A Proposed Categorial Semantics for Pure ML

A proposed categorical semantics for ML modules

Michael Fourman
and
Hayo Thielecke

Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

e-mail: mikef@dcs.ed.ac.uk and ht@dcs.ed.ac.uk

June 2, 1995

Abstract

We present a simple categorical semantics for ML signatures, structures and functors. Our approach relies on realizablity semantics in the category of assemblies. Signatures and structures are modelled as objects in slices of the category of assemblies. Instantiation of signatures to structures and hence functor application is modelled by pullback.

1 Introduction

Building on work on the semantics of programming languages in realizability models, in particular that of Wesley Phoa [Pho90] and John Longley [Lon95], we sketch a simple approach to elements of the ML modules system, such as signatures, structures and functors. Once the basic machinery is set up, we will need only quite basic category theory.

This paper is an updated and completely revised version of an earlier paper by Michael Fourman and Wesley Phoa [PF92]. The construction of "generic" (in a sense to be defined below) elements and types presented here is essentially the same as in that paper. However, our presentation is much more elementary. Instead of relying on the internal language of the effective topos, we give explicit descriptions in the realizability model. As in [Lon95], it is hoped that this will make our account comprehensible to Computer Scientists in general.

Apart from these stylistic changes, the main difference from the earlier paper is that we do *not* claim a characterization of the of the "generic predomain" construction in

terms of a universal property in a fairly large class of toposes—as a "classifying topos for predomains"; no hint at a proof of this is given in [PF92], and indeed, as stated there, this claim appears to be simply false. The point is that the "standard" construction of a generic predomain as an object in a polynomial category, sketched in [PF92], classifies *presentations* of predomains, and a given predomain has many presentations. It seems that a rigorous statement of the appropriate universal property would require careful attention to the internal *category* of predomains, which is technically delicate [Hyl88], and would make our account of ML modules yet more inaccessible.

Here we present a more simple-minded approach, inspired by the idea of polynomial categories, but worked out at a concrete level. Perhaps this is mathematically much less compelling than the claims in [PF92], but it is simpler, more accessible, and, we hope, correct.

A much weaker property of the generic predomain appears to be quite sufficient as a basis for all we wish to say about ML modules, and, we can often replace an account in terms of the slice categories used to model modules by talking, instead, about the objects in the base category used to construct these slices. So this paper is primarily an attempt to rescue the ideas on ML presented in [PF92], while basing them on a much humbler, but less dubious, mathematical framework.

We need to warn the reader, however, that this is still only a sketch of work in progress.

2 Semantics in the category of assemblies

We will model our ML-like language in the category of assemblies (over a *partial combinatory algebra*). Readers not familiar with realizability semantics may think of this as a mathematical setting that combines elements of domain theory and the PER model of polymorphism.

2.1 Assemblies

Let $\mathbb{A}$ be the set of natural numbers and $n, m \in \mathbb{A}$, let $n \cdot m$ denote the number that the n-th Turing machine prints on its tape when let loose on the number m or undefined when this does not terminate. In this context we do not think of $\mathbb{A}$ as a set of numbers; instead it is the partial applicative structure that we want to emphasize. Moreover, with some care most of what follows should generalize to other partial applicative algebras. The particular choice of $\mathbb{A}$ here is due only to ease of presentation, as it allows us to avoid technicalities about realizers, appealing instead to the reader's intuition that a certain operation can "obviously" be performed by a Turing Machine. (The category of assemblies for the particular choice of the natural numbers with Kleene application as the partial combinatory algebra is sometimes called ω-sets. For an introduction, see [Pho92]) Let $\langle \cdot, \cdot \rangle$ be a recursive encoding of pairs on $\mathbb{A}$. We define operations $\wedge$ and $\Rightarrow$ on its power set $\mathfrak{P}\mathbb{A}$ as follows: for B,$C \subseteq \mathbb{A}$, let

$$B \wedge C = \{\langle r, s\rangle \in \mathbb{A} \mid r \in B\,, s \in C\}$$
$$B \Rightarrow C = \{r \in \mathbb{A} \mid s \in B \text{ implies } r \cdot s \in C\}$$

2.1 Definition *The category $\mathcal{A}$ of assemblies is defined as follows: an assembly $A = (|A|, \|- \in A\|)$ consists of a set $|A|$ and a function*

$$\|- \in A\| : |A| \longrightarrow \mathfrak{P}\mathbb{A}$$

such that for all $a \in |A|$, $\|a \in A\| \neq \emptyset$.

An assembly A is called modest iff $a \neq b \in |A|$ implies $\|a \in A\| \cap \|b \in A\| = \emptyset$.

A morphism $A \xrightarrow{f} B$ of assemblies is a function $|A| \xrightarrow{f} |B|$ such that the set of its realizers

$$\bigcap_{a \in |A|} (\|a \in A\| \Rightarrow \|f(a) \in B\|)$$

is non-empty.

When defining morphism in the sequel, we will be somewhat sloppy and define only the underlying set-theoretic function, leaving it to the reader to convince himself that this is indeed realizable.

2.2 Slices of the category of assemblies

We will need to work with families of assemblies indexed over an assembly, *i. e.* in slice categories $\mathcal{A}/I$.

For a morphism $I \xrightarrow{f} J$, we will denoted the corresponding pullback functor

$$\mathcal{A}/I \xleftarrow{f^*} \mathcal{A}/J$$

Note that as the category of assemblies is locally cartesian closed, f^* has left and right adjoints $\sum_f \dashv f^* \dashv \prod_f$. In particular, for any I we have the pullback I^* along $I \longrightarrow \mathbf{1}$. We will frequently not distinguish notationally between an object A in $\mathcal{A}$ and its pullback I^*A along $I \longrightarrow \mathbf{1}$.

2.3 Predomains

Recall that the category of assemblies contains objects Σ, ω and $\overline{\omega}$, which in "domain-theoretic" terms one could think of as ascending chains of length 2, ω and $\omega + 1$, respectively.

Let Σ be the assembly $(\{\bot, \top\}, \|- \in \Sigma\|)$ where

$$\|\bot \in \Sigma\| = \{a \in \mathbb{A} \mid a \cdot 0 \uparrow\}$$
$$\|\top \in \Sigma\| = \{a \in \mathbb{A} \mid a \cdot 0 \downarrow\}$$

.

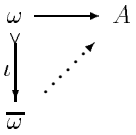
Let the lift $A_\bot$ of an assembly A be $\sum_! \prod_\top !_A$. The lift functor has an initial algebra ω and a terminal algebra $\overline{\omega}$, and there is a canonical injection $\omega \xrightarrow{\iota} \overline{\omega}$. For any assembly I, we can extend these concepts to the slice $\mathcal{A}/I$: the analogue of Σ in $\mathcal{A}/I$ is its pullback along $I \longrightarrow \mathbf{1}$, etc. (But note that one could define all the above objects in the internal language of the ambient topos. The fact that pullback functors preserve such definitions validates the above.)

2.2 Definition *An object $A \in \mathcal{A}/I$ is called a Σ-space iff*

$$\begin{array}{ccc} A & \xrightarrow{\eta} & \Sigma^{\Sigma^A} \\ a & \longmapsto & (f \longmapsto f(a)) \end{array}$$

is monic.

Roughly speaking, we will consider an object A to be domain-like in this setting if it has limits of increasing chains in the sense that every map from the "generic increasing chain" (*i. e.* ω) to A extends uniquely to a map from the "generic increasing chain with a limit point added" (*i. e.* $\overline{\omega}$) to A.



2.3 Definition *An object A in $\mathcal{A}/I$ is* complete *iff*

$$A^{\overline{\omega}} \xrightarrow{A^{\iota}} A^{\omega}$$

is an isomorphism.

This basic idea is elaborated (for somewhat technical reasons) to the following definition, due to John Longley and Alex Simpson [Lon95].

2.4 Definition *An object A in $\mathcal{A}/I$ is* well-complete *iff*

$$A_{\perp}{}^{\overline{\omega}} \xrightarrow{A_{\perp}{}^{\iota}} A_{\perp}{}^{\omega}$$

is an isomorphism.

Let $\mathfrak{WellCompl}\, A$ be the set of realizers for the inverse of $A_{\perp}{}^{\iota}$. We may regard this as the "realizability truth-value" of the proposition "A is well-complete". In particular, this is the empty set iff A is not well-complete; otherwise its elements constitute computational evidence of well-completeness.

We will adopt well-complete Σ-spaces as our notion of "predomain", combining Phoa's notion of Σ-space (which we need for "smallness", see below) with well-completeness (which is a particularly well-behaved notion of "domain-like" object).

3 Generic elements and generic predomains

We describe how to add generic elements and types to our categories. The former is standard, the latter appears to be basically folklore but is hard to find worked out in the literature in any level of detail.

3.1 Generic elements

In order to account for value declarations in ML signatures, which we shall view as being generic in the sense of being free to be instantiated to any value of the appropriate type, we need a way of adding an element of a given type.

The following well-known technique of adding an indeterminate element allows us to do that (this is topos-theoretic folklore, but does not require the full structure of a topos so that we can use it in our more modest setting).

3.1 Fact *For each object $A \in \mathcal{A}$, there exists a "generic element" $\mathbf{1} \xrightarrow{x} A^*A$ of A (in the slice over A), such that for every I in $\mathcal{A}$ and every element $\mathbf{1} \longrightarrow I^*A$ of A in the slice over I, $\mathbf{1} \longrightarrow I^*A$ is a pullback of $\mathbf{1} \xrightarrow{x} A^*A$ along a (unique) map $I \longrightarrow A$.*

The generic element $\mathbf{1} \xrightarrow{x} A^*A$ in $\mathcal{A}/A$ is just

$$\begin{array}{ccc} A & \xrightarrow{\langle 1_A, 1_A \rangle} & A \times A \\ & {\scriptstyle 1_A} \searrow \quad \swarrow {\scriptstyle \pi_1} & \\ & A & \end{array}$$

In the next section we shall address the much less trivial issue how and to what extent we can add not only an element of a given type, but a new *type* itself.

3.2 Construction of a generic predomain

Just as the generic element for an object A was obtained by taking the category of objects "varying" over A (*i. e.* the slice $\mathcal{A}/A$, we will now construct another slice category varying over all types. The reader will suspect that an object of *all* objects looks dubious, and indeed we can only obtain a category varying over a restricted class of objects. Two facts are crucial for this size restriction.

Modest assemblies form an essentially small collection. Let S be a sub-partition of $\mathbb{A}$, *i. e.* a set of non-empty and pairwise disjoint subsets of $\mathbb{A}$. Define the assembly $\hat{S}$ to be $(S, 1_S)$. For an assembly A, define $\breve{A}$ to be the set $\{\|a \in A\| \mid a \in |A|\}$. We have $\breve{\hat{S}} = S$ and, for modest A, $\hat{\breve{A}} \cong A$.

Hence a modest assembly may (up to isomorphism) be recovered from the subpartition of the set of realizers to which it gives rise. Hence we can form a small collection of objects that contains (isomorphic copies) of all modest assemblies.

The second crucial fact is due to Phoa [Pho90]:

3.2 Fact ([Pho90]) *Every Σ-space is modest.*

3.3 Proposition *There exists a family $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ in $\mathcal{A}$ that is generic in the sense that predomains (in any slice) are precisely the pullbacks of it. More precisely,*

1. $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ *is a well-complete* Σ*-space in* $\mathcal{A}/C_0$ *and*

2. *for any well-complete* Σ*-space* $\begin{pmatrix} A \\ \downarrow\alpha \\ I \end{pmatrix}$ *in a slice* $\mathcal{A}/I$ *there exists a morphism* $I \xrightarrow{\ulcorner\alpha\urcorner} C_0$ *such that* $\begin{pmatrix} A \\ \downarrow\alpha \\ I \end{pmatrix}$ *is a pullback of* $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ *along* $\ulcorner\alpha\urcorner$.

$$\begin{array}{ccc} A & \longrightarrow & G \\ {\scriptstyle\alpha}\downarrow & & \downarrow \\ I & \xrightarrow{\ulcorner\alpha\urcorner} & C_0 \end{array}$$

PROOF (Sketch) We define the generic predomain $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ as follows

$$|C_0| = \left\{ \{\|a \in A\| \mid a \in |A|\} \;\middle|\; A \text{ is a well-complete } \Sigma\text{-space} \right\}$$
$$\|S \in C_0\| = \mathfrak{WellCompl}(S, 1_S)$$

and

$$|G| = \{(S, s) | S \in |C_0|, s \in S\}$$
$$\|(S, s) \in G\| = \|S \in C_0\| \wedge s$$

The map $G \longrightarrow C_0$ is defined by $(S, s) \longmapsto S$.

Now a family $\begin{pmatrix} A \\ \downarrow\alpha \\ I \end{pmatrix}$ may be regarded as the indexed collection of the fibres of the map α, *i. e.* for each $i \in |I|$, we can define an assembly A_i by

$$|A_i| = \{a \in |A| \mid \alpha(a) = i\}$$
$$\|a \in A_i\| = \|a \in A\|$$

And it can be shown that $\begin{pmatrix} A \\ \downarrow\alpha \\ I \end{pmatrix}$ is a Σ-space iff all the A_i are and that it is well-complete iff the A_i are (uniformly) well-complete, *i. e.* if there is a realizer in

$$\bigcap_{i \in |I|} (\|i \in I\| \Rightarrow \mathfrak{WellCompl}\, A_i) \,.$$

It follows from this characterization that

1. $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ is a well-complete Σ-space.

2. Given $\begin{pmatrix} A \\ \downarrow\alpha \\ I \end{pmatrix}$, we can define maps

$$\begin{aligned} I &\longrightarrow C_0 \\ i &\longmapsto \mathfrak{WellCompl}\, A_i \\ A &\xrightarrow{\ulcorner\alpha\urcorner} G \\ a &\longmapsto (\{\|b \in A\| \mid \alpha(b) = \alpha(a)\}\ ,\ \|a \in A\|) \end{aligned}$$

   making the diagram above a pullback.

□

3.4 Remark The morphism $I \xrightarrow{\ulcorner\alpha\urcorner} C_0$ is unfortunately not unique; intuitively, it has a certain freedom which (isomorphic) representative of a fibre A_i to pick. So $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ is generic only in a weak sense and does not quite amount to a "predomain classifier".

3.5 Remark A similar construction could be used to introduce a generic complete Σ-space. This is the construction given in [PF92]. The lack of uniqueness of the map $\ulcorner\alpha\urcorner$ making the diagram a pullback could be used to construct a counterexample to the universal property which the corresponding $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ was claimed to enjoy.

3.6 Remark Alternatively, one could use the internal language of the ambient realizability topos to define the generic predomain. That would arguably have been more elegant, but perhaps less accessible.

4 The semantics of the module system

We are now in a position to explain how the generic elements and types of the preceding sections can be used to account for ML modules. (As we shall be quite informal anyway, we refer the reader to Paulson [Pau91] for details of these.)

In our account, signatures will be reified as objects (in suitably constructed slices). Matching, refining and functor application can then be modelled by generalized substitution, *i. e.* pullback functors. We hope that this depiction of signatures as concrete objects helps to formalize programmers' intuitions about them.

We hope that the systematic use of slice categories will facilitate the housekeeping of type and value environments and that the fact that pullback functors preserve all the relevant structure will give an easy way to ensure that denotations that ought to be "the same" are indeed isomorphic. This claim would need a more precise account of the denotational semantics of modules in order to be validated to any extent. We shall content ourselves with giving hints at such a definition.

4.1 Signatures

We begin by showing how signatures live in slice categories. For a signature `SIGA`, we need to distinguish between the object $[\![\texttt{SIGA}]\!]$, which we may regard as being just `SIGA` itself, and its *generic instance* , $(\!|\texttt{SIGA}|\!)$, which is an object in $\mathcal{A}/[\![\texttt{SIGA}]\!]$. Something defined in terms of the type and value environment represented by `SIGA` will be modelled as an object in the slice over $[\![\texttt{SIGA}]\!]$.

Consider the signature

$$\texttt{sig type t end}$$

We will model it as the generic predomain

$$(\!|t|\!) = \begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$$

in $\mathcal{A}/C_0$. Next, consider

$$\texttt{sig type t; val f : t} -> \texttt{t end}$$

Note that $(\!|t|\!)$ lives in $\mathcal{A}/C_0$. As every slice is cartesian closed, there is an exponential

$$(\!|t|\!)^{(\!|t|\!)}$$

in $\mathcal{A}/C_0$. Hence we have a generic element of $(\!|t|\!)^{(\!|t|\!)}$ in $\mathcal{A}/C_0/(\!|t|\!)^{(\!|t|\!)}$; and this is $(\!|f|\!)$.

So we obtain the category in which the denotation of a signature lives by successively slicing $\mathcal{A}$, which gives of course just another slice of A, as for $A \xrightarrow{f} B$ we have $\mathcal{A}/B/f \cong \mathcal{A}/A$. But note that in the example above we sliced $\mathcal{A}/C_0$ by an object that was not already in $\mathcal{A}$. So although all we get by this process are slices of $\mathcal{A}$, they may be over relatively more complicated objects than would be the case if we were just successively adding more indeterminate types to obtain

$$\mathcal{A}/C_0/\cdots/C_0 \quad \cong \quad \mathcal{A}/(C_0 \times \cdots \times C_0)$$

We may even regard this dependence of $[\![f]\!]$ on $[\![t]\!]$ as a rudimentary account of *scope* , inasmuch as the dependence of f on t is reflected in that we first have to slice to get a place where t lives. By contrast, the order of slicing could be permuted in cases such as

$$\texttt{sig type s; type t end}$$

or

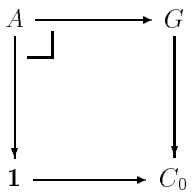
$$\texttt{sig val a : int; val b : int end}.$$

4.2 Structures matching a signature

Now consider a structure matching a signature in the previous example, e.g.

$$\texttt{struct type t = A end : sig type t end}$$

where $[\![\texttt{A}]\!] = A$ for some predomain A. We get a diagram expressing this matching



and by pulling back along $\mathbf{1} \longrightarrow C_0$ we can instantiate the generic t of the signature to the concrete $t = A$. By this instantiation the generic element of $(\!|t|\!)^{(\!|t|\!)}$ then becomes the generic element of A^A in $\mathcal{A}/A$, which can be further instantiated to any real element of A^A in $\mathcal{A}$.

In short we can thus match

$$\texttt{struct type t} = \texttt{A; fun f(x : t)} = \texttt{x end}$$

to

$$\texttt{sig type t; val f : t} -> \texttt{t end}.$$

4.3 Signatures matching a signature

If a signature `SIGB` *matches* another signature `SIGA`, this will allow us to construct a map

$$[\![\texttt{SIGB}]\!] \longrightarrow [\![\texttt{SIGA}]\!].$$

This means that whenever we have matched `SIGB`, then we have (by composition with $[\![\texttt{SIGB}]\!] \longrightarrow [\![\texttt{SIGA}]\!]$) also matched `SIGA`. And (in the opposite direction), whenever we have something implemented in terms of `SIGA`, *i. e.* a structure in the slice over $[\![\texttt{SIGA}]\!]$, then we can pull it back along $[\![\texttt{SIGB}]\!] \longrightarrow [\![\texttt{SIGA}]\!]$ to obtain a structure implemented in terms of `SIGB`.

4.4 Generic structures

Signatures can have substructure components (of a given signature). We can account for them by considering the signature as the type of the structure and then proceeding as we did above for constructing a generic element of a given type.

By the same method as we used for generic elements of a given type, we can construct a generic element, or rather generic *instance* of a signature `SIGA` in the slice $\mathcal{A}/[\![\texttt{SIGA}]\!]$.

This view of treating signatures as the type of the structures that are its instances appears to fit in quite well with the general ideas underlying the ML language.

4.5 Sharing constraints

Consider

```
signature SIGC = sig
   structure sa: SIGA;
   structure sb: SIGB;
   sharing type SIGA.ta = SIGB.tb
end
```

As the denotation of `SIGC`, we take the equalizer

$$[\![\texttt{SIGC}]\!] \rightarrowtail [\![\texttt{SIGA};\texttt{SIGB}]\!] \underset{[\![\texttt{SIGB.tb}]\!]}{\overset{[\![\texttt{SIGA.ta}]\!]}{\rightrightarrows}} C_0$$

If structures matching `SIGA` and `SIGB` are given, this will amount to maps $\cdot \longrightarrow [\![\texttt{SIGA}]\!]$ and $\cdot \longrightarrow [\![\texttt{SIGB}]\!]$ and hence to a map $\cdot \longrightarrow [\![\texttt{SIGA};\texttt{SIGB}]\!]$. And if the components `sa.ta` and `sb.tb` are the same type, the map $\cdot \longrightarrow [\![\texttt{SIGA};\texttt{SIGB}]\!]$ will equalize

$$[\![\texttt{SIGA};\texttt{SIGB}]\!] \underset{[\![\texttt{SIGB.tb}]\!]}{\overset{[\![\texttt{SIGA.ta}]\!]}{\rightrightarrows}} C_0$$

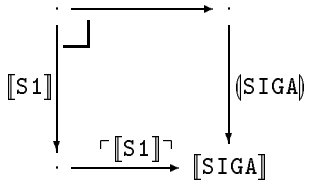
inducing a map $\cdot \longrightarrow [\![\texttt{SIGC}]\!]$.

4.6 Functors

A functor of the form

```
functor F(S: SIGA) = struct ... end
```

will be modelled as a structure in $\mathcal{A}/[\![\texttt{SIGA}]\!]$. Given any a structure `S1` matching the signature `SIGA`, we get



And we can instantiate the actual parameter `S1` for the formal parameter `S` in the body of `F` by pulling the structure in $\mathcal{A}/[\![\texttt{SIGA}]\!]$ representing the body of `F` back along $\ulcorner[\![\texttt{S1}]\!]\urcorner$.

4.7 Summary of the definitions

To summarize, we concentrate on the ML fragment in figure 1 and give a "Pidgin" semantics (see figure 2). This is *not* meant to be formal, we only try to give a more condensed presentation by abusing a familiar and concise formalism (in the style of Pidgin-Algol). In particular, we wanted to emphasize the dependence of each step on previous slice constructions.

Figure 1: A BNF fragment for signatures

$$
\begin{aligned}
Decl &::= \texttt{type}\ Ident \mid \texttt{val}\ Ident : Type \mid \texttt{structure}\ Ident : Sign \\
Decls &::= Decl \mid Decls;\ Decl \\
Sign &::= \texttt{sig}\ Decls\ \texttt{end}
\end{aligned}
$$

Figure 2: A Pidgin Semantics for signatures

$$
\begin{aligned}
[\![\texttt{val a} : \texttt{A}]\!]_{\mathcal{C}} &= [\![\texttt{A}]\!]_{\mathcal{C}} \\
(\!|\texttt{val a} : \texttt{A}|\!)_{\mathcal{C}} &= \mathbf{1} \xrightarrow{\langle 1,1 \rangle} [\![\texttt{A}]\!]^{*}_{\mathcal{C}} [\![\texttt{A}]\!]_{\mathcal{C}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{type t}]\!]_{\mathcal{C}} &= C_0 \\
(\!|\texttt{type t}|\!)_{\mathcal{C}} &= \begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{structure s} : \texttt{SIGA}]\!]_{\mathcal{C}} &= [\![\texttt{val s} : [\![\texttt{SIGA}]\!]]\!]_{\mathcal{C}} \\
(\!|\texttt{structure s} : \texttt{SIGA}|\!)_{\mathcal{C}} &= (\!|\texttt{val s} : [\![\texttt{SIGA}]\!]|\!)_{\mathcal{C}}
\end{aligned}
$$

$$
\begin{aligned}
[\![Decls;\ Decl]\!]_{\mathcal{C}} &= [\![Decl]\!]_{\mathcal{C}/[\![Decls]\!]_{\mathcal{C}}} \\
(\!|Decls;\ Decl|\!)_{\mathcal{C}} &= (\!|Decl|\!)_{\mathcal{C}/[\![Decls]\!]_{\mathcal{C}}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{sig}\ Decls\ \texttt{end}]\!]_{\mathcal{C}} &= [\![Decls]\!]_{\mathcal{C}} \\
(\!|\texttt{sig}\ Decls\ \texttt{end}|\!)_{\mathcal{C}} &= (\!|Decls|\!)_{\mathcal{C}}
\end{aligned}
$$

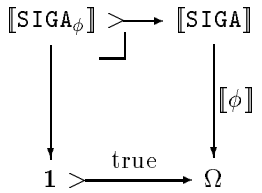
5 Concluding remarks

It remains to be seen to what extent Extended ML [San89] could be accommodated in this framework. In [PF92] a strong case was made for the internal language of the effective topos. We sketch how our ideas can be developed to include Extended ML. When dealing with signatures containing Extended ML axioms, we work (temporarily) in the ambient topos — but the objects thus constructed will again be assemblies.

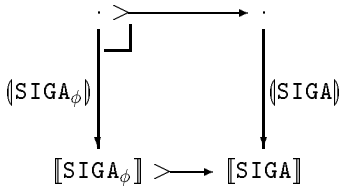
Let the formula ϕ be (the conjunction of) a set of Extended ML axioms imposed on a signature SIGA. Let $\mathtt{SIGA}_\phi$ be the signature SIGA augmented with the axioms in ϕ. By interpreting ϕ in the internal language, we obtain as its denotation a morphism into the object of truth-values

$$[\![\mathtt{SIGA}]\!] \xrightarrow{[\![\phi]\!]} \Omega$$

This classifies the object of all structures satisfying the axioms ϕ:



And its generic instance is the pullback



The object $(\!|\mathtt{SIGA}_\phi|\!)$ in the slice over $[\![\mathtt{SIGA}_\phi]\!]$ is the generic structure satisfying all the axioms. Just as structures matching SIGA arise as pullbacks of $(\!|\mathtt{SIGA}|\!)$, those structures that additionally satisfy the axioms in ϕ are pullbacks of $(\!|\mathtt{SIGA}_\phi|\!)$.

So the internal language allows us to continue our programme of reifying signatures, as we have reified not only matching, but also satisfaction of axioms. (Longley has suggested that a classical logic, which may be interpreted in the internal language, is more appropriate than the constructive internal logic for program verification. It may also be more appropriate in this context. But that issue is orthogonal to the concerns of this paper.)

References

[Hyl88] J.M.E. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40:135–165, 1988.

[Lon95] John Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1995.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[PF92] Wesley Phoa and Michael Fourman. A proposed categorical semantics for Pure ML. In W. Kuich, editor, *Automata, Languages and Programming*, pages 533–544. ICALP, Springer-Verlag, LNCS 623, July 1992.

[Pho90] Wesley Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, University of Cambridge, 1990.

[Pho92] Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, LFCS, April 1992.

[San89] Don Sannella. Formal program development in Extended ML for the working programmer. Technical Report ECS-LFCS-89-102, LFCS, December 1989.