

Higher-Order Functors with Transparent Signatures

Sandip K. Biswas*
Department of CIS
University of Pennsylvania
Philadelphia, PA 19104
sbiswas@saul.cis.upenn.edu

Abstract

The programming language Standard ML provides first-order *functors*, i.e. modules parameterized by modules. First-order functors in the language have a simple and elegant static semantics. The type structure of higher-order modules [3], i.e. modules parameterized by functors, is well understood. But it is only in the recent past that we have seen an implementation of higher-order functors with a formally defined static semantics in a dialect of Standard ML, SML/NJ. A study of this static semantics [7] shows it to be much more complicated than the static semantics of first-order functors. This paper investigates whether we can trade some semantic features in the module language to obtain a simpler static semantics, closer in spirit to that of first-order functors. This work helps in a conceptual understanding of the semantics of higher-order modules.

1 Introduction

Modules are an essential feature of any language which supports the development of large systems. This paper addresses the problem of extending the module system of Standard ML. The current module system of Standard ML provides for parameterized modules or *functors*. Functors in SML are first-order in nature: modules cannot be parameterized by parameterized modules. Here we remove this restriction by providing static semantics for higher-order functors.

1.1 Why higher-order modules?

In general, for any application using first-order functors, we can always make these functors more general by parameterizing the parameters to the functor. We now present two examples detailing specific uses of higher-order functors.

1. In SML/NJ, library routines are implemented as functors. For example, a set is a structure parameterized by a structure containing an order relation. The SML/NJ library has multiple set functors depending

*This research was supported by ONR Contract N00014-89-J-3155.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
POPL '95 1/ 95 San Francisco CA USA
© 1995 ACM 0-89791-692-1/95/0001....\$3.50

on the underlying representation of the set. For example,

```
functor BinarySet(K:ORD_KEY) : ORD_SET, for sets
represented by binary trees, and
functor SplaySet(K:ORD_KEY) : ORD_SET, for sets
represented by splay trees, where,
```

```
signature ORD_KEY =
sig
  type ord_key
  val cmpKey : (ord_key * ord_key)
    -> LibBase.relation
end
signature ORD_SET =
sig
  type item
  type set
  val empty : set
  val singleton : item -> set
  val union : (set * set) -> set
  val numItems : set -> int
  ...
end
```

The signature ORD_SET does not contain the function `closure:(set->set) -> set -> set`, which iterates a function on a set till there is no change in cardinality. But this may be defined using `union,numItems`. In the absence of higher-order functors, a functor with an extended signature containing `closure` has to be defined for every set functor in the library. But with higher-order functors available in the language, the following may be defined,

```
functor ExtendSet
(S : funsig (K:ORD_KEY) ORD_SET)
(K : ORD_KEY) =
struct
  structure Set = S(K);
  open Set;
  fun closure f s = ...
end
```

This functor `ExtendSet` may applied to the functors `BinarySet,SplaySet` to obtain functors whose result has been extended by the function `closure`.

2. There may be two independent software tools generating two functors F and G. A particular application may use the two functors in a very specific way. The

user of this application need not know the structure of the functors, F and G, and how they are joined together. A case in point being the parser generator in SML/NJ. A standard application of the parser generator uses the provided token structure, and the lexer for lexical analysis. Such an application should be able to call a routine which puts together the functors returned from the parser and the lexer. In the absence of higher-order functors this is not possible. For a calculator named Calc, ML-Yacc generates a functor CalcLrValsFun. ML-Lex generates a functor named CalcLexFun. To create a parser the following steps are necessary:

```
structure CalcLrVals =
  CalcLrValsFun
  (structure Token = LrParser.Token) ;
structure CalcLex =
  CalcLexFun
  (structure Tokens = CalcLrVals.Tokens) ;
structure CalcParser =
  Join
  (structure ParserData = CalcLrVals.ParserData
  structure Lex = CalcLex
  structure LrParser = LrParser);
```

If higher-order functors are allowed in the language we may provide a functor Join'. A user using this function need not know how the functors returned from the parser and lexer are brought together.

```
functor Join'(CalcLrValsFun,CalcLexFun) =
  struct
    structure CalcLrVals =
      CalcLrValsFun
      (structure Token = LrParser.Token) ;
    structure CalcLex =
      CalcLexFun
      (structure Tokens = CalcLrVals.Tokens) ;
    structure CalcParser =
      Join(
        structure ParserData = CalcLrVals.ParserData
        structure Lex = CalcLex
        structure LrParser = LrParser);
  end
```

1.2 Transparent Signatures

A key feature of the module system of SML is the presence of *transparent* signatures. This means that after a `structure` is constrained by a `signature` the identities of the components of the structure are not hidden as abstract entities. Similarly, during a functor application, identities of the components are propagated. The following example illustrates of the concept,

```
signature ORDER = sig
  type t
  val less : t -> t -> bool
end
functor Ord_List (K : ORDER) : ORDER =
  struct
    type t = K.t list
    fun less [] _ = true
      | less _ [] = false
```

```
  | less (a1 :: rest1) (a2 :: rest2) =
    if (K.less a1 a2)
    then (less rest1 rest2)
    else false
  end
structure int_order : ORDER =
  struct
    type t = int
    fun less (x:t) (y:t) = x < y
  end
structure int_list_order = Ord_List(int_order)
val _ = int_list_order.less [1,2,3] [2,3,5]
```

The above program statically type checks, even though the functor `Ord_List` constrains `int_list_order` to have the generic signature `ORDER`, because `int_list_order.t = int list` is transparent to the rest of the program.

Transparency in the higher-order case is more complicated. Consider the following program:

```
functor APP =
  functor(F : funsig (X : sig type t; val x:t end)
    sig type s; val x:s end)
    functor(X: sig type t ; val x:t end)
      (F X)
  functor F = Functor(X : sig type t; val x:t end)
    struct
      type s = X.t list
      val x = [ X.x ]
    end
  structure X = struct type t = int; val x = 4 end
  structure Y = ((APP F) X)
  val _ = car (Y.x)
```

If signatures are completely transparent this program must statically type check. This means that the semantic representation of `APP` must indicate that the dependency between `t` and `s` present in the argument `F`, even though not known statically, has to be propagated.

An important question to be addressed is, what are the benefits associated with completely transparent signatures? In [6] it is shown that completely opaque signatures or abstract datatypes are not acceptable for expressing modular structure of programs, as abstract types have no identity. More importantly no propagation of type information is possible across function boundaries. This means that elimination constructs associated with abstract types have restrictions on type variables which can be present in the final expression. In [1], it is shown that it is possible to give identities to abstract types but propagation of type information across function boundaries is not possible.

Translucent signatures as proposed in [5, 2] have certain limitations with regard to programming with higher-order modules. As shown in [5], translucent signatures are completely equivalent to transparent signatures in the first-order case. This equivalence fails when we have a functor taking in a functor as an argument because there is no uniform way of propagating information associated with the argument functor without restricting the set of functors which can be accepted as arguments. This problem of propagating information associated with argument functors in translucent signatures has been recently fixed. But there is still a fundamental problem associated with translucent signatures: programs have to explicitly state the type dependency to be propagated beyond a functor boundary.

Completely transparent signatures, as in SML/NJ, do not require tags in signatures stating type information. Hence the APP functor can be written the way it has been: with generic signatures.

2 Relation to Existing Work

The following is an extremely informal presentation of the semantics of first-order functors in SML. A formal version may be found in [9, 10]. A module or structure in SML has a semantic representation which is essentially a record with fields giving the types associated with the components of the structure. The semantic representation associated with a signature, i.e. specification of the interface of a module, is the same as that of a structure, with components which are left unspecified being represented by variables which are universally quantified. In SML, a functor with a name FF is defined as follows,

$$\text{functor FF } (X : \text{signature}_1) = \text{structure}_1 \quad (1)$$

If $\llbracket \text{signature}_1 \rrbracket = \forall \vec{N}. (\Sigma_1)$ then the semantic representation of the functor FF is given by $\forall \vec{N}. (\Sigma_1, \Sigma_2)$, where Σ_2 is the result of elaboration of structure_1 in an environment which maps X to Σ_1 . If a functor, with semantic representation $\forall \vec{N}. (\Sigma_1, \Sigma_2)$, is applied to a structure, with semantic representation S_1 , then the semantic representation of the resulting expression is computed as follows:

- Obtain a substitution ψ with domain \vec{N} , such that S_1 matches $\psi(\Sigma_1)$ as per certain rules.
- Return $\psi(\Sigma_2)$ as the semantic representation of the expression.

The semantics for higher-order functors in SML/NJ, as presented in [7], is much more complicated than the first-order case. In this case, the semantic representation of structure and functor signatures is no longer an abstraction of the semantic representation of structures and functors. A functor declaration for FF as in (1), is elaborated as follows: Let Σ_1 be the ‘free’¹ structure associated with signature_1 . Let Σ_2 be the result of elaboration of structure_1 in an environment which maps X to Σ_1 . But what is returned as the elaboration of the functor FF is $(\lambda X : \text{signature}_1. \text{Algorithm}_1)$, where Algorithm_1 is a step-by-step description for generating Σ_2 from structure_1 . This description is in an intermediate language, with a defined operational semantics, and contains references to X and its components. If a structure with representation S_1 is applied to FF then the elaboration proceeds as follows: First it is verified that S_1 matches signature_1 . Next, references to the variable X in Algorithm_1 are substituted by S_1 and the resulting expression is evaluated, as per the operational semantics of the intermediate language, to generate the semantic representation of the application expression.

Standard ML has a novel feature, *generativity*. What this means is that the same structure applied to the same functor returns structures which may differ in certain components, which are generated afresh at each application. The question which we would like to address is, if generativity, as a language feature is removed, can we obtain a simplification of the static semantics? If we wish to provide static

¹A free structure associated with $(\forall \vec{N}. \Sigma)$ is $[\vec{a}/\vec{N}](\Sigma)$, where \vec{a} is a fresh set of variables never used before

semantics to a higher-order module system for a strongly-typed language, which does not have a generativity in its semantics, are we tied to a semantics where functors have a two phase elaboration: one at definition time and the other at application time? Can we obtain a semantics similar to that of first-order functors, where the elaboration of functor application is merely the application of a substitution computed during the signature matching process? The rationale behind the semantics in [7] was to propagate dependencies present in functors. In the absence of generativity, when such dependencies are unknown, the obvious strategy is to represent them as universally quantified variables which can be instantiated during the signature matching process. To show that this intuitive strategy works out is non-trivial. This is because these dependency variables are going to be functions or simply-typed λ -terms and, as shown in [4] for the simply-typed λ -calculus most general unifiers do not exist and even computing whether a unifier exists is undecidable.

3 A Language for Higher-Order Modules

3.1 Syntax of the Language

The syntax of the language used, is the same as in [5]. As both structures and functors can be fields of a structure, we use a generic term *module* to refer to both structures and functors. As *val* expressions do not contribute to any change in the static semantics, the language presented here contains only *module* and *type* expressions as fields of structures.

Unlike SML, structures have no identity in the language to be discussed. Hence, sharing of structures is not allowed. Only sharing of types is allowed. As a specification of sharing, SML allows equating arbitrary paths. As discussed in [11], this immediately prevents a syntax-directed translation of a syntactic signature into its semantic representation. Instead, the translation has to be specified as a proof system for which it is necessary to prove the property of principality. In this paper, we would like to present a new semantic representation for the elaboration of functors. Sharing is attained in our language by assigning fields in a signature to a path name. Such a specification allows a simple syntax-directed translation of syntactic signatures to their semantic counterparts. This restriction on sharing specifications is merely for the purpose of exposition: it gives a simpler translation, while retaining the expressive power of the constraints. Accommodating sharing constraints as in SML, is merely a matter of changing the translation algorithm: an issue really orthogonal to the subject of our presentation.

As pointed in [2], having internal α -convertible names for fields is especially convenient, when sharing is obtained by explicit assignment. This is overlooked here, as this has no effect on the semantics.

The language is defined by the grammar below. It is assumed that the fields in a signature body S , or a structure body s are all distinct.

Type Expressions:

$$\tau ::= \text{int} \mid \text{bool} \mid \tau \rightarrow \tau \mid \text{path}$$

Paths:

$$\text{path} ::= t \mid x.\text{path}$$

Signature Components:

$$S_c ::= \text{type } t \mid \text{type } t = \tau \mid \text{module } x : \text{Sig}$$

Signature Body:

$S ::= S_c \mid S_c ; S$

Signature:

$Sig ::= \mathbf{sig} S \mathbf{end} \mid \mathbf{funsig}(x : Sig_1) Sig_2$

Structure Components:

$s_c ::= \mathbf{type} t = \tau \mid \mathbf{module} x = m$

Structure Body:

$s ::= s_c \mid s ; s$

Module Expressions:

$m ::= x \mid m.x \mid m_1 m_2 \mid \mathbf{struct} s \mathbf{end}$
 $\mid \mathbf{functor}(f : Sig) m \mid m : Sig$

3.2 Semantic Representation of Signatures

The semantic representation of signatures is similar to [9], with appropriate extensions for higher-order functors, and appropriate simplifications for the absence of equality between structures.

$$\begin{aligned} \mathcal{T} &\in \text{Mod} \equiv \text{StrEnv} + \text{Func} \\ (\text{TE}, \text{ME}) &\in \text{StrEnv} \equiv \text{TypEnv} \times \text{ModEnv} \\ \text{TE} &\in \text{TypEnv} \equiv \text{TypId} \xrightarrow{\text{fn}} \text{Type} \\ \text{ME} &\in \text{ModEnv} \equiv \text{ModId} \xrightarrow{\text{fn}} \text{Mod} \\ \forall S(\mathcal{T}_1 \Rightarrow \mathcal{T}_2) &\in \text{Func} \equiv \text{NameSet} \times \text{Mod} \times \text{Mod} \end{aligned}$$

Notation:

- $X \xrightarrow{\text{fn}} Y$, is the set of all partial functions from domain X to co-domain Y , where every function is defined on a finite subset of X .
- $\text{TypId}, \text{ModId}, \text{NameSet}$ are domains of the set of identifiers/variables.

In [9], variables are used in the semantic representation exclusively for types and stamps: structure identifiers, which are abstracted away at compile-time. When functors are passed as arguments, the argument-to-result dependency present in the functor is not known, but needs to be propagated. These dependencies are expressed by higher-order variables in the semantic representation.

Definition: $\text{TE} \in \text{Type}$ is defined by the following grammar,

$$\begin{aligned} \text{Type} &::= \text{Type}_{\text{atom}} \mid \text{Type}_1 \rightarrow \text{Type}_2 \\ \text{Type}_{\text{atom}} &::= \text{int} \mid \text{bool} \mid x \mid f(H_1, \dots, H_n) \\ H &::= \text{Type} \mid \lambda x. H \mid H_1 H_2 \\ &\text{where } f, x \text{ are variables} \end{aligned}$$

$\text{Type}_{\text{atom}}$ denotes the set of atomic type expressions.

Definition: The atomic subexpressions of a type T , $\text{ASExp}(\text{T})$, is a subset of $\text{Type}_{\text{atom}}$ defined as follows,

$$\begin{cases} \{\text{T}\} & \text{if } \text{T} \in \text{Type}_{\text{atom}} \\ \text{ASExp}(\text{T}_1) \cup \text{ASExp}(\text{T}_2) & \text{if } \text{T} \equiv \text{T}_1 \rightarrow \text{T}_2 \end{cases}$$

A syntactic signature Σ , as defined by the grammar Sig , is represented as a semantic object $\mathcal{T} \in \text{Mod}$.

- $\Sigma \equiv \mathbf{sig} S \mathbf{end}$, is represented as $(\text{TE}, \text{ME}) \in \text{StrEnv}$, with its type components generating an environment TE and its module components generating an environment ME .

- $\Sigma \equiv \mathbf{funsig}(x : \Sigma_1) \Sigma_2$, is represented as $\forall S(\mathcal{T}_1 \Rightarrow \mathcal{T}_2) \in \text{Func}$.

Every signature Σ has a unique semantic representation as defined by a syntax-directed translation function $\mathcal{E}[\cdot]$.

Notation:

- VarList denotes the set of lists of variables.
- Env is a partial function, with finite domain, from the set of variables to values, which are either elements of Type or Mod .
- For any two environments E_1, E_2

$$\text{E}_1 \sqcup \text{E}_2(x) = \begin{cases} \text{E}_2(x) & \text{if } \text{E}_2(x) \text{ is defined} \\ \text{E}_1(x) & \text{if } \text{E}_1(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$
- $l_1 @ l_2$, is the list formed by appending two lists l_1 and l_2 .

$$\mathcal{E}[\cdot] : \text{Sig} \rightarrow \text{Env} \rightarrow \text{VarList} \rightarrow \text{Mod} \times \text{VarList}$$

$$\mathcal{E}_L[\cdot] : S \rightarrow \text{Env} \rightarrow \text{VarList} \rightarrow (\text{TypEnv} \times \text{ModEnv}) \times \text{VarList}$$

$$\mathcal{E}_C[\cdot] : S_c \rightarrow \text{Env} \rightarrow \text{VarList} \rightarrow (\text{TypEnv} \times \text{ModEnv}) \times \text{VarList}$$

$$\mathcal{E}[\mathbf{sig} S \mathbf{end}]_{\Gamma \mathcal{W}} = \mathcal{E}_L[S]_{\Gamma \mathcal{W}}$$

$$\begin{aligned} \mathcal{E}[\mathbf{funsig}(x : \Sigma_1) \Sigma_2]_{\Gamma \mathcal{W}} &= \\ \text{let } (\mathcal{T}_1, \mathcal{V}_1) &= \mathcal{E}[\Sigma_1]_{\Gamma []} \\ (\mathcal{T}_2, \mathcal{V}_2) &= \mathcal{E}[\Sigma_2]_{\Gamma[x \mapsto \mathcal{T}_1] \mathcal{W} \circ \mathcal{V}_1} \\ \text{in } &(\forall \mathcal{V}_1 (\mathcal{T}_1 \Rightarrow \mathcal{T}_2), \mathcal{V}_2) \end{aligned}$$

$$\mathcal{E}_L[S_c]_{\Gamma \mathcal{W}} = \mathcal{E}_C[S_c]_{\Gamma \mathcal{W}}$$

$$\begin{aligned} \mathcal{E}_L[S_c; S]_{\Gamma \mathcal{W}} &= \\ \text{let } ((\text{TE}_1, \text{ME}_1), \mathcal{V}_1) &= \mathcal{E}_C[S_c]_{\Gamma \mathcal{W}} \\ ((\text{TE}_2, \text{ME}_2), \mathcal{V}_2) &= \mathcal{E}_L[S]_{\Gamma \sqcup \text{TE}_1 \sqcup \text{ME}_1 \mathcal{W}} \\ \text{in } &((\text{TE}_1 \sqcup \text{TE}_2, \text{ME}_1 \sqcup \text{ME}_2), \mathcal{V}_1 @ \mathcal{V}_2) \end{aligned}$$

$$\mathcal{E}_C[\mathbf{type} t]_{\Gamma \mathcal{W}} = ((t \mapsto f(\mathcal{W}), \emptyset), \{f\})$$

where f is a completely new variable.

$$\mathcal{E}_C[\mathbf{type} t = \tau]_{\Gamma \mathcal{W}} = ((t \mapsto [\tau]_{\Gamma}, \emptyset), \emptyset)$$

where $[\tau]_{\Gamma}$ merely replaces all free-variables and paths in τ by their values in the environment Γ .

$$\mathcal{E}_C[\mathbf{module} x : \Sigma]_{\Gamma \mathcal{W}} = \text{let } (\mathcal{T}, \mathcal{V}_1) = \mathcal{E}[\Sigma]_{\Gamma \mathcal{W}} \\ \text{in } ((\emptyset, x \mapsto \mathcal{T}), \mathcal{V}_1)$$

The following example illustrates the concept:

$$\begin{aligned} \mathcal{E}[\mathbf{funsig}(X : \mathbf{sig} \mathbf{type} t \mathbf{end}) \mathbf{sig} \mathbf{type} s \mathbf{end}]_{\emptyset []} &= \\ (\forall a((\{t \mapsto a\}, \emptyset) \Rightarrow (\{s \mapsto f(a)\}, \emptyset)), [f]) \end{aligned}$$

The idea being that, in the syntactic signature the dependency between t and s is not known statically. But the dependency can be abstracted by a function variable f .

4 Static Semantics

4.1 Elaboration Rules

The rules for elaboration of structure expressions in the language are presented in Figure 1. The critical set of rules here are the signature matching rules, (4) and (5), because they involve a substitution. In the presence of higher-order variables a most general substitution does not exist,² and

²For example, if $f \text{ int} = \text{int}$ then both $f/(\lambda x. \text{int})$ and $f/(\lambda x. x)$ are valid substitutions.

$$\Gamma \vdash x : \Gamma(x) \quad (1)$$

$$\frac{\Gamma \vdash m : (\text{TE}, \text{ME})}{\Gamma \vdash m.x : \text{ME}(x)} \quad (2)$$

$$\frac{\Gamma, f : \mathcal{T}_1 \vdash m : \mathcal{T}_2 \quad \text{where } (\mathcal{T}_1, \mathcal{V}) = \mathcal{E}[\Sigma]_{\Gamma \square}}{\Gamma \vdash \text{functor}(f : \Sigma) m : \forall \mathcal{V}(\mathcal{T}_1 \Rightarrow \mathcal{T}_2)} \quad (3)$$

$$\frac{\Gamma \vdash m_1 : \forall \mathcal{V}(\mathcal{T}_1 \Rightarrow \mathcal{T}_2) \quad \Gamma \vdash m_2 : \mathcal{T} \quad (\Gamma \triangleright \mathcal{T}, \mathcal{T}_1, \mathcal{V} \Downarrow \psi)}{\Gamma \vdash m_1 m_2 : \psi(\mathcal{T}_2)} \quad (4)$$

$$\frac{\Gamma \vdash m : \mathcal{T} \quad (\Gamma \triangleright \mathcal{T}, \mathcal{T}_1, \mathcal{V} \Downarrow \psi) \quad \text{where } (\mathcal{T}_1, \mathcal{V}) = \mathcal{E}[\Sigma]_{\Gamma \square}}{\Gamma \vdash (m : \Sigma) : \psi(\mathcal{T}_1)} \quad (5)$$

$$\frac{\Gamma \vdash s : \mathcal{T}}{\Gamma \vdash \text{struct } s \text{ end} : \mathcal{T}} \quad (6)$$

$$\frac{\Gamma \vdash s_c : (\text{TE}_1, \text{ME}_1) \quad \Gamma \square \text{TE}_1 \square \text{ME}_1 \vdash s : (\text{TE}_2, \text{ME}_2)}{\Gamma \vdash s_c ; s : (\text{TE}_1 \square \text{TE}_2, \text{ME}_1 \square \text{ME}_2)} \quad (7)$$

$$\Gamma \vdash \text{type } t = \tau : (\{t \mapsto [\tau]_{\Gamma}\}, \emptyset) \quad (8)$$

$$\frac{\Gamma \vdash m : \mathcal{T}}{\Gamma \vdash \text{module } x = m : (\emptyset, \{x \mapsto \mathcal{T}\})} \quad (9)$$

Figure 1: Rules for Elaboration

there is no decidable algorithm for matching. But we subsequently show that all our terms are in a particular syntactic form for which a most general unifier exists. The terms are in the format required for unification in L_λ , [8]: variables, for which substitution is required, always take distinct universally quantified variables as arguments. If the \Downarrow relation is *defined*, by structural induction on terms, then the entire semantics becomes completely definitional. A programmer has some intuitive notion of what subsumptions are valid. In a completely definitional semantics it is difficult to analyse whether the semantics manages to capture these intuitions. So instead we capture these intuitions as *specifications*. Then we go on to show that there is precisely a *unique* substitution which satisfies these specifications and this unique substitution may be computed by a simple algorithm.

The intuitive notions of subsumption for a programmer are:

- Record Polymorphism: If structure \mathcal{T}_1 has an extra set of fields over another structure \mathcal{T}_2 then \mathcal{T}_2 is more general than \mathcal{T}_1 , $\mathcal{T}_1 \prec \mathcal{T}_2$.
- Contravariant Functors: $\mathcal{T}_1 \Rightarrow \mathcal{T}_2 \prec \mathcal{T}'_1 \Rightarrow \mathcal{T}'_2$, if $\mathcal{T}'_1 \prec \mathcal{T}_1$ and $\mathcal{T}_2 \prec \mathcal{T}'_2$.
- Functorial Polymorphism: This is a new kind of polymorphism found in higher-order functors. The concept is best illustrated by an example. If a certain context expects a functor of signature:

$$\begin{array}{l} \text{funsig}(X : \text{sig type } t = \text{int end}) \\ (\text{sig type } t = \text{int end}) \end{array}$$

then a functor of signature
 $\text{funsig}(X : \text{sig type } t \text{ end})$
 $(\text{sig type } t = X.t \text{ end})$
 would do as well.

Intuitively, the relation $(\Gamma \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$, where ψ is a substitution with domain \mathcal{V} , is provable when $\psi(\mathcal{T}_2)$ is more general than \mathcal{T}_1 , by criteria to be described. Γ is a necessary parameter in the relation \Downarrow , as $\psi(\mathcal{T}_2)$ and \mathcal{T}_1 must be closed wrt the set free variables in $\text{range}(\Gamma)$. In the formal definition of the \Downarrow relation, Γ is replaced by Δ , where Δ is the set of free variables in $\text{range}(\Gamma)$.

Definition: $\Delta \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi$ iff

- The domain of ψ , $|\psi| = \mathcal{V}$.
- $\Delta \cap \mathcal{V} = \emptyset$.
- \mathcal{T}_1, ψ are closed wrt Δ .
- \mathcal{T}_2 is closed wrt $(\Delta \cup \mathcal{V})$ and $Fv(\mathcal{T}_2) - \Delta = \mathcal{V}$.
- $\Delta \triangleright \mathcal{T}_1 \prec \psi(\mathcal{T}_2)$

The subsumption relation \prec , as defined in Figure 2, is broken structurally into components \prec_T and \prec_L . \prec_T defines subsumption on TypEnv and \prec_L defines subsumption on ModEnv . For convenience, environments are assumed to be lists sorted by field names. To prevent capture of variables on a substitution, by the binding constructs in the language, it is assumed that variables bound by \forall, λ are distinct from

$$\frac{\Delta \triangleright TE_1 \prec_T TE_2 \quad \Delta \triangleright ME_1 \prec_L ME_2}{\Delta \triangleright (TE_1, ME_1) \prec (TE_2, ME_2)} \quad (1)$$

$$\Delta \triangleright \{\} \prec_L \{\} \quad (2)$$

$$\frac{\Delta \triangleright ME_1 \prec_L ME_2}{\Delta \triangleright \{(x \mapsto T), ME_1\} \prec_L ME_2} \quad (3)$$

$$\frac{\Delta \triangleright T_1 \prec_L T_2 \quad \Delta \triangleright ME_1 \prec_L ME_2}{\Delta \triangleright \{(x \mapsto T_1), ME_1\} \prec_L \{(x \mapsto T_2), ME_2\}} \quad (4)$$

$$\Delta \triangleright \{\} \prec_T \{\} \quad (5)$$

$$\frac{\Delta \triangleright TE_1 \prec_T TE_2}{\Delta \triangleright \{(t \mapsto T), TE_1\} \prec_T TE_2} \quad (6)$$

$$\frac{\Delta \triangleright TE_1 \prec_T TE_2}{\Delta \triangleright \{(t \mapsto T), TE_1\} \prec_T \{(t \mapsto T), TE_2\}} \quad (7)$$

$$\frac{\Delta \cup \mathcal{V}'_2 \triangleright T'_2, T'_1, \mathcal{V}'_1 \Downarrow \psi' \quad \Delta \cup \mathcal{V}'_2 \triangleright \psi'[T'_1] \prec T'_2}{\Delta \triangleright \forall \mathcal{V}'_1 (T'_1 \Rightarrow T'_1'') \prec \forall \mathcal{V}'_2 (T'_2 \Rightarrow T'_2'')} \quad (8)$$

Figure 2: A proof system for subsumption

each other and from the free variables in the environment. The subsumption relation \prec captures the intuitive notion of subsumption.

Rule 8 is the only non-trivial rule in Figure 2. It captures both functorial polymorphism and contravariance in functors. To prove $\forall \mathcal{V}'_1 (T'_1 \Rightarrow T'_1'') \prec \forall \mathcal{V}'_2 (T'_2 \Rightarrow T'_2'')$, it is necessary to prove $T \prec T'_2''$, where T is the result of the elaboration of the application of a functor of semantic signature $\forall \mathcal{V}'_1 (T'_1 \Rightarrow T'_1'')$ to a module of semantic signature T'_2 .

It is important to notice that we only have a specification of the \Downarrow relation. This specification does not provide an algorithm to compute the relevant substitution ψ . Any substitution ψ which satisfies the subsumption rules is a valid substitution. In the section 6, we prove that there is a unique substitution which satisfies the \Downarrow relation for any program which elaborates.

4.2 An Example

Consider the following program:

```

module APP = functor(F : funsig (X : sig type t end)
                        sig type s end)
  functor(X: sig type t end) (F X)

module F = functor(X : sig type t end)
  struct
    type s = X.t -> X.t
  end

module G = APP F

module X = struct

```

```

type s = bool;
type t = int;
end

```

module H = G X

The program given above does not have modules as components of structures anywhere, so for reasons of readability the empty sets denoting *ModEnv* are deleted. Hence, a structure $(\{t \mapsto \text{int}\}, \emptyset)$ is replaced by $\{t \mapsto \text{int}\}$.

$\llbracket \text{APP} \rrbracket = \forall f (\forall a (\{t \mapsto a\} \Rightarrow \{s \mapsto f(a)\}) \Rightarrow \forall b (\{t \mapsto b\} \Rightarrow \{s \mapsto f(b)\}))$.

$\llbracket F \rrbracket = \forall a (\{t \mapsto a\} \Rightarrow \{s \mapsto (a \rightarrow a)\})$.

As $(\triangleright \llbracket F \rrbracket, \forall a (\{t \mapsto a\} \Rightarrow \{s \mapsto f(a)\}), \{f\} \Downarrow \{f/\lambda a. a \rightarrow a\})$,

$\llbracket G \rrbracket = \forall b (\{t \mapsto b\} \Rightarrow \{s \mapsto (b \rightarrow b)\})$.

$\llbracket X \rrbracket = \{s \mapsto \text{bool}, t \mapsto \text{int}\}$

As $(\triangleright \llbracket X \rrbracket, \{t \mapsto b\}, \{b\} \Downarrow \{b/\text{int}\})$,

$\llbracket H \rrbracket = \{s \mapsto (\text{int} \rightarrow \text{int})\}$

5 Well-Formedness of Semantic Representations

The grammar for the semantic representation includes the entire simply-typed λ -calculus. Hence, as discussed in [4], in general the substitution ψ , involved in the subsumption process, may not be computable and may not be unique. But all semantic representations appearing in the elaboration of a term come from a very restricted subset of the defining grammar. In this section we characterise this restricted subset of semantic representations by a predicate.

In the next section, we show that for representations belonging to this restricted subset, the substitution ψ in the \Downarrow relation is unique and computable.

Because of the presence of higher-order variables, it is necessary to prove that at no point does normalisation, after substitution, fail because an attempt is made to apply a non-function value to a value. To prove this, we need to introduce kinds. It is to be kept in mind that kinds are artifacts introduced to get a simple formulation and proof of the statement: no errors occur during normalisation after a substitution. The changes, mentioned in this section, to functions like \mathcal{E} , and the elaboration rules, are merely for purposes of proof. The actual implementation remains unkinded. $T \in \text{Type}$ is considered to be of kind o , the only base kind in the system. From hereon, all variables are considered kinded. Binding constructs, \forall and λ , are assumed to bind kinded variables. The translation function for signatures has the type,

$$\mathcal{E}[\!]: \text{Sig} \rightarrow \text{Env} \rightarrow \text{VarList} \rightarrow \text{Mod} \times \text{VarList}$$

As mentioned, VarList in the result now needs to be a kinded list of variables. To achieve this, only one modification is required in the definition of \mathcal{E} :

$$\mathcal{E}_C[\text{type } t]_{\Gamma} \mathcal{W} = ((t \mapsto f(\mathcal{W}), \emptyset), [f : k_1 * \dots * k_n \rightarrow o])$$

where $\mathcal{W} \equiv [f_1 : k_1, \dots, f_n : k_n]$ and f is a completely new variable.

The inference rules for the static semantics now carry another parameter Δ , along with Γ . Δ is the set of all free variables in Γ , with their associated kinds. Calls to the \Downarrow relation now use Δ instead of Γ . All rules merely propagate Δ , except Rule (3), the rule for functor definitions.

$$\frac{\Delta \cup \mathcal{V}, (\Gamma, f : T_1) \vdash m : T_2 \quad \text{where } (T_1, \mathcal{V}) = \mathcal{E}[\Sigma]_{\Gamma} \square}{\Delta, \Gamma \vdash \text{functor}(f : \Sigma) m : \forall \mathcal{V}(T_1 \Rightarrow T_2)}$$

Definition: $\text{Aat}(T)$ defines the set of Accessible atomic type expressions in $T \in \text{Mod}$:

- $\text{Aat}(\forall \mathcal{V}(T_1 \Rightarrow T_2)) = \emptyset$
- $\text{Aat}(\text{TE} \equiv \{t_1 \mapsto T_1, \dots, t_m \mapsto T_m\}, \text{ME} \equiv \{s_1 \mapsto T_1, \dots, s_n \mapsto T_n\}) = \bigcup_{j=1}^m \text{ASExp}(T_j) \cup \bigcup_{i=1}^n \text{Aat}(T_i)$

To specify the well-formedness of semantic representations, two recursive predicates \mathcal{G} and \mathcal{S} are defined in Figure 3. $\mathcal{P}(X)$ for any set X denotes the powerset of X .

A semantic representation \mathcal{T} is well-formed wrt a set Δ , if $\mathcal{G}(\mathcal{T}, \Delta)$. A structure \mathcal{T} , without functor components, satisfies the relation $\mathcal{G}(\mathcal{T}, \Delta)$, if all the type components in \mathcal{T} are closed and well-kinded wrt Δ . We can give a precise characterisation of a semantic representation \mathcal{T} obtained from the translation of a syntactic signature Σ , wrt an environment Γ and a set of variables \mathcal{W} . The elaboration process merely modifies these semantic representations by instantiating variables bound by universal quantifiers. If $\mathcal{E}[\Sigma]_{\Gamma} \mathcal{W} = (\mathcal{T}, \mathcal{V})$ then we can always define a predicate such that $(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Gamma)$ satisfies it. But if Γ is replaced by its corresponding Δ , i.e. the set of free variables in the range of Γ , then for book-keeping reasons we need to introduce another parameter ρ , a set of atomic type expressions, in the predicate. This is the predicate \mathcal{S} defined in Figure 3.

Well-formedness of a functor signature $\forall \mathcal{V}(T_1 \Rightarrow T_2)$ wrt Δ , is more complicated. We require $\mathcal{G}(T_2, \Delta \cup \mathcal{V})$ and also $\mathcal{S}(T_1, \mathcal{V}, [], \Delta, \emptyset)$.

Note:

1. It is assumed that free variables in ρ are kinded. If $\mathcal{V}, \mathcal{W}, \Delta, \rho$ have variables in common then they have the same kind.
2. Both the predicates are monotone in the Δ component.
3. If $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta, \rho_1 \cup \rho_2)$ then $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta \cup \text{Fv}(\rho_1), \rho_2)$
4. If $\mathcal{S}((\text{TE}, \text{ME}), \mathcal{V}, \mathcal{W}, \Delta, \rho)$ then $\text{Aat}(\text{TE}, \text{ME})$ is given by the following grammar:

$$\begin{aligned} T_0 &::= T_{\text{atom}} \quad \text{where } T_{\text{atom}} \text{ of kind } o \text{ is closed w.r.t } \\ &\quad \Delta \text{ and kind-correct} \\ &| f(\mathcal{W}) \quad \text{where } f \in \mathcal{V} \text{ and } f(\mathcal{W}) \\ &\quad \text{is kind-correct of kind } o \\ &| t \quad \text{where } t \in \rho \end{aligned}$$

Lemma 5.1 *If $\mathcal{G}(\mathcal{T}, \Delta)$ then \mathcal{T} is closed w.r.t Δ and kind-correct.*

If $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta, \rho)$ s.t. ρ is kind-correct, then $\mathcal{G}(\mathcal{T}, \Delta \cup \mathcal{V} \cup \mathcal{W} \cup \text{Fv}(\rho))$.

Proof: By induction on the structure of \mathcal{T} , applying the observations made in the note above. \square

Lemma 5.2 *If $\mathcal{E}[\Sigma]_{\Gamma} \mathcal{W} = (\mathcal{T}, \mathcal{V})$, $\mathcal{W} \subseteq \text{Fv}(\Gamma)$ and $\mathcal{G}(\Gamma, \Delta)$ then $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta, \emptyset)$.*

Proof: The proof is an immediate corollary of the following lemma:

If $\mathcal{E}[\Sigma]_{\Gamma} \mathcal{W} = (\mathcal{T}, \mathcal{V})$, where $\mathcal{W} \subseteq \text{Fv}(\Gamma)$ and $\rho_{\Gamma} = \text{Aat}(\Gamma)$, then $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \emptyset, \rho_{\Gamma})$ and $\text{Fv}(\mathcal{T}) - \text{Fv}(\Gamma) = \mathcal{V}$. \square

Lemma 5.3 (Substitution Lemma)

1. *If $\mathcal{G}(\mathcal{T}, \Delta \cup \mathcal{V})$ and ψ is a well-kinded and closed substitution w.r.t Δ , s.t. $|\psi| = \mathcal{V}$, then $\mathcal{G}(\psi(\mathcal{T}), \Delta)$.*
2. *If $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta \cup \alpha, \rho)$, s.t. $\alpha \cap (\mathcal{V} \cup \mathcal{W} \cup \Delta \cup \text{Fv}(\rho)) = \emptyset$, $\mathcal{V} \cap \Delta = \emptyset$, and ψ is a well-kinded and closed substitution w.r.t Δ , st $|\psi| = \alpha$, then $\mathcal{S}(\psi(\mathcal{T}), \mathcal{V}, \mathcal{W}, \Delta, \rho)$.*
3. *If $\mathcal{S}(\mathcal{T}, \mathcal{V}, \mathcal{W}, \Delta, \{f_1(\mathcal{W}_1), \dots, f_n(\mathcal{W}_n)\} \cup \rho)$ s.t. $\mathcal{W}_i \subseteq \Delta$, $f_i \notin \mathcal{V} \cup \mathcal{W} \cup \Delta \cup \text{Fv}(\rho)$, $\Delta \cap \mathcal{V} = \emptyset$ and $\psi \equiv \{f_i / \lambda \mathcal{W}_i. t_i\}$, is a well-kinded and closed substitution w.r.t Δ , then $\mathcal{S}(\psi(\mathcal{T}), \mathcal{V}, \mathcal{W}, \Delta, \rho)$.*

Proof: By induction on the structure of \mathcal{T} . \square

Theorem 5.1 (Well-Formedness Theorem)

Let \mathcal{L} be a proof of $\Delta, \Gamma \vdash m : T$, where $\mathcal{G}(\Gamma, \Delta)$. If every occurrence of the \Downarrow relation in \mathcal{L} involves a closed and well-kinded substitution, i.e. for every $(\Delta_0 \triangleright T_1, T_2, \mathcal{V}_0 \Downarrow \psi_0)$ in \mathcal{L} , ψ_0 is closed and well-kinded wrt Δ_0 , then $\mathcal{G}(\mathcal{T}, \Delta)$

Proof: The proof is by induction on the height of the proof tree. A more general version of the proof is incorporated into the proof of Theorem 6.3. \square

$$\mathcal{G} : Mod \rightarrow \mathcal{P}(Var) \rightarrow Bool$$

$$\mathcal{S} : Mod \rightarrow \mathcal{P}(Var) \rightarrow VarList \rightarrow \mathcal{P}(Var) \rightarrow \mathcal{P}(Type_{atom}) \rightarrow Bool$$

- $\mathcal{G}((TE, ME), \Delta)$ iff $\mathcal{G}_T(TE, \Delta)$ and $\mathcal{G}_L(ME, \Delta)$,
 where $\mathcal{G}_T(TypId \xrightarrow{fn} T, \Delta)$ iff T , of kind o , is closed w.r.t Δ and kind-correct,
 $\mathcal{G}_L(\{x_1 \mapsto T_1, \dots, x_n \mapsto T_n\}, \Delta)$ iff $\forall i. \mathcal{G}(T_i, \Delta)$.
- $\mathcal{G}(\forall \mathcal{V}(\mathcal{T}_1 \Rightarrow \mathcal{T}_2), \Delta)$ iff
 $\mathcal{S}(\mathcal{T}_1, \mathcal{V}, [], \Delta, \emptyset)$ and $\mathcal{G}(\mathcal{T}_2, \Delta \cup \mathcal{V})$
- $\mathcal{S}((TE, ME), \mathcal{V}, \mathcal{W}, \Delta, \rho)$ iff
 $\mathcal{S}_T(TE, \mathcal{V}, \mathcal{W}, \Delta, \rho)$ and $\mathcal{S}_L(ME, \mathcal{V} - Fv(TE), \mathcal{W}, \Delta, \rho \cup \{f(\mathcal{W}) \mid f \in \mathcal{V} \text{ and } f(\mathcal{W}) \in ASExp(TE)\})$
 where $\mathcal{S}_T(TypId \xrightarrow{fn} T_0, \mathcal{V}, \mathcal{W}, \Delta, \rho)$ iff
 $T_0 ::= T$ where T of kind o is closed w.r.t Δ and kind-correct
 $\quad \mid f(\mathcal{W})$ where $f \in \mathcal{V}$ and $f(\mathcal{W})$ is kind-correct of kind o
 $\quad \mid t$ where $t \in \rho$
 $\quad \mid T_0 \rightarrow T_0$
 and $\mathcal{S}_L((x \mapsto T); ME', \mathcal{V}, \mathcal{W}, \Delta, \rho)$ iff
 $\mathcal{S}(T, \mathcal{V}, \mathcal{W}, \Delta, \rho)$ and
 $\mathcal{S}_L(ME', \mathcal{V} - Fv(T), \mathcal{W}, \Delta, \rho \cup \{f(\mathcal{W}) \mid f \in \mathcal{V} \text{ and } f(\mathcal{W}) \in Aat(T)\})$
- $\mathcal{S}(\forall \mathcal{V}_1(\mathcal{T}_1 \Rightarrow \mathcal{T}_2), \mathcal{V}, \mathcal{W}, \Delta, \rho)$ iff
 $\mathcal{S}(\mathcal{T}_1, \mathcal{V}_1, [], \Delta, \rho)$ and $\mathcal{S}(\mathcal{T}_2, \mathcal{V}, \mathcal{W} @ \mathcal{V}_1, \Delta \cup \mathcal{V}_1, \rho)$

Figure 3: Well-Formedness Predicate \mathcal{G}

6 Uniqueness of Elaboration

The \Downarrow relation is presented as a specification. Hence if we can prove that there is a unique computable substitution satisfying the \Downarrow relation, then proof of the deterministic nature of the elaboration is complete. This is because the other rules for elaboration of a term in the language are syntax-directed, hence completely deterministic.

Lemma 6.1 *Let $(\Delta \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$. If \mathcal{V} is partitioned into two disjoint sets \mathcal{V}_1 and \mathcal{V}_2 , st $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\psi_1 = \psi|_{\mathcal{V}_1}$ and $\psi_2 = \psi|_{\mathcal{V}_2}$ then $(\Delta \triangleright \mathcal{T}_1, \psi_1(\mathcal{T}_2), \mathcal{V}_2 \Downarrow \psi_2)$*

Proof: By definition, if $(\Delta \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$ then Δ and \mathcal{V} are disjoint, and ψ is closed wrt Δ . Hence $\psi(\mathcal{T}_2) = \psi_2(\psi_1(\mathcal{T}_2))$, because the co-domain of the substitution ψ_1 is disjoint from the domain of ψ_2 . Thus $\Delta \triangleright \mathcal{T}_1 \prec \psi_2(\psi_1(\mathcal{T}_2))$. \square

The strategy for proving the uniqueness of the substitution ψ in the \Downarrow relation is to do so incrementally. Let $(\Delta \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$. Let \mathcal{V} be partitioned into two disjoint sets \mathcal{V}_1 and \mathcal{V}_2 st $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\psi_1 = \psi|_{\mathcal{V}_1}$ and $\psi_2 = \psi|_{\mathcal{V}_2}$. We first prove that $\psi|_{\mathcal{V}_1}$ is unique and therefore must necessarily equal ψ_1 . But by the above lemma, we also have $(\Delta \triangleright \mathcal{T}_1, \psi_1(\mathcal{T}_2), \mathcal{V}_2 \Downarrow \psi_2)$. We next prove that ψ_2 is unique. Hence we have proved that ψ is unique.

6.1 A modified proof system for subsumption

Because of functorial polymorphism, it is seen that the relations \Downarrow and \prec are mutually recursive. This recursion is

unfolded to specify a single relation \Downarrow , broken down structurally into \Downarrow_T and \Downarrow_L . In the \Downarrow relation, as defined as Figure 4, the substitution ψ is applied only in the leaves of the proof tree: looking at the proof backwards, we may say that application of the substitution is delayed to the leaves. The relations \Downarrow and \prec were specified wrt a single environment Δ . For the specification of \Downarrow , Δ is partitioned into disjoint sets Δ_1, Δ_2 . From rule (9) in Figure 4, it is seen that only variables from Δ_2 can be introduced as bound variables in the semantic signature. Again, looking at the proof backwards, since the application of the substitution ψ is delayed, while we enter scope of bound variables, we need to ensure that the co-domain of ψ is disjoint from the bound variables: to avoid capture by binding constructs. Thus such a partition of Δ , into Δ_1 and Δ_2 , the set of free and bound variables, is essential.

As mentioned in Section 4.1, the \Downarrow relation is specified by a set of conditions and a proof system for the \prec relation to be used after the application of the substitution ψ . In the specification of the \Downarrow relation the substitution is held back. Hence these side conditions must be true for every tuple $(\Delta_1, \Delta_2 \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$ in a valid proof.

Definition: [Valid Proofs] A proof is considered **valid** if for every tuple $(\Delta_1, \Delta_2 \triangleright \mathcal{T}_1, \mathcal{T}_2, \mathcal{V} \Downarrow \psi)$ in the proof, the following holds,

- $|\psi| = \mathcal{V}$ and ψ is closed wrt Δ_1 .
- $(\Delta_1 \cup \Delta_2) \cap \mathcal{V} = \emptyset$.
- \mathcal{T}_1 is closed wrt $(\Delta_1 \cup \Delta_2)$ and \mathcal{T}_2 is closed wrt $(\Delta_1 \cup \Delta_2 \cup \mathcal{V})$.

$$\frac{\Delta_1, \Delta_2 \triangleright \text{TE}_1, \text{TE}_2, \mathcal{V}_1 \downarrow_T \psi_1 \quad \Delta_1, \Delta_2 \triangleright \text{ME}_1, \psi_1(\text{ME}_2), \mathcal{V}_2 \downarrow_L \psi_2}{\Delta_1, \Delta_2 \triangleright (\text{TE}_1, \text{ME}_1), (\text{TE}_2, \text{ME}_2), \mathcal{V}_1 \cup \mathcal{V}_2 \downarrow \psi_1 \sqcup \psi_2} \quad (1)$$

$$\Delta_1, \Delta_2 \triangleright \{\}, \{\}, \emptyset \downarrow_L \emptyset \quad (2)$$

$$\frac{\Delta_1, \Delta_2 \triangleright \text{ME}_1, \text{ME}_2, \mathcal{V} \downarrow_L \psi}{\Delta_1, \Delta_2 \triangleright \{(x \mapsto \text{T}_1), \text{ME}_1\}, \text{ME}_2, \mathcal{V} \downarrow_L \psi} \quad (3)$$

$$\frac{\Delta_1, \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V}_1 \downarrow_L \psi_1 \quad \Delta_1, \Delta_2 \triangleright \text{ME}_1, \psi_1(\text{ME}_2), \mathcal{V}_2 \downarrow_L \psi_2}{\Delta_1, \Delta_2 \triangleright \{(x \mapsto \text{T}_1), \text{ME}_1\}, \{(x \mapsto \text{T}_2), \text{ME}_2\}, \mathcal{V}_1 \cup \mathcal{V}_2 \downarrow_L \psi_1 \sqcup \psi_2} \quad (4)$$

$$\Delta_1, \Delta_2 \triangleright \{\}, \{\}, \emptyset \downarrow_T \emptyset \quad (5)$$

$$\frac{\text{T}_1 = \psi(\text{T}_2)}{\Delta_1, \Delta_2 \triangleright \{(t \mapsto \text{T}_1)\}, \{(t \mapsto \text{T}_2)\}, \mathcal{V} \downarrow_T \psi} \quad (6)$$

$$\frac{\Delta_1, \Delta_2 \triangleright \text{TE}_1, \text{TE}_2, \mathcal{V} \downarrow_T \psi}{\Delta_1, \Delta_2 \triangleright \{(t \mapsto \text{T}), \text{TE}_1\}, \text{TE}_2, \mathcal{V} \downarrow_T \psi} \quad (7)$$

$$\frac{\Delta_1, \Delta_2 \triangleright \{(t \mapsto \text{T}_1)\}, \{(t \mapsto \text{T}_2)\} \downarrow_T \psi_1 \quad \Delta_1, \Delta_2 \triangleright \text{TE}_1, \psi_1(\text{TE}_2), \mathcal{V}_2 \downarrow_T \psi_2}{\Delta_1, \Delta_2 \triangleright \{(t \mapsto \text{T}_1), \text{TE}_1\}, \{(t \mapsto \text{T}_2), \text{TE}_2\}, \mathcal{V}_1 \cup \mathcal{V}_2 \downarrow_T \psi_1 \sqcup \psi_2} \quad (8)$$

$$\frac{\Delta_1 \cup \Delta_2 \cup \mathcal{V}_2, \emptyset \triangleright \text{T}'_2, \text{T}'_1, \mathcal{V}_1 \downarrow \psi' \quad \Delta_1, \Delta_2 \cup \mathcal{V}_2 \triangleright \psi'(\text{T}''_1), \text{T}''_2, \mathcal{V} \downarrow \psi}{\Delta_1, \Delta_2 \triangleright \forall \mathcal{V}_1 (\text{T}'_1 \Rightarrow \text{T}''_1), \forall \mathcal{V}_2 (\text{T}'_2 \Rightarrow \text{T}''_2), \mathcal{V} \downarrow \psi} \quad (9)$$

Figure 4: A modified proof system for subsumption

- $Fv(\text{T}_2) - (\Delta_1 \cup \Delta_2) = \mathcal{V}$.

Similar side conditions must hold for, $(\Delta_1, \Delta_2 \triangleright \text{ME}_1, \text{ME}_2, \mathcal{V} \downarrow_L \psi)$ and $(\Delta_1, \Delta_2 \triangleright \text{TE}_1, \text{TE}_2, \mathcal{V} \downarrow_T \psi)$.

It should be noted that the definition of valid proofs is symmetric wrt Δ_1, Δ_2 except for the fact that the substitution ψ must be closed wrt Δ_1 .

These side conditions ensure that the partition of the \mathcal{V} component, in a valid proof, into \mathcal{V}_1 and \mathcal{V}_2 is unique in Rule (1), Rule (4) and Rule (8).

Lemma 6.2 $(\Delta_0 \cup \Delta_1, \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$, where ψ is closed wrt Δ_1 , has a valid proof iff $(\Delta_1, \Delta_0 \cup \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$ has a valid proof.

Proof: The proof is by induction on the height of the proof tree. \square

Theorem 6.1 If $\mathcal{G}(\text{T}_1, \Delta)$, $\mathcal{S}(\text{T}_2, \mathcal{V}, [\cdot], \Delta, \emptyset)$ where $(\Delta \cap \mathcal{V}) = \emptyset$ then $(\Delta \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$ iff there exists a valid proof of $(\Delta, \emptyset \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$.

Proof: The proof is by induction on the height of the proof tree and an application of Lemma 6.1 and Lemma 6.2. \square

6.2 A proof of unique elaboration

Given $\Delta_1, \Delta_2, \{(t \mapsto \text{T}_1)\}, \{(t \mapsto \text{T}_2)\}$ and \mathcal{V} , if we can compute a substitution ψ such that $\text{T}_1 = \psi(\text{T}_2)$ and the tuple $(\Delta_1, \Delta_2, \{(t \mapsto \text{T}_1)\}, \{(t \mapsto \text{T}_2)\}, \mathcal{V}, \psi)$ is valid, then the proof system in Figure 4, is essentially an algorithm for computing the substitution associated with the \downarrow relation.

By Theorem 6.1, it is also an algorithm for computing ψ associated with the \downarrow relation.

Theorem 6.2 (Unique Substitution Theorem)

If $\mathcal{G}(\text{T}_1, \Delta_1 \cup \Delta_2)$, $\mathcal{S}(\text{T}_2, \mathcal{V}, \Delta_2, \Delta_1 \cup \Delta_2, \emptyset)$ where $(\Delta_1 \cup \Delta_2) \cap \mathcal{V} = \emptyset$, and $(\exists \psi \Delta_1, \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$ then $(\exists! \psi [\Delta_1, \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi])$, where ψ is well-kinded).

Proof: The proof is an elaborate induction on the height of the derivation of $(\Delta_1, \Delta_2 \triangleright \text{T}_1, \text{T}_2, \mathcal{V} \downarrow \psi)$.

We incrementally prove the substitution ψ to be unique and well-kinded. Let $\Delta = \Delta_1 \cup \Delta_2$. We only discuss Rule (6), as this rule defines the substitution, and Rule (4), as this shows how assumptions propagate:

- (6) By assumption, $\mathcal{G}_T(\{(t \mapsto \text{T}_1)\}, \Delta)$ and $\mathcal{S}_T(\{(t \mapsto \text{T}_2)\}, \mathcal{V}, \Delta_2, \Delta, \emptyset)$.

Hence T_1 is closed wrt Δ . By the definition of \mathcal{S}_T , T_2 is generated by the grammar,

$$\begin{array}{l} T_0 ::= T \quad \text{where } T \text{ of kind } o, \text{ is closed} \\ \quad \quad \quad \text{w.r.t } \Delta \text{ and kind-correct} \\ \quad \quad \quad \text{where } f \in \mathcal{V} \text{ and } f(\Delta_2) \\ \quad \quad \quad \text{is kind-correct, of kind } o \\ \quad \quad \quad | f(\Delta_2) \\ \quad \quad \quad | T_0 \rightarrow T_0 \end{array}$$

A variable f in the domain of the substitution ψ , occurs only in atomic type expressions $f(\Delta_2) \in \text{ASExp}(\text{T}_2)$. The premise of this proof is, $\text{T}_1 = \psi(\text{T}_2)$. The equality is a *structural equality* on a single in-fix binary constructor, \rightarrow , i.e. the parse trees of T_1 and $\psi(\text{T}_2)$ are identical. Hence, if T is the type subexpression in T_1 that corresponds to $f(\Delta_2)$, at the leaf of the parse tree of T_2 , then $\text{T} = ((\psi f) \Delta_2)$.

Let $\Delta_2 \equiv [y_1, \dots, y_n]$. The normal form of $\psi(f)$ must be $(\lambda x_1 \dots x_n. T')$. Because of the validity condition, $\psi(f)$ is closed wrt Δ_1 . Hence T' must be closed wrt $\Delta_1 \cup \{x_1, \dots, x_n\}$. As Δ_2 is a list of distinct variables, $((\psi f) \Delta_2)$ is essentially a renaming substitution, $\{x_1/y_1, \dots, x_n/y_n\}$, on T' . Thus $\{x_1/y_1, \dots, x_n/y_n\}(T') = T$. Since T' does not contain variables from Δ_2 , $\psi(f)$ must be $(\lambda \Delta_2. T)$. This is a kind-correct substitution.

(4) By assumption, $\mathcal{G}_L(\{(x \mapsto T_1), ME_1\}, \Delta)$ and $\mathcal{S}_L(\{(x \mapsto T_2), ME_2\}, \mathcal{V}, \Delta_2, \Delta, \emptyset)$.

By the conditions on a valid proof, $\mathcal{V}_1 = Fv(\mathcal{T}_2) - \Delta$ and $\mathcal{V}_2 = \mathcal{V} - \mathcal{V}_1$. Let $\psi_1 = \psi|_{\mathcal{V}_1}$ and $\psi_2 = \psi|_{\mathcal{V}_2}$. By induction, ψ_1 is unique and well-kinded.

As $\mathcal{S}_L(ME_2, \mathcal{V}_2, \Delta_2, \Delta, \{f(\Delta_2)|f \in \mathcal{V}_2 \text{ and } f(\Delta_2) \in \text{Aat}(\mathcal{T}_2)\})$, by the Substitution Lemma, $\mathcal{S}_L(\psi_1(ME_2), \mathcal{V}_2, \Delta_2, \Delta, \emptyset)$. Since $\mathcal{G}_L(ME_1, \Delta)$, by induction, ψ_2 is a well-kinded substitution that is unique.

□

Theorem 6.3 (Unique Elaboration Theorem)

If $(\Gamma \vdash m : T)$, where $\exists \Delta. \mathcal{G}(\Gamma, \Delta)$, then \mathcal{T} is unique and $\mathcal{G}(\mathcal{T}, \Delta)$.

Proof: The entire proof of $(\Gamma \vdash m : T)$ is syntax-directed, hence completely deterministic. Hence to prove the uniqueness of \mathcal{T} , it is only necessary to show that whenever the \Downarrow relation is used, the assumptions associated with the Unique Substitution Theorem hold. It is for this reason we prove the clause $\mathcal{G}(\mathcal{T}, \Delta)$. The following non-trivial cases are considered here:

(3) We need to prove $\mathcal{G}(\forall \mathcal{V}(\mathcal{T}_1 \Rightarrow \mathcal{T}_2), \Delta)$ i.e. $\mathcal{S}(\mathcal{T}_1, \mathcal{V}, [], \Delta, \emptyset)$ and $\mathcal{G}(\mathcal{T}_2, \Delta \cup \mathcal{V})$.

By Lemma 5.2, $\mathcal{S}(\mathcal{T}_1, \mathcal{V}, [], \Delta, \emptyset)$. Hence, by Lemma 5.1, $\mathcal{G}(\mathcal{T}_1, \Delta \cup \mathcal{V})$. The induction hypothesis is now applicable, hence, \mathcal{T}_2 is unique and $\mathcal{G}(\mathcal{T}_2, \Delta \cup \mathcal{V})$.

(4) By induction, $\mathcal{G}(\forall \mathcal{V}(\mathcal{T}_1 \Rightarrow \mathcal{T}_2), \Delta)$ and $\mathcal{G}(\mathcal{T}, \Delta)$.

Thus, by definition, $\mathcal{S}(\mathcal{T}_1, \mathcal{V}, [], \Delta, \emptyset)$. We can now apply the Unique Substitution Theorem to obtain the unique ψ st, $(\Delta, \emptyset \triangleright \mathcal{T}, \mathcal{T}_1, \mathcal{V} \downarrow \psi)$. Since ψ is closed and well-kinded wrt Δ , by the Substitution Lemma, $\mathcal{G}(\psi(\mathcal{T}_2), \Delta)$.

(5) By induction, $\mathcal{G}(\Delta, \mathcal{T})$. Since $(\mathcal{T}_1, \mathcal{V}) = \mathcal{E}[\Sigma]_{\Gamma[]}$, by Lemma 5.2, $\mathcal{S}(\mathcal{T}_1, \mathcal{V}, [], \Delta, \emptyset)$. We can now apply the Unique Substitution Theorem to obtain the unique ψ st, $(\Delta, \emptyset \triangleright \mathcal{T}, \mathcal{T}_1, \mathcal{V} \downarrow \psi)$. By the Substitution Lemma, $\mathcal{G}(\psi(\mathcal{T}_1), \Delta)$.

□

7 Conclusion

There are two important directions for future research associated with this approach to providing semantics to higher-order functors. The first one being, by considering *gensym* as a primitive function and introducing environments, can we capture some form of generativity in the language? The second concerns parameterized types. SML allows declarations of parameterized types. The question to be answered here is, can we incorporate parameterized types into our system and still retain the principal substitution property?

The author conjectures this to be true. But in this extended system the order in which fields of a structure are matched will be significant.

The presentation in this paper provides a simple static semantics for higher-order functors, with transparent signatures, for languages which do not possess features like generativity. It is shown that in the absence of generativity, the semantics of higher-order functors comes very close to the semantics of first-order functors in SML.

Acknowledgements

The author is grateful for technical and editorial help from Carl Gunter.

References

- [1] L. Cardelli and X. Leroy. Abstract types and the dot notation. Research Report 56, DEC Systems, Palo Alto, March 1990.
- [2] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In H.-J. Boehm, editor, *Conference Record of the Twentyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1994.
- [3] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15:211–252, 1993.
- [4] G. Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [5] X. Leroy. Manifest types, modules, and separate compilation. In H.-J. Boehm, editor, *Conference Record of the Twentyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1994.
- [6] D. B. MacQueen. Using dependent types to express modular structure. In C. N. Fischer, editor, *Symposium on Principles of Programming Languages*, pages 277–286. ACM, 1986.
- [7] D. B. MacQueen and M. Tofte. A semantics for higher-order functors. In *European Symposium on Programming. Lecture Notes in Computer Science vol. ??*, Springer, 1994.
- [8] D. Miller. A logic programming language. *Journal of Logic and Computation*, 1:497–536, 1991.
- [9] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1991.
- [10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [11] M. Tofte. Principal signatures for higher-order program modules. In A. W. Appel, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 189–199. ACM, 1992.