A Type-Theoretic Approach to Higher-Order Modules with Sharing*

Robert Harper† Mark Lillibridge‡
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

The design of a module system for constructing and maintaining large programs is a difficult task that raises a number of theoretical and practical issues. A fundamental issue is the management of the flow of information between program units at compile time via the notion of an interface. Experience has shown that fully opaque interfaces are awkward to use in practice since too much information is hidden, and that fully transparent interfaces lead to excessive interdependencies, creating problems for maintenance and separate compilation. The "sharing" specifications of Standard ML address this issue by allowing the programmer to specify equational relationships between types in separate modules, but are not expressive enough to allow the programmer complete control over the propagation of type information between modules.

These problems are addressed from a type-theoretic viewpoint by considering a calculus based on Girard's system F_ω. The calculus differs from those considered in previous studies by relying exclusively on a new form of weak sum type to propagate information at compile-time, in contrast to approaches based on strong sums which rely on substitution. The new form of sum type allows for the specification of equational, as well as type and kind, information in interfaces. This provides complete control over the propagation of compile-time information between program units and is sufficient to encode in a straightforward way most uses of type sharing specifications in Standard ML. Modules are treated as "first-class" citizens, and therefore the system supports higher-order modules and some object-oriented programming idioms; the language may be easily restricted to "second-class" modules found in ML-like languages.

*This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168.

†Electronic mail address: `rwh@cs.cmu.edu`.

‡Electronic mail address: `mdl@cs.cmu.edu`.

1 Introduction

Modularity is an essential technique for developing and maintaining large software systems [46, 24, 36]. Most modern programming languages provide some form of module system that supports the construction of large systems from a collection of separately-defined program units [7, 8, 26, 32]. A fundamental problem is the management of the tension between the need to treat the components of a large system in relative isolation (for both conceptual and pragmatic reasons) and the need to combine these components into a coherent whole. In typical cases this problem is addressed by equipping each module with a well-defined interface that mediates all access to the module and requiring that interfaces be enforced at system link time.

The Standard ML (SML) module system [17, 32] is a particularly interesting design that has proved to be useful in the development of large software systems [2, 1, 3, 11, 13]. The main constituents of the SML module system are *signatures*, *structures*, and *functors*, with the latter two sometimes called *modules*. A structure is a program unit defining a collection of types, exceptions, values, and structures (known as *substructures* of the structure). A functor may be thought of as a "parameterized structure", a first-order function mapping structures to structures. A signature is an interface describing the constituents of a structure — the types, values, exceptions, and structures that it defines, along with their kinds, types, and interfaces. See Figure 1 for an illustrative example of the use of the SML module system; a number of sources are available for further examples and information [15, 39].

A crucial feature of the SML module system is the notion of *type sharing*[1] which allows for the specification

[1] The closely-related notion of *structure sharing* is not considered in this paper.

```
signature SYMBOL = sig
  type symbol
  val intern : string -> symbol
  val pname : symbol -> string
  val eq : symbol * symbol -> bool
end

structure Symbol : SYMBOL = struct
  structure HashTable : HASH_TABLE = ...
  type symbol = HashTable.hash_key
  fun intern id =
      HashTable.enter (HashTable.hash id) id
  fun pname sym = HashTable.retrieve sym
  fun eq (s1, s2) = HashTable.same_key s1 s2
end
```

Figure 1: Example of the SML Module System

of *coherence conditions* among a collection of structures that ensure that types defined in separate modules coincide. The classic example (adapted from MacQueen) is the construction of a parser from a lexer and a symbol table, each of which make use of a common notion of symbol (see Figure 2). The parser is constructed by a functor that takes as arguments two modules, a lexer and a symbol table manager. The parser composes functions from the lexer and symbol table manager; the composition is well-typed only if the two modules "share" a common notion of symbol. Within the body of the functor `Parser` the types `L.S.symbol` and `T.S.symbol` coincide, as specified by the type sharing specification in the parameter signature. See MacQueen's seminal paper for further examples and discussion [26, 17].

1.1 Transparency and Opacity

Module bindings in SML are "transparent" in the sense that the type components of a module are fully visible in the scope of the binding. For example, the structure declaration

```
structure S = struct
  type t = int
  type u = t -> t
  val f = fn x:t => x
end
```

introduces a structure variable `S` with type components `S.t` and `S.u` and value component `S.f`. Within the scope of `S`, the type `S.t` is equivalent to the type `int` and the type `S.u` is equivalent to the type `int->int`. These equivalences are not affected by the ascription of a signature to the binding. For example, the signature `SIG` defined by the declaration

```
signature LEXER = sig
  structure S : SYMBOL
  type token
  ⋮
end

signature SYMTAB = sig
  structure S : SYMBOL
  ⋮
end

signature PARSER = sig
  structure L : LEXER
  val parse : string -> Lexer.token stream
  ⋮
end

functor Parser
(structure L:LEXER and T:SYMTAB
 sharing type L.S.symbol=T.S.symbol):PARSER=
struct
  ⋮
end
```

Figure 2: Sharing Specifications

```
signature SIG = sig
  type t
  type u
  val f : u
end
```

may be correctly ascribed to the structure `S` without obscuring the bindings of `S.t` and `S.u`.

Functor bindings are similarly "transparent" in that the type components of the result of any application of the functor are fully visible within the scope of the functor binding. For example, consider the following functor declaration:

```
functor F(structure X:SIG):SIG = struct
  type t = X.t * X.t
  type u = X.u
  val f = X.f
end
```

The bindings of the `t` and `u` components of any application of `F` are fully visible as a function of the `t` and `u` components of the parameter `X`. For example, within the scope of the declaration

```
structure T:SIG = F(S)
```

the type `T.t` is equivalent to the type `int*int` and the type `T.u` is equivalent to the type `int->int`.

It is possible only to a very limited extent in SML to specify in a signature the bindings of the types in a module. For example, we may augment the signature SIG with a sharing specification to specify that t is int as follows:

```
signature SIG' = sig
  type t
  sharing type t=int
  type u
  val f : u
end
```

This method cannot be extended to specify the binding of u — sharing specifications may only involve type names, not general type expressions. To fully determine the type bindings in S requires a *transparent signature*:

```
signature FULL_SIG_S = sig
  type t = int
  type u = int -> int
  val f : int -> int
end
```

Note that this is not a legal SML signature because of the type equations. There is no way to express equations such as `u = int -> int` in SML signatures. The signature FULL_SIG_S is the "full" signature of S since it completely determines the bindings of its type components.

The importance of transparent signatures only becomes apparent when we consider functors and the closely-related abstraction bindings suggested by MacQueen [26, 17]. Functor parameters are *opaque* in the sense that the ascribed signature is the sole source of type information for those parameters (this property is the basis for the reduction of abstraction bindings to functor applications). Fine control over the "degree" of opacity of a functor parameter can be achieved by admitting transparent, or, more generally, *translucent*, signatures that allow for the partial (possibly full) determination of the type components of a module. For example, the translucent signature

```
signature PARTIAL_SIG_S_1 = sig
  type t
  type u = t -> t
  val f : u
end
```

is a partially transparent signature that leaves the type t unconstrained, but determines u up to the choice of t. The structure S matches the signature PARTIAL_SIG_S_1 since S.u is equal to S.t->S.t which is itself equal to int->int. Conversely, the translucent signature

```
signature PARTIAL_SIG_S_2 = sig
  type t = int
  type u
  val f : u
end
```

determines t, leaving u unconstrained. This signature is essentially equivalent to the signature SIG' above.[2] S matches this signature.

Translucent signatures are particularly useful in connection with higher-order functors [43]. Using equations in signatures it is possible to specify the dependency of the functor result on the functor argument. For example, a natural signature for the functor F defined above is the following functor signature which fully determines the type components of F:

```
funsig FULL_SIG_F =
(structure X:SIG):
 sig type t=X.t*X.t type u=X.u val f:u end
```

Another, less precise, signature for F is the functor signature

```
funsig PARTIAL_SIG_F =
(structure X:SIG):
 sig type t type u=X.u val f:u end
```

which only specifies the behavior of F on the component u, leaving its behavior on t unspecified.

Higher-order functors [43] are particularly important in connection with separate compilation. A separately-compiled module may be represented by a variable whose signature is the "full" transparent signature of the module itself [42]. By abstracting the client module with respect to these variables we obtain a (possibly higher-order) functor whose application models the process of linking the clients of the module with its actual implementation. The signature matching process ensures that the presumed signature of the separately-compiled module is consistent with the module itself so as to guarantee type safety. The full signature of the separately compiled module is necessary to ensure that separate and combined compilation yield the same result.

1.2 Static Semantics

The static semantics of SML [32] is defined by a collection of complex *elaboration rules* that specify the static well-formedness conditions for SML programs. The main techniques employed in the semantics are the use of "unique names" (or "generativity") to handle abstraction and sharing specifications, and the use of non-deterministic rules to handle polymorphism, sharing specifications, and signature matching. The static semantics has proved useful as a guide to implementation [28, 2, 41, 40], but is remarkably difficult to modify

[2] It seems plausible that most uses of type sharing specifications may be accounted for in this way, provided that we neglect local specifications and re-binding of variables in specifications, both of which are of questionable utility.

or extend (see, for example, [43]). The naïve attempt to enrich signatures as sketched above is incompatible with the crucial "principal signature" property [31]. But it is not clear whether this failure is a symptom of an intrinsic incoherence in the language, or is merely an artifact of the semantic method.[3]

In an effort to gain some insight into the complexities of the static semantics several authors have undertaken a type-theoretic analysis of SML, especially its module system [27, 33, 35, 7, 20, 19]. Previous studies of the module system focused on the transparency of SML-style structures through the use of "strong sum" or "dependent product" types. These are types of the form $\Sigma x{:}A.B(x)$ whose elements are pairs $\langle M_1, M_2\rangle$ that are accessed via projections $\pi_1(M)$ and $\pi_2(M)$. The crucial properties of strong sums [29] are that if $M : \Sigma x{:}A.B(x)$, then $\pi_2(M) : B(\pi_1(M))$, and that $\pi_1(\langle M_1, M_2\rangle) = M_1$. Together, these properties ensure that type information is propagated in rough accord with the SML static semantics. (See [27, 33, 20, 19] for further discussion.) Substitution-based methods are problematic in the presence of computational effects, unless care is taken to account for the phase distinction [20]. Moreover, strong sums fail to account for sharing specifications and the abstract nature ("generativity") of `structure` and `datatype` bindings.

In this paper we extend the type-theoretic analysis of SML-like module systems by presenting a calculus with the following features:

- Translucent sum types, which generalize weak sums by providing labeled fields and equations governing type constructors. These mechanisms obviate the need for substitution, and account for abstraction and common uses of SML-style sharing specifications.

- A notion of subsumption that encompasses a "coercive" pre-order associated with record fields and a "forgetful" pre-order associated with equations that represent sharing information.

- Treatment of modules as "first-class" values. The typing rules ensure that visibility of compile-time components is suitably restricted when run-time selection is used (see also [34]). If run-time selection is not used, modules behave exactly as they would in a more familiar "second-class" module system such as is found in SML.

Our calculus improves on previous work by providing a much greater degree of control over the propagation of type information at compile time so that we can achieve the effect of SML-like sharing specifications and provide direct support for abstraction.

[3] Based on the approach taken here, and a related idea due to Leroy, Tofte has recently devised a way to accommodate a form of type abbreviation in Standard MLs ignatures [44].

Kinds	K	$::=$	$\Omega \mid K{\Rightarrow}K'$
Constructors	A	$::=$	$\alpha \mid \Pi x{:}A.A' \mid \{D_1, \ldots, D_n\} \mid \lambda\alpha{::}K.A \mid A\,A' \mid V.\mathbf{b}$
Declarations	D	$::=$	$\mathbf{b} \triangleright \alpha{::}K \mid \mathbf{b} \triangleright \alpha{::}K{=}A \mid \mathbf{y} \triangleright x{:}A$
Terms	M	$::=$	$x \mid \lambda x{:}A.M \mid M\,M' \mid M{:}A \mid \{B_1, \ldots, B_n\} \mid M.\mathbf{y}$
Bindings	B	$::=$	$\mathbf{b} \triangleright \alpha{=}A \mid \mathbf{y} \triangleright x{=}M$
Values	V	$::=$	$x \mid \lambda x{:}A.M \mid \{B_{\mathrm{V}1}, \ldots, B_{\mathrm{V}n}\} \mid V.\mathbf{y}$
	B_{V}	$::=$	$\mathbf{b} \triangleright \alpha{=}A \mid \mathbf{y} \triangleright x{=}V$
Contexts	Γ	$::=$	$\bullet \mid \Gamma, \alpha{::}K \mid \Gamma, \alpha{::}K{=}A \mid \Gamma, x{:}A$

Figure 3: Syntax rules ($n \geq 0$)

2 Overview of the Calculus

Our system is based on Girard's F_ω [14] in much the same way that many systems are based on the second-order lambda calculus (F_2). That is to say, our system can be (roughly) thought of as being obtained from F_ω by adding more powerful constructs (translucent sums and dependent functions) and a notion of subtyping and then removing the old constructs (quantification[4] ($\forall$), weak sums ($\exists$), and non-dependent functions ($\rightarrow$)) superseded by the new ones. Subtyping interacts with the rest of the calculus via implicit subsumption. Bounded quantification is not supported.

Like F_ω, our system is divided into three levels: terms, (type) constructors, and kinds. Kinds classify constructors, and a subset of constructors, called types, having kind Ω classify terms. The kind level is necessary because the constructor level contains functions on constructors. Example: the constructor $\lambda\alpha{::}\Omega.\alpha{\rightarrow}\alpha$ has kind $\Omega{\Rightarrow}\Omega$ and when applied to type Int yields Int$\rightarrow$Int.

The syntax rules for our system are given in Figure 3. The meta variable α ranges over *constructor variables* and the meta variable x ranges over *term variables.* The meta variable $\mathbf{b}$ ranges over *constructor field names* and the meta variable $\mathbf{y}$ ranges over *term field names.* We

[4] Quantification is derivable from dependent functions and translucent sums. The basic idea is to transform $\Lambda\alpha{::}K.\ M$ into $\lambda x{:}\{\mathbf{b} \triangleright \alpha{::}K\}.\ [x.\mathbf{b}/\alpha]M$ where x is not free in M and $M\ [A]$ into $M\ \{\mathbf{b} \triangleright \alpha{=}A\}$. Note that this implements constructor abstraction as a delaying operation unlike the normal SML semantics. See [16] for a discussion of the differences between these two interpretations of constructor abstractions and why this choice seems to be preferable.

have placed field names in bold in order to help emphasize that they are names, not variables. The complete typing rules appear in appendix A.

Our handling of dependent functions ($\lambda x{:}A.M$) is standard [29] except that our elimination rule only allows for the application of functions having arrow-types[5] (non-dependent function types). The normal elimination rule for dependent functions does not have this restriction, requiring functions only to have a Π-type. We make this restriction because we intend to extend this system in future work with effect-producing primitives. In the presence of effects, the unrestricted rule is unsound because of interactions with the first-class nature of translucent sums.[6]

Translucent sums ($\{B_1, \ldots, B_n\}$), the central contribution of the calculus, will be discussed at length in the next section. Very briefly, they are n-ary labeled dependent sums whose types can optionally contain information about the contents of their constructor fields. Traditional records and weak sums (existentials) are degenerate forms of translucent sums.

A mechanism (written $M{:}A$) for forcing a term M to be coerced to a supertype A is provided. The subtyping relation allows for both generalized record subtyping [7, 6, 9] (fields which are not depended on by the other fields in a translucent sum may be dropped) and for the forgetting of information about the contents of constructor fields.

There are two basic forms of constructor definitions. A constructor definition is *opaque* if within the scope of the definition there is no information available about the identity of the constructor variable being defined. By contrast, a *transparent* constructor definition makes available the identity of the constructor variable that is being defined.

Contexts in our system can contain both opaque ($\alpha{::}K$) and transparent ($\alpha{::}K{=}A$) definitions. The effect of transparency is implemented by a typing rule (ABBREV') for the constructor equality relation judgement that establishes that $\alpha = A$ when $\alpha{::}K{=}A$ is in the context. (This is similar to mechanisms used in AUTOMATH and LEGO [12, 45, 25].)

Our calculus is intended to be interpreted using a call-by-value semantics. The typing system is not sound for call-by-name in the presence of effects.[7] We restrict terms in constructors to values in order to avoid the problem of trying to give a meaning to a constructor containing a side-effecting term.[8] In our system, values (V) are considered to be term variables, term λ-expressions, translucent sums containing only values and constructors, and selections of term fields from values. We allow $V.\mathbf{y}$ to be a value in order to allow paths like $x.\mathbf{y}.\mathbf{y}'$ to be values while still keeping the set of values closed under the substitution of a value for a term variable.

3 Translucent Sums

A translucent sum has the form of a possibly empty sequence of *bindings* written between curly braces ($\{B_1, \ldots, B_n\}$). The corresponding translucent sum type is similar except that *declarations* are used instead of bindings ($\{D_1, \ldots, D_n\}$).

Translucent sums differ from traditional records in a number of ways. In addition to normal term fields, they can contain constructor fields. The type or content of later fields in a translucent sum can depend on the content of earlier fields. As an example, consider the following translucent sum, call it P, that packages up a type with a value of that type:[9]

$$\{\mathbf{b} \triangleright \alpha{=}\mathrm{Int},\ \mathbf{y} \triangleright x{=}3\} \ : \ \{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}\alpha\}$$

Binding r to P would give $r.\mathbf{b} :: \Omega$ and $r.\mathbf{y} : r.\mathbf{b}$.[10]

The scope of variables bound in bindings and declarations is all the following bindings/declarations in that translucent sum (type). For example, the scope of x in the following translucent sum includes M' and A' but not M or A:

$$\{\mathbf{b} \triangleright \alpha{=}A,\ \mathbf{y} \triangleright x{=}M,\ \mathbf{y}' \triangleright x'{=}M',\ \mathbf{b}' \triangleright \alpha'{=}A'\}$$

Scoping for the other constructs is as normal. We regard terms, constructors, *etc.*, that differ only by α-conversion of variables as equivalent.

Note that field names cannot be α-converted. Changing a field name in a translucent sum term/type results in a different term/type because the set of legal field names which can be selected changes. Failure to distinguish between field names which cannot be α-converted and the internal names for fields which must be able to α-convert in order to permit substitution to work, leads to problems.[11] For example, the equivalent of the

[5] The arrow-type $A \to A'$ can be regarded as an abbreviation for $\Pi x{:}A.A'$ where x is not free in A'.

[6] Abbreviated example: $(((\lambda x.\, x.\mathbf{y})M).2)\ (((\lambda x.\, x.\mathbf{y})M).1)$ where M is the example in section 4.6.

[7] To see this consider the outermost β-redex of $(\lambda x.(x.\mathbf{y}.2)\ (x.\mathbf{y}.1))\ M$ where M is the example in section 4.6.

[8] It is not clear that allowing general terms in constructors would be that useful anyway since the substitution of general terms for term variables would be prohibited in a call-by-value setting in any case.

[9] We suggest pronouncing "$\triangleright$" as "as", ":" as "has type", and "::" as "has kind".

[10] Note the distinction between r and P here: r is a term variable (and hence a value) while P is a term meta-variable denoting a non-value. This distinction is important because the typing rules treat values specially.

[11] To avoid verbosity, a real programming language based on our system would probably provide that by default only one name need be given per field, to be used as both the field name and the internal name.

following cannot be written straightforwardly in SML:[12]

$$\begin{array}{ll} \{\mathbf{b} \triangleright \alpha{=}\mathrm{Int}, & \mathbf{y} \triangleright x{=}\{\mathbf{b} \triangleright \alpha'{=}\mathrm{Bool},\ \mathbf{y}' \triangleright x'{=}\lambda x{:}\alpha'.\ 3\}\} \ : \\ \{\mathbf{b} \triangleright \alpha{::}\Omega, & \mathbf{y} \triangleright x{:}\{\mathbf{b} \triangleright \alpha'{::}\Omega,\ \mathbf{y}' \triangleright x'{:}\alpha' {\to} \alpha\}\} \end{array}$$

Because SML does not distinguish the two kinds of names, it is problematic to express that the type of the $\mathbf{y}'$ field depends on both the outer and inner $\mathbf{b}$ fields.

The field names of any given translucent sum (type) are required to be distinct. Translucent sum types that differ only in that their declarations have been reordered without violating any dependences are considered equivalent. For example, the following first two types are equivalent but both are different from the third type:

$$\begin{array}{lll} \{\mathbf{b} \triangleright \alpha{::}\Omega, & \mathbf{b}_1 \triangleright \alpha_1{::}\Omega{=}\alpha, & \mathbf{b}_2 \triangleright \alpha_2{::}\Omega{=}\alpha\} \\ \{\mathbf{b} \triangleright \alpha{::}\Omega, & \mathbf{b}_2 \triangleright \alpha_2{::}\Omega{=}\alpha, & \mathbf{b}_1 \triangleright \alpha_1{::}\Omega{=}\alpha\} \\ \{\mathbf{b}_1 \triangleright \alpha_1{::}\Omega{=}\alpha, & \mathbf{b} \triangleright \alpha{::}\Omega, & \mathbf{b}_2 \triangleright \alpha_2{::}\Omega{=}\alpha\} \end{array}$$

It may help to think of translucent sum types as being directed acyclic graphs (DAGs) where the nodes are declarations and the edges are dependencies by one declaration on a variable declared in another declaration.

It is possible to include information in a translucent sum type on the contents of the constructor fields of its instances. This ability can be used to give a more expressive type to P:

$$\{\mathbf{b} \triangleright \alpha{=}\mathrm{Int},\ \mathbf{y} \triangleright x{=}3\} \ : \ \{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int},\ \mathbf{y} \triangleright x{:}\alpha\}$$

If it can be shown that r has this type, then it can be inferred that $r.\mathbf{b} = \mathrm{Int}$. This can not be inferred if it can only be shown that r has the less expressive type. The use of nested translucent sums and constructor field component information can give rise to more complex dependencies as the following example illustrates:

$$\begin{array}{llll} \{\mathbf{y} \triangleright x{=}\{\mathbf{b} \triangleright \alpha{=}\mathrm{Int}\}, & \mathbf{y}' \triangleright x'{=}3, & \mathbf{b} \triangleright \alpha{=}x.\mathbf{b}\} & : \\ \{\mathbf{y} \triangleright x{:}\{\mathbf{b} \triangleright \alpha{::}\Omega\}, & \mathbf{y}' \triangleright x'{:}x.\mathbf{b}, & \mathbf{b} \triangleright \alpha{::}\Omega{=}x.\mathbf{b}\} & \end{array}$$

3.1 Introduction and elimination rules

The introduction rule for translucent sums is as follows:

$$\frac{\begin{array}{c}\vdash \Gamma \text{ valid} \\ \forall i \in [1..n].\ \Gamma, \overline{D_1}, \ldots, \overline{D_{i-1}} \vdash B_i : D_i\end{array}}{\Gamma \vdash \{B_1, \ldots, B_n\} : \{D_1, \ldots, D_n\}} \quad \text{(TSUM)}$$

(The overline function ($\overline{D}$) merely strips off the field name.) Note that each of the bindings is being type checked under a context which takes account of the effect of all of the previous bindings. Constructor bindings result in transparent definitions, both when type checking later bindings and in the resulting type. Thus, the introduction rule gives P the more expressive type. The less-expressive type is obtained by the use of the subsumption rule afterwards.

There are two elimination forms with corresponding rules for translucent sums, one for constructor fields and one for term fields:

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K\}}{\Gamma \vdash V.\mathbf{b} :: K} \quad \text{(C-EXT-O)}$$

$$\frac{\Gamma \vdash M : \{\mathbf{y} \triangleright x{:}A\}}{\Gamma \vdash M.\mathbf{y} : A} \quad \text{(EXT-V)}$$

In order to apply these rules to translucent sums with multiple fields, it is first necessary to use the subsumption rule to drop the fields that are not being selected. The constructor field case may also require that type information about the field to be selected be dropped. Unlike traditional records, with translucent sums it is not always possible to drop all the other fields because the field we wish to select may depend in an essential way on them. Thus, the fact that M has a translucent sum type with a $\mathbf{y}$ field is not in itself sufficient to ensure that $M.\mathbf{y}$ is well-typed.[13] It is always possible to drop fields from the type of V because of the VALUE rules which we will discuss in section 3.3.

3.2 Translucency

When $x{:}A$ appears in the context where A is a translucent sum type which contains information about the contents of the constructor fields of its instances, it gives rise to equations via the following rule:

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K{=}A\}}{\Gamma \vdash V.\mathbf{b} = A :: K} \quad \text{(ABBREV)}$$

Thus it is possible to infer that $r.\mathbf{b} = \mathrm{Int}$ when it can be shown that r has the type $(\{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int},\ \mathbf{y} \triangleright x{:}\alpha\})$ but not when it can only be shown to have the type $(\{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}\alpha\})$.

This rule also gives rise to equations such as $\{\mathbf{b} \triangleright \alpha{=}\mathrm{Int},\ \mathbf{b}' \triangleright \alpha'{=}\mathrm{Bool}\}.\mathbf{b}' = \mathrm{Bool}$. These equations allow any valid constructor $V.\mathbf{b}$ to be reduced to a constructor which contains only values of the form $x.\mathbf{y}_1 \cdots \mathbf{y}_n$ $(n \geq 0)$. Because of this, it is not necessary in our system to consider the equality of arbitrary values (and hence terms) at type-checking time.

When the equality rules compare the parts of a constructor that are in the scope of a variable binding, they do so with the declaration associated with that variable in the context. For example, the equality rule for

[12] It is possible to write this in SML by using a combination of `local` specifications and type sharing in the signature. Thanks to Mads Tofte for pointing this out. Unfortunately, however, some SML implementations (*e.g.*, SML/NJ) do not implement `local` specifications in signatures properly so this is not very helpful in practice.

[13] For example, $(\{\mathbf{b} \triangleright \alpha{=}\mathrm{Int},\ \mathbf{y} \triangleright x{=}3\} \ : \ \{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}\alpha\}).\mathbf{y}$ is not well-typed.

Π-types below compares A_1' and A_2' with $x{:}A_1$ in the context:

$$\frac{\Gamma \vdash A_1 = A_2 :: \Omega \qquad \Gamma, x{:}A_1 \vdash A_1' = A_2' :: \Omega}{\Gamma \vdash \Pi x{:}A_1.\, A_1' = \Pi x{:}A_2.\, A_2' :: \Omega} \quad \text{(E-DFUN)}$$

This allows use of the ABBREV rule to obtain equations such as the following:

$$\begin{array}{rcl} \Pi x{:}\{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int}\}.\, x.\mathbf{b} & = & \Pi x{:}\{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int}\}.\, \mathrm{Int} \\ \{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int},\, \mathbf{y} \triangleright x{:}\alpha\} & = & \{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int},\, \mathbf{y} \triangleright x{:}\mathrm{Int}\} \end{array}$$

A similar effect occurs while typing terms. For example, in the following, we know that $x.\mathbf{b} = \mathrm{Int}$ while type checking M:

$$\{\mathbf{y} \triangleright x{:}\{\mathbf{b} \triangleright \alpha{::}\Omega{=}\mathrm{Int}\},\, \mathbf{y}' \triangleright x'{:}M\}$$

Because of the ability to substitute away transparently bound names using the equality rules, no dependency on a transparently bound name is ever truly essential. This allows many more field selections and function applications to type check than would otherwise be the case.

When translucent sums are given fully opaque types, they act like weak sums which can be used to create abstract data types (ADTs) [35]. Because we have dependent functions and a form of dependent pairs (a pair of terms where the type of the second term depends on the first component of the pair), our elimination form for weak sums is more powerful than usual [35, 7, 10]. Consider the following example in SML-like notation, where `weak` is used to construct a weak sum:

```
let structure S = struct
  structure Stack = weak
    type T = (int ref) list
    val makeStack:()->T = ...
    val push:(int,T)->() = ...
    ...
  end
  val myStack = Stack.makeStack()
end in
  S.Stack.push(1,S.myStack)
end
```

This example is well-typed in our system because we can determine that `S.myStack` has type `S.Stack.T` which is the argument type of the function `S.Stack.push`. Note that there is no way to type this example using the `open` elimination form for weak sums because there is no scope containing both the initialization of `myStack` and its use that is also inside the scope of `Stack`.

3.3 The VALUE rules

Suppose the typing context contains the following declaration:

$$r{:}\{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}\alpha\}$$

What types can we give to the expression r under this context? Because we have a name, r, for the translucent sum expression we are trying to type, we have a name for the contents of its $\mathbf{b}$ field, namely $r.\mathbf{b}$. This suggests that we can give r the type $\{\mathbf{b} \triangleright \alpha{::}\Omega{=}r.\mathbf{b},\, \mathbf{y} \triangleright x{:}r.\mathbf{b}\}$ which is a subtype of the context type for r.

This technique of giving a more expressive type to translucent sums when we have a name for their constructor components can be generalized to work on arbitrary translucent sum values. The name in this case is simply $V.\mathbf{b}$ where V is the value in question. Attempting to extend the technique to general terms requires dropping the restriction that only values may appear in constructors and results in unsoundness in the presence of effects.[14] The following two typing rules implement this technique:

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K, D_1, \ldots, D_n\}}{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K{=}V.\mathbf{b}, D_1, \ldots, D_n\}} \quad \text{(VALUE-O)}$$

$$\frac{\Gamma \vdash V.\mathbf{y} : A' \qquad \Gamma \vdash V : \{\mathbf{y} \triangleright x{:}A, D_1, \ldots, D_n\}}{\Gamma \vdash V : \{\mathbf{y} \triangleright x{:}A', D_1, \ldots, D_n\}} \quad \text{(VALUE-V)}$$

(The VALUE-V rule is used in cases of nested translucent sums to apply the technique recursively.)

By alternately applying the VALUE rules to convert an opaque binding into a transparent one and the subsumption rule to propagate that definition (and hence removing any dependencies on that binding), we can give any translucent sum value a fully transparent type (there are no constructor components for which information is lacking) with no dependencies between the fields (or subfields). Because of this, field selection on values as well as applications of functions to values do not run into problems due to the inability to remove dependencies. Without this kind of usage of the VALUE rules, expressions such as $r.\mathbf{y}$ would not type check.

The more expressive type given by the VALUE rules to translucent sum values is also critical to the propagation of typing information. For example, if s is bound to the result of the expression r, it will be given the more expressive type, allowing the fact that $s.\mathbf{b} = r.\mathbf{b}$ to be inferred.

[14] This would allow field selection to always succeed because it would permit all dependencies to be removed. Unsoundness example: $(M.\mathbf{y}.2)\,(M.\mathbf{y}.1)$ where M is the example in section 4.6.

4 Selected Examples

4.1 Simple structures

Typical SML structures can be translated straightforwardly into our system, with the only complication being the treatment of polymorphism (as discussed in [16].) Consider the following structure `S` considered in the introduction:

```
structure S = struct
  type t = int
  type u = t -> t
  val f = fn x:t => x
end
```

This translates as:

$$\mathtt{S} = \{\mathtt{t} \triangleright \mathtt{t}{=}\mathrm{Int},\ \mathtt{u} \triangleright \mathtt{u}{=}\mathtt{t}{\to}\mathtt{t},\ \mathtt{f} \triangleright \mathtt{f}{=}\lambda \mathtt{x}{:}\mathtt{t}.\,\mathtt{x}\}$$

The translations of the signatures considered earlier (only `SIG` here is actually a valid SML signature) are:

$$\begin{array}{rcl} \mathtt{FULL\_SIG\_S} & = & \{\mathtt{t} \triangleright \mathtt{t}{::}\Omega{=}\mathrm{Int},\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathrm{Int}{\to}\mathrm{Int}, \\ & & \ \mathtt{f} \triangleright \mathtt{f}{:}\mathrm{Int}{\to}\mathrm{Int}\} \\ \mathtt{PARTIAL\_SIG\_S\_1} & = & \{\mathtt{t} \triangleright \mathtt{t}{::}\Omega,\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathtt{t}{\to}\mathtt{t},\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\} \\ \mathtt{PARTIAL\_SIG\_S\_2} & = & \{\mathtt{t} \triangleright \mathtt{t}{::}\Omega{=}\mathrm{Int},\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega,\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\} \\ \mathtt{SIG} & = & \{\mathtt{t} \triangleright \mathtt{t}{::}\Omega,\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega,\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\} \end{array}$$

The subtyping rules for our system establish that $\mathtt{FULL\_SIG\_S} \le \mathtt{PARTIAL\_SIG\_S\_1} \le \mathtt{SIG}$ and $\mathtt{FULL\_SIG\_S} \le \mathtt{PARTIAL\_SIG\_S\_2} \le \mathtt{SIG}$. The signatures `PARTIAL_SIG_S_1` and `PARTIAL_SIG_S_2` are incomparable.

The signature given to `S` determines which equations on `S.t` and `S.u` can be deduced. By default our system, like SML, will give `S` its full signature, `FULL_SIG_S`. This means that we will be able to deduce that $\mathtt{S.t} = \mathrm{Int}$ and $\mathtt{S.u} = \mathrm{Int}{\to}\mathrm{Int}$. If we insert a coercion to one of the other signatures before the binding to `S`, fewer equations will be deducable in our system. In SML, by contrast, user-specified coercions never result in the loss of typing information. They can, however, result in the loss of fields. Thus, in order to translate a coercion from SML into our system, we need to enrich the target signature with all the available typing information.

4.2 Abstraction

SML/NJ [2] supports an extension to SML, called *abstraction*, which is an alternative to the normal structure binding mechanism. If the keyword `abstraction` is used instead of the keyword `structure` when binding a structure, all information about the constructor components of that structure is forgotten. Had `S` in the previous example been bound with an `abstraction` binding instead of the `structure` one we used, it would have been as if we had given `S` in our system the signature `SIG`. That is to say, `S.t` and `S.u` would have been bound opaquely. Note that it is not possible in SML/NJ to give `S` a partial signature using this mechanism. Only the fully transparent (via `structure`) and fully opaque (via `abstraction`) alternatives are available.

Abstraction bindings can be translated into our system by inserting a forced coercion just before the binding to the appropriate opaque type. For example, consider the following implementation of an abstract data type (ADT):

```
abstraction Stack = struct
  type T = (int ref) list
  val push:(T,int)->() = ...
  val pop:T->int = ...
  val isEmpty:T->bool = ...
end
```

This translates to:

$$\begin{array}{lll} \mathtt{Stack} = & (\{ & \mathtt{T} \triangleright \mathtt{T}{=}\mathtt{list}(\mathtt{ref}\,\mathrm{Int}), \\ & & \mathtt{push} \triangleright \mathtt{push}{=}(\ldots){:}(\mathtt{T},\mathrm{Int}){\to}(), \\ & & \mathtt{pop} \triangleright \mathtt{pop}{=}(\ldots){:}\mathtt{T}{\to}\mathrm{Int}, \\ & & \mathtt{isEmpty} \triangleright \mathtt{isEmpty}{=}(\ldots){:}\mathtt{T}{\to}\mathrm{Bool}\,\} \\ &):\{ & \mathtt{T} \triangleright \mathtt{T}{::}\Omega, \\ & & \mathtt{push} \triangleright \mathtt{push}{:}(\mathtt{T},\mathrm{Int}){\to}(), \\ & & \mathtt{pop} \triangleright \mathtt{pop}{:}\mathtt{T}{\to}\mathrm{Int}, \\ & & \mathtt{isEmpty} \triangleright \mathtt{isEmpty}{:}\mathtt{T}{\to}\mathrm{Bool}\,\} \end{array}$$

Note that because the type information about the identity of the `T` field is lost in the coercion, the rest of the program will be unable to break the abstraction. SML provides an abstraction mechanism, *abstype*, at the core language level. Because translucent sums are first-class in our system, we can achieve the effect of SML's `abstype` using the abstraction binding mechanism.

4.3 Sub-structures

Sub-structures are also easily translated. For example, suppose we wanted to use the Stack structure in a bigger structure as follows:

```
structure Big = struct
  structure ourStack = Stack
  type T = ourStack.T
  ...
end
```

This translates into:

$$\mathtt{Big} = \{\mathtt{ourStack} \triangleright \mathtt{ourStack}{=}\mathtt{Stack},\ \mathtt{T} \triangleright \mathtt{T}{=}\mathtt{ourStack.T},\ \ldots\}$$

`Big` will be given the following full signature:

$$\begin{array}{ll} \{\mathtt{ourStack} \triangleright \mathtt{ourStack}{:}\{ & \mathtt{T} \triangleright \mathtt{T}{::}\Omega{=}\mathtt{Stack.T}, \\ & \mathtt{push} \triangleright \mathtt{push}{:}(\mathtt{T},\mathrm{Int}){\to}(), \\ & \mathtt{pop} \triangleright \mathtt{pop}{:}\mathtt{T}{\to}\mathrm{Int}, \\ & \mathtt{isEmpty} \triangleright \mathtt{isEmpty}{:}\mathtt{T}{\to}\mathrm{Bool}\,\}, \\ \ \mathtt{T} \triangleright \mathtt{T}{::}\Omega{=}\mathtt{ourStack.T},\ \ldots\} & \end{array}$$

Note that we have that Big.ourStack.T = Big.T = Stack.T.

4.4 Functors

Functors translate into dependent functions in the expected way. Consider the following example from the introduction:

```
functor F(structure X:SIG):SIG = struct
  type t = X.t * X.t
  type u = X.u
  val f = X.f
end
```

This translates into:

$$\begin{aligned}\mathtt{F} = \lambda\mathtt{X}{:}\mathtt{SIG}.\,(\,&\{\mathtt{t} \triangleright \mathtt{t}{=}\mathtt{X.t} * \mathtt{X.t},\ \mathtt{u} \triangleright \mathtt{u}{=}\mathtt{X.u},\ \mathtt{f} \triangleright \mathtt{f}{=}\mathtt{X.f}\} : \\ &\{\mathtt{t} \triangleright \mathtt{t}{::}\Omega{=}\mathtt{X.t} * \mathtt{X.t},\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathtt{X.u},\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\})\end{aligned}$$

(The coercion on the result type of the functor is an abbreviation for a coercion on the functor body.) Note the enriched signature we have to give instead of SIG in order to make the coercion have the same effect as it does in SML. Translating the functor signatures we considered for F gives:

$$\begin{aligned}\mathtt{FULL\_SIG\_F} &= \Pi\mathtt{X}{:}\mathtt{SIG}.\,\{\mathtt{t} \triangleright \mathtt{t}{::}\Omega{=}\mathtt{X.t} * \mathtt{X.t},\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathtt{X.u},\\ &\qquad \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\}\\ \mathtt{PARTIAL\_SIG\_F} &= \Pi\mathtt{X}{:}\mathtt{SIG}.\,\{\mathtt{t} \triangleright \mathtt{t}{::}\Omega,\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathtt{X.u},\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\}\\ \mathtt{SIG\_F} &= \Pi\mathtt{X}{:}\mathtt{SIG}.\,\mathtt{SIG}\end{aligned}$$

Here, $\mathtt{FULL\_SIG\_F} \leq \mathtt{PARTIAL\_SIG\_F} \leq \mathtt{SIG\_F}$.

Suppose T was bound to the result of applying F to S. Before the APP rule can be applied to determine T's type, the subsumption rule must be used to coerce F's type (FULL_SIG_F) to an arrow type. One way this could be done is shown in Figure 4. First, F's argument type is coerced to a subtype (remember that $\mathtt{FULL\_SIG\_S} \leq \mathtt{SIG}$) using the fact that subtyping of function types is contra-variant. Next, the equality rules are used to remove the dependencies on X by the result type, resulting in an arrow type. The result is that T gets assigned the following type:

$$\{\mathtt{t} \triangleright \mathtt{t}{::}\Omega{=}\mathrm{Int} * \mathrm{Int},\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathrm{Int}{\rightarrow}\mathrm{Int},\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\}$$

If F had instead had the type PARTIAL_SIG_F, T would have been assigned the type:

$$\{\mathtt{t} \triangleright \mathtt{t}{::}\Omega,\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathrm{Int}{\rightarrow}\mathrm{Int},\ \mathtt{f} \triangleright \mathtt{f}{:}\mathtt{u}\}$$

4.5 Sharing specifications

The basic idea in translating sharing specifications is that for each set of names that are asserted to be equal, pick one with maximal scope as representative of the equivalence class and set the others equal to it using transparent definitions. For example, the following SML signature:

```
signature H = sig
  type t
  type u
  type v
    sharing type t = u
        and type v = u
end
```

translates into:

$$\mathtt{H} = \{\mathtt{t} \triangleright \mathtt{t}{::}\Omega,\ \mathtt{u} \triangleright \mathtt{u}{::}\Omega{=}\mathtt{t},\ \mathtt{v} \triangleright \mathtt{v}{::}\Omega{=}\mathtt{t}\}$$

A more interesting case is provided by the argument signature of the Parser functor in MacQueen's example from the introduction:

```
sig structure L:LEXER
    structure T:SYMTAB
  sharing type L.S.symbol=T.S.symbol
end
```

This translates into:

$$\{\mathtt{L} \triangleright \mathtt{L}{:}\mathtt{LEXER},\ \mathtt{T} \triangleright \mathtt{T}{:}\{\mathtt{S} \triangleright \mathtt{S}{:}\{\mathtt{symbol} \triangleright \mathtt{symbol}{::}\Omega{=}\mathtt{L.S.symbol},\ \ldots\},\ \ldots\}\}$$

The omitted parts are the usual translation of the rest of SYMTAB and SYMBOL. This translation method also works on sharing between constructors in the argument and result of a functor.

4.6 First-class modules

So long as we restrict ourselves to simple module operations like binding, functor application of a named functor to a named or fully transparent module, and selection from a named module, we never lose any typing information. In fact, the only module operation available in SML that causes a loss of information when used in our system is coercing a module to a user-specified type. This is not surprising, however, since the purpose of coercions is controlled information loss.

Due to the fact that modules are first-class in our system, it is possible to write module expressions which force the loss of typing information in order to preserve soundness. For example, consider the following:[15]

$$\begin{aligned}&\mathtt{if\ flip()\ then}\ \{\mathbf{b} \triangleright \alpha{=}\mathrm{Int},\ \ \mathbf{y} \triangleright x{=}(3,\quad \mathrm{succ})\}\\ &\qquad\quad \mathtt{else}\ \{\mathbf{b} \triangleright \alpha{=}\mathrm{Bool},\ \mathbf{y} \triangleright x{=}(\mathtt{true},\mathtt{not})\}\end{aligned}$$

While both parts of the if can be given fully transparent types, these types are not equal. In order to make the if type check, we must give them equal types. The only way to do this is to use the subsumption rule to coerce both of their types to $\{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}(\alpha, \alpha{\rightarrow}\alpha)\}$.

[15] For the unsoundness examples, flip is a function which alternates returning true and false. It is easily implemented using a global variable.

$$
\begin{array}{l}
\quad \texttt{FULL\_SIG\_F} \\
= \Pi\texttt{X:SIG}.\,\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\texttt{X.t} * \texttt{X.t},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\texttt{X.u},\ \texttt{f} \triangleright \texttt{f:u}\} \\
\leq \Pi\texttt{X:FULL\_SIG\_S}.\,\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\texttt{X.t} * \texttt{X.t},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\texttt{X.u},\ \texttt{f} \triangleright \texttt{f:u}\} \\
= \Pi\texttt{X}:\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\mathrm{Int},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\mathrm{Int}{\to}\mathrm{Int},\ \texttt{f} \triangleright \texttt{f}:\mathrm{Int}{\to}\mathrm{Int}\}.\,\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\texttt{X.t} * \texttt{X.t},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\texttt{X.u},\ \texttt{f} \triangleright \texttt{f:u}\} \\
= \Pi\texttt{X}:\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\mathrm{Int},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\mathrm{Int}{\to}\mathrm{Int},\ \texttt{f} \triangleright \texttt{f}:\mathrm{Int}{\to}\mathrm{Int}\}.\,\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\mathrm{Int} * \mathrm{Int},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\mathrm{Int}{\to}\mathrm{Int},\ \texttt{f} \triangleright \texttt{f:u}\} \\
= \texttt{FULL\_SIG\_S}{\to}\{\texttt{t} \triangleright \texttt{t}::\Omega{=}\mathrm{Int} * \mathrm{Int},\ \texttt{u} \triangleright \texttt{u}::\Omega{=}\mathrm{Int}{\to}\mathrm{Int},\ \texttt{f} \triangleright \texttt{f:u}\}
\end{array}
$$

Figure 4: Steps in coercing `F`'s type to an arrow type

The system described in [34] displays similar behavior, namely a forced loss of typing information when using modules in conditionals and other primitives. In that system, types are divided into two universes, U_1, the universe of "normal" types like Int and Bool→Int, and U_2, the universe of module types. The loss in this system is caused by the need to apply an implicit coercion from a strong sum (which belongs to U_2) to a weak sum (which belongs to U_1) because primitives operate only on terms with types in U_1. This coercion causes a total loss of typing information. Our system is more flexible than this because it only loses just enough information to ensure soundness.

The possible uses for first-class modules have not been well explored. One known use discussed in [35] is to select at runtime between two or more ADTs which implement the same abstraction using different algorithms based on expected usage conditions. For example, we could use one particular hash table implementation for small tables and another for large ones.

5 Related Work

An early influential attempt to give a comprehensive type-theoretic analysis of modularity and abstraction was undertaken by Burstall and Lampson with the experimental language Pebble [5]. Their work stresses the role of dependent types and the mechanisms required to support abstraction, but does not address the problem of controlling the "degree" of abstraction. In particular, Pebble supports type and value bindings as primitive notions, but with an "opaque" typing discipline, in contrast to our calculus.

Cardelli's language Quest [7] has exerted a strong influence on the present work. Our approach shares with Quest the emphasis on type-theoretic methods, and is similarly based on Girard's F_ω enriched with a notion of subsumption (though we depart from Cardelli's approach by omitting bounded quantification). Quest does not provide an adequate treatment of modularity; our work can be seen as providing the type-theoretic basis for an extension of Quest with an expressive module system.

Mitchell, *et al.* [34] consider an extension of the SML module system with first-class modules as a means of supporting certain object-oriented programming idioms. Their paper is primarily concerned with illustrating an interesting language design rather than with the type-theoretic underpinnings of such a language, though a brief sketch is provided. A comparison with their work is given in Section 4.6.

The type-theoretic analysis of the SML modules system was initiated by MacQueen [27], and further developed by Harper and Mitchell [33, 20, 19]. This work is summarized and compared with the present work in the introduction.

Our language bears some relationship to Russell [4] and Poly [30], but a detailed comparison seems difficult in the absence of a type-theoretic analysis of these languages (see [21] for an early attempt).

In an effort to address the problem of separate compilation, Leroy has independently developed a variant of the SML modules system based on the notion of a "manifest type" which is similar in spirit to our translucent sum types. See Leroy's paper in this volume [23] for a description of his system and some comments on its relationship to ours.

6 Conclusions

The main contribution of this work is the design of a calculus of modularity with the following features:

- Fine control over the "degree" of abstraction through the notion of a translucent sum type.
- A treatment of modules as first-class entities without sacrificing the control over type abstraction afforded by a second-class module system.
- Support for separate compilation in a form that ensures the complete equivalence between separate and integrated compilation of a large system.

The following are some important directions for future research:

- Establish the soundness of the type system by proving preservation of typing under a call-by-value operational semantics.
- Investigate the efficiency of type checking and develop practical algorithms that may be used in an implementation. We show in Appendix B that the subtyping problem for our system, and hence the type checking problem, is undecidable. There is reason to believe, however, that this will not be a problem in practice.
- Design an elaborator to translate an SML-like syntax into the calculus, including a systematic treatment of the reduction of symmetric `sharing` specifications to asymmetric definitions in signatures.
- Develop an treatment of structure sharing that accounts for structure generativity and interacts well with computational effects.

Acknowledgements

We are grateful to Andrew Appel, Luca Cardelli, Olivier Danvy, John Greiner, Nick Haines, Mark Leone, Xavier Leroy, Brian Milnes, John Mitchell, and Mads Tofte for their comments and suggestions.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[3] Edoardo Biagioni, Nicholas Haines, Robert Harper, Peter Lee, Brian G. Milnes, and Eliot B. Moss. ML signatures for a protocol stack. Fox Memorandum CMU–CS–93–170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[4] Hans-Jürgen Böhm, Alan Demers, and James Donahue. An informal description of Russell. Technical Report 80–430, Computer Science Department, Cornell University, Ithaca, New York, 1980.

[5] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Kahn et al. [22], pages 1–50.

[6] Luca Cardelli. A semantics of multiple inheritance. In Kahn et al. [22], pages 51–67.

[7] Luca Cardelli. Typeful programming. Technical Report 45, DEC SRC, 1989.

[8] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, DEC Systems Research Center, Palo Alto, CA, November 1989.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 18(4), December 1986.

[10] Luca Cardelli and Leroy Xavier. Abstract types and the dot notation. Technical Report 56, DEC SRC, Palo Alto, CA, March 1990.

[11] Eric Cooper, Robert Harper, and Peter Lee. The Fox project: Advanced development of systems software. Technical Report CMU–CS–91–178, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

[12] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[13] Emden Gansner and John Reppy. eXene. In Robert Harper, editor, *Third International Workshop on Standard ML*, Pittsburgh, PA, September 1991. School of Computer Science, Carnegie Mellon University.

[14] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[15] Robert Harper. Introduction to Standard ML. Technical Report ECS–LFCS–86–14, Laboratory for the Foundations of Computer Science, Edinburgh University, September 1986.

[16] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

[17] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS–LFCS–86–2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[18] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS–LFCS–86–2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[19] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. (See also [33].).

[20] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[21] James G. Hook. Understanding russell: A first attempt. In Kahn et al. [22], pages 69–85.

[22] Gilles Kahn, David MacQueen, and Gordon Plotkin, editors. *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1984.

[23] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994.

[24] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[25] Zhaolui Luo, Robert Pollack, and Paul Taylor. How to use Lego: A preliminary user's manual. Technical Report LFCS-TN-27, Laboratory for the Foundations of Computer Science, Edinburgh University, October 1989.

[26] David MacQueen. Modules for Standard ML. In *1984 ACM Conference on LISP and Functional Programming*, pages 198–207, 1984. Revised version appears in [18].

[27] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[28] David B. MacQueen. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 212–223. ACM Press, July 1988.

[29] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[30] David C. J. Matthews. POLY report. Technical Report 28, Computer Laboratory, University of Cambridge, 1982.

[31] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[32] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[33] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[34] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[35] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[36] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[37] Benjamin Pierce. Bounded quantification is undecidable. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque*. ACM, January 1992.

[38] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1991.

[39] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Addison Wesley, 1989.

[40] Nick Rothwell. Functional compilation from the Standard ML core language to lambda calculus. Technical Report ECS–LFCS–92–235, Laboratory for the Foundations of Computer Science, Edinburgh University, Edinburgh, Scotland, September 1992.

[41] Nick Rothwell. Miscellaneous design issues in the ML kit. Technical Report ECS–LFCS–92–237, Laboratory for the Foundations of Computer Science, Edinburgh University, Edinburgh, Scotland, September 1992.

[42] Zhong Shao and Andrew Appel. Smartest recompilation. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 439–450, Charleston, SC, January 1993.

[43] Mads Tofte. Principal signatures for higher-order program modules. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 189–199, January 1992.

[44] Mads Tofte. Type abbreviations in signatures. Unpublished manuscript, August 1993.

[45] Diedrik T. van Daalen. *The Language Theory of AUTOMATH*. PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.

[46] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.

A The Typing Rules

Definition A.1 (Judgements)

$\vdash \Gamma$ *valid*	*valid context*
$\Gamma \vdash A :: K$	*valid constructor*
$\Gamma \vdash A = A' :: K$	*equal constructors*
$\Gamma \vdash D = D'$	*equal declarations*
$\Gamma \vdash A \leq A'$	*subtype relation*
$\Gamma \vdash D \leq D'$	*subfield relation*
$\Gamma \vdash M : A$	*well-typed term*
$\Gamma \vdash B : D$	*well-typed binding*

Definition A.2 (The field name stripping function)

$$\begin{array}{lcl}
\overline{\mathbf{b} \triangleright \alpha{::}K} & = & \alpha{::}K \\
\overline{\mathbf{b} \triangleright \alpha{::}K{=}A} & = & \alpha{::}K{=}A \\
\overline{\mathbf{y} \triangleright x{:}A} & = & x{:}A
\end{array}$$

Definition A.3 (Context Formation Rules)

$$\vdash \bullet\ \mathit{valid} \qquad \text{(INITIAL-T)}$$

$$\frac{\vdash \Gamma\ \mathit{valid} \qquad \alpha \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, \alpha{::}K\ \mathit{valid}} \qquad \text{(DEF-O)}$$

$$\frac{\vdash \Gamma, \alpha{::}K\ \mathit{valid} \qquad \Gamma \vdash A :: K}{\vdash \Gamma, \alpha{::}K{=}A\ \mathit{valid}} \qquad \text{(DEF-T)}$$

$$\frac{\Gamma \vdash A :: \Omega \qquad x \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, x{:}A\ \mathit{valid}} \qquad \text{(DEF-V)}$$

Definition A.4 (Constructor Formation Rules)

$$\frac{\vdash \Gamma\ \mathit{valid} \qquad \alpha{::}K \in \Gamma}{\Gamma \vdash \alpha :: K} \qquad \text{(C-VAR-O)}$$

$$\frac{\vdash \Gamma\ \mathit{valid} \qquad \alpha{::}K{=}A \in \Gamma}{\Gamma \vdash \alpha :: K} \qquad \text{(C-VAR-T)}$$

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K\}}{\Gamma \vdash V.\mathbf{b} :: K} \qquad \text{(C-EXT-O)}$$

$$\frac{\Gamma, x{:}A \vdash A' :: \Omega}{\Gamma \vdash \Pi x{:}A.\, A' :: \Omega} \qquad \text{(C-DFUN)}$$

$$\frac{\vdash \Gamma\ \mathit{valid}}{\Gamma \vdash \{\} :: \Omega} \qquad \text{(C-UNIT)}$$

$$\frac{\Gamma, \overline{D} \vdash \{D_1, \ldots, D_n\} :: \Omega}{\Gamma \vdash \{D, D_1, \ldots, D_n\} :: \Omega} \qquad \text{(C-TSUM)}$$

$$\frac{\Gamma, \alpha{::}K \vdash A :: K'}{\Gamma \vdash \lambda\alpha{::}K.\, A :: K \Rightarrow K'} \qquad \text{(C-LAM)}$$

$$\frac{\Gamma \vdash A_1 :: K_2 \Rightarrow K \qquad \Gamma \vdash A_2 :: K_2}{\Gamma \vdash A_1\, A_2 :: K} \qquad \text{(C-APP)}$$

Definition A.5 (Constructor Equality Rules)

$$\frac{\Gamma \vdash A :: K}{\Gamma \vdash A = A :: K} \qquad \text{(E-REFL)}$$

$$\frac{\Gamma \vdash A' = A :: K}{\Gamma \vdash A = A' :: K} \qquad \text{(E-SYM)}$$

$$\frac{\Gamma \vdash A = A' :: K \qquad \Gamma \vdash A' = A'' :: K}{\Gamma \vdash A = A'' :: K} \qquad \text{(E-TRAN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 = A_2 :: \Omega \\ \Gamma, x{:}A_1 \vdash A_1' = A_2' :: \Omega\end{array}}{\Gamma \vdash \Pi x{:}A_1.\, A_1' = \Pi x{:}A_2.\, A_2' :: \Omega} \qquad \text{(E-DFUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash D = D' \\ \Gamma, \overline{D} \vdash \{D_1, \ldots, D_n\} = \{D_1', \ldots, D_n'\} :: \Omega\end{array}}{\Gamma \vdash \{D, D_1, \ldots, D_n\} = \{D', D_1', \ldots, D_n'\} :: \Omega} \qquad \text{(E-TSUM)}$$

$$\frac{\Gamma, \alpha{::}K \vdash A = A' :: K'}{\Gamma \vdash \lambda\alpha{::}K.\, A = \lambda\alpha{::}K.\, A' :: K \Rightarrow K'} \qquad \text{(E-LAM)}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_2 = A_2' :: K \\ \Gamma \vdash A_1 = A_1' :: K \Rightarrow K'\end{array}}{\Gamma \vdash A_1\, A_2 = A_1'\, A_2' :: K'} \qquad \text{(E-APP)}$$

$$\frac{\Gamma, \alpha{::}K \vdash A :: K' \qquad \Gamma \vdash A' :: K}{\Gamma \vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K'} \qquad \text{(E-BETA)}$$

$$\frac{\Gamma, \alpha{::}K \vdash A\, \alpha :: K' \qquad \Gamma \vdash A :: K \Rightarrow K'}{\Gamma \vdash \lambda\alpha{::}K.\, A\, \alpha = A :: K \Rightarrow K'} \qquad \text{(E-ETA)}$$

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K{=}A\}}{\Gamma \vdash V.\mathbf{b} = A :: K} \qquad \text{(ABBREV)}$$

$$\frac{\vdash \Gamma\ \mathit{valid} \qquad \alpha{::}K{=}A \in \Gamma}{\Gamma \vdash \alpha = A :: K} \qquad \text{(ABBREV')}$$

Definition A.6 (Declaration Equality Rules)

$$\frac{\vdash \Gamma, \alpha{::}K\ \mathit{valid}}{\Gamma \vdash \mathbf{b} \triangleright \alpha{::}K = \mathbf{b} \triangleright \alpha{::}K} \qquad \text{(EQ-O)}$$

$$\frac{\Gamma \vdash A = A' :: K \qquad \vdash \Gamma, \alpha{::}K{=}A\ \mathit{valid}}{\Gamma \vdash \mathbf{b} \triangleright \alpha{::}K{=}A = \mathbf{b} \triangleright \alpha{::}K{=}A'} \qquad \text{(EQ-T)}$$

$$\frac{\Gamma \vdash A = A' :: \Omega \qquad \vdash \Gamma, x{:}A\ \mathit{valid}}{\Gamma \vdash \mathbf{y} \triangleright x{:}A = \mathbf{y} \triangleright x{:}A'} \qquad \text{(EQ-V)}$$

Definition A.7 (Subtyping Rules)

$$\frac{\Gamma \vdash A = A' :: \Omega}{\Gamma \vdash A \leq A'} \qquad \text{(S-EQ)}$$

$$\frac{\Gamma \vdash A \leq A' \qquad \Gamma \vdash A' \leq A''}{\Gamma \vdash A \leq A''} \qquad \text{(S-TRAN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_2 \leq A_1 \\ \Gamma, x{:}A_2 \vdash A_1' \leq A_2' \qquad \Gamma \vdash \Pi x{:}A_1.\, A_1' :: \Omega\end{array}}{\Gamma \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_2'} \qquad \text{(S-DFUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash D \leq D' \qquad \Gamma \vdash \{D', D_1', \ldots, D_m'\} :: \Omega \\ \Gamma, \overline{D} \vdash \{D_1, \ldots, D_n\} \leq \{D_1', \ldots, D_m'\}\end{array}}{\Gamma \vdash \{D, D_1, \ldots, D_n\} \leq \{D', D_1', \ldots, D_m'\}} \qquad \text{(S-TSUM)}$$

$$\frac{\Gamma \vdash \{D_1, \ldots, D_n, D\} :: \Omega}{\Gamma \vdash \{D_1, \ldots, D_n, D\} \leq \{D_1, \ldots, D_n\}} \qquad \text{(S-THIN)}$$

Definition A.8 (Subfielding Rules)

$$\frac{\Gamma \vdash D = D'}{\Gamma \vdash D \leq D'} \qquad \text{(S-SAME)}$$

$$\frac{\Gamma \vdash A \leq A' \qquad \vdash \Gamma, x{:}A\ \mathit{valid}}{\Gamma \vdash \mathbf{y} \triangleright x{:}A \leq \mathbf{y} \triangleright x{:}A'} \qquad \text{(S-VALUE)}$$

$$\frac{\vdash \Gamma, \alpha{::}K{=}A\ \mathit{valid}}{\Gamma \vdash \mathbf{b} \triangleright \alpha{::}K{=}A \leq \mathbf{b} \triangleright \alpha{::}K} \qquad \text{(S-FORGET)}$$

Definition A.9 (Term Formation Rules)

$$\frac{\vdash \Gamma\ valid \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A} \quad \text{(VAR-V)}$$

$$\frac{\Gamma, x{:}A \vdash M : A'}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, A'} \quad \text{(LAM)}$$

$$\frac{\Gamma \vdash M_1 : A_1 \to A_2 \qquad \Gamma \vdash M_2 : A_1}{\Gamma \vdash M_1\, M_2 : A_2} \quad \text{(APP)}$$

$$\frac{\begin{array}{c}\vdash \Gamma\ valid \\ \forall i \in [1..n].\ \Gamma, \overline{D_1}, \ldots, \overline{D_{i-1}} \vdash B_i : D_i\end{array}}{\Gamma \vdash \{B_1, \ldots, B_n\} : \{D_1, \ldots, D_n\}} \quad \text{(TSUM)}$$

$$\frac{\Gamma \vdash M : \{\mathbf{y} \triangleright x{:}A\}}{\Gamma \vdash M.\mathbf{y} : A} \quad \text{(EXT-V)}$$

$$\frac{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K, D_1, \ldots, D_n\}}{\Gamma \vdash V : \{\mathbf{b} \triangleright \alpha{::}K{=}V.\mathbf{b}, D_1, \ldots, D_n\}} \quad \text{(VALUE-O)}$$

$$\frac{\begin{array}{c}\Gamma \vdash V.\mathbf{y} : A' \\ \Gamma \vdash V : \{\mathbf{y} \triangleright x{:}A, D_1, \ldots, D_n\}\end{array}}{\Gamma \vdash V : \{\mathbf{y} \triangleright x{:}A', D_1, \ldots, D_n\}} \quad \text{(VALUE-V)}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M{:}A : A} \quad \text{(COERCE)}$$

$$\frac{\Gamma \vdash M : A' \qquad \Gamma \vdash A' \leq A}{\Gamma \vdash M : A} \quad \text{(SUBS)}$$

Definition A.10 (Binding Formation Rules)

$$\frac{\vdash \Gamma, \alpha{::}K{=}A\ valid}{\Gamma \vdash \mathbf{b} \triangleright \alpha{=}A : \mathbf{b} \triangleright \alpha{::}K{=}A} \quad \text{(BIND-T)}$$

$$\frac{\Gamma \vdash M : A \qquad \vdash \Gamma, x{:}A\ valid}{\Gamma \vdash \mathbf{y} \triangleright x{=}M : \mathbf{y} \triangleright x{:}A} \quad \text{(BIND-V)}$$

Lemma A.11 (Properties of the typing system)

1. *if* $\Gamma \vdash A :: K$ *then* $\vdash \Gamma$ *valid*
2. *if* $\Gamma \vdash A_1 = A_2 :: K$ *then* $\Gamma \vdash A_1 :: K$ *and* $\Gamma \vdash A_2 :: K$
3. *if* $\Gamma \vdash D_1 = D_2$ *then* $\vdash \Gamma, \overline{D_1}$ *valid and* $\vdash \Gamma, \overline{D_2}$ *valid*
4. *if* $\Gamma \vdash A_1 \leq A_2$ *then* $\Gamma \vdash A_1 :: \Omega$ *and* $\Gamma \vdash A_2 :: \Omega$
5. *if* $\Gamma \vdash D_1 \leq D_2$ *then* $\vdash \Gamma, \overline{D_1}$ *valid and* $\vdash \Gamma, \overline{D_2}$ *valid*
6. *if* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A :: \Omega$
7. *if* $\Gamma \vdash B : D$ *then* $\vdash \Gamma, \overline{D}$ *valid*
8. *if* $\vdash \Gamma, \overline{D}$ *valid then* $\vdash \Gamma$ *valid*

B Undecidability of Subtyping

The subtyping relation for our system can be shown to be undecidable by a slight modification to Benjamin Pierce's proof of the undecidability of $F_{\leq}$ subtyping [38, 37]. The basic source of undecidability is the subtyping rule (FORGET) that allows the forgetting of information about the type components of translucent sums.

Even a vastly simpler system with transparent and opaque sums and a forgetting rule is undecidable. In order to demonstrate this as well as simplify the discussion, we consider now a very simple fragment of our full system.

B.1 The fragment $\lambda^{\to,\exists,\exists=}$

The fragment $\lambda^{\to,\exists,\exists=}$ is obtained from our system by restricting the set of constructors to include only types and restricting the methods of building types to only allow for arrow types, binary opaque sums (often called weak sums), and binary transparent sums. We use a slightly different notation to emphasize that these are simpler constructs. The syntax for $\lambda^{\to,\exists,\exists=}$ is as follows:

$$\text{Types} \qquad A ::= \alpha \mid A_1 \to A_2 \mid \exists\alpha.A \mid \exists\alpha{=}A_1.A_2$$

As before, the meta-variable α ranges over type variables and we identify types that differ only by α-conversion. The translation back to our earlier notation is as follows:

$$\begin{array}{lcl} \overline{\alpha} & = & \alpha \\ \overline{A_1 \to A_2} & = & \Pi x{:}\overline{A_1}.\overline{A_2} \\ \overline{\exists\alpha.A} & = & \{\mathbf{b} \triangleright \alpha{::}\Omega,\ \mathbf{y} \triangleright x{:}\overline{A}\} \\ \overline{\exists\alpha{=}A_1.A_2} & = & \{\mathbf{b} \triangleright \alpha{::}\Omega{=}\overline{A_1},\ \mathbf{y} \triangleright x{:}\overline{A_2}\} \end{array}$$

The effect of the subtyping rules of our system on this fragment is captured by the following simple set of rules:

Definition B.1 (Subtyping rules for $\lambda^{\to,\exists,\exists=}$)

$$\alpha \leq \alpha \quad \text{(VAR)}$$

$$\frac{A_1' \leq A_1 \qquad A_2 \leq A_2'}{A_1 \to A_2 \leq A_1' \to A_2'} \quad \text{(ARROW)}$$

$$\frac{A \leq A'}{\exists\alpha.A \leq \exists\alpha.A'} \quad \text{(SUM-O)}$$

$$\frac{[A/\alpha]A_1 \leq [A/\alpha]A_2}{\exists\alpha{=}A.A_1 \leq \exists\alpha{=}A.A_2} \quad \text{(SUM-T)}$$

$$\frac{[A/\alpha]A_1 \leq [A/\alpha]A_2}{\exists\alpha{=}A.A_1 \leq \exists\alpha.A_2} \quad \text{(FORGET')}$$

$$\mathcal{F}(\rho) = \begin{cases} \alpha_i & \text{if } \rho = \alpha_i \\ \exists\alpha,\alpha_1,\ldots,\alpha_n.\neg(\exists\alpha'{=}\alpha,\alpha_1'{=}\mathcal{F}(\rho_1),\ldots,\alpha_n'{=}\mathcal{F}(\rho_n).\neg\mathcal{F}(\rho_1)) & \text{if } \rho = [\alpha_1,\ldots,\alpha_n]{<}\rho_1\ldots\rho_n{>} \\ \exists\alpha,\alpha_1,\ldots,\alpha_n.\neg\alpha & \text{if } \rho = \text{HALT} \end{cases}$$

$$\begin{array}{lcll} \mathcal{F}(R) & = & \exists\alpha{=}\sigma,\alpha_1{=}\mathcal{F}(\rho_1),\ldots,\alpha_n{=}\mathcal{F}(\rho_n).\neg\mathcal{F}(\rho_1) \leq \sigma & \text{where } R = {<}\rho_1\ldots\rho_n{>} \text{ and} \\ \sigma & = & \exists\alpha,\alpha_1,\ldots,\alpha_n.\neg(\exists\alpha'{=}\alpha,\alpha_1'{=}\alpha_1,\ldots,\alpha_n' = \alpha_n.\neg\alpha) & \end{array}$$

Here, α, α', and α_1' through α_n' are fresh variables.

Figure 5: Modifications to Pierce's encoding of row machines

Note that this set of rules is completely syntax directed and does not require the use of a context because of the explicit use of substitution. The proof that this set of rules corresponds to the subtyping rules of the original system on this fragment is omitted. For the purposes of the undecidability proof, we will only need the following lemma:

Lemma B.2 *The subtyping relation for $\lambda^{\rightarrow,\exists,\exists=}$ is reflexive.*

The proof proceeds by structural induction on the size of the type using the following measure:[16]

$$\begin{array}{lcl} |\alpha| & = & 0 \\ |A_1{\rightarrow}A_2| & = & 1 + |A_1| + |A_2| \\ |\exists\alpha.A| & = & 1 + |A| \\ |\exists\alpha{=}A_1.A_2| & = & 1 + |[A_1/\alpha]A_2| \end{array}$$

B.2 Undecidability of $\lambda^{\rightarrow,\exists,\exists=}$ subtyping

Theorem B.3 *If the FORGET' rule is removed, then $\lambda^{\rightarrow,\exists,\exists=}$ subtyping is decidable.*

Proof: Each use of the other rules strictly decreases the following non-negative measure, so the simple syntax-directed procedure always terminates in this case: $|A_1 \leq A_2| = |A_1| + |A_2|$. □

Note that use of the FORGET' rule does not decrease this measure and in fact can increase it because the type on the right side can grow without limit in the recursive call. This fact can be used to construct examples that cause the simple syntax-directed procedure for checking $\lambda^{\rightarrow,\exists,\exists=}$ subtyping to loop. For example, consider the following definitions:

$$\begin{array}{lcll} \neg A & = & A{\rightarrow}\alpha' & \\ & & & \\ P(A) & = & \exists\alpha{=}A.\neg A & \text{where } \alpha \text{ fresh} \\ G_\alpha(A) & = & \exists\alpha.\neg A & \end{array}$$

The definition of $\neg A$ is chosen so that $\neg A_1 \leq \neg A_2$ iff $A_2 \leq A_1$. Any type constructor with a contravariant subtyping rule could be used here. An example which causes cyclic behavior is then as follows:

$$\begin{array}{llcl} & P(G_\alpha(P(\alpha))) & \leq & G_\alpha(P(\alpha)) \\ = & \exists\alpha{=}G_\alpha(P(\alpha)).\neg G_\alpha(P(\alpha)) & \leq & \exists\alpha.\neg P(\alpha) \\ \Rightarrow & [G_\alpha(P(\alpha))/\alpha](\neg G_\alpha(P(\alpha))) & \leq & [G_\alpha(P(\alpha))/\alpha](\neg P(\alpha)) \\ = & \neg G_\alpha(P(\alpha)) & \leq & \neg P(G_\alpha(P(\alpha))) \\ \Rightarrow & P(G_\alpha(P(\alpha))) & \leq & G_\alpha(P(\alpha)) \\ & & \vdots & \end{array}$$

Theorem B.4 *$\lambda^{\rightarrow,\exists,\exists=}$ subtyping is undecidable.*

Pierce's proof can be found in chapter 6 of his thesis [38] or in [37]. Space considerations prevent outlining it here. The modifications necessary to change his encoding of row machines so that it produces $\lambda^{\rightarrow,\exists,\exists=}$ subtyping questions instead are found in Figure 5. The key differences are as follows:

- Use of $\exists\alpha{=}A.A_1 \leq \exists\alpha.A_2$ instead of $\forall\alpha.A_2 \leq \forall\alpha{\leq}A.A_1$.
- Use of reflexivity to halt computation instead of the FTOP rule. (Compare the two definitions of $\mathcal{F}$(HALT))

[16] Note that $|[A_1/\alpha]A_2| = |[|A_1|/\alpha]A_2|$ if we define $|i| = i$.