Category Theory at Work
H. Herrlich, H.-E. Porst (eds.)

325-340

Products in PER: An Elementary Treatment of the Semantics of the Polymorphic Lambda Calculus

J. W. Gray

Abstract. In order to give a semantics for the polymorphic lambda calculus, a category is required that has products of the same size as the category itself. This is only possible using the notion of internal products in very special categories. One such category is the category PER of partial equivalence relations on the natural numbers. The syntax of the polymorphic lambda calculus is described here with examples, followed by a careful, elementary treatment of the related categories PER, MOD (of modest sets) and ω - SET (of omega sets.) Finally, there is an explicit description of the desired semantics functions.

1. INTRODUCTION.

This paper is an elementary expository account of an exciting development in category theory which has led to extensive interactions between category theorists and computer scientists. The category theory work began with a paper by Martin Hyland [10], based on an earlier work by Hyland, Johnstone and Pitts [12]. They showed that there was a nice category, called the *effective topos* whose morphisms were determined by partial recursive functions. Inside this category there is a very interesting small category, called the category of *modest sets*. Furthermore, there is a category object in the effective topos which "looks just like" the category of modest sets, so the category of modest sets occurs both as a subcategory and as a category object in the effective topos It was later shown by Hyland [11] that this category is complete with respect to internally indexed limits, which contradicts the old result of Freyd [4] saying that if a small category is complete then it is a preorder. Freyd's result, of course, is proved in classical logic whereas the completeness of the category of modest sets is meant in the sense of internal limits in the effective topos. The important thing from the standpoint of theoretical computer science is that the existence, in the category of modest sets (as a subcategory), of products indexed by the whole category itself (as a category object) means that this category can be used to give a semantics for the polymorphic lambda calculus. There is now a large literature concerned with this subject which will not be reviewed here. It can be found by following up the references in this paper. Note that the polymorphic lambda calculus is a logical language which provides the theoretical underpinnings for a number of actual programming languages, such as ML, Miranda, and Haskel.

The second section of this paper explains the syntax of the polymorphic lambda calculus and the third section is concerned with the categories PER and MOD. We make no use of the effective topos, but instead use a treatment of the category of modest sets based on the work of Bruce and Longo [2]. They begin with the category PER (for partial equivalence relations), which played a role in the original treatment by Hyland, and show that it is isomorphic to a subcategory of a simply described category of ω-sets. This category is isomorphic to a certain subcategory of the effective topos, called the category of $\neg\neg$ separated objects, but that plays no role in this paper. Finally, in the fourth section, we briefly describe an explicit semantics for the polymorphic lambda calculus in PER.

This work is an original contribution and will not appear elsewhere.

2. SYNTAX OF THE POLYMORPHIC TYPED λ-CALCULUS.

A *polymorphic typed λ-calculus* is a language consisting of a collection of types and for each type a collection of terms of that type. The types should be thought of as *objects* of some kind and the terms as *operations* or *morphisms* between appropriate objects.

2.1 Definition of a polymorphic typed λ-calculus.

i) The set, Type, of types is given recursively as follows:
   a) There is a finite or countable set B of basic constant types. Note that B may be empty.
   b) There is a countable set Tv of type variables. B + Tv is called the *set of basic types*.
   c) If σ and τ are types then $[\sigma \to \tau]$ is a type, called a *function-space* type.
   d) If σ is a type and t is a type variable, then $\forall t.\sigma$ is a type, called a *universally quantified* type. The variable t is bound in the expression $\forall t.\sigma$.
   Note: it is necessary to identify universally quantified types that differ by α-conversion; i.e., by renaming a bound variable t.

ii) For each type τ, there are
   a) a countable set Var^τ of variables of type τ,
   b) a finite or countable set $Const^\tau$ of constants of type τ. Note that it can be empty for all τ.
   c) the set $Atom^\tau = Var^\tau + Const^\tau$ of atoms of type τ.

iii) Write $f : \tau$ for "f is a term of type τ". The set $Terms^\tau$ of terms of type τ is described recursively as follows:
   a) $Terms^\tau \supseteq Atom^\tau$
   b) If $f : [\sigma \to \tau]$ and $g : \sigma$, then $(f\ g) : \tau$: i.e., if $f \in Terms^{[\sigma \to \tau]}$ and $g \in Terms^\sigma$ then $(f\ g) \in Terms^\tau$. Here, $(f\ g)$ is called an *application* term.
   c) If $g : \tau$ and $x \in Var^\sigma$, then $(\lambda x : \sigma\ .\ g) : [\sigma \to \tau]$; i.e., if $g \in Terms^\tau$ and $x \in Var^\sigma$, then $(\lambda x : \sigma\ .\ g) \in Terms^{[\sigma \to \tau]}$. Here, $(\lambda x : \sigma\ .\ g)$ is called an *abstraction* term.
   d) If $t \in Tv$ and $g : \sigma$ has the property that for all $x : \tau \in FV(g)$, $t \notin fv(\tau)$ then $\Lambda t\ .\ g : \forall t\ .\ \sigma$. Here, $\Lambda t\ .\ g$ is called a *polymorphic* term, or a *type abstraction* of a term.
   e) If $f : \forall t\ .\ \sigma$ and $\tau \in T$, then $f[\tau] : [\ \tau / t\ ]\ \sigma$. Here, $f[\tau]$ is called the *instantiation* of the term f at the type τ.

If the set B and the sets $Const^\tau$ are empty for every type τ, then the λ-calculus is called a *pure* polymorphic typed λ-calculus. That is the only kind of λ-calculus that concerns us here. Note that the above clauses without universally quantified types and without type abstractions and type instantiations of terms is called an *ordinary typed λ-calculus*. If there are no constant terms, it is called the *simple typed λ-calculus*. In these definitions, FV(g) means the set of ordinary free variables of g and $fv(\tau)$ means the set of free type variables of τ as defined below. $[\ \tau / t\ ]\ \sigma$ means substitution of τ for t in σ as usual. (See Hindley and Seldin [9].) We use the following notation:

$Var = \cup_{\tau \in Type} Var^\tau$ = the set of ordinary variables.
$P_f(Var)$ = the set of finite subsets of Var.
$P_f(Tv)$ = the set of finite subsets of Tv.
$Terms = \cup_{\tau \in Type} Terms^\tau$.
type : Terms → Type is the function taking terms of type τ to τ.

2.2 Definitions of free variable functions.

i) FV : Terms $\rightarrow P_f(Var)$ is the function defined by structural recursion as follows:
   a) If $x \in Var^\tau$, then $FV(x) = \{x\}$.
   b) If $c \in Const^\tau$, then $FV(c) = \emptyset$.
   c) $FV((f\ g)) = FV(f) \cup FV(g)$.
   d) $FV((\lambda x\ .\ g)) = FV(g) - \{x\}$.
   e) $FV(\Lambda t\ .\ g) = FV(g)$
   f) $FV(f[\tau]) = FV(f)$

ii) fv : Types $\rightarrow P_f(Tv)$ is the function defined by structural recursion as follows:
   a) If $t \in Tv$, then $fv(t) = \{t\}$,
   b) If $b \in B_I$, then $fv(b) = \emptyset$
   c) $fv(\sigma \rightarrow \tau) = fv(\sigma) \cup fv(\tau)$,
   d) $fv(\forall t.\sigma) = fv(\sigma) - \{t\}$.

A term or type with no free variables is called *closed*. Note that clause d) in the description of terms can also be written:

d') If $t \in Tv$ and $g : \sigma$ satisfy for all $x \in FV(g)$, $t \notin fv(type(x))$ then $\Lambda t\ .\ g : \forall t\ .\ \sigma$.

This condition says that t is not a free type variable in the type of any ordinary free variable of g.

2.3 Operational semantics for the polymorphic λ-calculus. The operational semantics for a λ-calculus consists of rewrite rules written with the symbol "⇒". The meaning is that an occurrence of the left hand side of a rule will be replaced by the right hand side. The transitive closure of ⇒ is denoted by "⇒*". The intention is that rewriting will continue until rewriting produces no change in the expression, resulting in a "normal form" for the original expression. A computer implementation of such a language consists in giving a concrete semantics to be executed by the computer which implements the operational semantics of the language.

i) (β - conversion) $(\lambda x\ .\ f)\ g \Rightarrow [g / x]f$.

ii) (α - conversion) $(\lambda x\ .\ f) \Rightarrow (\lambda y\ .[y / x]\ f)$ providing $y \notin FV(f)$.

iii) (rewrite schemes)

a) $\dfrac{h \Rightarrow k}{(h\ g) \Rightarrow (k\ g)}$ b) $\dfrac{h \Rightarrow k}{(f\ h) \Rightarrow (f\ k)}$ c) $\dfrac{f \Rightarrow g}{(\lambda x\ .\ f) \Rightarrow (\lambda x\ .\ g)}$

iv) (instantiation conversion) $(\Lambda t\ .\ f)[\tau] \Rightarrow [\tau / t]\ f$

Note that in β - conversion and instantiation conversion, substitution of g for x in f or of τ for t in f has to be carefully defined to prevent free variable capture. The meaning of the rewrite schemes is essentially that α and β - conversion can be carried out in any context. More restrictive schemes are sometimes used.

2.4 Examples of types in the pure polymorphic λ-calculus. Even if there are no basic constant types, there are certain universally quantified types built-up, as it were, from nothing. These examples are from [15].

void $= \forall t\ .\ t$
unit $= \forall t\ .\ [t \rightarrow t]$
bool $= \forall t\ .\ [t \rightarrow [t \rightarrow t]]$
int $= \forall t\ .\ [[t \rightarrow t] \rightarrow [t \rightarrow t]]$

The intended interpretations are indicated by the names. In section 4, we will define the type interpretation function D : Types → obj(C) in such a way that

$D_{\forall t\,.\,t} = \emptyset$
$D_{\forall t\,.\,[t \to t]} = 1$
$D_{\forall t\,.\,[t \to [t \to t]]} = \text{Bool}$
$D_{\forall t\,.\,[[t \to t] \to [t \to t]]} = \text{Nat}$

Similarly, given any many-sorted algebraic signature, there is a universally quantified type expression whose interpretation is the initial algebra for the signature. See [15] for further examples.

2.5 Examples of terms in the pure polymorphic λ-calculus. As with types, even if there are no basic constant types or terms, there are certain terms, given by Λ expressions which are also "built-up" out of nothing. We write them in bold face to make them easier to read.

2.5.1. There is one closed term of type unit given by the following expression:

$\mathbf{id} = \Lambda t\,.\,\lambda x : t\,.\,x.$

Since $\lambda x : t\,.\,x$ has type $[t \to t]$, id has type $\forall t\,.\,[t \to t]$. This is the polymorphic identity function. It has instantiations at any type; e.g.,

$\mathbf{id[bool]} = \Lambda t\,.\,\lambda x : t\,.\,x[\,\forall t\,.\,[t \to [t \to t]]\,].$

The type of id[bool] is

$[\,\text{bool} / t\,]\,[t \to t] = [\text{bool} \to \text{bool}]$
$= [\,\forall t\,.\,[t \to [t \to t]] \to \forall t\,.\,[t \to [t \to t]]\,]$

The rule for instantiation conversion says that

$\mathbf{id[bool]} = \Lambda t\,.\,\lambda x : t\,.\,x[\,\forall t\,.\,[t \to [t \to t]]\,]$
$\Rightarrow \lambda x : \forall t\,.\,[t \to [t \to t]]\,.\,x$

which is the identity function for type bool = $\forall t\,.\,[t \to [t \to t]]$.

2.5.2. There are two closed terms of type bool = $\forall t\,.\,[t \to [t \to t]]$ given by the following expressions:

$\mathbf{true} = \Lambda t\,.\,\lambda x : t\,.\,\lambda y : t\,.\,x$
$\mathbf{false} = \Lambda t\,.\,\lambda x : t\,.\,\lambda y : t\,.\,y$

(Cf., Böhm and Berarducci [1].)

Clearly, $\lambda y : t\,.\,x$ has type $[t \to t]$ so $\lambda x : t\,.\,\lambda y : t\,.\,x$ has type $[t \to [t \to t]]$, and hence **true** and **false** have type bool. Define a term for negation: not : [bool → bool] as follows:

$\mathbf{not} = \lambda b : \mathbf{bool}\,.\,\Lambda t\,.\,\lambda x : t\,.\,\lambda y : t\,.\,((b[t]\ y)\ x)$

To check the type of not, note that if b is of type bool, then b[t] has type $t \to t \to t$. Since x and y have type t, ((b[t] y) x) has type t. Thus, $\lambda x : t\,.\,\lambda y : t\,.\,((b[t]\ y)\ x)$ has type $t \to t \to t$ so

$\Lambda t\,.\,\lambda x : t\,.\,\lambda y : t\,.\,((b[t]\ y)\ x)$

has type bool. Therefore, not has type [bool → bool]. Homework: show that (not true) $\Rightarrow^*$ false.

2.5.3. There are countably many closed terms of type nat given by the following expressions:

$\underline{0} = \Lambda t\,.\,\lambda s : [t \to t]\,.\,\lambda z : t\,.\,z$
$\underline{1} = \Lambda t\,.\,\lambda s : [t \to t]\,.\,\lambda z : t\,.\,(s\ z)$
$\underline{2} = \Lambda t\,.\,\lambda s : [t \to t]\,.\,\lambda z : t\,.\,(s\ (s\ z))$

etc. (Cf., Girard [6].)

These are called *Church numerals* exactly as in the untyped λ-calculus. The successor function, succ : [nat → nat] is defined by

succ = λn : nat . Λt . λs : [t → t] . λz : t . s ((n[t] s) z)

The addition function for Church numerals, plus : [nat → [nat → nat]] is defined by

plus = λm : nat . λn : nat . (m[nat] succ) n

The multiplication function for Church numerals, times : [nat → [nat → nat]] is defined by

times = λm : nat . λn : nat . ((n[nat] (plus m)) 0)

Homework: show (succ $\underline{0}$) ⇒* $\underline{1}$.
show ((plus $\underline{1}$) $\underline{1}$) ⇒* $\underline{2}$.
show ((times 2) 2) ⇒* $\underline{4}$.

2.5.4. Here is an example of a bad term: Λt . λx : [t → t'] . (x z). Here z has to have type t, but z ∈ FV(λx : [t → t'] . (x z)) and t ∈ fv(z) so this term is not well-formed.

2.6 Discussion. This form of the polymorphic λ calculus is called F_2. It was originally described by Girard [6] and independently by Reynolds [16]. It is *explicitly* typed, in the sense that the types of all variables are given explicitly in the expressions and there is explicit type abstraction ∀t on types and Λt on terms. Furthermore, polymorphic expressions are applied explicitly to type arguments as in f[τ].

It is also possible to consider implicitly typed languages in which type information is omitted; e.g., ML, etc. In this case, type inference (or type derivation) becomes an important consideration. However, it may not always be possible to assign a unique type to an implicitly typed term. In ML there are type inference algorithms that derive principle types (or principle type schemes) for admissible terms. Admissible means that ∀t can only occur at the top level, and so, in fact, it is omitted from the language entirely. This restriction is crucial in order to have principle types.

Example: The untyped term λf . λx . f (f x) can yield the polymorphic typed term

Λt . λf : [t → t] . λx : t . f (f z)

which, as we have seen has the type nat = ∀t . [[t → t] → [t → t]]. However, it can also yield the polymorphic typed term

λf : (∀t . [t → t]) . Λt' . λx : t' . f[t'] (f[t'] x)

which has the type [∀t . [t → t] → ∀t' . [t' → t']] This takes a polymorphic function as argument and so is not allowed in ML. There can be no more general type which yields both of these types by instantiation. Hence, the full language F_2 does not admit principle type schemes. The decidability of type inference taking untyped λ-calculus into F_2 is open.

Note: F_2 extends to languages $F_3, F_4, \ldots F_\omega$ by introducing Kinds as well as types. In F_2, imagine that ∀t really means ∀t : Type. Then F_3 is given by a more complex grammar: there are Kinds K, types T and terms f described by the grammars:

K ::= Type | Type → K
T ::= t | T → T' | ∀t : K | λt : K . T | (TT')
f ::= x | λx : T . f | (f f') | Λt : K . f | f[T]

To get F_4, K is allow to also have expressions of the form [Type $\rightarrow$ Type] $\rightarrow$ K. Finally, F_ω differs from the earlier languages only in that K ::= Type | K $\rightarrow$ K'.

3. THE CATEGORIES PER AND MOD.

3.1 Review of partial recursive functions.

A partial recursive function is a partial function from N to N which is computed by some Turing machine. Let PRF = the set of all partial recursive function. (Recursive functions of several variables could be defined in the same way.) Since there is only a denumerable number of Turing machines, there is an effective surjective function $e : N \rightarrow PRF$. We use the notations:

$$e(n)(m) = n \cdot m = e_n(m).$$

for the value of the n'th partial recursive function at the argument m. All of these notations include the implication that $e_n(m)$ is defined. The number n is called the *code* for e_n.

We will make use of a number of properties of partial recursive functions.

i) There is a bijection $< -, - > : N \times N \rightarrow N$ whose inverse is given by two total recursive functions with codes p_1 and p_2; i.e.,
$p_1 \cdot <n, m> = n$, $p_2 \cdot <n, m> = m$, and $<p_1 \cdot m, p_2 \cdot m> = m$.

ii) Composition is recursive; i.e., there is p_{comp} such that
$n \cdot (m \cdot u) = (p_{comp} \cdot <n, m>) \cdot u$.

iii) Sections of $N \times N$ are recursive; i.e., for each p there is a code s_p such that
$s_p \cdot n = <p, n>$.

iv) Application is recursive; i.e., there is a code p_{app} such that $n \cdot m = p_{app} \cdot <n, m>$.

v) Induced functions into products are recursive: i.e., there is a code p_{ind} such that
$<n \cdot p , m \cdot p> = (p_{ind} \cdot <n, m>) \cdot p$.

3.2 The category PER.

The polymorphic λ-calculus can be interpreted in any cartesian closed category C, except for types of the form $\forall t$. expr(t) where expr(t) is some type expression in t. The idea is that $D_{\forall t \,.\, expr(t)}$ should involve $\prod_{A \in C} expr(A)$. However, it is a "classical" theorem of category theory that if a (small) category C has products of families of the same size as the category, then there is at most one morphism between any two objects; i.e., the category is a preorder. (Proof: Suppose C(A, B) has at least two elements. Then $C(A, \prod_{mor(C)} B) \approx \prod_{mor(C)} C(A, B)$ has cardinality at least as big as $2^{mor(C)}$ which is bigger that the cardinality of mor(C).)

This proof is classical and is not valid intuitionistically; i.e., it need not hold in a topos, where products are "internal" products. We will discuss a sequence of categories: EFF $\supseteq$ ω–SET $\supseteq$ PER where EFF is the effective topos, ω–SET is (isomorphic) to a certain (large) subcategory of EFF, and PER is a small subcategory which contains products indexed by ω–sets, but nevertheless is not a preorder. (Actually, we will only discuss the topos EFF in passing.)

3.2.1 Definition. A per A is a partial equivalence relation on the set N of natural numbers; i.e., a symmetric, transitive relation A contained in $N \times N$.

Notation: Write n A m iff $(n, m) \in A$ and $dom(A) = \{n \mid n\ A\ n\}$. Then A restricted to dom(A) is an equivalence relation. Denote the equivalence class of $n \in dom(A)$ mod A by $[n]_A = \{m \mid n\ A\ m\}$ and the set of equivalence classes (or the quotient set of dom(A) by the relation A) by $Q(A) = \{[n]_A \mid n \in dom(A)\}$.

3.2.2 Definition. If A and B are pers then a morphism f from A to B is a function $f : Q(A) \to Q(B)$ such that there is a $n_f \in N$ with the property that for all $p \in dom(A)$, $f([p]_A) = [n_f \cdot p]_B$. In particular, $n_f \cdot (-)$ is a total function on dom(A). The number n_f is said to *realize* f, or to be a *witness* for f.

3.2.3 Proposition. Composition of morphisms of pers is again a morphism of pers.

Proof. Given $f : Q(A) \to Q(B)$ and $g : Q(B) \to Q(C)$ with witnesses n_f and n_g, then

$$g(f(p)) = [n_g \cdot (n_f \cdot p)]C = [(p_{comp} \cdot \langle n_g, n_f \rangle) \cdot p]_C.$$

Hence, $(p_{comp} \cdot \langle n_g, n_f \rangle)$ is a witness for $g \cdot f$.

3.2.4 Definition. PER denotes the category of pers and morphisms of pers.

3.2.5 Theorem. PER is a cartesian closed category with all finite limits.

Proof.

1. PER has binary products. If A and B are pers then $A \times B$ is the per defined by

$$n\ A \times B\ m \text{ iff } (p_1 \cdot n)\ A\ (p_1 \cdot m) \text{ and } (p_2 \cdot n)\ B\ (p_2 \cdot m)$$

The projection morphisms $pr_A : Q(A \times B) \to Q(A)$ and $pr_B : Q(A \times B) \to Q(B)$ are given by $pr_A([n]_{A \times B}) = [p_1 \cdot n]_A$ and $pr_B([n]_{A \times B}) = [p_2 \cdot n]_B$. Thus p_1 and p_2 are witnesses for pr_A and pr_B. Note that $\langle pr_A, pr_B \rangle_{SET} : Q(A \times B) \to Q(A) \times_{SET} Q(B)$ is a bijection since, given $[n]_A$ and $[m]_B$, then $\langle n, m \rangle \in dom(A \times B)$ and $[\langle n, m \rangle]_{A \times B}$ is well defined with $pr_A([\langle n, m \rangle]_{A \times B}) = [n]_A$ and $pr_B([\langle n, m \rangle]_{A \times B}) = [m]_B$. Now, given $f : Q(C) \to Q(A)$ and $g : Q(C) \to Q(B)$ with witnesses n_f and n_g, then there is a function

$$\langle f, g \rangle : Q(C) \to Q(A) \times_{Set} Q(B) \approx Q(A \times B)$$

satisfying

$$\langle f, g \rangle([r]_C) = [\langle n_f \cdot r, n_g \cdot r \rangle]_{A \times B} = [(p_{ind} \cdot \langle n_f, n_g \rangle) \cdot r]_{A \times B}$$

so $p_{ind} \cdot \langle n_f, n_g \rangle$ is a witness for $\langle f, g \rangle$.

2. PER has function space objects. Let A and B be pers. Then the function space per, $[A \to B]$, is defined as follows:

$$n\ [A \to B]\ m \text{ iff } \forall p.\ \forall q.\ p\ A\ q \Rightarrow (n \cdot p)\ B\ (m \cdot q)$$

In particular, $n \in dom([A \to B])$ iff $\forall p.\ \forall q.\ p\ A\ q \Rightarrow (n \cdot p)\ B\ (n \cdot q)$ so n is the witness of a morphism $f_n : Q(A) \to Q(B)$. Furthermore, n and m are witnesses of the same morphism if and only if $n\ [A \to B]\ m$, so $Q([A \to B]) \approx PER(A, B)$.

3. The application morphism is defined as follows. Define $app_{A,B} : Q([A \to B]) \times Q(A) \to Q(B)$ by the formula: $app_{A,B}([n]_{[A \to B]}, [m]_A) = [n \cdot m]_B = [p_{app} \cdot \langle n, m \rangle]_B$ so p_{app} witnesses $app_{A,B}$.

4. Curried morphisms are morphisms. Let $h : C \times A \to B$ be a morphism witnessed by n_h, and consider $h([<p, n>]_{C\times A}) = [n_h \cdot <p, n>]_B$. Since h is really a function $h : Q(C) \times Q(A) \to Q(B)$, it determines a curried function $h^{\#} : Q(C) \to [Q(A) \to Q(B)]_{SET}$. Then

$$h^{\#}([p]_C)([n]_A) = [n_h \cdot <p, n>]_B = [n_h \cdot (s_p \cdot n)]_B = [(p_{comp} \cdot <n_h, s_p>) \cdot n]_B$$

Hence, $h^{\#}([p]_C)(-)$ is a morphism from A to B witnessed by $p_{comp} \cdot <n_h, s_p>$. Furthermore, $F(p) = p_{comp} \cdot <n_h, s_p>$ is a recursive function of p, so $h^{\#} : C \to [A \to B]$ is a morphism. Therefore PER is cartesian closed.

5. Equalizers. Let $f, g : A \to B$ be morphisms of pers. Then $eq(f, g) : Eq(f, g) \to A$ is constructed as follows: Chose witnesses n_f and n_g for f and g and form the set theoretic equalizer of the restrictions of n_f and n_g to dom(A). This is the domain of Eq(f, g) and Eq(f, g) itself is the restriction of the relation A to this set. The morphism eq(f, g) is witnessed by the code for the identity function. The existence of arbitrary finite limits follows as usual from the existence of binary products and equalizers. See also [18].

3.3 The category of ω-sets.

3.3.1 Definition. An ω-set is a set X together with a surjective relation $\Vdash_X$ contained in $N \times X$; i.e., $\forall x.\ \exists n.\ n \Vdash_X x$. This relation is read "n realizes x" or "n witnesses x". Note that a particular n can realize many different x's. The only condition is that every x is realized by some n.

3.3.2 Definition. Let $(X, \Vdash_X)$ and $(Y, \Vdash_Y)$ be ω-sets. A morphism $f : (X, \Vdash_X) \to (Y, \Vdash_Y)$ is a function $f : X \to Y$ such that there exists $p_f \in N$ satisfying the condition: if $n \Vdash_X x$ then $(p_f \cdot n) \Vdash_Y f(x)$. As usual, we say p_f realizes or witnesses f.

3.3.3 Definition. ω - SET is the category of ω-sets and morphisms of ω-sets.

3.3.4 Theorem. ω - SET is cartesian closed with all finite limits.

Proof. $(X, \Vdash_X) \times (Y, \Vdash_Y) = (X \times Y, \Vdash_{X\times Y})$ where

$$n \Vdash_{X\times Y} (x, y) \text{ iff } (p_1 \cdot n) \Vdash_X x \text{ and } (p_2 \cdot n) \Vdash_Y y.$$

The function space object is $(\omega \text{ - SET}(X, Y), \Vdash_{[X \to Y]})$ where $p \Vdash_{[X \to Y]} f$ iff p witnesses f as above. Equalizers are "full" subobjects; i.e., if $f, g : (X, \Vdash_X) \to (Y, \Vdash_Y)$ are morphisms of ω-sets, then the equalizer Eq(f, g) of f and g is constructed set theoretically as the subset of A on which f and g agree, with the relation given by $n \Vdash_{Eq(f,g)} e$ if and only if $n \Vdash_A e$

3.3.5 Definition. MOD denotes the full subcategory of ω - SET determined by the "functional" ω-sets; i.e., those $(X, \Vdash_X)$ for which $\Vdash_X$ is the graph of a function. Thus, if $n \Vdash_X x$ and $n \Vdash_X y$ then $x = y$. Such an $(X, \Vdash_X)$ is called a *modest set*. In a modest set, a number n witnesses at most one element. MOD is called the category of modest sets.

3.3.6 Proposition. MOD is isomorphic to PER.

Proof. We define functors PerToMod : PER → MOD and ModToPer : MOD → PER by the formulas: PerToMod(A) = $(Q(A), \in_A)$, and PerToMod(f) = f. Here, $(\in_A)$ is the subset of $N \times Q(A)$ given by $(n, [p]_A) \in (\in_A)$ iff $n \in [p]_A$. In the other direction, ModToPer$(X, \Vdash_X)$ is the per defined by

n ModToPer(X, $\vdash_X$) m iff $\exists x \in X$ with n $\vdash_X$ x and m $\vdash_X$ x and ModToPer(f) = f. Thus, n $\in$ dom(ModToPer(X, $\vdash_X$)) iff $\exists x \in X$ with n $\vdash_X$ x, and

$$[n]_{\text{ModToPer}(X, \vdash_X)} = \{m \mid \exists x \in X \text{ with } n \vdash_X x \text{ and } m \vdash_X x\}.$$

One checks easily that ModToPer(PerToMod(A)) = A and PerToMod(ModToPer(X, $\vdash_X$)) $\approx$ (X, $\vdash_X$); i.e.,.ModToPer $\cdot$ PerToMod = id_{PER} and PerToMod $\cdot$ ModToPer $\approx$ id_{MOD}.

3.4. MOD as an internal category object in ω-SET.

3.4.1 The "equational" definition of a category.

A *category* C consists of six components:

i) A class obj(C) whose elements are called *objects* of C.

ii) A class mor(C) whose elements are called *morphisms* of C.

iii) Two functions dom_C, cod_C : mor(C) $\rightarrow$ obj(C) called the *domain* and *codomain* functions. The expression f : X $\rightarrow$ Y for a morphism f in C abbreviates the statements: $\text{dom}_C(f)$ = X and $\text{cod}_C(f)$ = Y.

iv) A function id_C : obj(C) $\rightarrow$ mor(C). We will write $\text{id}_C(X)$ as id_X when C is clear from the context.

v) A function comp_C : compPair(C) $\rightarrow$ mor(C) called *composition*. Here compPair(C) denotes the class of pairs of morphisms (f, g) such that $\text{cod}_C(f) = \text{dom}_C(g)$. As before, we write $\text{comp}_C(f, g) = g \circ f$, or g f, or f ; g. Sometimes for clarity we write it as $g \circ_C f$.

These data satisfy four axioms.

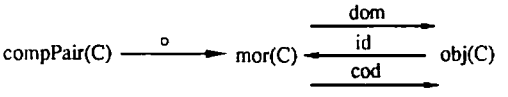
a) $\text{dom}_C(\text{id}_X) = \text{cod}_C(\text{id}_X) = X$; i.e., $\text{id}_X : X \rightarrow X$.

b) If f : X $\rightarrow$ Y, then $\text{id}_Y \circ f = f \circ \text{id}_X = f$.

c) $\text{dom}_C(g \circ f) = \text{dom}_C(f)$ and $\text{cod}_C(g \circ f) = \text{cod}_C(g)$; i.e., if f : X $\rightarrow$ Y and g : Y $\rightarrow$ Z, then $g \circ f : X \rightarrow Z$.

d) If f : X $\rightarrow$ Y, g : Y $\rightarrow$ Z, and h : Z $\rightarrow$ W, then $h \circ (g \circ f) = (h \circ g) \circ f$.

The functions described in this definition can be illustrated by the following picture.

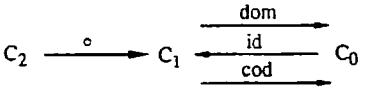


In this version of the definition, the collection of morphisms f such that dom(f) = X and cod(f) = Y could be a proper class. If it does form a set for each pair of objects X and Y, then C is called *locally small*, and the set of such morphisms is denoted by C(X, Y). C is called *small* if both obj(C) and mor(C) are sets.

The definition of CompPair says that CompPair is a pullback of the two morphisms dom, cod : mor(C) $\rightarrow$ obj(C), so we can generalize this description of a category to the notion of a category object, or an internal category in any category D.

3.4.2 Definition.

An internal category object C in a category D is a diagram



of objects and morphisms in D, C_2 is a pullback in D of the morphisms dom and cod, as above, and the same equations as in the existential definition of a category are satisfied.

3.4.3 The internal category object M in ω-SET.

Let $M_0 = (\text{obj}(\text{MOD}), \Vdash_{M_0})$ where $\Vdash_{M_0} = N \times \text{obj}(\text{MOD})$; i.e., every $n \in N$ witnesses every modest set. Such an ω-set is *chaotic* or *codiscrete*. Let

$$M_1 = (\{< (X, \Vdash_X), f, (Y, \Vdash_Y) > | f \in \text{MOD}(X, Y)\}, \Vdash_{M_1})$$

where $n \Vdash_{M_1} <X, f, Y>$ iff $n \Vdash_{[X \to Y]} f$. Finally, $\text{dom}(<X, f, Y>) = X$, $\text{cod}(<X, f, Y>) = Y$, $\text{id}(X) = <X, \text{id}_X, X>$, and $\text{comp}(<X, f, Y, g, Z>) = <X, f ; g, Z>$. Then

$$\underline{M} = (M_0, M_1, \text{dom}, \text{cod}, \text{id}, \text{comp})$$

denotes this category object. Note that there is also the discrete category object

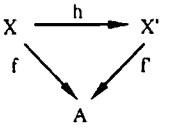
$$\underline{M_0} = (M_0, M_0, \text{id}, \text{id}, \text{id}, \text{id}).$$

3.4.4 Relations with EFF. The categories MOD and ω-SET can be imbedded in a natural way in the effective topos. Under this embedding, MOD is equivalent to the category of "effective objects" and ω-SET is equivalent to the category of "$\neg\neg$ – separated objects". See Hyland [10].

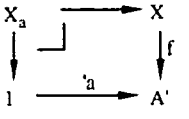
3.5. Products of modest sets indexed by MOD.

3.5.1 Definition.

Let C be a category and let A be an object in C. The *slice* (or *comma*) category $C{\downarrow}A$ is the category whose objects are pairs $(X, f : X \to A)$, abbreviated by (X, f). A morphism from (X, f) to (X', f') is a morphism $h : X \to X'$ in C such that $f' \cdot h = f$. Objects and morphisms in $C{\downarrow}A$ are called *objects and morphisms over* A.



A useful way to think about objects over A is in terms of indexed families of objects. If C has pullbacks and a terminal object, then given $f : X \to A$ and a global element $'a : 1 \to A$, we can form the pullback X_a of f along 'a, as illustrated. X_a is called the *fibre* of $f: X \to A$ over a.



If $f' : X' \to A$ is another object over A and $h : X \to X'$ is a morphism over A, then it is easily checked that h determines a family of morphisms $\{h_a : X_a \to X'_a\}$. Thus, for each global element a of X, h_a maps the fibre of X over a to the fibre of X' over a.

It is easily seen that a category $C{\downarrow}A$ has a terminal object given by the identity map on A and that C has pullbacks if and only if $C{\downarrow}A$ has binary products for all A in C. The relations between the slice categories over objects A and A' determined by a morphism $g : A \to A'$ are our main concern here. If $g : A \to A'$, then the *change of base functor* $\Sigma_g : C{\downarrow}A \to C{\downarrow}A'$ is the functor whose value on objects is given by $\Sigma_g(X, f) = (X, g \cdot f)$ and whose value on morphisms is given by $\Sigma_g(h) = h$; i.e., Σg takes

objects and morphisms over A to objects and morphisms over A' by composing the objects with g. If C has chosen pullbacks then Σ_g has a right adjoint functor, the *inverse change of base functor* $g^* : C{\downarrow}A' \to C{\downarrow}A$ given by *pulling back along* g; i.e.,

$$g^*(X, f : X \to A') = (g^*X, p_1 : g^*X \to A)$$

where p_1 is the projection from the pullback to A. A category C with finite limits is called *locally cartesian closed* if for all $g : A \to A'$ in C, g^* has a further right adjoint functor $\Pi_g : C{\downarrow}A \to C{\downarrow}A'$. If $f : X \to A$, then we frequently write $\Pi_g(X)$ instead of the more precise $\Pi_g(X, f)$. See Freyd [4], or Taylor [19]. Note that local cartesian closedness is equivalent to all slice categories being cartesian closed.

3.5.2 Example.

The category ω-SET is locally cartesian closed. This can be proved directly, or by using topos theory.

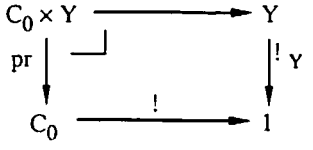
3.6 Internal functors.

Just as there are internal category objects in a category, one can also consider internal functors between them (i.e., small functors) as well as internal functors from an internal category object to the ambient category (i.e., locally small functors).

3.6.1 Definition. Let $C_{int} = (C_0, C_1, \text{dom}, \text{cod}, \text{id}, \text{comp})$ be an internal category object in a category D. An internal functor F from C_{int} to the ambient category D is an object $(F, p : F \to C_0)$ in $D{\downarrow}C_0$ together with an action $\alpha : F \times_{C_0} C_1 \to F$ over C_0, satisfying easily derived equations. For a discrete category object, there are no equations and the action is the identity morphism.

3.6.2 Definition.

A constant internal functor over C is one that is pulled back from something over 1; i.e., one of the form



where the action is given by the projection onto Y. (It represents the functor constantly equal to Y.)

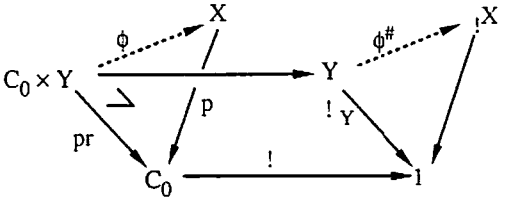
3.6.3 Internal products.

Recall the usual definition of a product of a family of objects X_i indexed by $i \in I$, which says that $C(Y, \prod_{i \in I} X_i) \approx \prod_{i \in I} C(Y, X_i)$;i.e., morphisms from Y to $\prod_{i \in I} X_i$ are bijectively equivalent to cones from Y to the family $\{X_i : i \in I\}$. Stated differently, Y can be regarded as the constant family indexed by I and $\prod_{i \in I} C(Y, X_i)$ is the collection of "natural transformations" from the constant family $\{Y_i = Y\}$ to the family $\{X_i\}$. Turning to the case of internal products, instead of the index set I, one uses a discrete internal category object $\underline{C_0}$ (which is identified with its object C_0 of objects) as the indexing object. Furthermore, the family $\{X_i \mid i \in I\}$ is replaced by an object over C_0 (i.e., a morphism $p : X \to C_0$), thought of as an internal functor on the discrete category object $\underline{C_0}$, while the constant family equal to Y is represented by a constant internal functor $\text{pr} : C_0 \times Y \to C_0$. The product of the *family* $p : X \to C_0$ is the value of the right adjoint functor $\Pi_! : C{\downarrow}C_0 \to C{\downarrow}1$ on (X, p). Thus,

the description of an internal product in terms of slice categories says that "internal natural transformations" from the constant internal functor $(C_0 \times Y, pr)$ to the internal functor (X, p) are bijectively equivalent to morphisms over 1 from Y to $\prod_!(X, p)$; i.e.,

$$(C{\downarrow}C_0)(\,(C_0 \times Y, pr),\,(X, p)\,) \approx (C{\downarrow}1)(\,(Y, !_Y),\,\prod_!(X, p)\,) \approx C(Y, \prod_! X)$$

as illustrated below.



Therefore, there has to be a natural bijection between morphisms ϕ and $\phi^{\#}$ as illustrated.

3.7 Products in MOD.

In the category ω-SET, suppose that C_0 above is M_0 and consider an object $p : X \to M_0$ where the fibres of X are modest sets; i.e., X looks like a family $\{X_A \mid A \in M_0\}$ of modest sets indexed by the "ω-set of all modest sets". More explicitely, for each global element $'A : 1 \to X$, the pullback X_A constructed as in 2.3.4 is a modest set.

3.7.1 Theorem. $\prod_! X = \mathrm{PerToMod}(\cap_{A \in M_0} \mathrm{ModToPer}(X_A))$.

Proof. Recall that $\mathrm{ModToPer}(X_A)$ is a per for each A; i.e., a symmetric, transitive subset of $N \times N$. The intersection of such subsets is again symmetric and transitive, so it is a per. (Note: in PER, for any family A_i of pers, $\cap_{i \in I} A_i$ is defined by $n \cap_{i \in I} A_i\, m$ iff for all $i \in I$, $n\, A_i\, m$. Then $\mathrm{dom}(\cap_{i \in I} A_i) = \cap_{i \in I} \mathrm{dom}(A_i)$ and there are morphisms $pr_i : Q(\cap_{i \in I} A_i) \to Q(A_i)$ which are witnessed by the code for the identity function.) Hence $\mathrm{PerToMod}(\cap_{A \in M_0} \mathrm{ModToPer}(X_A))$ is a modest set.

To prove that this modest set is isomorphic to the internal product $\prod_! X$, let Y be a modest set, and consider a morphism $\phi : M_0 \times Y \to X$ over M_0 in ω-SET as in 2.6.3. Let m be a witness for ϕ; i.e., $m \Vdash_{[M_0 \times Y \to X]} \phi$. Let $(A, y) \in M_0 \times Y$ and let $n \Vdash_{M_0 \times Y} (A, y)$, so $n_1 = p_1 \cdot n \Vdash_{M_0} A$, $n_2 = p_2 \cdot n \Vdash_Y y$ and $n = \langle n_1, n_2 \rangle$. Then $m \cdot n \Vdash_X \phi(A, y)$. The point of the following argument is that the condition $n_1 \Vdash_{M_0} A$ is no condition at all since M_0 is a chaotic ω-set. It is satisfied by all $n_1 \in N$. Hence, for all $n_1 \in N$ and for all $A \in M_0$, $m \cdot \langle n_1, n_2 \rangle \Vdash_X \phi(A, y)$. In particular, we can choose $n_1 = 0$, so $m \cdot \langle 0, n_2 \rangle \Vdash_X \phi(A, y)$ for all A. But $\phi(A, y) \in X_A$ since ϕ is a morphism over M_0. Furthermore, $m \cdot \langle 0, n_2 \rangle \Vdash_{X_A} \phi(A, y)$ since X_A is full in X, because it is a pullback. Thus there is a number $r = m \cdot \langle 0, n_2 \rangle$, independent of A, which witnesses the value of $\phi(A, y)$ in each X_A. Therefore,

$$r \in \mathrm{dom}(\cap_{A \in M_0} \mathrm{Out}(X_A)).$$

Now consider $\mathrm{PerToMod}(\cap_{A \in M_0} \mathrm{ModToPer}(X_A))$ and define $\phi^{\#} : Y \to \mathrm{PerToMod}(\cap_{A \in M_0} \mathrm{ModToPer}(X_A))$ by

$$\begin{aligned}\phi^{\#}(y) &= [r]_{\cap_{A \in M_0} \mathrm{ModToPer}(X_A)} = [m \cdot \langle 0, n_2 \rangle]_{\cap_{A \in M_0} \mathrm{ModToPer}(X_A)} \\ &= [(p_{comp} \cdot \langle m, s_0 \rangle) \cdot n_2]_{\cap_{A \in M_0} \mathrm{ModToPer}(X_A)}\end{aligned}$$

so $\phi^{\#}$ is witnessed by $p_{comp} \cdot \langle m, s_0 \rangle$. (Note: one still has to verify that the correspondence between ϕ and $\phi^{\#}$ is a natural bijection.)

3.7.2 Corollary: MOD, (or PER) has internal products indexed by chaotic ω-sets.

4. SEMANTICS OF THE POLYMORPHIC λ-CALCULUS IN PER.

4.1 Semantics of the simple typed λ-calculus in a cartesian closed category.

For a more complete treatment of certain aspects, see Gray [7] and [8]. If we ignore universally quantified types, type abstractions of terms and type instantiations in terms, then what is left is an ordinary typed λ-calculus. This has a standard interpretation in any cartesian closed category C.

4.1.1 Semantics of ordinary types.
A *type interpretation* of an ordinary typed λ-calculus in C consists of a function $D : Type \to obj(C)$ such that

i) If $b \in B$, then $D(b) \in obj(C)$ is some user chosen object in C.
ii) $D([\sigma \to \tau] = [D(\sigma) \to D(\tau)]_C$.

We frequently write D_σ for $D(\sigma)$.

4.2.2 Semantics of ordinary terms.

i) If f is any term, then $Env(f) = \prod_{x \in FV(f)} D_{type(x)}$.
ii) For each $x \in Var$ and $f \in Terms$, $update_{x,f}$ is defined by two cases:
   a) If $x \in FV(f)$, then $update_{x,f} : Env(\lambda x \,.\, f) \times D_{type(x)} \to Env(f)$ is the unique morphism such that for every $w \in FV(f)$,
      $pr_w \circ update_{x,f} = pr_w \circ pr_1$, if $w \neq x$, and $pr_x \circ update_{x,f} = pr_2$.
   b) If $x \notin FV(f)$, then $FV(\lambda x \,.\, f) = FV(f)$, so $Env(\lambda x \,.\, f) = Env(f)$ and
      $update_{x,f} = pr_1 : Env(\lambda x \,.\, f) \times 1 \to Env(f)$.

iii) The (*local environments*) *term interpretation* function $[\![-]\!] : Terms \to mor(C)$ for a simply typed λ-calculus assigns to each term f in L a morphism $[\![f]\!] : Env(f) \to D_{type(f)}$. These morphisms are defined recursively by the clauses:
   a) If $x \in Var$, then $[\![x]\!] = id_{D_{type(x)}} : D_{type(x)} \to D_{type(x)}$.
   b) If $f : [\sigma \to \tau]$ and $g : \sigma$, then
      $[\![(f\ g)]\!] = app_{\sigma,\tau} \circ \langle [\![f]\!] \circ pr_{FV(f)}, [\![g]\!] \circ pr_{FV(g)} \rangle : Env(\,(f\ g)\,) \to D_\tau$
   c) Let $x \in Var$ and $f \in Terms$. If $x \in FV(f)$, then
      $[\![\lambda x \,.\, f]\!] = curry([\![f]\!] \circ update_{x,f}) : Env\,(\lambda x \,.\, f\,) \to [D_{type(x)} \to D_{type(f)}]$.
      else
      $[\![\lambda x \,.\, f]\!] = curry([\![f]\!] \circ update_{x,f}) : Env\,(\lambda x \,.\, f\,) \to [1 \to D_{type(f)}]$.

Here, $curry : C(X \times Y, Z) \to C(X, [A \to B])$ is the isomorphism describing the function-space construction $[A \to B]$ as right adjoint to the cartesian product. Part b) makes sense since
$[\![f]\!] : Env(f) \to [D_\sigma \to D_\tau]$ and $[\![g]\!] : Env(g) \to D_\sigma$.
Now $Env(\,(f\ g)\,) = \prod_{w \in FV(f) \cup FV(g)} D_{type(w)}$ so there are generalized projections
$pr_{FV(f)} : Env(\,(f\ g)\,) \to Env(f)$ and $pr_{FV(g)} : Env(\,(f\ g)\,) \to Env(g)$

Thus

$$\langle [\![f]\!] \circ pr_{FV(f)}, [\![g]\!] \circ pr_{FV(g)} \rangle : Env((f\ g)) \to [D_\sigma \to D_\tau] \times D_\sigma$$

while $app_{\sigma,\tau} : [D_\sigma \to D_\tau] \times D_\sigma \to D_\tau$.

To make sense of part c), we have to clarify what morphism is being curried. If $x \in FV(f)$, then

$$update_{x,f} : Env(\lambda x . f) \times D_{type(x)} \to Env(f),$$

so $[\![f]\!] \circ update_{x,f} : Env(\lambda x . f) \times D_{type(x)} \to D_{type(f)}$. Hence,

$$curry([\![f]\!] \circ update_{x,f}) : Env(\lambda x . f) \to [D_{type(x)} \to D_{type(f)}],$$

as desired. If $x \notin FV(f)$, then $update_{x,f} = pr_1 : Env(\lambda x . f) \times 1 \to Env(f)$, so as before,

$$curry([\![f]\!] \circ update_{x,f}) : Env(\lambda x . f) \to [1 \to D_{type(f)}],$$

4.2 Semantics of the polymorphic λ-calculus.

If X is a discrete category and C is cartesian closed, then $[X \to C]$ is cartesian closed with $(F \times G)(X) = F(X) \times G(X)$ and $[F \to G](X) = [F(X) \to G(X)]$. We use this with C = PER and $X = |\mathbf{PER}|^{Tv}$ where $|C|$ means the underlying discrete category of C, and Tv is the set of type variables for the polymorphic lambda calculus. The product category $|\mathbf{PER}|^{Tv}$ can be thought of as the (discrete) category of type environments.

4.2.1 Polymorphic types. Assume that $B_1 = \emptyset$. Note that $obj([|\mathbf{PER}|^{Tv} \to \mathbf{PER}])$ consists of functors $F : |\mathbf{PER}|^{Tv} \to \mathbf{PER}$. Such a functor has a value for each type environment $S \in |\mathbf{PER}|^{Tv}$, called F(S). Define $D_{(\ -\ )} : Type \to obj([|\mathbf{PER}|^{Tv} \to \mathbf{PER}])$ by the recursive clauses:

$$D_t = pr_t : |\mathbf{PER}|^{Tv} \to \mathbf{PER}, \text{ i.e., } D_t(S) = pr_t(S) = S(t).$$

$$D_{[\sigma \to \tau]} = [\ D_\sigma \to D_\tau\ ]$$

$$D_{\forall t.\sigma}(S) = \prod_{A \in |\mathbf{PER}|} D_\sigma(S[A/t])$$

Here $S \in |\mathbf{PER}|^{Tv}$ is a type environment and S[A/t] is the environment in which t is updated by A; i.e., $S[A/t](t) = A$ and $S[A/t](t') = S(t')$ for $t' \neq t$.

Example: Consider $int = \forall t . [[t \to t] \to [t \to t]]$. Then

$$\begin{aligned} D_{\forall t . [[t \to t] \to [t \to t]]}(S) &= \prod_{A \in |\mathbf{PER}|} D_{[[t \to t] \to [t \to t]]}(S([A/t])) \\ &= \prod_{A \in |\mathbf{PER}|} [[D_t(S[A/t] \to D_t(S[A/t]] \to [D_t(S[A/t] \to D_t(S[A/t]]] \\ &= \prod_{A \in |\mathbf{PER}|} [[[S[A/t](t) \to S[A/t](t)] \to [S[A/t](t) \to S[A/t](t)]] \\ &= \prod_{A \in |\mathbf{PER}|} [[[A \to A] \to [A \to A]] \\ &= \bigcap_{A \in |\mathbf{PER}|} [[[A \to A] \to [A \to A]]. \end{aligned}$$

It is a theorem, first proved by Hyland, Robinson, and Rosolini [13] that this per is exactly the natural numbers. Later, Freyd [5] gave a much simpler proof for this case.

4.2.2 Polymorphic terms. In the case of the polymorphic λ-calculus, terms are represented by morphisms in the category $[|\mathbf{PER}|^{Tv} \to \mathbf{PER}]$. These are natural transformations between functors, but since $|\mathbf{PER}|^{Tv}$ is a discrete category, natural transformations are just families of morphisms. Thus, let $F, G : |\mathbf{PER}|^{Tv} \to \mathbf{PER}$ be functors. This means that F and G are functions with values in PER for each type environment $S \in |\mathbf{PER}|^{Tv}$. A morphism $\phi : F \to G$ is a family of morphisms $\phi_S : F(S) \to G(S)$ for each S. Since $[|\mathbf{PER}|^{Tv} \to \mathbf{PER}]$ is cartesian closed, the interpretation of ordinary terms is

the usual one, and we only have to worry about type abstractions and type instantiations. These will be determined by induction as usual. In general, if $g \in \text{Terms}^{\sigma}$, then $[[\ g\ ]] : \text{Env}(g) \to D_{\sigma}$ where

$$\text{Env}(g) = \prod_{x \in FV(g)} D_{type(x)}.$$

Regarded as functors, $\text{Env}(g)(S) = (\prod_{x \in FV(g)} D_{type(x)}).(S) = \prod_{x \in FV(g)} (D_{type(x)}.(S))$, so

$$[[\ g\ ]]_S : \prod_{x \in FV(g)} (D_{type(x)}.(S)) \to D_{\sigma}(S).$$

4.2.3 Type abstractions of terms.

Assume that $[[\ g\ ]]$ has already been specified. Now, a term of the form $\Lambda t\ .\ g$ has type $\forall t\ .\ \sigma$, where $type(g) = \sigma$, and for each $S \in |\text{PER}|^{Tv}$, by 3.1, $D_{\forall t.\sigma}(S) = \prod_{A \in |\text{PER}|} D_{\sigma}(S[A/t])$ so the component of $[[\Lambda t\ .\ g]]$ at S has to be a morphism

$$[[\Lambda t\ .\ g]]_S : \prod_{x \in FV(\Lambda t\ .\ g)} (D_{type(x)}(S)) \to \prod_{A \in |\text{PER}|} D_{\sigma}(S[A/t])$$

Such a morphism is determined by its projections onto the objects $D_{\sigma}(S[A/t])$ for each $A \in \text{PER}$. These are given by the formula:

$$pr_A \circ [[\Lambda t\ .\ g]]_S = [[g]]_{S[A/t]}$$

This makes sense since $FV(\Lambda t\ .\ g) = FV(g)$ so the domain of $[[g]]$ is $(\prod_{x \in FV(\Lambda t\ .\ g)} D_{type(x)})$. Furthermore, for all $x \in FV(g)$, $t \notin fv(type(x))$, so $D_{type(x)}(S[A/t]) = D_{type(x)}(S)$ by induction on the structure of type(x). Hence the domain of $[[\ g\ ]]_{S[A/t]}$, which is $\prod_{x \in FV(\Lambda t\ .\ g)} (D_{type(x)}(S[A/t]))$, in fact equals $\prod_{x \in FV(\Lambda t\ .\ g)} (D_{type(x)}(S))$. Thus, for all A,

$$[[\ g\ ]]_{S[A/t]} : \prod_{x \in FV(\Lambda t\ .\ g)} (D_{type(x)}(S)) \to D_{\sigma}(S[A/t])$$

as desired.

4.2.4 Type instantiations of terms.

If $f \in \text{Terms}^{\forall t.\sigma}$, then $[[\ f\ ]]_S : (\prod_{x \in FV(f)} D_{type(x)})(S) \to \prod_{A \in |\text{PER}|} D_{\sigma}(S[A/t])$. Assuming that $[[\ f\ ]]$ has already been determined, then we must specify $[[\ f[\tau]\ ]]$ for any type τ. But $D_{\tau} \in \text{PER}$ so there is a factor $D_{\sigma}(S[D_{\tau}/t])$ in $\prod_{A \in |\text{PER}|} D_{\sigma}(S[A/t])$. Hence it makes sense to define $[[\ f[\tau]\ ]]_S = pr_{D_{\tau}} \circ [[f]]_S$. This completes the description of the semantics. It is a non-trivial exercise to show that this semantics is invariant under the operational semantics of the polymorphic λ-calculus.

5. References:

[1] C. Böhm and A. Berarducci, Automatic synthesis of typed λ-programs on term algebras, Theor. Comp. Sci., 39 (1985), 135 - 154.

[2] K. Bruce and G. Longo, Modest models for inheritance and explicit polymorphism, in Logic in Computer Science, Proceedings Third Annual Symposium, Edinburgh, Scotland, 1988, IEEE Computer Society Press, Los Almitos, CA, 1988, 38 - 50.

[3] P. J. Freyd, *Abelian Categories*, Harper and Row, New York, 1964.

[4] P. J. Freyd, Aspects of Topoi, Bull. Austral. Math. Soc. 7 (1972), 1 - 72.

[5] P. J. Freyd, POLYNAT in PER, in [CCSL], 67 - 68. In *Categories in Computer Science and Logic*, J. W. Gray and A. Scedrov, Eds., Contemporary Mathematics 92, Amer. Math. Soc., Providence, R. I. 1989, 67 - 68.

[6] J.-Y. Girard, Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J.E.Fenstad, editor, Proceedings of the Second Scandinavian Logic symposium, North-Holland Pub. Co., Amsterdam, London, 1971, 63 - 92.

[7] J. W. Gray, The Integration of Logical and Algebraic Types, Proceedings of the International Workshop on Categorical Methods in Computer Science with Aspects from Topology, Sept. 1988, Lecture Notes in Computer Science 393, Springer-Verlag, New York, 1989, 16 - 36.,

[8] J. W. Gray, Initial Algebra Semantics for Lambda Calculi, in *Mathematical Foundations of Programming Language Semantics*, M. Main, A Melton, M. Mislove and D. Schmidt, Eds., 5rd Workshop, New Orleans 1989, Lecture Notes in Computer Science 442, Springer-Verlag, New York, 1989., 418 - 439.

[9] J. Hindley and J. Seldin, Introduction to Combinators and λ-calculus, London Mathematical Society Student Texts: 1, Cambridge University Press, 1986.

[10] J. M. E. Hyland, The effective topos, in *The L.E.J. Brouwer Centenary Symposium*, A. S. Troelstra and D. van Dalen, Eds., North-Holland, Amsterdam, 1982.

[11] M. Hyland, A small complete category, in Proc. of the Conference on Church's Thesis: Fifty Years Later, 1987.

[12] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos theory, Math. Proc. Cambridge Philos. Soc. 88 (1980), 205 - 232.

[13] J. M. E. Hyland, E. P. Robinson, and G. Rosolini, Algebraic types in PER Models, in Mathematical foundations of Programming Semantics, 5th International conference, Tulane 1989, M. Main, A. Melton, M. Mislove and D. Schmidt, Eds., Lecture Notes in computer Science 442, Springer-Verlag, New York, 1990, 333 - 350.

[14] G. Longo and E. Moggi, Constructive natural deduction and its "Modest" interpretation, preprint 1987.

[15] B. Pierce, S. Dietzen, and S. Michaylov, Programming in Higher-Order Typed Lambda-Calculi, Carnegie -Mellon Technical Report, CMU-CS-89-111.

[16] J. C. Reynolds, Towards a theory of type structures, in *Colloque sur la Programmation*, Lecture Notes in Computer Science 19, Springer-Verlag 1974, 408 - 425.

[17] J. C. Reynolds, An introduction to the polymorphic lambda calculus. In G. Huet, editor, Logical foundations of Functional Programming, Proceedings of the Year of Programming Institute, Addison-Wesley, 1988.

[18] E. P. Robinson, How complete is PER?, in Proceedings of the 4th IEEE Symposium on Logic in Computer Science, Computer Science Press of the IEEE, 1989, 106 - 111.

[19] P. Taylor, Recursive Domains, Indexed Category Theory, and Polymorphism, dissertation University of Cambridge, Cambridge, England, 1986.