Functor Categories and Two-Level Languages

E. Moggi

DISI - Univ. di Genova, via Dodecaneso 35, 16146 Genova, Italy
phone: +39 10 353-6629, fax: +39 10 353-6699, e-mail: moggi@disi.unige.it

Abstract. We propose a denotational semantics for the two-level language of [GJ91, Gom92], and prove its correctness w.r.t. a standard denotational semantics. Other researchers (see [Gom91, GJ91, Gom92, JGS93, HM94]) have claimed correctness for lambda-mix (or extensions of it) based on denotational models, but the proofs of such claims rely on imprecise definitions and are basically flawed. At a technical level there are two important differences between our model and more naive models in **Cpo**: the domain for interpreting dynamic expressions is more abstract (we interpret code as λ-terms modulo α-conversion), the semantics of *newname* is handled differently (we exploit functor categories). The key idea is to interpret a two-level language in a suitable functor category $\mathbf{Cpo}^{\mathcal{D}^{op}}$ rather than **Cpo**. The semantics of *newname* follows the ideas pioneered by Oles and Reynolds for modeling the stack discipline of Algol-like languages. Indeed, we can think of the objects of $\mathcal{D}$ (i.e. the natural numbers) as the states of a name counter, which is incremented when entering the body of a λ-abstraction and decremented when coming out. Correctness is proved using Kripke logical relations (see [MM91, NN92]).

Introduction

Two-level languages are an important tool for analyzing programs. In the context of partial evaluation they are used to identify those parts of the program that can be reduced statically, and those that have to be evaluated dynamically. We take as representative of these two-level languages that described in [GJ91], which we call PCF_2, since it can be considered as the "PCF of two-level languages". The main aims of this paper are: to point out the flaws in the semantics and correctness proof given in [Gom92], and to propose an alternative semantics for which one can prove correctness.
The interpretation of dynamic λ-abstraction given in [GJ91, Gom92] uses a *newname* construct "informally". Indeed, Gomard and Jones warn that "the generation of new variable names relies on a side-effect on a global state (a name counter). In principle this could have been avoided by adding an extra parameter to the semantic function, but for the sake of notational simplicity we use a less formal solution". Because of this informality, [GJ91, Gom92] are able to use a simplified semantic domain for dynamic expressions, but have to hand wave when it comes to the clause for dynamic λ-abstraction. This informality is maintained also in the correctness proof of [Gom92]. It is possible to fix the informal semantics using a name-counter (as suggested by Gomard and Jones), but then

it is unclear how to fix the correctness proof. In fact, several experts were unable to propose a patch. Lack of precision in the definition of denotational semantics and consequent flaws in correctness proofs are not confined to [Gom92], indeed

- Chapter 4 of [Gom91] and Chapter 8 of [JGS93] contain the same definitions, results and proofs
- [GJ91] quotes the same definitions and results (but without proofs)
- while [HM94] adapts Gomard's technique to establish correctness for a polymorphic binding-time analysis (and introduces further flaws in the denotational semantics).

The specific model we propose is based on a functor category. In denotational semantics functor categories have been advocated by [Ole85] to model Algol-like languages, and more generally they have been used to model locality and dynamic creation (see [OT92, PS93, FMS96]). For this kind of modeling they outperform the more traditional category **Cpo** of cpos (i.e. posets with lubs of ω-chains and ω-continuous maps). Therefore, they are a natural candidate for modeling the *newname* construct of [GJ91].

In the proposed functor category model the domain of residual programs is a bit more abstract than expected, namely α-convertible programs are identified. This identification is necessary for defining the category $\mathcal{D}$ of dynamic expressions, but it seems also a *desirable abstraction*. Functor categories are definitely more complex than **Cpo**, but one can avoid most of the complexities by working in a metalanguage (with computational types). Indeed, it is only in few critical places, where it is important to know which category (and which monad) is used. The graduate textbook [Ten91] gives the necessary background on functor categories for denotational semantics to understand our functor category model. In **Cpo** models the renaming of bound dynamic variables (used in the interpretation of dynamic λ-abstraction) is modeled via a side-effect monad with a name-counter as state, on the contrary in the functor category model renaming is handled by the functor category itself (while non-termination at specialization-time is modeled by the lifting monad).

The paper is organized as follows: Section 1 recall the two-level language of [GJ91, Gom92] which we call PCF_2; Section 2 describes a general way for interpreting PCF_2 via translation into a metalanguage with computational types, and explains what's wrong with previously proposed semantics of PCF_2; Section 3 describes our functor category model for PCF_2 and proves correctness; Section 4 make a comparison of the semantics.

Acknowledgments. I wish to thank Olivier Danvy and Neil Jones for e-mail discussions, which were very valuable to clarify the intended semantics in [GJ91, Gom92], and to identify the critical problem in the correctness proof. Neil Jones has kindly provided useful bibliographic references and made available relevant internal reports.

1 The two-level language of Gomard and Jones

In this section we recall the main definitions in [GJ91, Gom92], namely: the untyped object language λ_o and its semantics, the two-level language PCF_2 and its *semantics* (including with the problematic clause for $\underline{\lambda}$). Both semantics are given via a translation into a metalanguage with computational type (see [Mog91, Mog97b]). In the case of λ_o the monad corresponds to dynamic computations, while in the case of PCF_2 it corresponds to static computations.

1.1 The untyped object language

The object language λ_o is an untyped λ-calculus with a set of ground constants c (which includes the truth values)

$$M ::= x \mid \lambda x.M \mid M_1 @ M_2 \mid \mathit{fix}\ M \mid \mathit{if}\ M_1\ M_2\ M_3 \mid c$$

There is a canonical CBN interpretation of λ_o in $D = (\mathit{const} + (D \rightarrow D))_\bot$, where *const* is the flat cpo of ground constants (ordered by equality). This interpretation can be described via a CBN translation $_^n$ into a suitable metalanguage with computational types, s.t. a λ_o-term M is translated into a meta-term M^n of type TV with $V = \mathit{const} + (TV \rightarrow TV)$, or more precisely $\overline{x} : TV \vdash_{ML} M^n : TV$ when M is a λ_o-term with free variables included in the sequence $\overline{x}$:

- $x^n = x$
- $c^n = [\mathit{inl}(c)]$
- $(\lambda x.M)^n = [\mathit{inr}(\lambda x : TV.M^n)]$
- $(M_1 @ M_2)^n = \text{let } u \Leftarrow M_1{}^n \text{ in case } u \text{ of } \mathit{inr}(f) \Rightarrow f(M_2{}^n)$
  $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _ \Rightarrow \bot$
  where $\bot : TV$ is the least element of TV
- $(\mathit{if}\ M\ M_1\ M_2)^n = \text{let } u \Leftarrow M^n \text{ in case } u \text{ of } \mathit{inl}(\mathit{true}) \Rightarrow M_1{}^n$
  $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{inl}(\mathit{false}) \Rightarrow M_2{}^n$
  $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _ \Rightarrow \bot$
- $(\mathit{fix}\ M)^n = \text{let } u \Leftarrow M^n \text{ in case } u \text{ of } \mathit{inr}(f) \Rightarrow Y(f)$
  $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _ \Rightarrow \bot$
  where $Y : (TV \rightarrow TV) \rightarrow TV$ is the least fixed-point of TV

Note 1. T can be any strong monad on **Cpo** s.t.: (i) each TX has a bottom element $\bot$; (ii) $\text{let } x \Leftarrow \bot \text{ in } e = \bot$, i.e. $\bot$ is preserved by $f^* : TX \rightarrow TY$ for any $f : X \rightarrow TY$. With these properties one can interpret recursive definitions of programs and solve domain equations involving T.

The interpretation by Gomard amounts to take $TX = X_\bot$ and $D = TV$.

1.2 The two-level language PCF_2

The two-level language PCF_2 can be described as a simply typed λ-calculus over the base types *base* and *code* with additional operations. The raw syntax of PCF_2 is given by

- types τ ::= $base \mid code \mid \tau_1 \rightarrow \tau_2$
- terms e ::= $x \mid \lambda x:\tau.e \mid e_1@e_2 \mid fix_\tau\ e \mid if_\tau\ e_1\ e_2\ e_3 \mid c \mid$
  $lift\ e \mid \underline{\lambda} x.e \mid e_1\underline{@}e_2 \mid \underline{fix}\ e \mid \underline{if}\ e_1\ e_2\ e_3 \mid \underline{c}$

The well-formed terms of PCF_2 are determined by assigning types to constants:

- $fix_\tau : (\tau \rightarrow \tau) \rightarrow \tau$
- $if_\tau : base, \tau, \tau \rightarrow \tau$
- $c : base$
- $lift : base \rightarrow code$
- $\underline{\lambda} : (code \rightarrow code) \rightarrow code$, following Church we have taken $\underline{\lambda}$ to be a higher order constant rather than a binder (all the binding is done by λ). The two presentations are equivalent: the term $\underline{\lambda} x.e$ of [GJ91] can be replaced by $\underline{\lambda}(\lambda x : code.e)$, while the constant $\underline{\lambda}$ can be defined as $\lambda f : code \rightarrow code.\underline{\lambda} x.f@x$.
- $\underline{@} : code, code \rightarrow code$
- $\underline{fix} : code \rightarrow code$
- $\underline{if} : code, code, code \rightarrow code$
- $\underline{c} : code$

Remark. The language PCF_2 corresponds to the well-annotated expressions of Gomard and Jones. For two-level languages with dynamic type constructors (e.g. that in [HM94]) it is necessary to distinguish between static and dynamic types. In PCF_2 the only dynamic type is *code*, and there is no need to make this explicit.

2 Models of PCF_2 in Cpo

The interpretation of PCF_2 is described by a translation $_^s$ into a suitable metalanguage with computational types, s.t. $x_1 : \tau_1{}^s, \ldots, x_n : \tau_n{}^s \vdash_{ML} e^s : \tau^s$ when $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{PCF_2} e : \tau$. The translation highlights that static computations take place only at ground types (just like in PCF and Algol).

- $base^s = T(const)$, where *const* is the flat cpo of ground constants
- $code^s = T(exp)$, where *exp* is the flat cpo of open λ_o-terms with (free and bound) variables included in $var = \{x_n \mid n \in N\}$. When translating terms of PCF_2 we make use of the following expression-building operations:
  - *build_const* : $const \rightarrow exp$ is the inclusion of ground constants into terms
  - *build_var* : $var \rightarrow exp$ is the inclusion of variables into terms.
  - *build_@* : $exp, exp \rightarrow exp$ is the function $M_1, M_2 \mapsto M_1@M_2$ which builds an application. There are similar definitions for *build_fix* and *build_if*.

- $build_\lambda : var, exp \to exp$ is the function $x, M \mapsto \lambda x.M$ which builds a λ-abstraction.

– $(\tau_1 \to \tau_2)^s = {\tau_1}^s \to {\tau_2}^s$
– $x^s = x$
– $c^s = [c]$
– $(\lambda x : \tau.e)^s = \lambda x : \tau^s.e^s$
– $(e_1 @ e_2)^s = {e_1}^s @ {e_2}^s$
– $(if_\tau\ e\ e_1\ e_2)^s = \text{let } u \Leftarrow e^s \text{ in case } u \text{ of } \begin{array}{l} true \Rightarrow {e_1}^s \\ false \Rightarrow {e_2}^s \\ _ \Rightarrow \bot \end{array}$
  where $\bot$ is the least element of τ^s
– $(fix_\tau\ e)^s = Y(e^s)$, where Y is the least fixed-point of τ^s
– $(\ell ift\ e)^s = \text{let } x \Leftarrow e^s \text{ in } [build_const(x)]$
– $\underline{c}^s = [build_const(c)]$
– $(\underline{op}\ \overline{e})^s = \text{let } \overline{M} \Leftarrow \overline{e}^s \text{ in } [build_op\ \overline{M}]$, where $op \in \{fix, @, if\}$
* $(\underline{\lambda}\ e)^s = \text{let } x \Leftarrow newname \text{ in let } M \Leftarrow e^s([build_var(x)]) \text{ in } [build_\lambda(x, M)]$
  where $newname : T(var)$ generates a *fresh variable* of the object language.

The monad T for static computations should satisfy the same additional properties stated in Note 1.

Remark. In the above interpretation/translation the meaning of *newname* (and $\underline{\lambda}$) is not fully defined, indeed one should fix first the interpretation of computational types TX.
The interpretation of [GJ91, Gom92] uses *simplified* semantic domains (which amount to use the lifting monad $TX = X_\bot$), but with these domains there is no way of interpreting *newname* (consistently with the informal description). Therefore, most of the stated results and proofs are inherently faulty.
Gomard and Jones are aware of the problem and say that "the generation of new variables names relies on a side effect on a global state (a name-counter)...but for the sake of notational simplicity we have used a less formal solution". Their proposed solution amounts to use a side-effect monad $TX = (X \times N)_\bot^N$, and to interpret $newname : T(var)$ as $newname = \lambda n : N.up(\langle x_n, n+1 \rangle)$, where $up_X : X \to X_\bot$ is the inclusion of X into its lifting.
A simpler solution, suggested by Olivier Danvy, uses a state-reader monad $TX = X_\bot^N$. In this case one can interpret the operation $newname'_X : (TX)^{var} \to TX$ as $newname'_X(f) = \lambda n : N.fx_n(n+1)$, and use it for translating $\underline{\lambda}$

– $(\underline{\lambda}\ e)^s = newname'_{exp}(\lambda x : var.\text{let } M \Leftarrow e^s([build_var(x)]) \text{ in } [build_\lambda(x, M)])$.

The only place where a name-counter is really needed is for generating code, so we could use the simpler translation $base^s = const_\bot$ and $code^s = T(exp)$. This is similar to what happens in Algol, where expressions cannot have side-effects, while commands can.

2.1 Correctness: attempts and failures

Informally speaking, correctness for PCF_2 should say that for any $\emptyset \vdash_{PCF_2} e : code$ if the static evaluation of e terminates and produces a λ_o-term M : exp, then λ_o-terms M and e^{ϕ} are *equivalent*, where $_^{\phi}$ is the translation from PCF_2 to λ_o erasing types and annotations. In fact, this is an over-simplified statement, since one want to consider PCF_2-terms $\overline{x} : code \vdash_{PCF_2} e : code$ with free dynamic variables.
In a denotational setting one could prove correctness by defining a logical relation (see [MW85, Mit96]) between two interpretations of PCF_2

$$\begin{array}{ccc} PCF_2 & \xrightarrow{\phi} & \lambda_o \\ {\scriptstyle s}\downarrow & \nearrow_{R} & \downarrow{\scriptstyle I^o} \\ ML_T(\Sigma) & \xrightarrow[I]{} & \mathbf{Cpo} \end{array}$$

The parameterized logical relation $R^{\tau}_{\rho} \subseteq [\![\tau^s]\!] \times D$, where $\rho : var \rightharpoonup D$, proposed by [Gom92] is defined as follows

- $\bot\ R^{base}_{\rho}\ d$ and $up(b)\ R^{base}_{\rho}\ d \overset{\Delta}{\iff} d = up(in_1\ b)$
- $\bot\ R^{code}_{\rho}\ d$ and $up(M)\ R^{code}_{\rho}\ d \overset{\Delta}{\iff} d = [\![M]\!]^o_{\rho}$
- $f\ R^{\tau_1 \to \tau_2}_{\rho}\ d \overset{\Delta}{\iff} x\ R^{\tau_1}_{\rho}\ y \supset (f@x)\ R^{\tau_2}_{\rho}\ (d@^o y)$, this is the standard way of defining at higher types a logical relation between typed applicative structures.

Gomard interprets types according to the informal semantics, i.e. $[\![base^s]\!] = const_{\bot}$ and $[\![code^s]\!] = exp_{\bot}$. According to the fundamental lemma of logical relations, if the two interpretations of each operation/constant of PCF_2 are logically related, then the two interpretations of each PCF_2-term are logically related. It is easy to do this check for all operations/constants except $\underline{\lambda}$. In the case of $\underline{\lambda}$ one can only hand wave, since the interpretation is informally given. Therefore, Gomard concludes that he has proved correctness.

Remark. Gomard does not mention explicitly logical relations. However, his definition of R is given by induction on the structure of PCF_2-types, while correctness is *proved* by induction of the structure PCF_2-terms $\Gamma \vdash_{PCF_2} e : \tau$. This is typical of logical relations.

In order to patch the proof one would have to change the definition of R^{code}_{ρ}, since in the intended semantics $[\![code^s]\!] = exp^N_{\bot}$ or $(exp \times N)^N_{\bot}$, and check the case of $\underline{\lambda}$ (which now has an interpretation). We doubt that this can be done, for the following reasons (for simplicity we take $[\![code^s]\!] = exp^N_{\bot}$):

- The interpretation of $\underline{\lambda}$ may capture variables that ought to remain free. For instance, consider the interpretation of $x : code \vdash_{PCF_2} \underline{\lambda} y.x : code$, which is a function $f : exp_\perp^N \to exp_\perp^N$, and the element $[M] = \lambda n.up(M)$ of $exp_\perp^N$, then $f([M]) = \lambda n.up(\lambda x_n.M)$ (here there is some overloading in the use of λ, since λn is a semantic lambda while λx_n is syntactic). Depending on the choice of n we may bind a variable free in M, therefore the semantics of $\underline{\lambda}$ fails to ensure *freshness* of x_n.
- The semantic domain $exp_\perp^N$ has junk elements in comparison to $exp_\perp$, and so there are several ways of defining $u\ R_\rho^{code}\ d$, e.g.
  - $\forall n : N.\forall M : exp.u(n) = up(M) \supset [\![M]\!]_\rho^o = d$
  - $\exists n : N.\forall M : exp.u(n) = up(M) \supset [\![M]\!]_\rho^o = d$
  - $\exists M : exp.\forall n : N.u(n) \equiv_\alpha up(M) \supset [\![M]\!]_\rho^o = d$

  but none of them works (nor is more canonical than the others).

If there is a way to prove correctness using (Kripke) logical relations, it is likely to involve something more subtle than parameterization by $\rho : var \to D$.

3 A functor category model of PCF_2

In this section we define a categorical model of PCF_2 in a **Cpo**-enriched functor category $\widehat{\mathcal{D}} = \mathbf{Cpo}^{\mathcal{D}^{op}}$, where $\mathcal{D}$ is a *syntactic* category corresponding to λ_o, and the objects of $\mathcal{D}$ can be viewed as states of a name-counter. The main property of this model is that the hom-set $\widehat{\mathcal{D}}(exp^n, exp)$ is isomorphic to the set of λ_o-terms modulo α-conversion whose free variables are included in $\{x_0, \ldots, x_{n-1}\}$.

3.1 The dynamic category

We define $\mathcal{D}$ like the category associated to an algebraic theory (as proposed by Lawvere in [Law63]), i.e.:

- an object of $\mathcal{D}$ is a natural number; we identify a natural number n with the set $\{0, \ldots, n-1\}$ of its predecessors;
- an arrow from m to n, which we call **substitution**, is a function $\sigma : n \to \Lambda(m)$, where $\Lambda(m)$ is the set of λ_o-terms modulo α-conversion with free variables included in $\{x_0, \ldots, x_{m-1}\}$; thus $\mathcal{D}(m, n) = \Lambda(m)^n$;
- composition is given by composition of substitutions with renaming of bound variables (which is known to respect α-conversion). Namely, for $\sigma_1 : m \to n$ and $\sigma_2 : n \to p$ the substitution $(\sigma_2 \circ \sigma_1) : m \to p$ is given by $(\sigma_2 \circ \sigma_1)(i) = N_i[\sigma_1]$, where $i \in p$, $N_i = \sigma_2(i) \in \Lambda(n)$, $N_i[\sigma_1] \in \Lambda(m)$ is the result of applying in *parallel* to N_i the substitutions $x_j := M_j$ with $j \in m$.
  Identities are given by identity substitutions $id : n \to \Lambda(n)$.

It is easy to see that $\mathcal{D}$ has finite products: the terminal object is 0, and the product of m with n is $m + n$. Therefore, the object n is the product of n copies of the object 1, moreover $\mathcal{D}(m, 1) = \Lambda(m)$.

Remark. We can provide an informal justification for the choice of $\mathcal{D}$. The objects of $\mathcal{D}$ correspond to the states of a name-counter: state m means that m names, say $x_0, \ldots, x_{m-1}$, have been created so far.
For the choice of morphisms the justification is more technical: it is *almost* forced when one wants $\widehat{\mathcal{D}}(exp^m, exp)$ to be isomorphic to the set of λ_o-terms whose free variables are included in $\{x_0, \ldots, x_{m-1}\}$. In fact, the natural way of interpreting exp in $\widehat{\mathcal{D}}$ is with a functor s.t. $exp(m)$ = the set of λ_o-terms with free names among those available at state m. If we require $F = Y(1)$, i.e. the image of $1 \in \mathcal{D}$ via the Yoneda embedding, and m to be the product in $\mathcal{D}$ of m copies of 1, then we have $\widehat{\mathcal{D}}(exp^m, exp) = \widehat{\mathcal{D}}(Y(1)^m, Y(1)) = \widehat{\mathcal{D}}(Y(m), Y(1)) = \mathcal{D}(m, 1) = exp(m)$. Therefore, we can conclude that $\mathcal{D}(m, n) = exp(m)^n$. Moreover, to define composition in $\mathcal{D}$ we are forced to take λ_o-terms modulo α-conversion.

3.2 The static category

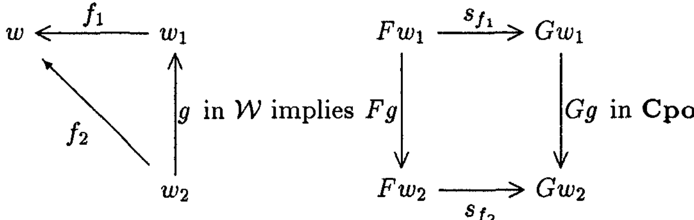
We define $\widehat{\mathcal{D}}$ as the functor category $\mathbf{Cpo}^{\mathcal{D}^{op}}$, which is a variant of the more familiar topos of presheaves $\mathbf{Set}^{\mathcal{D}^{op}}$. Categories of the form $\widehat{\mathcal{W}}$ (where $\mathcal{W}$ is a small category) have been used in [Ole85] for modeling local variables in Algol-like languages. $\widehat{\mathcal{W}}$ enjoys the following properties:

- it has small limits and colimits (computed pointwise), and exponentials;
- it is **Cpo**-enriched, thus one can interpret fix-point combinators and solve recursive domain equations by analogy with **Cpo**;
- there is a full and faithful embedding $Y : \mathcal{W} \to \widehat{\mathcal{W}}$, which preserves limits and exponentials. This is basically the Yoneda embedding $Y(w) = \mathcal{W}(_, w)$.
- the functor $\Delta : \mathbf{Cpo} \to \widehat{\mathcal{W}}$ s.t. $(\Delta X)(_) = X$ has left and right adjoints.

Since $\mathcal{D}$ has a terminal object, $\Delta : \mathbf{Cpo} \to \widehat{\mathcal{D}}$ is full and faithful, and its right adjoint is the global section functor $\Gamma : \widehat{\mathcal{D}} \to \mathbf{Cpo}$ s.t. $\Gamma F = \widehat{\mathcal{D}}(1, F) = F(0)$.
A description of several constructions in $\widehat{\mathcal{W}}$ relevant for denotational semantics can be found in [Ten91]. Here we recall only the definition of exponentials.

Definition 2. The exponential object G^F in $\widehat{\mathcal{W}}$ is the functor s.t.

- $G^F(w)$ is the cpo of families $s \in \prod_{f:w' \to w} \mathbf{Cpo}(Fw', Gw')$ ordered pointwise and satisfying the **compatibility condition**



- $(G^F f s)_g = s_{f \circ g}$ for any $w'' \xrightarrow{g} w' \xrightarrow{f} w$ in $\mathcal{W}$.

We recall also the notion of ω-inductive relation in a **Cpo**-enriched functor category $\widehat{\mathcal{W}}$, which is used in the correctness proof.

Definition 3. Given an object $X \in \widehat{\mathcal{W}}$, a (unary) ω-inductive relation $R \subseteq X$ in $\widehat{\mathcal{W}}$ consists of a family $\langle R_w \subseteq Xw | w \in \mathcal{W}\rangle$ of ω-inductive relations in **Cpo** satisfying the **monotonicity condition**:

- $f : w' \to w$ in $\mathcal{W}$ and $x \in R_w \subseteq Xw$ implies $Xfx \in R_{w'} \subseteq Xw'$.

3.3 Interpretation of PCF_2

By analogy with Section 1, we parameterize the interpretation of PCF_2 in $\widehat{\mathcal{D}}$ w.r.t. a strong monad T on **Cpo** satisfying the additional properties stated in Note 1. Any such T induces a strong monad $T^{\mathcal{D}^{op}}$ on $\widehat{\mathcal{D}}$ satisfying the same additional properties. With some abuse of language we write T for its *pointwise extension* $(T^{\mathcal{D}^{op}}F)(m) = T(F(m))$.
In the proof of correctness we take $TX = X_\perp$, since the monad has to account only for the possibility of non-termination at specialization-time, while the interpretation of $\underline{\lambda}$ exploits only the functor category structure (and not the monad, as done for the interpretations in **Cpo**).
Also in this case the interpretation of PCF_2 can be described by a *standard* translation $_^s$ into a suitable metalanguage with computational types (which play only a minor role). The key differences w.r.t. the interpretation/translation of Section 2 are: the interpretation of *exp* (which is not the image of a cpo via the functor Δ), and the expression-building operation *build_*λ (which has type $(exp \to exp) \to exp$, as expected in a higher-order syntax encoding of λ_o).

- $base^s = T(\Delta(const))$, where *const* is the flat cpo of ground constants. Therefore, $base(n) = T(const)$ and so global elements of *base* correspond to elements of the cpo $T(const)$.
- $code^s = T(exp)$, where $exp = Y(1)$, i.e. the image of $1 \in \mathcal{D}$ via the Yoneda embedding $Y : \mathcal{D} \to \widehat{\mathcal{D}}$. Therefore, $exp(n) = \Lambda(n)$ and $code(n) = T(\Lambda(n))$. It is also immediate to show that $\widehat{\mathcal{D}}(exp^n, exp)$ is isomorphic to $\Lambda(n)$:
  - $\widehat{\mathcal{D}}(Y(1)^n, Y(1)) \cong$ because Y preserves finite products
  - $\widehat{\mathcal{D}}(Y(n), Y(1)) \cong$ because Y is full and faithful
  - $\mathcal{D}(n, 1) \cong \Lambda(n)$ by definition of $\mathcal{D}$.

  When translating terms of PCF_2 we make use of the following expression-building operations (which are interpreted by morphisms in $\widehat{\mathcal{D}}$, i.e. natural transformation):
  - *build_const* : $\Delta(const) \to exp$ s.t. $build_const_n : const \to \Lambda(n)$ is the obvious inclusion of ground constants. Alternatively, one can define *build_const* via the isomorphism $\widehat{\mathcal{D}}(\Delta(const), exp) \cong \mathbf{Cpo}(const, \Lambda(0))$ induced by the adjunction $\Delta \dashv \Gamma$.
  - *build_@* : $exp, exp \to exp$ s.t. $build_@_n : \Lambda(n), \Lambda(n) \to \Lambda(n)$ is the function $M_1, M_2 \mapsto M_1@M_2$ which builds an application. Alternatively, one can define *build_@* as the natural transformation corresponding to the term $x_0@x_1 \in \Lambda(2)$, via the isomorphism $\widehat{\mathcal{D}}(exp^2, exp) \cong \Lambda(2)$. There are similar definitions for *build_fix* and *build_if*.
  - *build_*λ : $exp^{exp} \to exp$ is the trickiest part and is defined below.

- the interpretation of static operations/constants is obvious, in particular we have least fixed-points because $\widehat{\mathcal{D}}$ is **Cpo**-enriched.
- $(\mathit{lift}\ e)^s = \text{let } x \Leftarrow e^s \text{ in } [\mathit{build_const}(x)]$
- $\underline{c}^s = [\mathit{build_const}(c)]$
- $(\underline{op}\ \overline{e})^s = \text{let } \overline{M} \Leftarrow \overline{e}^s \text{ in } [\mathit{build_op}\ \overline{M}]$, where $op \in \{\mathit{fix}, @, \mathit{if}\}$

* $\underline{\lambda} : \mathit{code}^{\mathit{code}} \to \mathit{code}$ is defined in terms of $\mathit{build_\lambda} : \mathit{exp}^{\mathit{exp}} \to \mathit{exp}$ as explained below.

To define the components of the natural transformation $\mathit{build_\lambda} : \mathit{exp}^{\mathit{exp}} \to \mathit{exp}$ we use the following fact, which is an easy consequence of Yoneda's lemma.

Lemma 4. *For any $u \in \mathcal{W}$ and $F \in \widehat{\mathcal{W}}$ there is a natural isomorphism between the functors $F^{Y(u)}$ and $F(_ \times u)$.*

By Lemma 4, $\mathit{build_\lambda}$ amounts to a natural transformation from $\mathcal{D}(_ + 1, 1)$ to $\mathcal{D}(_, 1)$. We describe $\mathit{build_\lambda}$ through a diagram:

$$
\begin{array}{cccc}
m & M \in \Lambda(m+1) & \xmapsto{\mathit{build_\lambda}_m} & (\lambda x_m.M) \in \Lambda(m) \\
\uparrow \sigma \in \mathcal{D} & \downarrow _ \circ (\sigma + 1) & \text{in } \mathbf{Cpo} & \downarrow _ \circ \sigma \\
n & M[\sigma + 1] \in \Lambda(n+1) & \xmapsto[\mathit{build_\lambda}_n]{} & (\lambda x_n.M)[\sigma] \in \Lambda(n)
\end{array}
$$

Observe that $\mathcal{D}(_, 1) = \Lambda(_)$, the substitution $(\sigma + 1) : m + 1 \to \Lambda(n+1)$ is like σ on m and maps m to x_n, while the commutativity of the diagram follows from $(\lambda x_n.M[\sigma + 1]) \equiv_\alpha (\lambda x_m.M)[\sigma]$.
To define $\underline{\lambda} : T(\mathit{exp})^{T(\mathit{exp})} \to T(\mathit{exp})$ we need the following lemma.

Lemma 5. *For any functor $T : \mathbf{Cpo} \to \mathbf{Cpo}$, $u \in \mathcal{W}$ and $F \in \widehat{\mathcal{W}}$ there is a natural isomorphism between the functors $(TF)^{Y(u)}$ and $T(F^{Y(u)})$.*

Proof. For any $v \in \mathcal{W}$ we give an isomorphism between $(TF)^{Y(u)}(v)$ and $T(F^{Y(u)})(v)$:

- $(TF)^{Y(u)}(v) =$ by Lemma 4
- $(TF)(u \times v) =$ since T is extended pointwise to $\widehat{\mathcal{W}}$
- $T(F(u \times v)) =$ by Lemma 4
- $T(F^{Y(u)}(v)) =$ since T is extended pointwise to $\widehat{\mathcal{W}}$
- $T(F^{Y(u)})(v)$

It is immediate to see that this family of isomorphisms is natural in v.

By exploiting the isomorphism $i : T(\mathit{exp})^{\mathit{exp}} \to T(\mathit{exp}^{\mathit{exp}})$ given by Lemma 5, one can define $\underline{\lambda} : T(\mathit{exp})^{T(\mathit{exp})} \to T(\mathit{exp})$ in a metalanguage with computational types as

$$\underline{\lambda}(f) = \text{let } f' \Leftarrow i(\lambda x : \mathit{exp}.f([x])) \text{ in } [\mathit{build_\lambda}(f')]$$

Remark. The category $\widehat{\mathcal{D}}$ has two full sub-categories $\mathcal{D}$ and **Cpo**, which have a natural interpretation: $\mathcal{D}$ corresponds to dynamic types, while **Cpo** corresponds to pure static types, i.e. those producing no residual code at specialization time (e.g. *base*). A key property of pure static expressions is that they cannot depend on dynamic expressions. Semantically this means that the canonical map $(\Delta X) \to (\Delta X)^{Y(u)}$, i.e. $x \mapsto \lambda y : Y(u).x$, is an isomorphism. In fact, by Lemma 4 $(\Delta X)^{Y(u)}$ is naturally isomorphic to $(\Delta X)(_ \times u)$, which is (ΔX).

3.4 Correctness and logical relations

The semantics for the two-level language PCF_2 was used in [GJ91, Gom92] to prove a correctness theorem for partial evaluation. The correctness theorem relates the interpretation I^o of the object language λ_o in **Cpo** to the interpretation I^2 of the two-level language PCF_2 in $\widehat{\mathcal{D}}$.
The first step is to define a translation $_^{\phi}$ from PCF_2 to λ_o, i.e. $\overline{x} : \overline{\tau} \vdash_{PCF_2} e : \tau$ implies $\overline{x} \vdash_{\lambda_o} e^{\phi}$, which erases types and annotations, so $(\lambda x : \tau.e)^{\phi} = \lambda x.e^{\phi}$, $(op_{\tau}\ \overline{e})^{\phi} = op\ \overline{e}^{\phi}$, $(\underline{op}\ \overline{e})^{\phi} = op\ \overline{e}^{\phi}$ and $(\ell ift\ e)^{\phi} = e^{\phi}$. By composing the translation ϕ with the interpretation I^o we get an interpretation of I^1 of PCF_2 in **Cpo**, where every type is interpreted by the cpo $D = (const + (D \to D))_{\bot}$.
At this stage we can state two correctness criteria (the first being a special case of the second), which exploit in an essential way the functor category structure:

- Given a closed PCF_2-expression $\emptyset \vdash e : code$, its I^2 interpretation is a global element d of $exp_{\bot} \in \widehat{\mathcal{D}}$, and therefore $d_0 \in \Lambda(0)_{\bot}$. Correctness for e means: $d_0 = up(M)$ implies $[\![M]\!]^o = [\![e^{\phi}]\!]^o \in D$, for any $M \in \Lambda(0)$.
- Given an open PCF_2-expression $\overline{x} : code \vdash e : code$ where $\overline{x} = x_0, \ldots, x_{n-1}$, its I^2 interpretation is a morphism $f : exp_{\bot}^n \to exp_{\bot}$, and therefore $f_n : \Lambda(n)_{\bot}^n \to \Lambda(n)_{\bot}$. Correctness for e means: $f_n(up(x_0), \ldots, up(x_{n-1})) = up(M)$ implies $[\![\overline{x} \vdash M]\!]^o = [\![\overline{x} \vdash e^{\phi}]\!]^o : D^n \to D$, for any $M \in \Lambda(n)$.

The proof of correctness requires a stronger result, which amounts to prove that *the two interpretations of* PCF_2 *are logically related*. However they live in different categories. Therefore, before one can relate them via a (Kripke) logical relation R between typed applicative structures (see [MM91]), they have to be moved (via limit preserving functors) to a common category $\widehat{\mathcal{E}}$.

$$
\begin{array}{ccc}
PCF_2 & \xrightarrow{I^1} & \hat{1} = \mathbf{Cpo} \\
{\scriptstyle I^2}\downarrow & \overset{R}{\nearrow\!\!\swarrow} & \downarrow {\scriptstyle \hat{!} = \Delta} \\
\widehat{\mathcal{D}} & \xrightarrow[\hat{\pi}]{} & \widehat{\mathcal{E}}
\end{array}
$$

- $\mathcal{E}$ is the category whose objects are pairs $\langle m \in \mathcal{D}, \rho \in D^m \rangle$, while morphisms from $\langle m, \rho \rangle \to \langle n, \rho' \rangle$ are those $\sigma : m \to n$ in $\mathcal{D}$ s.t. $\rho' = [\![\sigma]\!]_{\rho}$
- $\pi : \mathcal{E} \to \mathcal{D}$ is the obvious projection functor $\langle m, \rho \rangle \mapsto m$.

The Kripke logical relation R is a family of ω-inductive relations (see Definition 3) R^τ in $\hat{\mathcal{E}}$ defined by induction on the structure of types τ in PCF_2.

base $R^{base}_{\langle m,\rho\rangle} \subset const_\perp \times D$ s.t. $\perp R_{\langle m,\rho\rangle} d$ and $up(c) R_{\langle m,\rho\rangle} d \overset{\Delta}{\Longleftrightarrow} d = up(inl\ c)$

code $R^{code}_{\langle m,\rho\rangle} \subset \Lambda(m)_\perp \times D$ s.t. $\perp R_{\langle m,\rho\rangle} d$ and $up(M) R_{\langle m,\rho\rangle} d \overset{\Delta}{\Longleftrightarrow} d = [\![M]\!]_\rho$

We must check that R^{code} satisfies the monotonicity property of a Kripke relation, i.e. $\sigma : \langle m,\rho\rangle \to \langle n,\rho'\rangle$ in $\mathcal{E}$ and $up(M) R^{code}_{\langle n,\rho'\rangle} d$ implies $up(M[\sigma]) R^{code}_{\langle m,\rho\rangle} d$. This follows from $\rho' = [\![\sigma]\!]_\rho$, i.e. from the definition of morphism in $\mathcal{E}$, and $[\![M[\sigma]]\!]_\rho = [\![M]\!]_{[\![\sigma]\!]_\rho}$, i.e. the substitution lemma for the interpretation of λ_o. More diagrammatically this means

$$\begin{array}{cccc}
\mathcal{D} & \mathcal{E} & code\quad R^{code} & D \\
m & \langle m,\rho\rangle & up(M[\sigma]) R_{\langle m,\rho\rangle} & [\![M[\sigma]]\!]_\rho \\
\sigma\downarrow & \sigma\downarrow & \uparrow code(\sigma) & \| \\
n & \langle n,\rho'\rangle & up(M)\ \ R_{\langle n,\rho'\rangle} & [\![M]\!]_{\rho'} = d
\end{array}$$

The family R on functional types is defined (in the internal language) in the standard way, i.e. $f R^{\tau_1\to\tau_2} g \overset{\Delta}{\Longleftrightarrow} \forall x,y.x R^{\tau_1} y \supset f @^2 x R^{\tau_2} g @^1 y$, where $@^i$ is the binary application of the applicative structure used for the interpretation I^i. The definition of the Kripke logical relation at types *base* and *code* says that partial evaluation is only partially correct, namely if it terminates it gives the expected result.

By the fundamental lemma of logical relations, to prove that the interpretations I^1 and I^2 of PCF_2 are logically related it suffices to show that the interpretation of all higher-order constants (besides @ and λ) are logically related. This is a fairly straightforward check, therefore we consider only few cases, including the critical one of dynamic λ-abstraction.

$\underline{@}$ Since $\underline{@}^2$ is strict, we need to prove only that $up(M_i) R_{\langle m,\rho\rangle} d_i$ (for $i = 1,2$) implies $up(M_1)\underline{@}^2 up(M_2) \overset{\Delta}{=} up(M_1 @ M_2) R_{\langle m,\rho\rangle} d_1 @^1 d_2 \overset{\Delta}{=} d_1 \underline{@}^1 d_2$

By definition of R at type *code*, we have to prove that $[\![M_1 @ M_2]\!]_\rho = d_1 @^1 d_2$

- $[\![M_i]\!]_\rho = d_i$, because $up(M_i) R_{\langle m,\rho\rangle} d_i$
- $[\![M_1 @ M_2]\!]_\rho = @^1([\![M_1]\!]_\rho, [\![M_2]\!]_\rho)$, by definition of I^1
- therefore $[\![M_1 @ M_2]\!]_\rho = d_1 @^1 d_2$

fix_τ We need to prove that $f R^{\tau\to\tau} g$ implies $(\sqcup_i x_i) R^\tau (\sqcup_i y_i)$, where $x_0 = y_0 = \perp$ and $x_{i+1} = f @^2 x_i$ and $y_{i+1} = g @^1 y_i$.

This follows immediately from ω-inductivity of R^τ, i.e.

- $\perp R^\tau \perp$ and
- $(\sqcup_i x_i) R^\tau (\sqcup_i y_i)$ when $x_{i\in\omega}$ and $y_{i\in\omega}$ are ω-chains and $\forall i. x_i R^\tau y_i$

ω-inductivity of R^τ can be proved by a straightforward induction on τ.

$\underline{\lambda}$ The case of $\underline{\lambda} : (code \to code) \to code$ is the most delicate one. Suppose that $f R^{code \to code}_{(m,\rho)} g$, we have to prove that $\underline{\lambda}_m(f) R^{code}_{(m,\rho)} up(inr(\lambda d : D.g@^1 d))$.
For this we need an explicit description of $\underline{\lambda}_m(f) \in \Lambda(m)_\perp$

- $\underline{\lambda}_m(f) = \perp$ when $f_{\pi:m+1 \to m}(up\ x_m) = \perp$, where $\pi : m+1 \to m$ is the first projection in $\mathcal{D}$ and we exploit the definition of exponentials in $\widehat{\mathcal{D}}$;
- $\underline{\lambda}_m(f) = up(\lambda x_m.M)$ when $up(M) = f_{\pi:m+1 \to m}(up\ x_m) \in \Lambda(m+1)_\perp$.

We can ignore the first case, since when $\underline{\lambda}_m(f) = \perp$ there is nothing to prove.
In the second case, we have to prove that $[\![\lambda x_m.M]\!]_\rho = up(inr(\lambda d : D.g@^1 d))$, i.e. $[\![M]\!]_{\rho[m \mapsto d]} = g@^1 d$ for any $d \in D$

- $up(x_m) R^{code}_{(m+1,\rho[m \mapsto d])} d$, by definition of R
- $up(M) \stackrel{\Delta}{=} f_{\pi:m+1 \to m}(up\ x_m) R^{code}_{(m+1,\rho[m \mapsto d])} g@^1 d$, because $f R^{code \to code}_{(m,\rho)} g$
- $[\![M]\!]_{\rho[m \mapsto d]} = g@^1 d$, by definition of R.

4 Comparisons

In this section we make a comparative analysis of the interpretations of PCF_2 in **Cpo** and $\widehat{\mathcal{D}}$. In fact, to highlight more clearly the differences in the interpretations of *code* and dynamic λ-abstraction (and ignore orthogonal issues), it is better to work in a simplified setting, where

- λ_o is the pure untyped λ-calculus;
- PCF_2 is the simply typed λ-calculus with atomic type *code*, and additional operations $\underline{@} : code, code \to code$ and $\underline{\lambda} : (code \to code) \to code$.

With this simplification one can ask for total correctness of the interpretation of PCF_2 w.r.t. an interpretation of λ_o in **Cpo** (say in the standard model $D = (D \to D)_\perp$ for the lazy λ-calculus). Moreover, the interpretation of PCF_2 without fix_τ can be given in **Set** or $\mathbf{Set}^{\mathcal{D}^{op}}$, where the syntactic category $\mathcal{D}$ has to be changed to reflect the simplifications in λ_o.
The following table summarizes the key differences between the original interpretation proposed by Gomard (Gomard's naive), its patching (Gomard's patched) and the interpretation in $\widehat{\mathcal{D}}$ (functor category).

Semantics	Gomard's patched	Gomard's naive	functor category
category	**Set**	**Set**	$\mathbf{Set}^{\mathcal{D}^{op}}$
$[\![code]\!]$	exp^N	exp	$\Lambda(n)$ at stage n
$[\![code \to code]\!]$	$(exp^N)^{(exp^N)}$	exp^{exp}	$\Lambda(n+1)$ at stage n
$[\![\underline{\lambda}]\!]$	use counter	not defined	use functor category
R^{code}	not defined	$R_{\rho:N \to D}$	$R_{n:N,\rho:n \to D}$
correctness proof	not stated	not meaningful	by Kripke log. rel.

Where exp is the set of λ-terms with variables in N, $\Lambda(n)$ is the set of λ-terms modulo α-conversion with free variables in n, and $D \in$ **Cpo** is a domain for interpreting the lazy λ-calculus, i.e. $D = (D \to D)_\perp$. When describing the functor in $\widehat{\mathcal{D}}$ interpreting a certain type of PCF_2, we have given only its action on objects.
The comparison shows that:

- The functor category interpretation is very similar to Gomard's naive interpretation, when it comes to the definition of $[\![code]\!]$ and R^{code}, though more care is taken in spelling out what object variables may occur free in an object expression.
- The advantage of working in a functor category becomes apparent in the interpretation $code \rightarrow code$, this explains also why the functor category can handle the interpretation of $\underline{\lambda}$.
- Gomard's patched has strong similarities with the simple-minded semantics in **Cpo** for modeling local variables in Algol-like languages. In fact, Gomard's patched semantics parameterizes the meaning of expressions, but not that of types, w.r.t. the number of names generated used so far.

Conclusions and future work

The first part of the paper recalls the main definitions and results in [Gom92], points out the problems with the published interpretation of the two-level language PCF_2, presents possible ways of fixing the interpretation (these were proposed by Olivier Danvy, Fritz Henglein and Neil Jones during several e-mail exchanges) along the lines hinted by Gomard. After fixing the interpretation of PCF_2, there are however problems in fixing the correctness proof in [Gom92]. In the second part of the paper we propose an alternative semantics, and prove correctness for it. We have also cast doubts on the possibility of giving an interpretation of PCF_2 in **Cpo** and prove its correctness w.r.t. the standard interpretation of λ_o **using a logical relation.**
An alternative approach to correctness is proposed in [Wan93]. This avoids any explicit use of operational or denotational semantics, instead he proves correctness modulo β-conversion. Wand uses logical relations, and represents dynamic expressions using higher-order abstract syntax (while [Gom92] uses concrete syntax, and can distinguish α-convertible expressions).
Similar problems to those pointed out in Section 2 are present in other correctness proofs (e.g. [HM94]), which adapt Gomard's approach to more complex two-level languages. We would like to test whether the functor category approach scales up to these languages.

References

[FMS96] M. Fiore, E. Moggi, and D Sangiorgi. A fully-abstract model for the pi-calculus. In *11th LICS Conference*. IEEE, 1996.

[GJ91] K. Gomard and N. Jones. A partial evaluator for the untyped lambda calculus. *J. of Func. Program.*, 1(1), 1991.

[Gom91] Carsten Krogh Gomard. *Program Analysis Matters*. PhD thesis, DIKU, November 1991. DIKU report 91/17.

[Gom92] K. Gomard. A self-applicable partial evaluator for the lambda calculus. *ACM Trans. on Progr. Lang. and Systems*, 14(2), 1992.

[HM94] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sanella, editor, *ESOP'94*, volume 788 of *LNCS*. Springer Verlag, 1994.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[Law63] F.W. Lawvere. Functorial semantics of algebraic theories. *Proc. Nat. Acad. Sci. U.S.A.*, 50, 1963.

[Mit96] John C. Mitchell. *Foundations of Programming Languages.* The MIT Press, Cambridge, MA, 1996.

[MM91] J. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Journal of Pure and Applied Algebra*, 51, 1991.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mog97a] E. Moggi. A categorical account of two-level languages. In *MFPS XIII*, ENTCS. Elsevier, 1997.

[Mog97b] E. Moggi. Metalanguages and applications. In *Semantics and Logics of Computation*, Publications of the Newton Institute. CUP, 1997.

[MW85] A. Meyer and M. Wand. Continuation semantics in typed lambda calculus. In R. Parikh, editor, *Logics of Programs '85*, volume 193 of *LNCS*. Springer Verlag, 1985.

[NN92] F. Nielson and H.R. Nielson. *Two-Level Functional Languages.* Number 34 in Cambridge Tracts in Theoretical Computer Science. CUP, 1992.

[Ole85] F.J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, 1985.

[OT92] P.W. O'Hearn and R.D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, number 177 in L.M.S. Lecture Notes Series. CUP, 1992.

[PS93] A.M. Pitts and I.D.B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Math. Found. of Comp. Sci. '93*, volume 711 of *LNCS*. Springer Verlag, 1993.

[Ten91] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.