Mixin Modules

Dominic Duggan* Constantinos Sourelis

Abstract

Mixin modules are proposed as a new construct for module languages, allowing recursive definitions to span module boundaries. Mixin modules are proposed specifically for the Standard ML language. Several applications are described, including the resolution of cycles in module import dependency graphs, as well as functionality related to Haskell type classes and CLOS generic functions, though without any complications to the core language semantics. Mixin modules require no changes to the core ML type system, and only a very minor change to its run-time semantics. A type system and reduction semantics are provided, and the former is verified to be sound relative to the latter.

1 Introduction

The Standard ML module language has gained some interest because of its state-of-the-art module language. The module language provides a clean separation between modules and interfaces, allowing a module to export several interfaces. A particular strength of the language is its treatment of parameterized modules (functors). Standard ML provides an innovative mechanism for combining abstraction and sharing: sharing constraints in a functor building block can be used to place some graph structure on the import hierarchy over which it abstracts. The original Standard ML module system provided weak support for separate compilation. Modula-like manifest type definitions in opaque interfaces have been proposed as an alternative module design, facilitating both separate compilation [18] and first-class modules [13, 26].

An aspect of the Standard ML module system which remains problematic is its interaction with recursive constructs which cross module boundaries. The problem is that such constructs are not allowed; the module dependency graph is required to be acyclic. This is an instance of a problem shared by all other module systems. The essence of the problem is that recursion cannot cross module boundaries. As a result, for example, mutually recursive datatypes in ML must be defined in the same module. This results in a somewhat monolithic module structure, in which one module collects the datatype definitions, while client modules implement the functionality required for these datatypes. This monolithic structure is at odds with data abstraction, which would entail encapsulating the types, and the operations on those types, behind opaque interfaces.

Various languages take different approaches to reconciling the need to allow recursion to cross module boundaries. Haskell and Modula-3 allow cycles in the module import dependency graph. Although some such approach might be investigated for the Standard ML module system, this is not a general solution to what we would like achieve in breaking up recursive constructs across modules. For example the Haskell language also has *type classes* [31]. In the semantics provided in [10], type classes allow recursive definitions to cross module boundaries by breaking up recursive functions into implementations of an overloaded operator. The type system allows these implementations to be type-checked independently and then implicitly combined at points where an overloaded operation is used in an application.

In this paper we present an extension to Standard ML modules which supports modular building blocks where recursive definitions cross module boundaries, but which is a conservative extension of the existing module system. This approach allows SML modules to be "decomposed" into smaller parts, called *mixin modules*. Mixins allow recursive definitions to be broken up into modular fragments: both recursive function and recursive type definitions may be broken up and spread across modules. A combining operation allows these fragments to be combined, and a closure operation allows an ordinary Standard ML module to be produced from the composition of several mixins. At the core language level, we allow empty `datatype` definitions. These are primarily for mixins which rely on other mixins to provide the definitions of such datatypes (Sect. 2.2). We introduce an `inner` pseudo-variable, which allows access to further extensions of a recursive function within a mixin module body.

We provide several examples of the usefulness of our approach. Beyond resolving circular dependencies in import graphs, we demonstrate how our approach provides similar functionality to Haskell type classes [14] and CLOS generic functions [17]. This is done without extending the core ML type system or operational semantics. The former fact means that, with our approach, the core language programmer sees a familiar and widely-accepted type system. The latter fact means that our approach allows a straightforward efficient run-time implementation.

Section 2 provides several examples to motivate mixin modules Section 3 presents the formal type system for mixin modules; this type system is an extension of recent reformulations which have been proposed for SML modules [13, 18, 19]. Section 4 presents an operational semantics for mixin modules, in the form of a reduction system which maps mixin modules to ordinary SML modules. A semantic soundness result verifies that the semantics does not "fail" due to type errors during reduction. Section 5 considers related and future work, and provides our conclusions.

2 Examples of Mixin Modules

The addition of mixin modules to the ML module system requires the introduction of a new kind of module construct, in addition to structures and functors:

structure M = **mixin** $\langle defns_1\rangle$ **body** $\langle defns_2\rangle$ **init** $\langle defns_3\rangle$ **end**

The body of the mixin, $\langle defns_2\rangle$, is required to be a collection of datatype and recursive function definitions. These definitions may use type and value definitions from $\langle defns_1\rangle$, which we refer to as the *prelude* of the mixin. The definitions in $\langle defns_1\rangle$ and $\langle defns_2\rangle$ are both visible in $\langle defns_3\rangle$, which we refer to as the *initialization*

*Address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. dduggan@uwaterloo.ca

ICFP '96 5/96 PA, USA
© 1996 ACM 0-89791-771-5/96/0005...$3.50

```
structure Num =
mixin body
   datatype term = CONST of int
   datatype value = NUM of int
   type env = string -> value
   fun eval (CONST i) _ = NUM i
     | eval tm (e:env) = inner tm e
end (* Num *)

structure Func =
mixin
   fun bind(x,v,e) = fn y =>
       if x=y then v else e y
body
   datatype term = VAR of string |
      ABS of string * term | APP of term * term
   datatype value = CLOS of term * env
   withtype env = string -> value
   fun eval (VAR x) (env:env) = env x
     | eval (f as ABS _) e = CLOS (f,e)
     | eval (APP (rator,rand)) e =
       let val CLOS (ABS(x,b),e') = eval rator e
        in eval b (bind (x,eval rand e,e'))
       end
     | eval tm e = inner tm e
end (* Func *)
```

Figure 1: Simple Interpreter Mixins

section of the mixin. ⟨*defns*$_1$⟩ and ⟨*defns*$_3$⟩ contain type, function and module definitions, and have the usual ML semantics. The interesting part of the mixin is the *mixin body*, ⟨*defns*$_2$⟩. The definitions in this part are assumed to be recursive. More importantly, these definitions are assumed to be open to further extension, by combining the mixin with other mixins. When two mixins are combined, recursive definitions with the same names in the mixin bodies are merged. Corresponding to mixin modules we have *mixin signatures*:

signature S = **mixsig** ⟨*decls*$_1$⟩ **body** ⟨*decls*$_2$⟩ **init** ⟨*decls*$_3$⟩ **end**

Corresponding to the composition rules for mixin modules, our module calculus also has a composition rule for mixin signatures. Our module calculus also has an equality theory for signatures which allows the composition of mixin signatures to be transformed into a mixin signature of the aforesaid form. The closure of a mixin module produces a module with an ordinary SML signature. Like structures, mixins may occur inside functor bodies and may be arguments to functors, so there is no problem with their interaction with the SML module system. An extension we do not consider is *mixin functors*, mixins which may be combined and closed into a functor. We leave this for future work. We now provide several examples to demonstrate the use of mixins.

2.1 Simple Interpreter Mixins

Figure 1 provides a simple example of the use of mixins, for providing a modular implementation of an interpreter. The example is a simple version of that used by Steele to motivate pseudo-monads [29]. The example provides two building blocks, one for numbers and the other for functions. Both building blocks provide cases for the definition of the `term` and `value` datatypes. The calculus of mixin modules includes an operator for combining mixins, ⊗. In the composition `Num`⊗`Func`, the two definitions for the `term` datatype are combined into a single datatype definition. Similarly the two definitions for `eval` are combined. The pseudo-variable `inner`, available within a function in the body of a mixin module, provides programmer control over how this combination is performed. In the result of composing the definitions of `eval`, the resulting single `eval` function first checks for the `CONST` constructor (using the `Num` implementation) and then delegates to the `Func` implementation to check for the other constructors. Because of `inner`, composition ⊗ is not commutative.

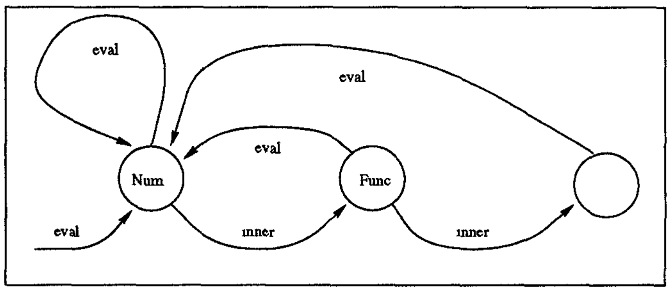

Figure 2: Binding of `inner`, and of recursive references to `eval`, in definition of `eval` in `Num` and `Func`

The `Func` mixin demonstrates the important points of the example. Both the `term` datatype and the `eval` function contain recursive references. However these recursive references are not necessarily to the current definitions of these datatypes. The final fixed point of these recursive definitions may include definitions provided by other mixins, and the recursive references within the `Func` mixin refer to these final fixed points. The definition of `inner`, on the other hand, is not a recursive reference to the definition of `eval`, but a non-recursive reference to further extensions of `eval` contributed by other mixins composed to the right of `Func`.

Figure 2 illustrates this graphically. An initial call to the `eval` function exported by the composition of these mixins begins by executing the code defined in the `Num` mixin. Invocations of `inner` transfer control to the code defined in the `Func` mixin. At any point, recursive references to `eval` transfer control back to the code defined in `Num`.

2.2 Cyclic Import Dependencies

The Standard ML module system prohibits cycles in the module import dependency graph. Figure 3 provides a practical example of the difficulties this causes, from the Standard ML of New Jersey compiler [1]. This example shows some of the import hierarchy for the modules comprising the front end of the compiler, modified by the addition of a type explanation facility to the type-checker [9]. The type explainer stores explanations in extra information fields in type variables. As a result, the `Types` module is now (through `TypeExplain`) a client of `Absyn`, which introduces a cycle in the import hierarchy. Our implementation of type explanation avoided this cycle by using unsafe type casting! The only alternative was to merge the `Types`, `Variable` and `Absyn` modules into a single "mega-module," since the types and abstract syntax datatypes are mutually recursive (with type explanation). Even so, this "disguised" import cycle shows up in other parts of the hierarchy. For example, `TypesUtil` exports operations for creating new type variables; these variables are initialized with null explanations, so the

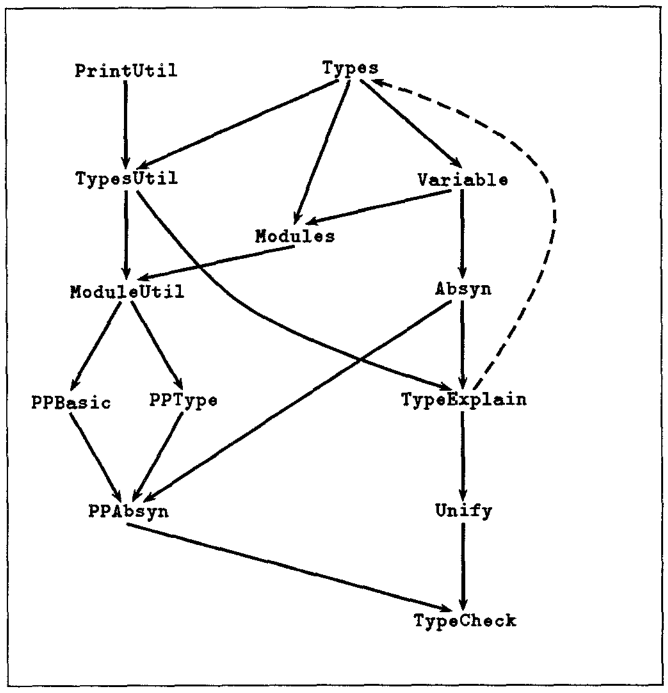

Figure 3: Import Hierarchy of SML/NJ

operations for creating these null explanations must be placed in `TypesUtil` rather than `TypeExplain`, where they really belong (where the type explanation data structures are declared).

One might be tempted to believe that these problems can be solved using functors, which have received a great deal of attention in the design and implementation of SML modules. For example, we might try to make `Types` into a functor, parameterized over the type explanations which are recorded with type variables. This functor is then instantiated at the point where `Absyn` is defined. Unfortunately this approach only serves to illustrate further problems with SML modules:

```
functor TypesF (T: TYPE_EXPLAIN) : TYPES = ...
functor TypeExplainF (A: ABSYN) : TYPE_EXPLAIN = ...
functor AbsynF (T: TYPES) : ABSYN = ...
structure Absyn = AbsynF (Types)
      and TypeExplain = TypeExplainF (Absyn)
      and Types = TypesF (TypeExplain)
```

Functorizing the code only delays the problems described earlier until the point at which the functors need to be instantiated. Again we face the problem that the abstract syntax and type datatypes are mutually recursive, so their definitions cannot span different modules.

The obvious solution to these problems is to allow recursive structures and functors. Modula-3 [24] is an example of a language which allows cycles in module dependency graphs. This has significant implications for the semantics of modules. Type definitions in modules may be mutually recursive, so type-checking must check for equality of rational trees. Allowing rational trees, as in Modula-3, would involve adding circular unification to the type inference algorithm. Although this can be done, experience suggests that some type errors are no longer detected statically in the resulting type system; furthermore the inferred types can be somewhat complicated [27]. The semantics of module initialization in Modula-3 are that, within a strongly connected component in the import dependency graph, the order of initialization is left undefined. This is clearly unacceptable in ML-like languages, where variables are immutable and bound to clearly defined initial values. In fact current implementations of Standard ML modules explicitly rely on the absence of circularities in the initialization of modules [2].

Essentially the semantics of module composition in Standard ML (and almost all other module languages) is based on functor application. This is so even if functors are not explicitly involved; in the SML/NJ implementation of linking [2], an ordinary ML structure is implemented as a function which is applied to a vector of imports at link-time, and produces a vector of exports. Mixin modules introduce a new form of module composition, based on "merging" the fixed points of recursive definitions. Within the body of the mixin, all value definitions are required to be λ-abstractions; since evaluation is delayed, order-of-initialization ambiguities are avoided. Outside of the body of the mixin, the semantics of value definitions are the same as that for Standard ML. As noted, mixin composition is non-commutative because of the use of `inner`. The initialization order for definitions outside of the mixin bodies in a mixin composition is to initialize the "child" (rightmost) definitions first, for reasons made clear in Sect. 4.

The solution to the original problem is then to declare `Absyn`, `TypeExplain` and `Types` as mixins. This allows for example the type explainer to be reused with different abstract syntaxes and type systems, by combining it with different `Absyn` and `Types` mixins. The types module specifies a datatype for object language types, and some useful functions over that representation which are required by clients. The module makes use of the undetermined type explanation datatype by giving a null definition for this datatype, depending on a type explanation mixin to complete its definition. This requires a mild extension of core ML, allowing empty datatype definitions. A particular implementation of a compiler frontend with type explanation is then constructed by combining these mixins and "closing up" the result (using the mixin operations provided in Sect. 3):

```
structure Types = mixin body datatype explain;
                             datatype ty = ... end
structure TypeExplain = ...
structure Absyn = ...
structure Front = clos (Types⊗TypeExplain⊗Absyn)
```

Consider if the frontend requires an operation for copying abstract syntax trees. Within the `Types` mixin, we define a function `copyTy` which recurses over a `ty` data structure. At some point this function needs to copy explanations. To do this, `copyTy` makes a recursive call to `copyExplain`, where the latter is defined by:

```
fun copyExplain (exp:explain):explain = inner(exp)
```

The `Types` mixin then relies on the type explanation mixin to complete the definition of `copyExplain` (and the latter will in turn call `copyAbsyn`; the implementation of `copyAbsyn` in `Absyn` will in turn call `copyTy`).

2.3 Parametric Overloading and Generic Functions

The next example compares mixin modules with parametric overloading [16], as exemplified by Haskell type classes [14], and with CLOS generic functions [17]. Although parametric overloading is well-understood for single-parameter overloading, the implementation of type classes remains problematic, necessitating as they do call-site closure construction at the uses of overloaded functions. Type classes remain the main performance bottleneck in Haskell

```
structure Linear =
mixin body
   datatype car = VECTOR of car list |
      MATRIX of car list list
   fun (VECTOR vs) * (VECTOR vs') = ...
     | (MATRIX ms) * (VECTOR vs) = ...
     | v * (VECTOR vs) =
       VECTOR (map (fn v' => v * v') vs)
     | v * v' = inner(v,v')
end (* Linear *)

structure Number =
mixin body
   datatype car = REAL of real | INT of int
   fun (REAL x) * (INT y) = ...
end (* Number *)
```

Figure 4: Numerical Algebra Mixins

compilers. Furthermore type-checking for some extensions of parameter overloading remains unclear. Figure 4 demonstrates a use of mixin modules for an application of "multi-parameter" parametric overloading. The **Linear** mixin implements linear algebra operations for vectors and matrices, while the **Number** mixin implements operators for integers and reals. In the definition of multiplication in **Linear**, there is a case for vector-vector and matrix-vector multiplication, with the third clause handling scalar-vector multiplication (where scalars are meant to be provided by another mixin). Duggan and Ophel [11] demonstrate that, in any type system for multi-parameter parametric overloading which is rich enough to support the example in Fig. 4, type-checking is undecidable.

Mixin modules avoid these decidability and performance problems. Type-checking at the core language level is exactly the same as in Standard ML. Furthermore our module calculus avoids all of the implementation problems associated with type classes. The definitions of a mixin module cannot be used until the module is closed up to an ordinary ML module. At the point where this is done, the datatype and function definitions in the mixin body are closed to any further extensions, and code generation and optimization may proceed without needing to accomodate future extensions to the definitions. As a result mixin modules introduce no complications for runtime efficiency.

Mixin modules can also model other extensions of parametric overloading. For example, a facility analogous to type constructor classes [15] can be modelled using mixins with collection datatypes:

```
structure List = mixin body
   datatype α carr = LIST of α list
   fun map f (LIST xs) = ...
     | map f xs = inner f xs
end (* List *)

structure Tree = mixin body
   datatype α carr = LEAF |
      NODE of α * α carr * α carr
   fun map f LEAF = LEAF
     | map f (NODE(x,t1,t2)) =
       NODE(f x,map f t1,map f t2)
     | map f xs = inner f xs
end (* Tree *)
```

Mixin modules and Haskell type classes each have their relative advantages. There is an essential difference between the two: the latter assumes homogeneous collection data structures (for example, all elements of a vector are integers, or all are matrices, etc). The approach of mixin modules allows each element of a vector to be in the union of the types which make up the carrier type, so the approach of mixins allows heterogeneous collections. Run-time dispatching is then based on the tags associated with the data constructors. The approach of mixin modules appears preferable for e.g. graphical user interface applications. For example, we can structure an extensible window library as a collection of mixin modules, with an extensible window type; the basic window library may be extended by adding mixin modules defining new window types. The advantage of the mixin modules approach may be seen by considering e.g., window hierarchies, which can be represented as a list of subwindows associated with each window. A function which operates over the window hierarchy dispatches based on the data constructor tag associated with each window in a list of subwindows. In this way mixin modules may be seen as in some sense analogous to CLOS generic functions [17]; what is lacking from mixin modules is any notion of subtyping or subclassing, although subclassing is available with an extension of mixin modules which we consider in Sect. 5.

The approach of type classes cannot handle this example of an extensible window system, because it requires that collection types be homogeneous: the immediate subwindows of a given window would all have to be of the same window type, which is clearly useless. On the other hand the approach of type classes is preferable for applications involving homogeneous collection datatypes (e.g. vectors and matrices), since for example in mapping an overloaded operation over a vector, dispatching is done once for the entire vector, rather than repeatedly for each element of the vector.

3 Typing Mixin Modules

In this section we provide the static semantics for mixin modules. Section 3.1 summarizes the syntax of an ML-like minilanguage, while Sect. 3.2 provides the type rules for mixins. The dynamic semantics are provided in the next section.

3.1 Syntax

Figure 5 contains the grammar of the calculus used in the presentation of mixin semantics and typing. Names **s**, **t**, **x** denote module, type, and value components of modules respectively, while identifiers $s_{\mathbf{s}}$, $t_{\mathbf{t}}$, $t_{\mathbf{t}}$ bind to modules, types and values. Identifiers $i_{\mathbf{i}}$ consist of an *external name* **i** and an *internal name* *i*. Following the approaches of Leroy [18] and Harper and Lillibridge [13], only the internal name of a bound identifier admits α-conversion; this permits module access by external name and composition of corresponding fields in mixin bodies. This renaming is implicit in the modules calculus (similarly to other calculi such as the λ-calculus), but is necessary for some of the preconditions of the type rules which require internal names to be distinct in the concatenation of environments and signatures.

The two main classes of expression are modules M_t and expressions E. A structure is a collection of definitions M, and a functor is a mapping from modules to modules. In addition we now have mixins as a new module construct, denoted by $M_1\varsigma(M)M_2$. This is abstract syntax for mixin module expressions of the form `mixin` M_1 `body` M `init` M_2 `end` used in the previous section. M_1 and M_2 denote the prelude and initialization parts, respectively, of the mixin. $\varsigma(M)$ denotes the body, in which all (type and value) definitions are mutually recursive. Module types are defined by the S_t

$$
\begin{array}{rcl}
M_t & ::= & \textbf{struct}\, M\, \textbf{end} \mid \textbf{functor}(\textbf{structure}\, s_{\mathrm{s}}{:}S_t)\, M_t \mid (M_t{:}S_t) \\
 & & p \mid p(p) \mid M\varsigma(M_b)M \mid M_t \otimes M_t \mid \textbf{clos}(M_t) \\
S_t & ::= & \textbf{sig}\, S\, \textbf{end} \mid \textbf{funsig}\, (\textbf{structure}\, s_{\mathrm{s}}{:}S_t)S_t \mid S\varsigma(S)S \\
S & ::= & \textbf{type}\, t_{\mathrm{t}} \mid t = \tau \mid t\, \textbf{is}\, \phi \mid \textbf{val}\, x_{\mathbf{x}}{:}\tau \\
 & & \mid \textbf{structure}\, s_{\mathrm{s}}{:}S_t \mid S;S \mid \varepsilon \\
p & ::= & \varepsilon \mid s \mid p.\mathrm{s} \\
 & & \\
M & ::= & \textbf{structure}\, s_{\mathrm{s}} = M_t \mid \textbf{type}\, t_{\mathrm{t}} \mid t = \tau \mid t\, \textbf{is}\, \phi \mid M;M \mid D \\
D & ::= & \textbf{fun}\, f^1_{\mathrm{f}_1} = \lambda x_1.E_1 \,\|\, \ldots \,\|\, f^n_{\mathrm{f}_n} = \lambda x_n.E_n \\
 & & \mid \textbf{val}\, x_{\mathbf{x}} = E \mid D;D \mid \varepsilon \\
M_b & ::= & \textbf{val}\, f_{\mathrm{f}} = \lambda x.E \mid \textbf{type}\, t_{\mathrm{t}} \mid t\, \textbf{is}\, \phi \mid M^1_b;M^2_b \mid \varepsilon \\
 & & \\
E & ::= & \bot \mid x \mid p.\mathbf{x} \mid p.\mathbf{c}(\overline{E_n}) \mid E_1\, E_2 \mid \lambda y.E \\
 & & \mid \textbf{let}\, D\, \textbf{in}\, E \mid \textbf{case}\, E\, \textbf{of}\, R_1\, `\mid\textrm{'}\ldots`\mid\textrm{'}\, R_n \\
R & ::= & P \Rightarrow E \\
P & ::= & x \mid p.\mathbf{c}(\overline{x_n}) \\
 & & \\
\tau & ::= & t \mid p.\mathbf{t} \mid \tau_1 \rightarrow \tau_2 \\
\phi & ::= & \mathbf{c}(\overline{\tau_n}) \mid \phi_1 \cup \phi_2
\end{array}
$$

Figure 5: Syntax

$$
\begin{array}{rcl}
\mathtt{dom}(\Gamma) & \equiv & \{x \mid \textbf{val}\, x : \tau \in \Gamma\} \cup \\
 & & \{t \mid \textbf{type}\, t \in \Gamma\} \cup \\
 & & \{s \mid \textbf{structure}\, s : S_t \in \Gamma\} \\
\mathtt{dom}(\phi) & \equiv & \{\mathbf{c} \mid \phi = \phi_1 \ldots \cup \mathbf{c}(\overline{\tau_n}) \cup \ldots \phi_n\} \\
\mathtt{cons}(\Gamma) & \equiv & \bigcup\{\mathtt{dom}(\phi) \mid (t\, \textbf{is}\, \phi) \in \Gamma\} \\
\mathtt{cons}(S) & \equiv & \bigcup\{\mathtt{dom}(\phi) \mid (t\, \textbf{is}\, \phi) \in S\} \\
\mathtt{cons}(M) & \equiv & \bigcup\{\mathtt{dom}(\phi) \mid (t\, \textbf{is}\, \phi) \in M\} \\
S_1 \oplus S_2 & \equiv & S_1, S_2 \quad \text{if} \left\{ \begin{array}{l} \mathtt{IN}(\mathtt{BV}(S_1)) \cap \mathtt{IN}(\mathtt{BV}(S_2)) = \{\} \\ \mathtt{cons}(S_1) \cap \mathtt{cons}(S_2) = \{\} \end{array} \right. \\
\Gamma_1 + \Gamma_2 & \equiv & \Gamma_1 \cup \Gamma_2 \quad \text{if} \left\{ \begin{array}{l} \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) = \{\} \\ \mathtt{cons}(\Gamma_1) \cap \mathtt{cons}(\Gamma_2) = \{\} \end{array} \right. \\
\mathtt{NEW}(\Gamma, t) & \equiv & (\textbf{type}\, t) \in \Gamma \text{ and } (t\, \textbf{is}\, \phi'), (t = \tau) \notin \Gamma \\
\mathtt{EN}(N) & \equiv & \{\mathbf{x} \mid y_{\mathbf{x}} \in N\} \\
\mathtt{IN}(N) & \equiv & \{y \mid y_{\mathbf{x}} \in N\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{BV}(S;\, \textbf{type}\, t_{\mathrm{t}}) & \equiv & (\mathtt{BV}(S) - \{t'_{\mathrm{t}}\}_{t'}) \cup \{t_{\mathrm{t}}\} \\
\mathtt{BV}(S;\, \textbf{val}\, x_{\mathbf{x}} : \tau) & \equiv & (\mathtt{BV}(S) - \{x'_{\mathbf{x}}\}_{x'}) \cup \{x_{\mathbf{x}}\} \\
\mathtt{BV}(S;\, t\, \textbf{is}\, \phi) & \equiv & \mathtt{BV}(S) \\
\mathtt{BV}(S;\, t = \tau) & \equiv & \mathtt{BV}(S) \\
\mathtt{BV}(S;\, \textbf{structure}\, s_{\mathrm{s}} : S') & \equiv & (\mathtt{BV}(S) - \{s'_{\mathrm{s}}\}_{s'}) \cup \{s_{\mathrm{s}}\} \\
\mathtt{BV}(M;\, \textbf{type}\, t_{\mathrm{t}}) & \equiv & \cdots \text{(similar)}
\end{array}
$$

Figure 6: Definitions

syntax class, and consist of signatures for structures, functor signatures for functors, and a new form of module type for mixins, $S_1\varsigma(S)S_2$. Definitions M in a module consist of structure definitions, type definitions and value definitions. Value definitions are separated into a separate syntax class since they may also appear in a `let`-expression. Value definitions include a special `fun` form for defining mutually recursive functions; each body E_i is assumed to be a λ-abstraction. In type definitions, we separate the declaration of a new type name (**type** t_{t}) from its definition as a type abbreviation ($t = \tau$) or a datatype (t **is** ϕ), as explained in the next subsection.

Our calculus contains several simplifications for the sake of exposition. Functor applications are only allowed to have the form $p(p')$ where p and p' are paths; this enables us to give a reduction semantics which preserves type identity, a crucial issue in the design of a module type system with generativity. This simplification is inessential, and we may assume a source-level translation which converts from a more liberal syntax to this one [18]. We only allow simple paths $s.\mathrm{s}_1 \ldots \mathrm{s}_n$; in particular we do not have functor applications in paths, as suggested by Leroy [19] for typing higher-order functors. We do this to simplify the presentation, since higher-order functors do not materially affect the semantics of mixin modules. We make the simplifying assumption that functors are first-order, as in Standard ML, although this is not strictly enforced in the type rules. This assumption is only necessary to simplify reasoning about type soundness in the next section, due to the fact that we work with a reduction semantics rather than the usual denotational semantics. The core language types are simple type names, projections of type components of structures, and arrow types. This type system can be extended very straightforwardly to more sophisticated core language types (polymorphic types, parameterized types, constrained types, etc) without contributing anything to the exposition. Finally we assume that each functor takes a single module as its argument; this simplifies the type rule for functor application.

The body of a mixin may only contain type and value definitions. We make this restriction because of our decision that it should always be possible to transform a combination of mixins into an ordinary ML module. For simplicity we only allow datatype definitions, although type abbreviations can also be added (with some restriction on recursive references having to go "through" a datatype). M_b gives the syntax of mixin body definitions. All value definitions, and all type definitions, are assumed to be mutually recursive. As with function definitions, value definitions are required to be λ-abstractions. This prevents circularities in initialization.

Figure 6 provides several metafunctions which are used in the type rules in the next subsection. $\mathtt{dom}(\Gamma)$ denotes the internal names in the domain of Γ. $\mathtt{dom}(\phi)$ denotes the constructors in a datatype definition. `cons` extends this to environments, signatures and modules. $\mathtt{BV}(S)$ denotes the set of (internal name,external name) pairs exported by the sequence of declarations S; this operation excludes names which are "shadowed" by declarations to the right in S (i.e., a variable $x_{\mathbf{x}}$ is excluded if there is another declaration with external name $\mathbf{x}$ to the right of it in S). $\mathtt{BV}(M)$ is defined similarly for sequences of definitions. $\mathtt{IN}(N)$ returns the internal names in a set of names, while $\mathtt{EN}(N)$ returns the external names. $\mathtt{NEW}(\Gamma, t)$ is a predicate which checks that there is a declaration for the name t, but that there is no type or datatype assertion for t. $S_1 \oplus S_2$ denotes the concatenation of the declarations of S_1 and S_2; the side-condition on this operation ensures that internal names in the declarations are renamed apart beforehand, and that the declarations have no data constructors in common. Similarly $\Gamma_1 + \Gamma_2$ denotes the concatenation of two type contexts, with the proviso that they have no internal names in common in their domains.

3.2 Mixin Typing Rules

We base the module type system on the systems of manifest types proposed independently by Leroy [18] and by Harper and Lillibridge [13] (see also [26]). We base our formulation on the approach of manifest types because of its relative simplicity in modeling generativity.

In the approach of manifest types, a definition in a structure **type** $t = \tau$ gives rise to the manifest type **type** $t = \tau$ in the signature of the structure. A subtyping relation on type declarations includes the rule (**type** $t = \tau$) <: (**type** t). Extended to signatures, this rule allows type information to be "forgotten" in signatures, so that manifest types may be made opaque. "Generativity" is obtained by requiring that every top-level module be given a unique "stamp" (or, be bound to a unique "internal name"), and making generative types opaque and with their identity determined by their path beginning at a top-level structure.

We modify this approach in the following ways. We break a type declaration into two parts, the binding assertion **type** t and the defining assertions $t = \tau$ and t **is** ϕ. Each bound type name has no more than one defining assertion. In the aforesaid approaches to module typing, only the first form of defining assertion (for type abbreviations) is allowed. We introduce the second form of defining assertion for generative datatypes. With this approach, datatypes with identical forms but different "stamps" are distinguished; datatypes occupy a middle ground between manifest and opaque types. The final modification is to *type strengthening* [18]: given a structure with path p and signature S_t, type strengthening S_t/p allows the replacement of an opaque type **type** t_t in S_t with the manifest type **type** $t_t = p.t$. Type strengthening in our module system is given by:

$$\begin{array}{rcl}
(\textbf{sig}\ S\ \textbf{end})/p & = & \textbf{sig}\ S/p\ \textbf{end} \\
(\textbf{funsig}\ (S)S_t)/p & = & \textbf{funsig}\ (S)S_t \\
(S\varsigma(S)S)/p & = & S\varsigma(S)S \\
(\textbf{structure}\ s_s : S_t\ ;\ S)/p & = & \textbf{structure}\ s_s : (S_t/p.s)\ ;\ S/p \\
(\textbf{type}\ t_t;\ S)/p & = & \textbf{type}\ t_t;\ t = p.t;\ S/p \\
(t = \tau;\ S)/p & = & S/p \\
(t\ \textbf{is}\ \phi;\ S)/p & = & S/p \\
(\textbf{val}\ x_x : \tau;\ S)/p & = & \textbf{val}\ x_x : \tau;\ S/p
\end{array}$$

Type strengthening allows a datatype to be shared between modules:

```
structure S1 =
   struct datatype foo = bar; val x = bar end
structure S2 =
   S1 : sig type foo = S1.foo; val x: foo end
if true then S1.bar else S2.x (* type checks *)
```

Figure 7 provides the type rules for the modules language. The rules for typing declarations of modules (R-MOD,R-MOD-MR), structures (R-MOD-SS) and functors (R-MOD-FS), as well as the rule for functor application (R-MOD-PA) are as usual and self-explanatory. The rule for projecting a module s from within a structure (R-MOD-SP) performs a substitution to construct the signature of s. All occurrences of an internal name bound before the declaration of s in the signature of the enclosing structure are prefixed by the path to this structure *and* replaced with the corresponding external name. This prefixing is also done for data constructors defined before the declaration of s. This prefixing step is necessary to properly remove the dependency of s on its local context since (as mentioned) internal names admit α-conversion whereas external names are fixed. Finally Rule R-MOD-TS gives the type strengthening rule for the calculus, Rule R-MOD-MAT gives the type rule for explicit signature matching, while Rule R-MOD-SUB gives the type subsumption rule. We omit type rules for checking the well-formedness of signatures ($\Gamma \vdash S_t$ **modtype**) and for subtyping ($\Gamma \vdash S_t <: S_t'$ and $\Gamma \vdash S <: S'$), since these are essentially standard [13, 18, 19].

$$\frac{\Gamma \vdash M_t : S_t \qquad \Gamma + \{\textbf{structure}\ s : S_t\} \vdash M : S}{\Gamma \vdash (\textbf{structure}\ s_s = M_t\ ;\ M) : (\textbf{structure}\ s_s : S_t;\ S)} \quad \text{(R-MOD)}$$

$$\frac{\textbf{structure}\ s : S_t \in \Gamma}{\Gamma \vdash s : S_t} \quad \text{(R-MOD-MR)}$$

$$\frac{\Gamma \vdash p : (\textbf{sig}\ S';\ \textbf{structure}\ s_s : S_t\ ;\ S''\ \textbf{end}) \qquad s \notin \mathtt{EN}(\mathtt{BV}(S'')) \qquad \theta = \{p.\mathrm{n}/n \mid n_{\mathrm{n}} \in \mathtt{BV}(S')\} \cup \{p.\mathrm{c}/\mathrm{c} \mid \mathrm{c} \in \mathtt{cons}(S')\}}{\Gamma \vdash p.s : \theta(S_t)} \quad \text{(R-MOD-SP)}$$

$$\frac{\Gamma \vdash p : S}{\Gamma \vdash p : S/p} \quad \text{(R-MOD-TS)}$$

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash \textbf{struct}\ M\ \textbf{end}\ : \textbf{sig}\ S\ \textbf{end}} \quad \text{(R-MOD-SS)}$$

$$\frac{\Gamma + \{\textbf{structure}\ s : S_t\} \vdash M_t\ : S_t' \qquad \Gamma \vdash S_t\ \textbf{modtype}}{\Gamma \vdash \textbf{functor}(\textbf{structure}\ s_s : S_t)M_t\ : \textbf{funsig}(\textbf{structure}\ s_s : S_t)S_t'} \quad \text{(R-MOD-FS)}$$

$$\frac{\Gamma \vdash p : \textbf{funsig}(\textbf{structure}\ s_s : S_t')S_t \qquad \Gamma \vdash p' : S_t'}{\Gamma \vdash p(p') : \{p'/s\}S_t} \quad \text{(R-MOD-PA)}$$

$$\frac{\Gamma \vdash M_t : S_t}{\Gamma \vdash (M_t : S_t) : S_t} \quad \text{(R-MOD-MAT)}$$

$$\frac{\Gamma \vdash M_t : S_t \qquad \Gamma \vdash S_t'\ \textbf{modtype} \qquad \Gamma \vdash S_t <: S_t'}{\Gamma \vdash M_t : S_t'} \quad \text{(R-MOD-SUB)}$$

$$\frac{\Gamma \vdash M_p : S_p \qquad \Gamma + \overline{S_p} \Vdash M_b : S_b \qquad \Gamma + \overline{S_p} + \overline{S_b} \Vdash M_b\ \textbf{ok} \qquad \Gamma + \overline{S_p} + \overline{S_b} \vdash M_i : S_i}{\Gamma \vdash M_p \varsigma(M_b) M_i : S_p \varsigma(S_b) S_i} \quad \text{(R-MOD-MIX)}$$

$$\frac{\Gamma \vdash M_1 : S_p^1 \varsigma(S_b^1) S_i^1 \qquad \Gamma \vdash M_2 : S_p^2 \varsigma(S_b^2) S_i^2 \qquad S_p^2 \oplus S_p^1 = S_p \qquad \Gamma + \overline{S_p} \vdash S_b^1 \otimes S_b^2 = S_b \qquad S_i^2 \oplus S_i^1 = S_i}{\Gamma \vdash M_1 \otimes M_2 : S_p \varsigma(S_b) S_i} \quad \text{(R-MOD-COMP)}$$

$$\frac{\Gamma \vdash M_t : S_p \varsigma(S_b) S_i}{\Gamma \vdash \textbf{clos}(M_t) : S_p \oplus S_b \oplus S_i} \quad \text{(R-MOD-CL)}$$

Figure 7: Modules Language Typing Rules

Rule R-MOD-MIX in Figure 7 determines the signature of an atomic mixin module:

$$\frac{\Gamma \vdash M_p : S_p \qquad \Gamma + \overline{S_p} \Vdash M_b : S_b \qquad \Gamma + \overline{S_p} + \overline{S_b} \Vdash M_b\ \textbf{ok} \qquad \Gamma + \overline{S_p} + \overline{S_b} \vdash M_i : S_i}{\Gamma \vdash M_p \varsigma(M_b) M_i : S_p \varsigma(S_b) S_i} \quad \text{(R-MOD-MIX)}$$

This rule first constructs the signature S_p of the prelude section M_p. The operation $\overline{S_p}$ returns the set of all declarations in S_p after removing external names from identifiers. These declarations are added to the context to perform a "first pass" over the body of the mixin in order to construct a signature S_b for the body. The second pass over the body adds $\overline{S_p}$ and $\overline{S_b}$ to the context. The latter is needed because declarations within the body are mutually recursive. The second pass can therefore verify the correctness of types ascribed to mutually recursive functions. The final step is the derivation of a signature for the initialization section after the context has been extended with the declarations of the prelude and body sections. Type inference for the functions in the mixin body is exactly the same as type inference for mutually recursive functions in Hindley-Milner type inference. Some explicit type information is necessary if polymorphic recursion is desired.

The first and second passes over the mixin body are performed using the rules shown in Figure 8. Rule R-MFP-T checks a binding assertion for a datatype, ensuring that the external name introduced for the type does not occur elsewhere in the mixin module body. This requirement is appropriate since declarations within the body section are mutually recursive and, moreover, constructs with the same external name are extended during composition. Rule R-MFP-I handles the defining assertion of a datatype, checking that a binding assertion has already been encountered and that no other defining assertions exist for the datatype. The rule also ensures that data constructors are all new and distinct (checked as a side-condition of the $+$ operation). Rule R-MFP-V checks a value declaration, requiring that the value be a λ-abstraction as explained in section 3.1.

The remaining rules of Figure 8 perform the second pass over the body section. Rule R-MSP-V can verify the type of a function since the bindings for the identifiers of any recursive references made by the function exist in the context. Before checking the function, the rule also adds an appropriate binding for the pseudo-variable **inner** which, as demonstrated in Section 2, acts as a placeholder of extensions supplied by other mixins.

Rule R-MOD-COMP in Figure 7 gives the rule for typing the composition of two mixins $M_1 \otimes M_2$:

$$\frac{\Gamma \vdash M_1 : S_p^1 \varsigma(S_b^1) S_i^1 \quad \Gamma \vdash M_2 : S_p^2 \varsigma(S_b^2) S_i^2 \\ S_p^2 \oplus S_p^1 = S_p \quad \Gamma + \overline{S_p} \vdash S_b^1 \otimes S_b^2 = S_b \quad S_i^2 \oplus S_i^1 = S_i}{\Gamma \vdash M_1 \otimes M_2 : S_p \varsigma(S_b) S_i} \quad \text{(R-MOD-COMP)}$$

The signatures for the prelude and initialization sections are composed as $S_p^2 \oplus S_p^1$ and $S_i^2 \oplus S_i^1$ respectively, with '$\oplus$' defined in Figure 6. This operation is essentially sequential composition.

Figure 9 contains the rules used by Rule R-MOD-COMP to compose the signatures of the bodies of the mixins being composed. The composition of two signatures S_1 and S_2 is denoted by $S_1 \otimes S_2$. The rules in Figure 9 allow the composition of two signatures $S_1 \otimes S_2$ to be rewritten to a normal signature. At the heart of these rules is the R-CB-I rule which joins together the constructors from the definition of a datatype in two different mixin bodies. The first premise in this rule ensures that there is no overlap in the constructors defined for that datatype in the two different mixins. The addition of the combined datatype assertion to the context, $\Gamma + \{t \textbf{ is } \phi\}$, checks that there is no overlap between the constructors being defined and the constructors already defined in the context Γ (recall that Γ has already combined the constructor definitions to the left of these datatype definitions in the mixin prelude and body).

The R-CB-TI, R-CB-II and R-CB-VI rules in Figure 9 allow a type constructor or program variable which is defined in one mixin but not the other, to be added to the signature of the mixin resulting from their composition. We refer to these rules as the *inflation rules*. They essentially allow us to inflate the interface of a mixin with declarations that are missing from its body, in order to allow us to compose that mixin with other mixins that provide the missing declarations.

In the composition $M_1 \otimes M_2$ of two mixins M_1 and M_2, we refer to M_1 and M_2 as the *parent* and *child* modules, respectively. In the sequential composition of the prelude and initialization parts of the mixins, the parent mixin's declarations follow the child mixin's declarations. Consider for example:

```
structure m1 = mixin
      type t = int
  body val x = fn w:int => w+1
  init val y = x
  end
structure m2 = mixin
      type t = bool
  body val x = fn w:int => 3
  init val x = true
  end
```

Composing $m_1 \otimes m_2$ and closing results in a structure:

```
struct
    type t' = bool; type t = int
    val x = fn w:int => w+1
    val x' = true; val y = x
end
```

with signature:

```
sig type t = int
    val x: int -> int
    val y: int -> int
end
```

Note that some renaming of the child mixin's shadowed fields is required. This approach is taken to make the overriding of definitions in the prelude and initialization part consistent with composition of bodies: since the parent does not use the `inner` pseudo-variable in the definition of `x`, the definition of `x` in the child is "discarded." So in some sense the parent is "in control" and determines what parts of the child mixin are visible in the final structure.

Rule R-MOD-CL determines the signature of an ML structure resulting from closure of a mixin module. The signature is the concatenation '$\oplus$', as defined in Figure 6, of the signatures for the three sections of the mixin module.

$$\frac{\mathbf{t} \notin \mathtt{EN}(\mathtt{BV}(S)) \quad \Gamma + \{\mathbf{type}\, t\} \Vdash M : S}{\Gamma \Vdash (\mathbf{type}\, t_{\mathfrak{t}};\ M) : (\mathbf{type}\, t_{\mathfrak{t}};\ S)} \quad \text{(R-MFP-T)}$$

$$\frac{\mathtt{NEW}(t, \Gamma) \quad \Gamma + \{t \textbf{ is } \phi\} \Vdash M : S}{\Gamma \Vdash (t \textbf{ is } \phi;\ M) : (t \textbf{ is } \phi;\ S)} \quad \text{(R-MFP-I)}$$

$$\frac{\mathbf{f} \notin \mathtt{EN}(\mathtt{BV}(S)) \quad \Gamma + \{\mathbf{val}\, f : \tau\} \Vdash M : S}{\Gamma \Vdash (\mathbf{val}\, f_{\mathfrak{f}} = E;\ M) : (\mathbf{val}\, f_{\mathfrak{f}} : \tau;\ S)} \quad \text{(R-MFP-V)}$$

$$\frac{\Gamma \Vdash M \textbf{ ok}}{\Gamma \Vdash (\mathbf{type}\, t_{\mathfrak{t}};\ M) \textbf{ ok}} \quad \text{(R-MSP-T)}$$

$$\frac{\Gamma \Vdash M \textbf{ ok}}{\Gamma \Vdash (t \textbf{ is } \phi;\ M) \textbf{ ok}} \quad \text{(R-MSP-I)}$$

$$\frac{(f : \tau) \in \Gamma \quad \Gamma + \{\mathbf{inner} : \tau\} \Vdash E : \tau \quad \Gamma \Vdash M \textbf{ ok}}{\Gamma \Vdash (\mathbf{val}\, f_{\mathfrak{f}} = E;\ M) \textbf{ ok}} \quad \text{(R-MSP-V)}$$

Figure 8: Atomic Mixin Typing Rules (See Rule R-MOD-MIX in Figure 7)

The SML module system includes a subtyping relation based on interface containment. As mentioned, approaches based on manifest types augment this with the ability to "forget" manifest types. For mixins the subtype relation must be made invariant in the top-level definitions in all parts of the mixin:

$$\frac{\mathtt{BV}(S_p^1) = \mathtt{BV}(S_p^2) \quad \mathtt{BV}(S_i^1) = \mathtt{BV}(S_i^2) \\ \Gamma \vdash S_p^1 <: S_p^2 \quad \Gamma \vdash S_i^1 <: S_i^2}{\Gamma \vdash S_p^1 \varsigma(S_b) S_i^1 <: S_p^2 \varsigma(S_b) S_i^2} \quad \text{(R-MIX-S)}$$

$$\frac{\Gamma + \{\mathbf{type}\, t\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (\mathbf{type}\, t_{\mathtt{t}};\, S_1) \otimes (\mathbf{type}\, t_{\mathtt{t}};\, S_2) = (\mathbf{type}\, t_{\mathtt{t}};\, S)} \quad \text{(R-CB-T)}$$

$$\frac{\mathtt{t} \notin \mathtt{EN}(\mathtt{BV}(S_2)) \quad \Gamma + \{\mathbf{type}\, t\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (\mathbf{type}\, t_{\mathtt{t}};\, S_1) \otimes S_2 = (\mathbf{type}\, t_{\mathtt{t}};\, S)} \quad \text{(R-CB-TI)}$$

$$\frac{\mathtt{dom}(\phi_1) \cap \mathtt{dom}(\phi_2) = \{\} \quad \Gamma + \{t\, \mathbf{is}\, \phi_1 \cup \phi_2\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (t\, \mathbf{is}\, \phi_1;\, S_1) \otimes (t\, \mathbf{is}\, \phi_2;\, S_2) = (t\, \mathbf{is}\, \phi_1 \cup \phi_2;\, S)} \quad \text{(R-CB-I)}$$

$$\frac{\mathtt{NEW}(t, \Gamma) \quad \Gamma + \{t\, \mathbf{is}\, \phi\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (t\, \mathbf{is}\, \phi;\, S_1) \otimes S_2 = (t\, \mathbf{is}\, \phi;\, S)} \quad \text{(R-CB-II)}$$

$$\frac{\Gamma + \{\mathbf{val}\, f : \tau\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (\mathbf{val}\, f_{\mathtt{f}} : \tau;\, S_1) \otimes (\mathbf{val}\, f_{\mathtt{f}} : \tau;\, S_2) = (\mathbf{val}\, f_{\mathtt{f}} : \tau;\, S)} \quad \text{(R-CB-V)}$$

$$\frac{\mathtt{f} \notin \mathtt{EN}(\mathtt{BV}(S_2)) \quad \Gamma + \{\mathbf{val}\, f : \tau\} \vdash S_1 \otimes S_2 = S}{\Gamma \vdash (\mathbf{val}\, f_{\mathtt{f}} : \tau;\, S_1) \otimes S_2 = (\mathbf{val}\, f_{\mathtt{f}} : \tau;\, S)} \quad \text{(R-CB-VI)}$$

Figure 9: Mixin Composition Typing Rules (See Rule R-MOD-COMP in Figure 7)

This restriction is necessary because of the composition rule for mixins: If interface containment is allowed on the definitions in a mixin, then the meaning of the composition of the following is ambiguous:

```
structure m1 = mixin val x = true end
structure m2 = mixin val x = 3 end
structure m3 = mixin val x = 4 end
structure s = clos (m1 ⊗ m2 ⊗ m3)
```

For example, without the restriction on subtyping, the type of `s.x` could be `int` or `bool`; if its type were `int`, its value could be 3 or 4. For example, we could use subtyping as follows:

`m1 :   mixsig val x:bool end   <:   mixsig end`

to "forget" the value of `m1` so that in the final composition the type of `x` would be `int` instead of `bool` (using inflation in mixin composition). The restriction in rule R-MIX-S dissallows this since the two signatures do not share the same collection of identifiers.

Figure 10 provides typing rules for the core language. We omit rules for expressions E. These are standard and their inclusion would be straightforward. Mixin modules are intended as an extension to the module system of ML, which leaves the type system and semantics of the core language unmodified.

The key property that is required for type-checking is the existence of principal types for modules.

Definition 3.1 *Given a context* Γ.

An expression E *has a principal type* τ *if* $\Gamma \vdash E : \tau$, *and for any other* τ' *such that* $\Gamma \vdash E : \tau'$, *we have* $\Gamma \vdash \tau <: \tau'$.

A module M_t *has a principal signature* S_t *if* $\Gamma \vdash M_t : S_t$, *and for any other* S'_t *such that* $\Gamma \vdash M_t : S'_t$, *we have* $\Gamma \vdash S_t <: S'_t$.

Lemma 3.1 *Assume all well-typed expressions have principal types. Then all well-formed modules have principal signatures.*

The statement of principality deliberately abstracts from the details of core language types. For the simple core language used here,

$$\frac{\Gamma + \{\mathbf{type}\, t\} \vdash M : S}{\Gamma \vdash (\mathbf{type}\, t_{\mathtt{t}};\, M) : (\mathbf{type}\, t_{\mathtt{t}};\, S)} \quad \text{(R-C-T)}$$

$$\frac{\mathtt{NEW}(t, \Gamma) \quad \Gamma + \{t\, \mathbf{is}\, \phi\} \vdash M : S}{\Gamma \vdash (t\, \mathbf{is}\, \phi;\, M) : (t\, \mathbf{is}\, \phi;\, S)} \quad \text{(R-C-I)}$$

$$\frac{\mathtt{NEW}(t, \Gamma) \quad \Gamma + \{t = \tau\} \vdash M : S}{\Gamma \vdash (t = \tau;\, M) : (t = \tau;\, S)} \quad \text{(R-C-E)}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + \{\mathbf{val}\, x : \tau\} \vdash M : S}{\Gamma \vdash (\mathbf{val}\, x_{\mathbf{x}} = E;\, M) : (\mathbf{val}\, x_{\mathbf{x}} : \tau\, ;\, S)} \quad \text{(R-C-V)}$$

$$\frac{\begin{array}{c}\Gamma' = \{\mathbf{val}\, f^1 : \tau_1, \ldots, \mathbf{val}\, f^n : \tau_n\} \\ \Gamma + \Gamma' \vdash E_i : \tau_i,\ i = 1 \ldots n \quad \Gamma + \Gamma' \vdash M : S\end{array}}{\begin{array}{c}\Gamma \vdash (\mathbf{fun}\, f^1_{\mathtt{f}_1} = E_1 \| \ldots \| f^n_{\mathtt{f}_n} = E_n\, ;\, M) \\ : (\mathbf{val} f^1_{\mathtt{f}_1} : \tau_1;\, \ldots;\, \mathbf{val} f^n_{\mathtt{f}_n} : \tau_n;\, S)\end{array}} \quad \text{(R-C-R)}$$

Figure 10: Core Language Typing Rules

some form of type polymorphism must be added to the core language to admit principality, with the subtype relation $<:$ extended to an instance relation over polymorphic types. Various other enrichments of the subtype relation with core language subtyping are possible [25, 12].

4 Semantics of Mixin Modules

In this section, we provide the operational semantics for our minilanguage in terms of a rewrite rule system. The rules for normalizing mixin expressions are given in Fig. 11. Rule COMP normalizes a composition of two mixins. This forms the sequential composition of the respective preludes and initialization parts, and the parallel composition of the mixin bodies. This latter composition is defined by the COMP-T, COMP-I and COMP-V rules. If a field is present in one mixin body and missing in the other, then a "default" definition is inserted during composition (either an empty datatype definition or a function definition `fn x ⇒ inner x`). We refer to this as the *inflation rule* for mixin composition; we omit the obvious rules which perform this inflation.

Rule CLOS closes up a mixin module to an ordinary structure. The value definitions in the body of the mixin are composed into a collection of mutually recursively defined functions. The CLOS rule must also "close up" occurrences of **inner** in these value definitions. We use the special constant $\perp$ to close up these definitions; evaluation fails at run-time if $\perp$ is ever evaluated. This is analogous to the situation in ML where pattern-matching may fail because the clauses in a function definition do not cover all possible cases.

The reduction semantics is based on the definition of evaluation contexts given in Fig. 13, with the rules:

$$\begin{array}{rcl} ME_t[M] \longrightarrow ME_t[M'] & \text{if} & M \longrightarrow M' \\ ME_t[M_t] \longrightarrow ME_t[M'_t] & \text{if} & M_t \longrightarrow M'_t \\ ME[E] \longrightarrow ME[E'] & \text{if} & E \longrightarrow E' \\ ME[M] \longrightarrow ME[M'] & \text{if} & M \longrightarrow M' \end{array}$$

and so on for evaluation contexts for expressions (omitted). These rules prevent evaluation within a functor body or a mixin, and ensure that evaluation of a structure body proceeds from left to right. The rules for ordinary expressions are essentially similar and familiar [32], and are omitted. The reduction semantics also needs to accomodate structure projection and functor application. Rules PROJ-

$$(M_p^1 \varsigma(M_b^1) M_i^1) \otimes (M_p^2 \varsigma(M_b^2) M_i^2) \longrightarrow (M_p^2;M_p^1)\varsigma(M_b^1 \otimes M_b^2)(M_i^2;M_i^1) \qquad (\textsc{comp})$$

$$(\textbf{type}\ t_{\mathbf{t}};M_b^1) \otimes (\textbf{type}\ t_{\mathbf{t}};M_b^2) \longrightarrow \textbf{type}\ t_{\mathbf{t}};M_b^1 \otimes M_b^2 \qquad (\textsc{comp-t})$$

$$(t\ \textbf{is}\ \phi_1;M_b^1) \otimes (t\ \textbf{is}\ \phi_2;M_b^2) \longrightarrow t\ \textbf{is}\ \phi_1 \cup \phi_2;M_b^1 \otimes M_b^2 \qquad (\textsc{comp-i})$$

$$(\textbf{val}\ x_{\mathbf{x}} = E_1;M_b^1) \otimes (\textbf{val}\ x_{\mathbf{x}} = E_2;M_b^2) \longrightarrow (\textbf{val}\ x_{\mathbf{x}} = \{E_2/\textbf{inner}\}E_1);M_b^1 \otimes M_b^2 \qquad (\textsc{comp-v})$$

$$\frac{\begin{array}{c} M_b = (\textbf{type}\ \overline{t_{\mathbf{t}}};\ \bar{t}\ \textbf{is}\ \overline{\phi};\ \textbf{val}\ f^1_{\mathbf{f}_1} = E_1;\ \ldots\ \textbf{val}\ f^n_{\mathbf{f}_n} = E_n) \\ M = (\textbf{type}\ \overline{t_{\mathbf{t}}};\ \bar{t}\ \textbf{is}\ \overline{\phi};\ \textbf{fun}\ f^1_{\mathbf{f}_1} = \{\lambda x.\bot/\textbf{inner}\}E_1 \parallel \ldots \parallel f^n_{\mathbf{f}_n} = \{\lambda x.\bot/\textbf{inner}\}E_n) \end{array}}{\textbf{clos}(M_p \varsigma(M_b) M_i) \longrightarrow \textbf{struct}\ M_p;M;M_i\ \textbf{end}} \qquad (\textsc{clos})$$

$$EE[\bot] \longrightarrow \bot \qquad (\textsc{bot})$$

$$(\lambda x.E_1)\,E_2 \longrightarrow \{E_2/x\}E_1 \qquad (\textsc{beta})$$

$$\textbf{let}\ \varepsilon\ \textbf{in}\ E \longrightarrow E \qquad (\textsc{let-1})$$

$$\textbf{let}\ \textbf{val}\ x_{\mathbf{x}} = E_1;\ D\ \textbf{in}\ E_2 \longrightarrow \{E_1/x\}(\textbf{let}\ D\ \textbf{in}\ E_2) \qquad (\textsc{let-2})$$

$$\textbf{let}\ D';\ D''\ \textbf{in}\ E \longrightarrow \textbf{let}\ \theta(D'')\ \textbf{in}\ \theta(E) \qquad (\textsc{let-3})$$

$$\text{where} \left\{ \begin{array}{rcl} D' & = & \textbf{fun}\ f^1_{\mathbf{f}_1} = E_1 \parallel \ldots \parallel f^n_{\mathbf{f}_n} = E_n \\ \theta & = & \{(\lambda x^i.\textbf{let}\ D'\ \textbf{in}\ E_i)/f^i \mid i = 1,\ldots,n\} \end{array} \right.$$

$$(\textbf{case}\ p.\textbf{c}(\overline{E_n})\ \textbf{of}\ \ldots\ \text{‘}|\text{’}\ p.\textbf{c}(\overline{x_n}) \Rightarrow E\ \text{‘}|\text{’}\ \ldots) \longrightarrow \{\overline{E_n}/\overline{x_n}\}E \qquad (\textsc{case-1})$$

$$(\textbf{case}\ E\ \textbf{of}\ \ldots\ \text{‘}|\text{’}\ x \Rightarrow E'\ \text{‘}|\text{’}\ \ldots) \longrightarrow \{E/x\}E' \qquad (\textsc{case-2})$$

$$MV;\ \textbf{structure}\ s_{\mathbf{s}} = MV_t;ME[s.p] \longrightarrow MV;\ \textbf{structure}\ s_{\mathbf{s}} = MV_t;ME[MV_t(p,\varepsilon)] \qquad (\textsc{proj-v})$$

$$MV;\ \textbf{structure}\ s_{\mathbf{s}} = MV_t;ME[s.p'(p)] \longrightarrow MV;\ \textbf{structure}\ s_{\mathbf{s}} = MV_t;ME[(MV_t(p',\varepsilon))(p)] \qquad (\textsc{proj-f})$$

Figure 11: Computation Rules

V and PROJ-F denote the operations of projecting a value and a functor, respectively, out of the global context of bound structures. Note that a structure is never copied by projection, because of the fact that type generativity is based on the syntactic identity of paths. These rules use the operation of applying a module to a path, defined in Fig. 12.

Module application $M_t(p,p')$, where M_t is applied to p, is written using an accumulating parameter p' which records the path so far followed from the top-level environment to reach the module M_t. The first case in the definition in Fig. 12 corresponds to projecting a module out of a structure; however this projection is only defined if the operation terminates in one of the following cases. As a module is projected out of a structure, the definitions it imports from the bindings to the left of it in the structure are prefixed by the path to the structure from the top-level environment. This is completely analogous to the R-MOD-SP rule in Figure 7.

The second case in the definition in Figure 12 corresponds to the case where a path is being projected out of a structure. This is allowed since no type identities are lost by this projection. The third case corresponds to a value being projected out of a structure, while the fourth and fifth cases correspond to a functor and a mixin being projected, respectively. Finally the last case corresponds to a functor application being β-reduced.

The formulation of subject reduction is complicated by abstraction in the type system. Consider:

```
structure s : sig type t val x:t end =
    struct datatype t=foo val x=foo end
```

With this declaration, `s.x` has type `s.t`. `s.x` reduces to the constructor `s.foo`, but the datatype declaring `s.foo` is hidden behind the opaque type `s.t`. We refer to `s.foo` as a *hidden value*, since it cannot be given a type outside of the body of `s`. Nevertheless, in reasoning about subject reduction for this example, we need to expose the type information in the definition of `t`.

To reason about subject reduction, we use a reformulation of the type system presented in the previous section, with the following

$$
\begin{array}{rcl}
(\textbf{struct}\, M^1; \textbf{structure}\, s_{\mathrm{s}} = MV_t\ ; M^2\ \textbf{end})(\mathrm{s}.p', p) & = & \theta(MV_t)(p', p.\mathrm{s}) \\
\text{where} & & \theta = \{p.\mathrm{n}/n \mid n_{\mathrm{n}} \in \mathtt{BV}(M^1)\} \cup \{p.\mathrm{c}/\mathrm{c} \mid \mathrm{c} \in \mathtt{cons}(M^1)\} \\
\text{and} & & \mathrm{s} \notin \mathtt{EN}(\mathtt{BV}(M^2)) \\
 & & \\
p\,(p', p'') & = & p.p' \\
 & & \\
(\textbf{struct}\, M^1; \textbf{val}\, x_{\mathrm{x}} = EV\ ; M^2\ \textbf{end})(\mathrm{x}, p) & = & \theta(EV) \\
\text{where} & & \theta = \{p.\mathrm{n}/n \mid n_{\mathrm{n}} \in \mathtt{BV}(M^1)\} \cup \{p.\mathrm{c}/\mathrm{c} \mid \mathrm{c} \in \mathtt{cons}(M^1)\} \\
\text{and} & & \mathrm{x} \notin \mathtt{EN}(\mathtt{BV}(M^2)) \\
 & & \\
(\textbf{functor}\ (\textbf{structure}\, s_{\mathrm{s}} : S_t)M_t)(\varepsilon, p) & = & \textbf{functor}\ (\textbf{structure}\, s_{\mathrm{s}} : S_t)M_t \\
 & & \\
(M_p\, \varsigma(M_b)M_t)(\varepsilon, p) & = & M_p\, \varsigma(M_b)M_t \\
 & & \\
(\textbf{functor}\ (\textbf{structure}\, s_{\mathrm{s}} : S_t)M_t)(p) & = & \{p/s\}M_t
\end{array}
$$

Figure 12: Applying Modules to Paths

$$
\begin{array}{lcl}
ME_t & ::= & [\,] \mid \textbf{struct}\, ME\ \textbf{end} \\
 & & \mid ME_t \otimes M_t \mid MV_t \otimes ME_t \mid \textbf{clos}(ME_t) \\
 & & \\
ME & ::= & [\,] \mid MV;\ \textbf{val}\, x_{\mathrm{x}} = EE; M \\
 & & \mid MV;\ \textbf{structure}\, s_{\mathrm{s}} = ME_t; M \\
 & & \\
MV_t & ::= & \textbf{functor}(S_t)M_t \mid M\varsigma(M_b)M \mid \textbf{struct}\, MV\ \textbf{end} \\
 & & \\
MV & ::= & \textbf{structure}\, s_{\mathrm{s}} = MV_t \\
 & & \mid \textbf{type}\, t_{\mathrm{t}} \mid t = \tau \mid t\ \textbf{is}\ \phi \\
 & & \mid \textbf{val}\, x_{\mathrm{x}} = EV \\
 & & \mid \textbf{fun}\, f^1_{\mathrm{f}_1} = EV_1 \| \ldots \| f^n_{\mathrm{f}_n} = EV_n \\
 & & \mid MV; MV \\
 & & \\
EE & ::= & \ldots\text{omitted} \\
EV & ::= & \bot \mid \ldots\text{omitted}
\end{array}
$$

Figure 13: Evaluation Context

changes:

1. We remove the construct for allowing explicit type declarations for modules, $(M_t : S_t)$, and we omit the corresponding signature matching rule Rule R-MOD-MAT in Figure 7.
2. We fold the type subsumption rule, Rule R-MOD-SUB, into the rule for functor application. Rule R-MOD-PA in Figure 7. With the omission of explicit signature matching, this is the only place where subsumption is required (even with the addition of mixin modules).

To simplify the exposition of type soundness, we make the simplifying assumption that functors are first-order, as in Standard ML. It is then trivial to define a syntactic transformation which removes explicit signature matching from programs. This transformation is defined as the homomorphic extension of the following:

$$[\![(M_t : S_t)]\!] \quad = \quad [\![M_t]\!]$$

Let $\Gamma \vdash_M M : S$ denote the derivability of type judgements in this restricted (or *minimal*) type system. Then we have:

Lemma 4.1 *If $\Gamma \vdash M : S$ then $\Gamma \vdash_M [\![M]\!] : S'$ for some S' such that $\Gamma \vdash S' <: S$.*

Theorem 1 (Subject Reduction) *Given Γ and M not containing signature matching. Given $\Gamma \vdash_M M : S$ and $M \longrightarrow M'$. Then $\Gamma \vdash_M M' : S'$ for some S' such that $\Gamma \vdash S' <: S$.*

Fig. 13 also defines value expressions. A *faulty term* is an expression containing an irreducible subexpression which is not a value.

Lemma 4.2 *Given Γ and M not containing signature matching. Given $\Gamma \vdash_M M : S$, then M contains no faulty terms.*

Finally soundness follows from the fact that any reduction sequence in the original type system can be simulated by a corresponding reduction sequence in the minimal type system:

Theorem 2 (Soundness) *Given Γ and M not containing signature matching. If $\Gamma \vdash M : S$ then evaluation of M does not go wrong, i.e.,there does not exist a reduction sequence*

$$M \longrightarrow \cdots \longrightarrow M_i \longrightarrow \cdots$$

of reductions starting from M such that some M_i contains a faulty term.

5 Related Work and Conclusions

Mixin modules are clearly influenced by work in implementation inheritance, and in particular mixin-based inheritance, in the object-oriented languages community. As already discussed, mixin modules bear some relationship to CLOS generic functions, but with data constructor tags replacing the type tags that are used for dispatching in CLOS. The `inner` construct bears some relationship to `call-next-method` in CLOS, which allows a method in a class to invoke the instance of that method in the next class in the inheritance chain. Mixin modules are also related to BETA patterns [21], and indeed our `inner` construct is deliberately named to suggest the analogy with the BETA construct. Like BETA, implementation inheritance with mixin modules is based on incremental extensions only, and does not allow the overriding of existing definitions (Smalltalk-style method overrides). Bracha and Cook [3] propose "mixin classes" as a construct for object-oriented languages. Bracha and Lindstrom [4] extend the work of Bracha and Cook by proposing "modules" (object generators) as a more general notion than classes, from which classes and mixins may be derived using their suite of inheritance operations. Mitchell, Meldal and Madhav [22] consider the addition of object-oriented constructs to the ML module language. Their approach amounts to adding F-bounded quantification and implementation inheritance [7, 23] to the ML module language, and implementing objects as modules rather than as closures. All of this work is based on providing some notion of implementation inheritance for object-oriented languages.

We are the first to suggest a form of implementation inheritance for a functional language, based on adding inheritance mechanisms to the module system. Burstall [5] proposed a functional language NPL (a predecessor to HOPE [6]) with extensible datatypes and function definitions, in which constructors could be incrementally added to the definition of a datatype, and clauses could be incrementally added to the definitions of functions that operated on that datatype. However unlike our approach he did not base his extensions on the module system, and his approach to providing extensions is less general. Beyond the fact that our approach solves some open problems with the SML module system, our commitment to providing inheritance in the module system has important benefits for the implementation of mixins. One of the reasons for the increasing acceptance of functional languages has been the very efficient implementation of pattern-matching [30]. This efficiency in turn relies on the fact that all of the constructors for a datatype are known when compiling the clauses of a function definition. This is manifested in the SML/NJ compiler, for example, where the implementation of pattern-matching in exception handlers is somewhat less efficient than that for the ordinary `case` construct. By postponing optimization and code generation for mixin modules until they are closed up to ordinary ML structures, we may similarly benefit from efficient compilation while providing extensible datatypes. We are currently investigating the design of a suitable intermediate form into which to compile mixin modules, in order to support this implementation strategy.

The other implementation issue with mixin modules is the inferring of properties of extensible datatypes defined in mixins. Given a datatype definition in one mixin, how are we to infer whether it admits equality, since another mixin may declare a constructor for that datatype with a functional domain? Similar issues arise with other datatype attributes, for example, variance properties of datatypes if we introduce subtyping [8]. One possible approach here is to require that these attributes be declared by the programmer. Analogous issues arise with datatype properties that are only used internally in the compiler, in the representation analysis of datatypes. It appears plausible that work on cross-module optimization may be applicable in this situation.

In a large sense we have only told half of the story of mixin modules. We have only reported on *vertical extensions*, adding cases to the definitions of datatypes and functions. However there is also a notion of *horizontal extensions* for mixin modules, which also allows the domain of a data constructor to be extended during mixin composition [28]. This provides us with the form of subclassing provided in CLOS that is missing from this account. Using this mechanism, for example, we can define modular interpreter building blocks in the style of Liang et al [20]. Furthermore mixin modules can be extended to provide a class construct for ML extended with objects [8]. In contrast to simply adding classes to ML, this approach provides implementation inheritance for all of ML, not just some object-oriented fragment of it. We intend to report on this in a subsequent paper.

Acknowledgements: Thanks to Bob Harper and David MacQueen for stimulating discussions.

References

[1] Andrew Appel and David MacQueen. Standard ML of New Jersey. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1991.

[2] Andrew Appel and David MacQueen. Separate compilation for Standard ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM Press, June 1994. SIGPLAN Notices volume 30, number 6.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, pages 303–311. ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10.

[4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages*, pages 282–290. IEEE, 1992.

[5] R. M. Burstall. Design considerations for a functional programming language. In *Infotech State of the Art Conference*, Copenhagen, Denmark, 1977. Infotech.

[6] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. HOPE: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, 1980.

[7] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[8] Dominic Duggan. Polymorphic methods with self types for ML-like languages. Technical Report CS-95-03, University of Waterloo, Department of Computer Science, 1995.

[9] Dominic Duggan and Frederick Bent. Explaining type inference. Technical Report CS-94-14, University of Waterloo, Waterloo, Ontario, Canada, 1994. To appear in *Science of Computer Programming*.

[10] Dominic Duggan and John Ophel. Kinded parametric overloading. Technical Report CS-94-35, University of Waterloo, Department of Computer Science, September 1994.

[11] Dominic Duggan and John Ophel. On type-checking multi-parameter type classes. To be submitted, 1995.

[12] Jonathan Eifrig, Scott Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*. ACM Press, October 1995.

[13] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.

[14] Paul Hudak, Simon Peyton-Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict purely functional language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[15] Mark Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*, pages 1–10. Springer-Verlag, 1993. Lecture Notes in Computer Science 594.

[16] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *European Symposium on Programming*, pages 131–144. Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[17] S. C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.

[18] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994. acmp.

[19] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–163, San Francisco, California, January 1995. ACM Press.

[20] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, January 1995. ACM Press.

[21] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[22] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML with subtyping and inheritance. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 270–278. ACM Press, January 1991.

[23] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–124, 1990.

[24] Greg Nelson. *Systems Programming in Modula-3*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, 1991.

[25] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 54–67, St. Petersburg Beach, Florida, January 1996. ACM Press.

[26] John Ophel. A polymorphic language with first-class modules. *Australian Computer Science Communications*, 17(1):422–430, February 1995.

[27] Didier Rémy. Programming objects with ML-ART: An extension to ml with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[28] Constantinos Sourelis. Mixin modules. Master's thesis, University of Waterloo, 1995.

[29] Guy Steele. Building interpreters by composing monads. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 472–492. ACM Press, 1994.

[30] Philip Wadler. Efficient compilation of pattern-matching. pages 78–103. Prentice-Hall, 1987. Chapter contributed to *The Implementation of Functional Programming Languages*, Simon Peyton-Jones.

[31] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[32] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, April 1991.