V such that $M \rightarrow_s^* V$, and $d \lesssim_t V$ where
 - $up(n) \lesssim_{nat} n$
 - $d \lesssim_{nat} V'$, for each V'
 - $f \lesssim_{t_1 \rightarrow t_2} \lambda x.M'$, if for each $c \in \llbracket t_1 \rrbracket$ and $N : t_1$

$$c \lesssim_{t_1} N \Rightarrow f(c) \lesssim_{t_2} (\lambda x.M' N)$$
 - $d \lesssim_{d_1 \Rightarrow d_2} V'$, for each V' .

This relation will be used to prove that if the semantics of a term is not undefined, then it can be statically reduced to a value. This is very important for dynamic terms, because we can show that all their static sub-terms are completely evaluated and discarded during compilation. Note how the relation is essentially independent from the actual interpretation of dynamic constructs, since it only checks if the interpretation is defined.

Lemma 11 *If $c \lesssim_t M'$ and $M \rightarrow_s^* M'$, then $c \lesssim_t M$.*

Proof. The trivial case is when $c = \perp$. If $c \neq \perp$, then there exists a unique (by Lemma 9) V such that $M' \rightarrow_s^* V$. But also $M \rightarrow_s^* M' \rightarrow_s^* V$, hence $c \lesssim_t M$. \square

Lemma 12 *The relation \lesssim_t is monotone and complete: for each type t and closed term $M : t$, the following hold.*

- 1 If $b \lesssim_t M$ and $c \leq b$, then $c \lesssim_t M$.

2 If $(c_i)_{i \in \omega}$ is an ω -chain and $c_i \lesssim_t M$ for each $i \in \omega$, then $\bigsqcup_{i \in \omega} c_i \lesssim_t M$.

Proof. The only interesting case is point 2 when t is of the form $t_1 \rightarrow t_2$. If $\bigsqcup_{i \in \omega} c_i = \perp$, there is nothing to prove. Otherwise there exists $k \in \omega$ such that $d_k \neq \perp$, so $M \rightarrow_s^* V$ for some (unique by Lemma 9) value V . Take $a \in \llbracket t_1 \rrbracket$ and $U: t_1$ such that $a \lesssim_{t_1} U$. By definition for each $i \in \omega$, $c_i(a) \lesssim_{t_2} (V U)$, hence by induction hypothesis $(\bigsqcup_{i \in \omega} c_i)(a) = \bigsqcup_{i \in \omega} (c_i(a)) \lesssim_{t_2} (V U)$. \square

The following lemma is valid in general for open terms, but in the particular case of closed terms it states that the interpretation of a term is related with the term itself. The proof of the various cases is developed in quite a natural way from the previous lemmas. The only exception is the $\underline{\lambda}$ case, where some effort is required.

Lemma 13 Suppose $\Gamma = \vec{x} : \vec{t}$ and $\Gamma \vdash M : t$. If $c_i \in \llbracket t_i \rrbracket$ and $c_i \lesssim_{t_i} M_i$, then

$$\llbracket M \rrbracket[\vec{x} \mapsto \vec{c}] \lesssim_t M[\vec{M}/\vec{x}].$$

Proof. By induction on the structure of M . Let σ be the substitution $[\vec{M}/\vec{x}]$, and let ρ be the environment $[\vec{x} \mapsto \vec{c}]$. We consider only some important cases:

Case $M \equiv \lambda x.M' : t' \rightarrow t''$.

Let f be $\llbracket \lambda x.M' \rrbracket \rho$. If $f = \perp$, the conclusion is immediate.

Otherwise, since $\lambda x.M'$ is a value, we have to show, for each $c \in \llbracket t' \rrbracket$ and $N : t'$ such that $c \lesssim_{t'} N$, that $f(c) \lesssim_{t''} (\sigma(\lambda x.M') N)$.

Taken any such c and N , the induction hypothesis gives

$$\llbracket M' \rrbracket \rho[x \mapsto c] \lesssim_{t''} \sigma[x \mapsto N](M').$$

Since $f(c) = \llbracket M' \rrbracket \rho[x \mapsto c]$, and $(\sigma(\lambda x.M') N) \rightarrow_s^* \sigma[x \mapsto N](M')$ by an application of the β rule, Lemma 11 implies $f(c) \lesssim_{t''} (\sigma(\lambda x.M') N)$.

Case $M \equiv (L N)$.

Suppose that $L : t' \rightarrow t''$ and $N : t'$. Let f be $\llbracket L \rrbracket \rho$, and a be $\llbracket N \rrbracket \rho$. We have to prove that $f(a) \lesssim_{t''} \sigma(L N)$. If $f(a) = \perp$, the conclusion is immediate. Otherwise, also $f \neq \perp$, and by induction hypothesis $\sigma(L) \rightarrow_s^* \lambda x.L'$ for some L' , and for all $c \in \llbracket t' \rrbracket$ and $U : t'$, $c \lesssim_{t'} U \Rightarrow f(c) \lesssim_{t''} (\lambda x.L' U)$. But by induction hypothesis $a \lesssim_{t'} \sigma(N)$, so $f(a) \lesssim_{t''} (\lambda x.L' \sigma(N))$. By operational rules $\sigma(L N) = (\sigma(L) \sigma(N)) \rightarrow_s^* (\lambda x.L' \sigma(N))$, so Lemma 11 gives the result.

Case $M \equiv \underline{\lambda} x.M' : d' \Rightarrow d''$.

Let f be $c \mapsto \llbracket M' \rrbracket \rho[x \mapsto c]$. Then $\llbracket M \rrbracket \rho = \text{dyn}(f)$. If $\text{dyn}(f) = \perp$, the proof is immediate. Otherwise, by definition of dyn , for each $c \neq \perp$, $f(c) \neq \perp$. We have to show that there exists a value V such that $\sigma(\underline{\lambda} x.M') \rightarrow_s^* V$. Because d' is a dynamic type, it can be written as $d_1 \Rightarrow \dots \Rightarrow d_n \Rightarrow \text{nat}$. Consider the term $Z_{d'} \equiv \underline{\lambda} y_1. \dots \underline{\lambda} y_n. \text{lift}(0)$. It is easy to show that $\llbracket Z_{d'} \rrbracket \lesssim_{d'} Z_{d'}$, and $\llbracket Z_{d'} \rrbracket \neq \perp$, hence $f(\llbracket Z_{d'} \rrbracket) \neq \perp$.

By induction hypothesis on M' , $\llbracket M' \rrbracket \rho[x \mapsto \llbracket Z_{d'} \rrbracket] \lesssim_{d'} \sigma[x \mapsto Z_{d'}](M')$. Since $f(\llbracket Z_{d'} \rrbracket) = \llbracket M' \rrbracket \rho[x \mapsto \llbracket Z_{d'} \rrbracket]$ and $f(\llbracket Z_{d'} \rrbracket) \neq \perp$, there exists a value V

such that $\sigma[x \mapsto Z_{d'}](M') \rightarrow_s^* V$, and using Lemma 7 we obtain $\sigma(M') \rightarrow_s^* V'$ for some value V' . By operational rules, $\sigma(\underline{\lambda}x.M') \rightarrow_s^* \underline{\lambda}x.V'$.

Case $M \equiv \mathbf{fix} x.M' : t$.

Let f be $c \mapsto \llbracket M' \rrbracket \rho[x \mapsto c]$, let $e_0 = \perp$ and $e_{i+1} = \llbracket M' \rrbracket \rho[x \mapsto e_i]$. Then $e_i = f^{(i)}(\perp)$, and $\llbracket M \rrbracket \rho = \bigsqcup_{i \in \omega} e_i$.

Clearly $e_0 \lesssim_t \sigma(M)$. Suppose $e_i \lesssim_t \sigma(M)$. Then by induction hypothesis on M' we have $e_{i+1} = \llbracket M' \rrbracket \rho[x \mapsto e_i] \lesssim_t \sigma[x \mapsto \sigma(M)](M')$.

By the operational semantics of \mathbf{fix} , $\sigma(M) \rightarrow_s \sigma[x \mapsto \sigma(M)](M')$, hence by Lemma 11 we obtain $e_{i+1} \lesssim_t \sigma(M)$. We have shown that, for each $i \in \omega$, $e_i \lesssim_t \sigma(M)$, thus, by Lemma 12, $\llbracket M \rrbracket \rho = \bigsqcup_{i \in \omega} e_i \lesssim_t \sigma(M)$. \square

Lemma 14 *Static soundness*

Given a closed term M , if $M \rightarrow_s M'$ then $\llbracket M \rrbracket = \llbracket M' \rrbracket$.

Proof. The proof is a simple case analysis, involving the Substitution Lemma. \square

Proof of Theorem 5

Consider a closed term M of type nat .

By Lemma 13, $\llbracket M \rrbracket \lesssim_{nat} M$, hence if $\llbracket M \rrbracket = n$ then $M \rightarrow_s^* n$.

Conversely, if $M \rightarrow_s^* n$, then $\llbracket M \rrbracket = n$ by Lemma 14. \square

3.3 Proof of Dynamic Adequacy

In this section we prove how the interpretation of purely dynamic terms is essentially the same as that of call-by-name PCF. We define a translation between 2-level PCF and standard PCF, a relation between the two denotational semantics, and conclude with the proof of the Dynamic Adequacy Theorem.

From now on we will refer to dynamic terms and dynamic types simply as terms and types, and introduce a new notation: $\llbracket - \rrbracket^{2LPCF}$ indicates the interpretation of terms of 2-level PCF and $\llbracket - \rrbracket^{PCF}$ the usual interpretation of terms of call-by-name PCF, i.e. the continuous function model over N_\perp given in [19].

We define a translation $\ulcorner - \urcorner$ from dynamic values of 2-level PCF to terms of standard PCF. The translation of types simply removes the annotations.

$$\begin{aligned} \ulcorner \underline{nat} \urcorner &\stackrel{\Delta}{=} nat \\ \ulcorner d_1 \rightarrow d_2 \urcorner &\stackrel{\Delta}{=} \ulcorner d_1 \urcorner \rightarrow \ulcorner d_2 \urcorner \end{aligned}$$

The translation of type assignments is point-wise, and the definition for terms is the following.

$$\begin{aligned}
\lceil x \rceil &\triangleq x \\
\lceil \text{lift}(n) \rceil &\triangleq n \\
\lceil \text{succ } V \rceil &\triangleq \text{succ } \lceil V \rceil \\
\lceil \text{pred } V \rceil &\triangleq \text{pred } \lceil V \rceil \\
\lceil \text{ifz } V \text{ then } V_1 \text{ else } V_2 \rceil &\triangleq \text{ifz } \lceil V \rceil \text{ then } \lceil V_1 \rceil \text{ else } \lceil V_2 \rceil \\
\lceil \lambda x. V \rceil &\triangleq \lambda x. \lceil V \rceil \\
\lceil U @ V \rceil &\triangleq \lceil U \rceil \lceil V \rceil \\
\lceil \text{fix } x. V \rceil &\triangleq \text{fix } x. \lceil V \rceil
\end{aligned}$$

To understand the following logical relation, note that the semantics of a dynamic type d is the lift of the corresponding static type:

$$\llbracket d \rrbracket^{2LPCF} \cong (\llbracket \lceil d \rceil \rrbracket^{PCF})_{\perp}.$$

Definition 15 *Logical relation R .*

For each dynamic type d , define a logical relation R_d between $\llbracket d \rrbracket^{2LPCF}$ and $\llbracket \lceil d \rceil \rrbracket^{PCF}$:

- $b R_{\text{nat}} c$, if $(b = \underline{\perp} \text{ and } c = \perp)$ or $b = \text{up}(\text{up}(n))$ and $c = \text{up}(n)$ for $n \in N$.
- $f R_{d_1 \rightarrow d_2} g$, if for all b and c , $b R_{d_1} c$ implies $\text{dapp}(f, b) R_{d_2} g(c)$

The relation is intended to be used on purely dynamic terms, i.e. terms that compile to themselves, hence are always defined. Thus \perp in 2LPCF is not in relation with anything, and at the upper levels, the relation is essentially one-to-one.

Lemma 16 For each type d , $b R_d c$ if and only if $b = \text{up}(c)$ (or, equivalently, $\text{down}(b) = c$).

Lemma 17 Suppose $\Gamma = x_i : t_i$ and $\Gamma \vdash D : t$. If $b_i R_{t_i} c_i$ for each i , then

$$\llbracket D \rrbracket^{2LPCF} [x_i \mapsto b_i] R_t \llbracket \lceil D \rceil \rrbracket^{PCF} [x_i \mapsto c_i].$$

Proof. By induction on the structure of D . Let ρ be the environment $[x_i \mapsto b_i]$ and let ρ' be the environment $[x_i \mapsto c_i]$. We consider only some key cases:

Case $D \equiv \lambda x. D' : d_1 \rightarrow d_2$.

Let f be $b \mapsto \llbracket D' \rrbracket^{2LPCF} \rho [x \mapsto b]$ and let g be $c \mapsto \llbracket \lceil D' \rceil \rrbracket^{PCF} \rho' [x \mapsto c]$.

Suppose $\text{dyn}(f) = \perp$. Then, by definition of dyn , there exists $a \in \llbracket d_1 \rrbracket^{2LPCF}$ such that $a \neq \perp$ and $f(a) = \perp$, but $a R_{d_1} \text{down}(a)$ by Lemma 16, hence by induction hypothesis on D' we have $f(a) R_{d_2} g(\text{down}(a))$. By Lemma 16, $f(a) = \text{up}(g(\text{down}(a)))$, but $f(a) = \perp$. This is a contradiction, hence we

conclude that $\text{dyn}(f) \neq \perp$.

We have to show $\text{dyn}(f) R_{d_1 \Rightarrow d_2} g$. Suppose $b R_{d_1} c$, then we have to show that $\text{dapp}(\text{dyn}(f), b) R_{d_2} g(c)$.

By induction hypothesis on D' , $f(b) R_{d_2} g(c)$. But $\text{dyn}(f) \neq \perp$ and $b \neq \perp$, since $b = \text{up}(c)$ by Lemma 16, so $\text{dapp}(\text{dyn}(f), b) = f(b)$. We have shown that $\text{dyn}(f) R_{d_1 \Rightarrow d_2} g$.

Case $D \equiv \mathbf{fix} x.D' : d$.

Let f be $\llbracket \lambda x.D' \rrbracket^{2LPCF} \rho$ and g be $\llbracket \lambda x.\ulcorner D' \urcorner \rrbracket^{PCF} \rho'$.

From the $\underline{\lambda}$ case, $\text{dyn}(f) R_{d \Rightarrow d} g$, so $b R_d c$ implies $\text{dapp}(\text{dyn}(f), b) R_d g(c)$; but from Lemma 16, $b = \text{up}(c)$, so $\text{dapp}(\text{dyn}(f), b) = \text{up}(\text{down}(\text{dyn}(f))(c))$, hence $\text{up}(\bigsqcup_{i \in \omega} \text{down}(\text{dyn}(f))^{(i)}(\perp)) = \bigsqcup_{i \in \omega} g^{(i)}(\perp)$. \square

Proof of Theorem 6

Take a closed term M of type nat .

If $\llbracket M \rrbracket \neq \perp$, then by Lemma 13 $M \rightarrow_s^* D$ for some dynamic value D .

Conversely if $M \rightarrow_s^* D$, by Lemma 14 $\llbracket M \rrbracket^{2LPCF} = \llbracket D \rrbracket^{2LPCF}$ and by Lemma 17 $\llbracket D \rrbracket^{2LPCF} \neq \perp$.

Suppose now that $\llbracket M \rrbracket^{2LPCF} = \text{up}(\text{up}(n)) \in N$; then by Lemma 17 $\llbracket \ulcorner M \urcorner \rrbracket^{PCF} = \text{up}(n)$, hence by adequacy of PCF $D \rightarrow_d^* \mathbf{lift}(n)$.

Conversely if $D \rightarrow_d^* \mathbf{lift}(n)$, then by soundness of PCF we have $\llbracket \ulcorner D \urcorner \rrbracket^{PCF} = \text{up}(n)$, and by Lemma 17 $\llbracket D \rrbracket^{2LPCF} = \text{up}(\text{up}(n))$. \square

4 A Model in CPO^\rightarrow

In this final section we introduce a new model for 2-level PCF, showing the advantages of this on the previous one. Then we show the limits of the new model and outline what an ideal model would be.

So far we have seen how to interpret 2-level PCF in the category CPO of complete partial orders and continuous functions. We have seen a function dyn to interpret dynamic types, and its definition was not completely natural. The use of CPO^\rightarrow allows to give a more natural interpretation.

Definition 18 *The category CPO^\rightarrow is defined as follows*

- *Objects are triples (X, A, p) where X and A are CPOs, and $p : X \rightarrow A$ is a continuous function.*
- *Morphisms between (X, A, p) and (Y, B, q) are pairs $\langle f, g \rangle$ such that the following diagram commutes*

$$\begin{array}{ccc} X & \xrightarrow{g} & Y \\ p \downarrow & & \downarrow q \\ A & \xrightarrow{f} & B \end{array}$$

- *The identity on (X, A, p) is the pair $\langle \text{id}_A, \text{id}_X \rangle$.*

- *Composition is point-wise:* $\langle h, l \rangle \circ \langle f, g \rangle = \langle h \circ f, l \circ g \rangle$.

The category CPO^\rightarrow is cartesian closed, where the product is defined point-wise, and the exponential object $(X, A, p) \Rightarrow (Y, B, q)$ is

$$\begin{array}{c} \{\langle f, g \rangle \mid q \circ g = f \circ p\} \\ \downarrow \Pi_1 \\ A \rightarrow B \end{array}$$

where Π_1 is the first projection, and the order in the upper CPO is defined as the conjunction of the two orders between functions.

We use additional notation: $!$ is the unique morphism $X \rightarrow 1$, where 1 is the CPO with one element; $*$ is the element above \perp in 1_\perp ; if $f : X \rightarrow Y$ is a continuous function, then $f_\perp : X_\perp \rightarrow Y_\perp$ is the extension of f that maps \perp to \perp ; $(-\rightarrow -)$ indicates the exponential in CPO and $(-\Rightarrow -)$ the exponential in CPO^\rightarrow .

With this machinery we can give another model of 2-level PCF. The types are interpreted as follows

$$\begin{array}{ccc} \llbracket \text{nat} \rrbracket \triangleq \begin{array}{c} N_\perp \\ \downarrow \text{id} \\ N_\perp \end{array} & \llbracket \underline{\text{nat}} \rrbracket \triangleq \begin{array}{c} N_{\perp\perp} \\ \downarrow !_\perp \\ 1_\perp \end{array} & \llbracket t_1 \rightarrow t_2 \rrbracket \triangleq \llbracket t_1 \rrbracket \Rightarrow \llbracket t_2 \rrbracket \\ \\ \llbracket d_1 \Rightarrow d_2 \rrbracket \triangleq \begin{array}{c} (X \rightarrow Y)_\perp \\ \downarrow !_\perp \\ 1_\perp \end{array} & \text{, where } \begin{array}{c} X_\perp \\ \downarrow !_\perp \\ 1_\perp \end{array} = \llbracket d_1 \rrbracket & \text{ and } \begin{array}{c} Y_\perp \\ \downarrow !_\perp \\ 1_\perp \end{array} = \llbracket d_2 \rrbracket. \end{array}$$

The \rightarrow that CPO^\rightarrow uses for parametrisation can be visualized as

$$\text{static} \xrightarrow{\text{compile}} \text{dynamic}.$$

In the model the top CPO represents the information before compilation, while the bottom one represents the information after compilation. In the nat case, the bottom part says whether the program can be compiled or not.

To give a characterization of the interpretation of dynamic types, consider the notation $\text{in}_\perp^2(X)$, given a CPO X , for

$$\begin{array}{c} X_\perp \\ \downarrow !_\perp \\ 1_\perp \end{array}$$

Then the interpretation of each dynamic type d has the following property:

$$\llbracket d \rrbracket^{\text{CPO}^\rightarrow} \cong \text{in}_\perp^2(\llbracket \ulcorner d \urcorner \rrbracket^{\text{CPO}}).$$

We can now define a family of morphisms dyn' , used to interpret dynamic terms.

Definition 19 For each pair of CPOs X and Y , define

$$\text{dyn}' : (\text{in}'_{\perp}(X) \Rightarrow \text{in}'_{\perp}(Y)) \longrightarrow \text{in}'_{\perp}(X \rightarrow Y)$$

In diagrams this is

$$\left(\begin{array}{c} X_{\perp} \\ \text{!}_{\perp} \downarrow \\ 1_{\perp} \end{array} \Rightarrow \begin{array}{c} Y_{\perp} \\ \text{!}_{\perp} \downarrow \\ 1_{\perp} \end{array} \right) \xrightarrow{\text{dyn}'} \begin{array}{c} (X \rightarrow Y)_{\perp} \\ \downarrow \text{!}_{\perp} \\ 1_{\perp} \end{array}$$

For the nature of the objects involved, it is enough to give the upper function of the morphism, because the lower one is determined by the other.

$$\begin{aligned} \langle \perp, \perp \rangle &\mapsto \langle \perp, \perp \rangle \\ \langle \text{id}, g \rangle &\mapsto \langle *, \text{up}(\text{down} \circ g \circ \text{up}) \rangle \\ \langle \top, g \rangle &\mapsto \langle *, \text{up}(\text{down} \circ g \circ \text{up}) \rangle. \end{aligned}$$

Note that if $f = \perp$, then also $g = \perp$. If $f = \text{id}$ then $g(\perp) = \perp$ and $g(X) \subset Y$. If $f = \top$ then $g(X_{\perp}) \subset Y$.

We will not give the formal interpretation of terms, since it is very similar to the CPO case. Note that considering global elements, i.e. morphisms from 1 to A , the interpretations of base types and of dynamic types in CPO^{\rightarrow} are order-isomorphic to the interpretations in CPO, hence the interpretation of dynamic terms and of most of the static constructs is given essentially in the same way. In particular, a type assignment is interpreted as the product of the interpretations of its types, and static λ -abstraction and application are interpreted in the usual way using the cartesian closed structure of CPO^{\rightarrow} . The remaining cases are the dynamic λ -abstraction that is interpreted like the CPO case but using dyn' instead of dyn , and the static fix point is interpreted using the CPO-enrichment of CPO^{\rightarrow} .

Another example illustrates how the model in CPO^{\rightarrow} is an improvement of the previous one. Consider the type $u \equiv \underline{\text{nat}} \rightarrow \text{nat}$. By definition $\llbracket u \rrbracket^{\text{CPO}} = N_{\perp\perp} \rightarrow N_{\perp}$, while the intuition from the operational semantics suggests that the only definable functions of that type should be the constant ones, because there is no way for a dynamic sub-term to influence the compilation of a static term. On the other side, the elements of $\llbracket u \rrbracket^{\text{CPO}^{\rightarrow}}$ are pairs $\langle f, g \rangle$ such that the following diagram commutes

$$\begin{array}{ccc} N_{\perp\perp} & \xrightarrow{g} & N_{\perp} \\ \text{!}_{\perp} \downarrow & & \downarrow \text{id} \\ 1_{\perp} & \xrightarrow{f} & N_{\perp} \end{array}$$

If $f(\perp) \neq \perp$, then g is a constant function; otherwise g maps \perp to \perp and is constant on the remaining elements.

4.1 Limits of the Model

We have seen how the introduction of the CPO^{\rightarrow} model eliminates some junk present in the CPO one. But the new model is not completely satisfactory, as will be clear from the following example.

Consider the simple language where the only base types are booleans; the interpretation of the type $\underline{\text{bool}} \rightarrow \underline{\text{bool}}$ will contain the pairs $\langle f, g \rangle$ such that

$$\begin{array}{ccc} B_{\perp\perp} & \xrightarrow{g} & B_{\perp\perp} \\ !\perp \downarrow & & \downarrow !\perp \\ 1_{\perp} & \xrightarrow{f} & 1_{\perp} \end{array}$$

commutes, where B is the set $\{\text{true}, \text{false}\}$.

In particular we can have

$$\begin{aligned} f(x) &= *, \text{ for all } x \\ g(\perp) &= \perp \\ g(y) &= \text{true}, \text{ if } y \neq \perp. \end{aligned}$$

Clearly this is not definable, because a term would be required able to distinguish between failure at compile-time and at run-time of its argument. This shows that not all the elements of the model are definable.

As a final remark, we make a case-study of what an optimal interpretation should be for an even easier type. Consider the type $\underline{\text{unit}}$, whose only constant is $*$; we want to study the definable functions of type $\underline{\text{unit}} \rightarrow \underline{\text{unit}}$. To do so we consider the global elements of $\llbracket \underline{\text{unit}} \rightarrow \underline{\text{unit}} \rrbracket^{\text{CPO}^{\rightarrow}}$ and say which ones are definable.

$$\begin{array}{ccc} 1_{\perp\perp} & \xrightarrow{g} & 1_{\perp\perp} \\ !\perp \downarrow & & \downarrow !\perp \\ 1_{\perp} & \xrightarrow{f} & 1_{\perp} \end{array}$$

There are the following cases:

Case $f = \perp$. Then also $g = \perp$ and is defined by $\lambda x. \text{lift}(\text{snd})$.

Case $f = \text{id}$. Then g has a component $g_0 : 1_{\perp} \rightarrow 1_{\perp}$:

- $g_0 = \perp$, defined by $\lambda x. ((\lambda y. \text{dund}) \text{@ } x)$;
- $g_0 = \text{id}$, defined by $\lambda x. x$;
- $g_0 = \top$, defined by $\lambda x. ((\lambda y. \text{lift}(*)) \text{@ } x)$.

Case $f = \top$. Then g has a component $g_1 : 1_{\perp\perp} \rightarrow 1_{\perp}$:

- $g_1(\perp, \perp, *) = (\perp, \perp, \perp)$, defined by $\lambda x. \text{dund}$;

- $g_1(\perp, \perp, *) = (\perp, \perp, *)$, **not** definable;
- $g_1(\perp, \underline{\perp}, *) = (\underline{\perp}, *, *)$, **not** definable;
- $g_1(\perp, \underline{\perp}, *) = (*, *, *)$, defined by $\lambda x.\mathbf{lift}(*)$.

In summary there are 6 definable functions of type $\underline{unit} \rightarrow \underline{unit}$, the CPO^\rightarrow interpretation contains 8 functions, and the CPO one contains 10 functions.

5 Conclusion

In this work we presented an operational and denotational semantics of a 2-level language. Both dynamic and static operational semantics were call-by-name. The denotational model, however, was a bit unusual; the model of the static part is standard, while the model of the dynamic part is the lift of the standard call-by-name model. This lifting was used to capture the interplay between static and dynamic evaluation. The adequacy theorem confirms the accuracy of the model in this regard.

The model, although adequate, is not very precise, because it contains some junk. For example the type $\underline{nat} \rightarrow \underline{nat}$ is interpreted as the CPO $N_{\perp\perp} \rightarrow N_{\perp\perp}$, while the actual definable functions are either undefined or send each defined element to a defined result.

In conclusion, the relation between static and dynamic is subtle and we do not understand it perfectly yet from a semantic point of view. In particular, the nature of $\llbracket - \rightrightarrows - \rrbracket$ is still unclear: it is not the cartesian closed arrow, nor it is the usual interpretation in call-by-value or lazy λ -calculus. A future work would be to clarify its meaning from a categorical point of view.

Another area of further work is the choice of evaluation strategies: we have chosen call-by-name at both static and dynamic levels for simplicity, but we expect the same could work for call-by-value or lazy λ -calculus.

ACKNOWLEDGEMENTS.

I want to thank Peter O’Hearn for the many fruitful discussions. I am also grateful to Eugenio Moggi and the anonymous referees for their useful comments on an early draft of this paper.

Paul Taylor’s macros for diagrams have been used.

References

1. S. Abramsky, Dov. M. Gabbay and T.S.E. Maibaum, editors. Handbook of logic in computer science. Oxford University Press, 1992.
2. P. Benton. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Preliminary Report). University of Cambridge, Computer Laboratory, No. 352, 1994.
3. G. Bierman. What is a Categorical Model of Intuitionistic Linear Logic?. University of Cambridge Computer Laboratory Technical Report 333, April 1994.

4. O. Danvy. Type-directed partial evaluation. In Steele, 1996.
5. O. Danvy. Pragmatics of Type-Directed Partial Evaluation. LNCS 1110, 1996.
6. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial evaluation*, volume 1110 of *LNCS*. Springer-Verlag, 1996.
7. A. Filinski. A Semantic Account of Type-Directed Partial Evaluation. In Principles and Practice of Declarative Programming: International Conference PPDP'99, pp. 378–395, Paris, France (September 1999). Lecture Notes in Computer Science, vol. 1702.
8. C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda calculus. *J. of Func. Prog.*, 1(1), 1991.
9. C. A. Gunter. Semantics of Programming Languages: Structures and Techniques. MIT Press, Foundations of Computing, 1992.
10. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sanella, editor, *ESOP'94*, volume 788 of *LNCS*. Springer Verlag, 1994.
11. J. Hughes. Type specialization for the lambda calculus. In Danvy et al., *Partial Evaluation*, 1996.
12. N.D. Jones, P. Sestoft and H. Søndergaard. An Experiment in Partial Evaluation: the Generation of a Compiler Generator. *Rewriting Techniques and Applications*, Springer Lecture Notes in Computer Science, vol. 202: pp. 124–140, 1985.
13. J. van Leeuwen, editor. Handbook of Theoretical Computer Science. Elsevier, 1990.
14. J. Launchbury. Projection Factorisations in Partial Evaluation. CUP, 1991.
15. E. Moggi. A categorical account of two-level languages. In *MFPS XIII*, ENTCS. Elsevier, 1997.
16. E. Moggi. Functor categories and two-level languages. In *FOSSACS'98*, volume 1378 of LNCS. Springer Verlag, 1998.
17. F. Nielson. Multi-level lambda-calculi. In Danvy et al.
18. F. Nielson and H.R. Nielson. Two-Level Functional Languages. Number 34 in Cambridge Tracts in Theoretical Computer Science. CUP, 1992.
19. G.D. Plotkin. LCF as a programming language. *Theoretical Computer Science* 5, pp. 223–255, 1977.
20. G.L. Steele, editor. *POPL'96*. ACM Press, 1996.