True Higher-Order Modules, Separate Compilation, and Signature Calculi*

George Kuan

January 15, 2009

Abstract

In the past three decades, the ML module system has been the focal point of tremendous interest in the research community. The combination of parameterized modules and fine-grain data abstraction control have proven to be quite powerful in practice. Mainstream languages have slowly adopted features inspired by the ML module system. However, programmers have run into various limitations and complexities in implementations of the ML module system. In the presence of common extensions such as true higher-order modules, true separate compilation becomes a problem. This conflict reflects a fundamental tension in module system design. Module systems should both propagate as much type information across module boundaries as is unconstrained by the programmer and be able to separately typecheck modules.

1 Background

Modularity in programming dates back at least to Parnas and his information hiding criteria [44]. Linguistic support for program modularity took many forms. There are two main lines in the development of module systems. First, the Modula [61] and Cedar/Mesa [21] line emphasized approach based on *ad hoc* and extralinguistic schemes for modeling linking of program modules,*i.e.*, such languages used separate tool, an object file format sensitive linkloader, to compose separately compiled modules. Second, the ML line of modularity [35–37] described modularity in terms of a small functional programming language where application is the main form of linkage. Common to both of these approaches is the idea that the separation of interface and implementation leads to better program structure.

The most basic form of modularity is namespace management. Simple namespace management only requires a means to declare namespaces at the top-level of program structure (namespace N), functions that reside inside those namespaces, and a notation for projecting those functions (N.f). Each namespace is a module in such a system. One can neither manipulate nor reference a namespace outside of the projection notation. A programmer can

*Draft v.4

call functions by projecting out the desired components from these rudimentary modules. Namespace management, however, is only a convenience for the programmer, providing no additional expressive flexibility or power in the language. In particular, namespace management by itself does not support separate compilation, which is crucial for independent development of components of a large software project and efficient compilation. In fact, without separate compilation, some programs would have been impossible to compile with the limited resources on early machines. Even on modern machines, large software compilation can be quite taxing. Many languages did not gain even a rudimentary module system until the late 1980s and early 1990s. For example, the venerable FORTRAN did not include a rudimentary module system until FORTRAN 90.

A record of functions and variables can be used as a slightly richer form of modularity. In functional languages, regular records may also contain functions as fields. Records aggregate labeled fields that can be projected. Unlike namespaces, records are typically first-class, so functions on records enable a degree of generic programming especially in conjunction with row polymorphism [47]. Unfortunately, records normally do not support data abstraction. Pierce's model of object-oriented languages leverages records as its core construct [45].

Modula-2 pioneered many of the major ideas behind module systems. It supported hierarchical composition (*i.e.*, nesting), type components, and an exception handling mechanism [61]. The exception handling mechanism consisted of exception declarations as components of modules and a module-level exception handler. Compositions of modules were limited to explicit imports and exports of definite modules (as in a specific concrete module with all its components filled out) and their components.

Ada is an example of an imperative and object-oriented language that supports modularity in the form of basic and generic packages [6]. Packages are specified in two parts, specifications (akin to interfaces or signatures) and bodies that define the implementation of the package. Generic packages map basic packages to basic packages, thus providing a form of parameterization. Basic packages can be nested.

Object-oriented languages use the class or object system to provide a degree of modularity for functions and data. Most object-oriented languages do not support type components, thus limiting their expressiveness. Scala's object system [42] appears to be the only OO language that supports type components as members of a class. Beyond simple namespace management, OO languages support information hiding via private data members. Unfortunately, this is a rather coarse-grain encoding even with multiple levels of privacy, visibility to an external client is all-or-nothing. Scala's structured types offer another approach to modularity. There has also been a considerable amount of interest in Bracha's mixin approach to modularity [4, 5, 16] and traits [18, 51].

Type classes [59] can be modeled using a very stylized use of modules coupled with a dictionary-passing inference [14]. Where type classes apply, this inference scheme makes code significantly more succinct. However, as the set of instances of classes at any one point in the program is global, type classes sometimes interfere with modular programming.

2 The ML Module System

The ML module system consists of structures, signatures, functors (parameterized modules), and functor signatures. Structures are sequences of type, value, structure, and functor bindings. In Standard ML, structures can be named or anonymous. Signatures are sets of specifications for types, the type of values, structure component signatures, and functor component signatures. Similar to structures, signatures can be either named and anonymous. Anonymous signatures are either written inline (*i.e.*, **sig** ... **end**) or inferred or extracted from a structure. There is a many-to-many relationship between structures and signatures. Ascribing the signature to the structure verifies that that structure satisfies the signature. Alternatively, one can use a signature SIG to seal a structure M which makes types in M abstract or transparent as required by SIG and hides omitted components. Type specifications in signatures can be transparent (**type** t = unit), abstract types (**type** u), and relatively defined in terms of abstract types (**type** v = u $*$ int), usually considered a kind of transparent specification. The possibility that a signature contains both transparent and abstract specification is called translucency.

The distinguishing characteristics of the ML module system are functors and the interaction of the module system and core language type inference. Ada supports type components in modules (called packages) and "generic" modules parameterized on a single restricted type and some associated operator definitions. The module system of ML can express much of System F. Indeed, the SML/NJ compiler compiles source into a System F-like core language. Although the Definition [40] does not require nor define support for higher-order modules, they can be found in many implementations of Standard ML including SML/NJ, Moscow ML, and MLton. Apart from implementations of ML, higher-order module systems are not found elsewhere. In the ML module system, functors can be typechecked independently from their uses (*i.e.*, applications) unlike related programming constructs such as C++ templates. Separate typechecking is necessary for true independent development of large software systems. It also makes it possible to gain confidence in the type safety of the functor before even beginning to write functor arguments and applications of functors, thus revealing type errors earlier in the development process when they can be resolved more easily. The ML module system also supports type abstraction via opaque ascription. With this feature, programmers can define abstract data types and data structures with their corresponding suite of operations.

Another important property of ML module systems is the phase distinction [25]. If a language respects phase distinction, then the static and dynamic parts of the program can be separated such that that former does not depend on the latter. Because the ML module system can be compiled into System F_ω, it naturally respects the phase distinction. At first glance, it may appear that types and values are entangled in type definitions such as **type** t = M.u where M is a module contains value as well as type components. Because type components always only refer to other type components in modules, one can split any module into one that only contains type components and the other only value components. The proviso to this argument is that modules cannot be first-class. Because in ML, the module and core languages are stratified, this proviso holds.

2.1 Principles for evaluating module system design

Module system designs are often evaluated subjectively and qualitatively. However, one can define our design goals around more objective criteria. One goal of the module system can be to achieve as much raw performance as is possible by admitting as many different kinds of optimizations as is practical. Although it is true that performance attributable module system design is difficult to isolate, If I consider type-based optimizations, this goal would mean that I would like to have types propagate across module boundaries after compilation as far as is possible. True higher-order functors propagate types across transparent higher-order functor applications. The performance benefit of such type propagation across module boundaries can be measured at least in terms of synthetic benchmarks.

We can consider the notational convenience of the module system design. Minimizing the amount of syntactic overhead would help programmers in rapid prototyping and software maintenance. In some sense, the very principle of module systems runs against the spirit of rapid prototyping. The popular misconception is that requiring the programmer to confront interfaces and types stifles prototyping. Type inference mitigates this concern for the core language somewhat. A new module system design ought to optimize and balance the syntax for brevity and readability. Syntactic overhead is always tricky to measure accurately. Raw lines of code for a series of realistic programs give a very rough and potentially misleading idea of overhead. Verbosity may improve readability and maintainability. Moreover, if automated tools such as IDEs and preprocessors can produce with little or no programmer intervention some of the syntax, then the actual significant overhead may be minimal. Alternatively, I may derive syntactic overhead by comparing the best practice encodings of very common programming patterns to gain a relative measure of notational convenience. Again, one falls into the trap of leaving open the possibility of only solving very contrived problems.

The software engineering discipline has devised a number of additional metrics for measuring program source complexity including McCabe's cyclomatic complexity (MCC), coupling, cohesion, and Martin's software package metrics. MCC measures complexity of a program's control flow graph. Of these, coupling, cohesion, and software package metrics were devised expressly to measure complexity in the presence of a some modularity mechanism, though especially in the case of software package metrics, the measures tend to be designed for class-based object-oriented languages. More fundamentally, most of these metrics were intended to measure code quality for software projects and not the complexity of programming languages in which they are written.

A related design criterion is the predictability and transparency of module system design. Because compilers are becoming more sophisticated and complex, it is sometimes difficult to figure out what a compiler would generate for a given source. Although much of the complexity resides in the compiler optimization phases, it is important that elaboration and compilation for the module system are not overly sensitive, producing wildly different code for otherwise roughly semantically equivalent source. A trained programmer should be able to easily understand the source of errors and performance bottlenecks. Although this criterion is somewhat qualitative, one can perhaps measure the complexity of elaboration and compilation by quantifying how much information must be reconstructed in order to

predict the general outcome of elaboration or compilation. If one considers programmer understanding of elaboration as an abstract interpretation of program source, our metric of predictability might be analogous to how precise such an abstraction interpretation will have to be to identify the source of elaboration errors.

One goal of module systems is to support extensible software design. Thus the module system must promote ease of extensibility in multiple dimensions. The Expression Problem is a particularly well-known instance of this issue [60]. While a good module system design might not necessarily produce a solution to the expression problem, it will provide a best practice that permits programmers to extend software without touching parallel cases. Again, this criterion can be loosely quantified as the number of expressions or statements that must be changed to extend source in a certain direction. A closely related objective is to permit programmers to safely compose program modules together in a flexible way.

2.1.1 Garcia *et al.*

Garcia *et al.* [20] evaluated how well 8 major programming languages support the implementing type-safe polymorphic containers by generic programming techniques. Their chief criticisms of the ML module system are three-fold. They argue that the signature language should permit semantic compositions of signatures beyond the syntactic include inheritance mechanism similar to Ramsey's signature language design [46]. The authors also remark that the module system would be impractical for programming-in-the-small because the syntactic overhead is excessive even when compared to other languages that require explicit instantiation, another criticism. The lack of "functions with constrained genericity aside from functions nested within functors" was also cited as a disadvantage. In other words, Garcia *et al.* suggests that some bounded polymorphism would be desirable.

3 Principal Research Problems

This dissertation will address three main subjects: the formalization of the true higher-order module system in SML/NJ, a formal investigation of the relationship between true higher-order modules and true separate compilation, and the development of a rich foundational signature calculus that solves some of the shortcomings of the module system without radically altering it. The implementation of the higher-order module system has evolved since MacQueen and Tofte's paper [38]. The proposed dissertation will also formalize, simplify, extend, and improve the static semantics implicit in the SML/NJ elaborator. I will focus on changes in module representations and improve elaboration algorithms such as the instantiation algorithm for solving type sharing constraints. This dissertation will study what this intuition means precisely and formally by giving a modern formal account of true higher-order modules, possible signature languages, the incompleteness of signature languages, and the relationship to separate compilation. These results will open the way to exploration of the design space of **full transparency** (i.e., exactly what can and should propagate through higher-order functor applications) and separate compilation.

```
module type T = sig type t end
module Id = functor(X:T) -> X

module type FS = functor (X:T) -> T

module Apply =
   functor(X: sig
                module F : FS
                module M:T
              end) ->
         X.F(X.M)

module M0 = struct type t = unit end
module M1 = Apply(struct
                    module F=Id
                    module M=M0
                  end)

let x : M1.t = ();;
```

Figure 1: Apply functor example in OCaml

3.1 Higher-order module system

Higher-order modules extend the ML module system with higher-order functors, a natural extension. There are many different applications and three general approaches for higher-order modules [2, 11, 31, 32, 34, 38, 48, 56, 57] which differ mainly on how they handle type sharing in the apply functor (fig. 1).

In the apply functor example, the higher-order functor apply simply applies its first argument, functor F, to its second, a module M. The main problem of interest is how one types the apply functor and the application of the apply functor on line 14. Tofte [56] introduced the first cut at the semantics for higher-order modules with **non-propagating** functors. Under Tofte's semantics, the apply functor does not propagate the type M0.t through the functor application X.F(X.M), thus M1.t $\neq$ M0.t = unit. It assigns the signature in fig. 2. to the apply functor. Notice that the signature for the body does not say anything more about **type** t.

MacQueen-Tofte [38] argues that the type sharing M1.t = M0.t = unit should hold. The strong sums model of modules also predicts this behavior of full type propagation [37]. The semantics in MacQueen-Tofte, **true higher-order functors** or **fully transparent generative functors**, re-elaborates the functor body of apply given the contents of X.M at the point of the functor application on line 14. Although this re-elaboration has the desired effect of propagating types, MacQueen-Tofte assigns exactly the same functor signature as the Tofte semantics. Leroy [31] offered an alternative approach, **applicative functors**, that

```
functor Apply(X: sig
                   functor F : (X: T) : T
                   structure M : T
                 end) :
               sig type t end
```

Figure 2: Signature assigned to Apply functor under Tofte and MacQueen-Tofte semantics

```
functor Apply(X : sig
                    functor F : FS
                    structure M : T
                  end) :
          sig type t = X.F(X.M).t end
```

Figure 3: Signature assigned to Apply functor under applicative functor semantics

enriched the notion of type paths in functor signatures with functor applications such as F(M).t. Applicative functor semantics assigns the functor signature in fig. 3.

However, applicative functors only solve the type propagation problem under certain circumstances and loses the generative semantics of functors, *i.e.*, functor applications do not generate fresh abstract types to enforce abstraction. To address this shortcoming, Moscow ML and Dreyer's module system [13] combined applicative and non-propagating generative higher-order functors in the same language. Fully transparent generative functors both solve the type propagation problem under all circumstances and do not have to compromise on generative functor semantics. In the last decade, researchers have gained significant experience in engineering non-fully transparent generative functors and transparent applicative functors in compilers such as Moscow ML and OCaml. Although they differ internally, both SML/NJ and MLton compilers support some variant of true higher-order functors semantics. I take OCaml and SML/NJ as representatives of the former and latter groups respectively.

Under both OCaml and SML/NJ compilers, the apply functor example typechecks. Furthermore, although applicative functors cannot directly handle applications to nonpaths, lambda lifting the offending nonpath generally solves that issue. Applicative functor semantics lifts this restriction in the case where $m_1(m_2)$ such that m_1 does not have a dependent type,*i.e.*, the formal parameter does not occur free in the signature of m_2. Together with signature subtyping, such a technique can remove the restriction on nonpaths for dependently typed functors in certain cases but only at the cost of principality. However, if I complicate the apply functor slightly by applying a formal functor F to **struct type** t = int **end** as in fig. 4, then applicative functors fail to propagate enough type information. When the typechecker gets to line 3 in fig. 4, it does not have enough information about F to give a stronger signature for ApplyToInt's functor body. Neither is there a path to **struct type** t = int **end** to construct an applicative functor path. Because the typechecker must give the functor body a signature immediately in order to give HO functor ApplyToInt a complete signature,

```
module ApplyToInt =
   functor (F : functor (X:T) -> T)
    -> F(struct type t = int end)

module R = ApplyToInt(Id)
let x : R.t = 5;;
```

Figure 4: ApplyToInt functor example

OCaml functor signature for H

```
module ApplyToInt :
  functor (F : functor (X : T) -> T)
    -> sig type t end
```

OCaml type error:

```
This expression has type unit but is here used with type R.t = ApplyToInt(Id).t
```

SML/NJ functor signature for ApplyToInt

```
functor ApplyToInt(functor F :
   (X: sig type t end)
    : sig type t end end)
: sig type t end
```

SML/NJ types x as R.t

Figure 5: A higher-order ApplyToInt functor fails to properly propagate types under Leroy's applicative functor semantics. Consequently, the last line fails to typecheck.

it can only give the weakest signature, **sig type** t **end** (fig. 5). A-normalization gets the program to typecheck but unnecessarily clutters up the code. Thus, in this sense, applicative functors cannot be said to be fully transparent. The SML/NJ compiler has no such restrictions.

The example in figs. 4 and 5 illustrates the fundamental problem with applicative functors. They work well as long as the relationship between functor parameter and body is simple, *i.e.*, can be captured in the extended notion of a path with functor application or a lambda lifted path. However, not all possible functors fit into this mold. Having to A-normalize functor applications in itself is an unnecessary shortcoming. In contrast, true higher-order functors propagate types across all transparent functor applications with no change the source. The solution that MacQueen and Tofte [38] advocates is a re-elaboration of the functor body given the actual argument module.

In instances such as the SymbolTable functor (fig. 6), applicative functors admit too much sharing as noted by Dreyer [11]. Applicative functors would permit the symbols from one SymbolTable ST1 to be used to index another ST2 despite the sealing of the functor

```
signature SYMBOL_TABLE =
sig
        type symbol
        val string2symbol : string -> symbol
        val symbol2string : symbol -> string
        ...
end
functor SymbolTable () =
struct
        type symbol = int
        val table : HashTable.t =
                (* allocate internal hash table *)
                HashTable.create (initial size, NONE)
        fun string2symbol x =
                (* lookup (or insert) x *) ...
        fun symbol2string n =
        (case HashTable.lookup (table, n) of
                  SOME x => x
                | NONE => raise (Fail "bad symbol"))
        ...
end :> SYMBOL_TABLE
structure ST1 = SymbolTable()
structure ST2 = SymbolTable()
```

Figure 6: SymbolTable functor example from Dreyer [11]

body to signature SYMBOL_TABLE. Both OCaml and SML/NJ elaborators will make the symbol type abstract, but it is only abstract with respect to external clients and not other instances of SymbolTable. Because SymbolTable is applied to the same argument for both ST1 and ST2, the two instances also share the same symbol type according to applicative functor semantics. This behavior breaks an important abstraction. Generative functors are more appropriate for enforcing the exact kind of abstraction desired. In contrast, applicative functor semantics are appropriate for some purposes such as the Set functor in fig. 7 where the type sharing of Item.item is desirable and it is acceptable to use items in Sets of the same type interchangeably. However, it is debatable whether the Set functor is a common case. I will argue that the set of programs for which true higher-order functors propagate types is exactly those one would want to propagate types. In contrast, one should not propagate types in the set Applicative functors - True HO functors. Dreyer [11] noted correctly that applicative functors and non-fully transparent generative functors are incomparable. Neither applicative nor true higher-order functors can subsume the other. However, there remains the question whether the programs that propagate types under applicative functors but not under true higher-order functors ought to have propagated the types in those cases.

The original criticisms of the MacQueen-Tofte semantics are the lack of support for true

```
signature COMPARABLE =
sig
        type item
        val compare : item * item -> order
end
functor Set (Item : COMPARABLE) =
struct
        type set = Item.item list
        val emptyset : set = []
        fun insert (x : Item.item, S : set) : set = x::S
        fun member (x : Item.item, S : set) : bool =
                ... Item.compare(x,y) ...
        ...
end
```

Figure 7: Set functor example from Dreyer [11]

separate compilation and that the stamp-based operational semantics makes it difficult to extend the module system and to reason about it. Many recent treatments of ML module systems abandons true higher-order functors completely due to these issues. The claim is that the type-theoretic presentations of the module system with applicative functors address these problems. This dissertation will consider the question whether an operational semantics account must necessarily be more complicated and if so, why. In contrast to recent work, this dissertation will take true higher-order module behavior and my revised elaboration-based semantics (figs. 18, 20, and 21) as the starting point for developing a formal semantics while addressing these criticisms and concerns. My formalism follows the implicit semantics in SML/NJ compiler which enriches the internal representation of functors and functor signatures to express the static actions of the functor, thus not having to do a full re-elaboration of the functor body. This approach will also yield some practical benefits. The SML/NJ and MLton implementations have not kept up with the pace of the progress in module system design at least partially due to the fact that most of the research has been a radical departure from true higher-order module semantics. Reframing the state-of-the-art in terms of true higher-order module semantics will bring recent developments closer a practical extension in these production-quality compilers.

3.2 Full transparency and true separate compilation

The main issue I will study in this dissertation is the the exact nature of the tension between true higher-order modules and separate compilation. Since MacQueen and Tofte introduced true higher-order modules, many researchers [15, 30, 48] have studied how to downgrade higher-order functors to regain true separate compilation, by which I mean Modula-2-style separate compilation. Although Standard ML enjoys separate typechecking and compilation for the most part, the MacQueen-Tofte re-elaboration semantics of true higher-order functors

necessitates the availability of the functor body source at the point of functor application. To workaround this limitation, SML/NJ uses cut-off incremental recompilation [1, 23] via a powerful compilation manager CM [3]. Incremental recompilation, however, does not solve the true separate compilation problem.

The separate compilation problem can be reframed as a completeness problem for the signature language, *i.e.*, can the source-level signature language adequately describe all possible modules including functors. Currently, the SML/NJ compiler elaborates module syntax into internal semantic objects. These semantic objects are expressive enough to encode the functor body relationships that eluded the source signature language. The source and these semantic objects are then compiled to a predicative System F_ω-like calculus [52]. This suggests that an F_ω-like calculus should be expressive enough to characterize all the static semantic actions of functors.

Intuitively, true higher-order modules cannot be fully expressed in the syntactic signature language because it is limited to definitional specs and type sharing. For example, there is no way to express a functor signature for the apply functor that accounts for all sharing due to full transparency. Therefore true higher-order modules cannot in general be separately compiled. In particular, HO functor applications cannot always be separately compiled from the functor definition. In the past, various researchers have approached this problem by incorporating applicative functors into the language to varying degrees [2, 13, 31, 48] sometimes limiting the generative functors in the process. In this dissertation, I will argue that applicative functors cannot replace true higher-order functors in the general case. Moreover, if fully transparent generativity is the goal, then applicative functors only serve to support true separate compilation in a limited number of cases.

Aside from describing the static semantic actions of HO functors, a signature language supporting separate compilation needs a mechanism to ensure coherence of the abstract types in imported modules. ML presently solves the coherence problem through a combination of type sharing constraints, definitional type specs, and where type clauses. In simpler module systems, the problem of coherence is not as pronounced because compilation units only import external units using definite references [54]. The current implementation for type sharing constraints resolution, called **instantiation**, is a complicated process with a considerable amount of folklore. Simply put, instantiation constructs the free instantiation of a functor formal parameter that imposes the required amount of type sharing but no more. The instantiation phase in SML/NJ imposes a semantics stricter than that of the Definition. Instantiation guarantees inhabitability of signatures where the Definition does not. Although Harper and Pierce [26] claim that type sharing constraints can be superseded by definitional type specs in all cases, my study has identified examples (fig. 8) where this is not the case absent a mechanism for signature self-reference, *e.g.*, **structure** M : S0 where **type** t = self.N.u.

Both Russo [48] and Swasey *et al.* [54] have suggested that the separate compilation problem can be solved by identifying an alternate form of compilation unit, one that is not a module. OCaml, in fact, already implements such a regime. However, in the presence of true higher-order functors, these approaches merely punt on the real problem by compelling the programmer to put otherwise independent modules together in a single compilation unit

```
signature S0 = sig datatype s = A type t end
signature S1 = sig datatype u = B type v end

signature S3 =
sig
   structure M : S0
   structure N : S1
   sharing type M.t = N.u and N.v = M.s
end
```

Figure 8: Criss-crossing type sharing constraints cannot be reduced to definitional type specifications or uses of where type.

and then abstracting over that unit. This dissertation will either prove definitely that true higher-order functors and true separate compilation are incompatible or develop a module system integrating these two features.

3.3 Signature calculus

Since the study of the true separate compilation problem points in the direction of the signature calculus, it will be fruitful to take this opportunity to reconsider the design of ML's signature language. After Harper-Lillibridge and Leroy, despite the continuing pace of the development of ML module systems, the signature language generally did not see much attention except for Ramsey *et al.*'s paper [46]. Ramsey *et al.* [46] describe a signature language that includes operations for post hoc manipulation such as adding, removing, rebinding components, and merging signatures. SML/NJ's semantics for **include** is richer than the simple syntactic inclusion found in the Definition [40]. In particular, certain kinds of compatible signatures can be merged. In the SML/NJ 110.68 compiler, two signatures are **compatible** when their overlapping specifications (*i.e.*, specifications with the same name) have the same arity and follow the rules summarized in table 1. However, the current compatibility rules are inconsistent and incomplete. For example, merging an eqtype and a type specification results in an eqtype in one direction and a type in the other as shown in fig. 9.

Despite its present incomplete state, SML/NJ can do the appropriate consistent merge for Garcia *et al.*'s example (fig. 10). In the case of Garcia *et al.*'s example, the typechecker needed to do is to note that the repeated components t and u due to inclusion are identical specifications.

The SML/NJ semantics goes further and merges consistent yet unequal specifications such as abstract types and datatypes. The merging semantics can be substantially improved by making the table more symmetrical and adjusting some of the precedences to something more sensible. Both Ramsey [46] and Dreyer and Rossberg [15] offer language support for a signature calculus that can safely compose signatures, effectively permitting a kind of multiple signature inheritance. Both accounts only model signature merging for a small language without support to features such as eqtype and generative datatypes. In particular,

```
signature S0 = sig eqtype t end
signature S1 = sig type t end
signature S2 = sig include S0 include S2 end

S2 : sig eqtype t end

signature S0 = sig type t end
signature S1 = sig eqtype t end
signature S2 = sig include S0 include S2 end

S2 : sig type t end
```

Figure 9: Unsound behavior of SML/NJ signature merging by **include**

	type	eqtype	datatype	deftype
type	✓	eqtype	✗	✗
eqtype	type	✓	✗	✗
datatype	✓	datatype	✗	✗
deftype	✗	✗	✗	✗
datatype withtype	✓	datatype withtype	✗	✗

Table 1: SML/NJ 110.68 Signature elaboration consistent signature merging: ✓ can be merged, ✗ cannot be merged, otherwise indicates specs mergable but indicated spec takes precedence

a fine-grain merging of eqtype can be nontrivial. For example, it is safe to merge **eqtype** t and **datatype** t = K where K is a data constructor. In contrast, merging **eqtype** t and **datatype** t = K **of** int -> int is unsafe. Consistent merge rules of this flavor can already be found elsewhere in the compiler, namely in signature matching. This dissertation will develop a formal semantics for a safe but flexible consistent signature merging that covers these features of ML.

The signature merging semantics found in Ramsey *et al.* is quite aggressive. In one example (fig. 11), the merging semantics creates a new definitional type spec **type** u = t in order to merge two signatures that disagree on an entangled value specification **val** x : t list and **val** x : u list . This kind of aggressiveness likely goes beyond the intention or expectation of the programmer. The programmer may have difficulty deciphering typechecking errors relating to S2.u and S2.x after this point because of this aggressive induced type sharing. It would be more sensible to have the typechecker complain that S0.x and S1.x are incompatible value specifications because as far as the typechecker and programmer are concerned, S0.t and S1.u are simply flexible type specifications.

Inspired by Ramsey *et al.*, my study of signature calculi will go beyond the semantics of consistent signature merging to consider the design implications of adding parameterized signatures [27], signature variables, and related features to the ML signature language. The ML signature language permits type definitions that may refer to general type expressions. Type

```
signature S0 =
  sig
    type t
    eqtype u
  end

signature S1 =
  sig
    include S0
    val x : int
  end

signature S2 =
  sig
    include S0
    val y : unit
  end

signature S3 =
  sig
    include S1
    include S2
  end
```

Figure 10: The naive macro expansion semantics of the Definition rejects S3. SML/NJ accepts it. This example was derived from Garcia *et al.*'s GraphSig, IncidenceGraphSig, and VertexList-GraphSig [20].

expressions may involve both primitive type constructors such as $\rightarrow$ and programmer-defined type operators. It is the inclusion of type operators that gives the signature language much of its expressiveness. The semantics of type sharing constraints differs significantly between SML90 and SML97. Type sharing constraints could be imposed on two type constructors without restriction in SML90. In SML97, the designers partitioned the semantics of type sharing into type definitions which expressed sharing between an abstract type and an arbitrary type expression, and regular type sharing constraints which can only be imposed between two flexible (or primary) types whose names must be in scope.

A module system that permits both type definitions and type sharing constraints in signatures introduces significant new complexity. For example, whereas in Leroy's [32] TypModl language, which only permits SML90-style definitional type sharing constraints and no type definitions, type sharing constraints can be "normalized" by pushing them up the signature and eliminated by turning them into type definitions, type sharing constraints cannot be eliminated in a language that permits both type definitions and type sharing constraints.

In ML modules, structures can be arranged in a hierarchy. This feature enables flexible namespace management. In contrast, signatures cannot be arranged in such a hierarchy. Signatures must be defined at the top-level and can never be enclosed in any other signature or module. For complex hierarchies such the SML/NJ's Control module that contains layers of submodules, the corresponding signature CONTROL and the signatures of the submodules PRINT and ELAB are related only incidentally by occurrence in structure specifications in CONTROL. This shortcoming in the signature language unnecessarily pollutes the signature namespace and complicates browsing through and working with highly nested hierarchies. It would be desirable to permit (transparent) signature specifications within signatures. For added flexibility and perhaps increased expressiveness, it may be useful permit signature definitions within structures and functors. Furthermore, in order for modules to match

```
signature S0 =
sig
  type t
  type u
  val x : t list
end
```

```
signature S1 =
sig
  type t
  type u
  val x : u list
end
```

```
signature S2 =
sig
  type t
  type u = t
  val x : t list
end
```

Figure 11: This is an example from Ramsey *et al.* [46]. S2 is the merge (greatest lower bound) of S0 and S1 according to their semantics.

```
structure M =
  struct
    type t = int
    type u = bool * string
    val a : u * t
  end
signature S = sign(M) removing u adding val b : t * t
```

Figure 12: Accessing and modifying (via Ramsey *et al.* signature operations) an inferred signature

these signatures enriched with signature specifications, modules must permit corresponding signature definitions.

Since the semantics of ML already supports the extraction or inference of module signatures from the implementation, perhaps it makes sense to permit the programmer to operate directly on the inferred signatures of implementations (modules) or generate/synchronize module implementation based on the signature. Programmers often skip the step of writing proper signatures for modules because the necessary notation is cumbersome and potentially repetitive with respect to the module implementations of signatures. Some programming environments can help programmers automatically generate interfaces, but changes usually are not propagated bidirectionally. Synchronization of modules and signatures is ill-defined in the ML module system which supports a many-to-many relationship between modules and signatures. However, this is exactly where the existence of a principal signature or of a full signature might be useful. If programmers can leverage the structure of existing module structure when writing signatures, this might lower the barrier of entry for programmers with

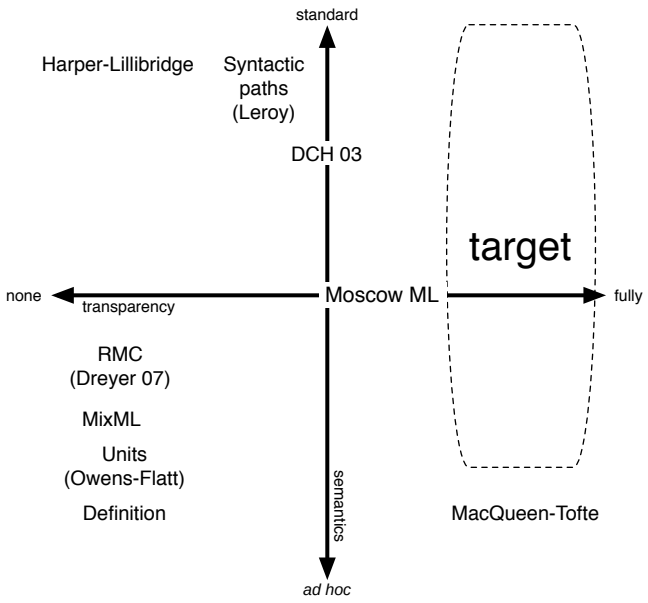
existing non-modularized source thus providing a path to "gradual modularization". For example, in fig. 12 an inferred signature can be modified after the fact to serve as a template for future structures without having to explicitly write out any signature. This signature language will enable programmers to quickly integrate modular and non-modular code by facilitating rapid construction of variations on inferred signatures. Admittedly, this feature may run against the very spirit of splitting out explicit interfaces from implementations.

4 Related work

Module systems have generally followed either in the extralinguistic style of Modula and Mesa or the functional style of ML. Dreyer and Rossberg's recent module system leans towards the extralinguistic style with the subtle mixin merge linking construct that serves many roles including encoding functor application. Cardelli [7] brought some amount of formalization to this *ad hoc* approach to modularity, but module systems in this style still vary considerably in terms of semantics. Moreover, the extralinguistic semantics of linking are typically fairly involved and delicate.

The ML module system has inspired a bevy of formalisms describing its semantics and common extensions. These formalisms run the gamut from Leroy's presentation based on syntactic mechanisms alone, to Harper-Lillibridge [24] and Dreyer, Crary, and Harper's [13] type-theoretic approach, to the elaboration semantics approach as represented by the Definition [40]. As I have discussed at length in sec. 3.1, full transparency is a desirable property for module system designs. Designs range from no support for transparency (type propagation) such as non-propagating functors and complete support for MacQueen-Tofte full transparency. In between are the various gradations,*i.e.*, different combinations of applicative and generative functors. Another important quality of the account of module system design is the formalism used for expressing the semantics. In the early days, module system semantics were quite *ad hoc*, each relying on their own peculiar notation and collection of semantic objects [21, 56, 61]. Since then, there has been a strong move toward more standardized logical frameworks that are more amenable to encoding in a theorem proving system such as Isabelle/HOL and Coq [13, 24]. Curiously, the most recent accounts have been moving back towards a semantic objects-based approach [12, 15]. Fig. 13 shows very roughly where the major families of module system designs fall in relationship to each other in terms of support for full transparency and adherence to some formal logic framework. In the figure, it appears that the left-hand side has been well-explored. I will study the right-hand side, and perhaps the upper-right-hand quadrant which represents a fully transparent semantics in something close to a mechanized metatheory for Coq. This study will ideally result in a module system design that reconciles a current understanding of fully transparent functors with a simpler and more accessible semantics, a clear evolutionary step from the state-of-the-art.

Besides the main module system features described earlier in this proposal, there are a number of other useful properties developed in the different module languages. The **phase distinction** [25] plays a fundamental role in ensuring that module systems can be type-



A semantics is more standard in the sense that it uses more formal logic frameworks. It may potentially be easier to mechanize the metatheory of such semantics.

Figure 13: The spectrum of major module system families

checked fully at compile-time. This property is desirable, being consistent with our goal of static type safety. It says that a language, in this case a module, can be split into static and dynamic parts such that the static part does not depend on the dynamic. Some accounts of module systems respect phase distinction at the surface language level [31, 48]. Others respect the phase distinction in an internal language but not the surface language [38].

The other key property in a module system design is the existence of a **principal signature** for modules. The term principal signature has been overloaded in meaning. Because of the many-to-many relationship between signatures and modules and the signature subtyping relationship, many signatures may be safely ascribed to a single structure. I will call the most precise signature for a structure (i.e., one that constrains all components of the structure with exact, most constraining types) the **full signature**. A related concept is the **free instantiation** of a functor formal parameter. The free instantiation is an instance of the functor formal parameter signature S that admits exactly enough type sharing to satisfy S and no more. In particular, it avoids any extraneous type sharing that would constrain the free instantiation more than is necessary. What Dreyer [11] calls the principal signature is the full signature in this terminology. Tofte defines principal signature of a signature expression *sigexp* as one whose flexible components (*i.e.*, abstract types) can be instantiated to obtain all instantiations of *sigexp* [40, 56]. I will adopt Tofte's terminology.

4.1 Type-theoretic approach

Beginning with Harper-Lillibridge's translucent sums module calculus [24], this large, prolific family of module systems pushed the state-of-the-art in terms of the type theoretic approach to module system design. Although Harper-Lillibridge originally explored first-class modules, the bulk of the research in this family was directed towards an applicative higher-order module semantics for type-directed compilers TIL/TILT and adding recursion.

Crary *et al.* [8] and later Dreyer [10, 12] have explored adding support for recursion. Harper-Lillibridge [24] and then Russo [49] studied mechanisms for making modules first-class entities in the core language. With first-class modules, programmers can leverage the familiar module system to take advantage of the System F-like power of the module system for programming-in-the-small. Unfortunately, as Garcia [20] remarks, the syntactic overhead of the ML module system makes this undesirable and impractical. When used in a similar context, Garcia suggests that the type inference used for type classes makes modular programming-in-the-small more succinct. This observation holds only in specialized use case of type classes.

4.2 Syntactic paths approach

One of the first formal accounts of an ML-like module system is Leroy's manifest types calculus [30] where manifest types are definitional type specifications. The surface language for the manifest types calculus is equivalent to that of the translucent sums calculus described by Harper-Lillibridge. The key observation in the manifest types calculus is that one can typecheck manifest types by comparing the rooted syntactic paths to those types which uniquely determine type identity. Thus, type equivalence is syntactic path equivalence. Leroy introduced the notion of applicative functors which held types in the result of a functor application to be equivalent to corresponding types in all other results of applications of that functor to the "same" argument [31]. There is a design space for module equivalence, from static equivalence to full observational equivalence. Several designs [13, 48, 53] have tried to incorporate both applicative and generative functors in a single calculus. Cregut [9] enriched signatures with structure equalities to obtain complete syntactic signatures for separate compilation.

Leroy introduces a relatively simple approach to module system semantics that precludes shadowing of core and module bindings [32, 33]. The semantics supports type generativity and SML90-style definitional sharing by reducing them to solving path equivalence by way of A-normalization (for functor applications) and S-normalization (a consolidation of sharing constraints by reordering). In his module system, Leroy claims that all type sharing can be rewritten in his calculus with generative datatypes and manifest types. However, Leroy's simplified module system does not include value specifications and datatype constructors both of which can constrain the order in which specifications must be written and therefore result in situations where sharing constraints cannot be in general reduced to manifest types.

For full transparency, Leroy proved that there is a type-preserving encoding of a stratified calculus with strong sums without generativity using applicative functors [31], claiming that

$$\frac{f \text{ is a fresh higher-order dependency variable}}{\Gamma, \mathcal{W} \vdash \text{type t} \Rightarrow ((t \mapsto f(\mathcal{W}), \emptyset), \{f\})}$$

Figure 14: Biswas's elaboration rule for abstract type specifications: Γ is the type environment mapping program variables to types. $\mathcal{W}$ is a list of formal parameters variables the specification may depend on.

the existence of such an encoding is a strong hint that applicative functors support full transparency. My HO apply functor example in fig. 4 casts some doubt to this claim. As Leroy pointed out, under the strong sums model, first-class modules is at odds with phase distinction because of the typechecker would have to do arbitrary reductions [30]. In contrast, because the weak sum model of the manifest types calculus does not require any reductions at typecheck time regardless of the presence of first-class modules, it does not violate the phase distinction [30]. In the most recent paper in the manifest types series [33], Leroy abstracts away most of the core language details from the manifest calculus to obtain a mostly core language independent module system.

Shao [53] offers a signature language based on gathering (and internally factoring out) all flexible components (*i.e.*, abstract type components unconstrained by sharing) in a higher-order type constructor that can be applied to obtain a signature that expresses functor body semantic actions at a later point. The resultant signature language superficially resembles applicative functors. However, type constructor applications in the signature language must be on paths. Consequently, it does not support full transparency in the general case.

Although the syntactic paths approach may very well provide the simplest account of module systems, this is at the cost of some very fundamental shortcomings such as the inability to support shadowing and full transparency. Because the account's support for type sharing is incomplete, there may also be limits to how the semantics deals with the coherency issue. The proposed dissertation will address these issues which are fundamental to the power of the ML module system.

4.3 Moscow ML

Biswas gave a static semantics for a simpler form of higher-order modules [2]. The account relies on semantic objects and a stamp-based semantics similar to the Definition. The type propagation in higher-order functors is captured by a "higher-order" dependency variable that abstracted possible dependencies on the argument. These variables are only present in the internal language produced during elaboration. Consequently, Biswas's semantics does not support true separate compilation and neither does it enrich the surface syntax for signatures. Biswas's elaboration rule in fig. 14 maps an abstract type name t to the fresh higher-order abstract dependency variable f applied to the list of all abstract dependency variables $\mathcal{W}$ it could possibly depend upon. For example, the functor parameter of the Apply functor, **functor** F(X:**sig type** t **end**): **sig type** t **end**, is given the semantic representation $\forall f_0(\{t \mapsto f_0\} \Rightarrow \{t \mapsto f_1(f_0)\})$.

Biswas's formal account was extended in a somewhat more type-theoretic style to the full

SML language and implemented by Russo in the Moscow ML compiler. Moscow ML also adds support for first-class modules [49] and a form of recursion [50]. This family of semantics attempts to incorporate both non-propagating generative functors and applicative functors. The main limitation, as pointed out of Dreyer [11], is that Moscow ML's combination of generative and applicative functors is unsound. In particular, any generative functor can be η-expanded into an applicative functor thereby circumventing the generative functor abstraction.

4.4 Units and other extralinguistic linking system

Flatt-Felleisen [19] and Owens-Flatt [43] develop a module system semantics based on a calculus with stratified distinct hierarchically composable modules and recursively linkable units. Because both accounts appeal to extralinguistic linking semantics, they fall under the Module/Mesa line of module systems. As pointed out of Dreyer [15], the fundamental limitation in this semantics is that the stratification of units and modules makes precludes using unit linking and hierarchical composition together. One strength of their module system design is that it is one of the few accounts that includes an operational dynamic semantics, unlike all the other accounts discussed in this section. All other accounts of module systems merely give a static semantics and perhaps a typechecking algorithm which they prove is sound with respect to the static semantics. Owens and Flatt prove type soundness of their semantics.

Swasey *et al.* [54] described a calculus SMLSC that is modeled after Cardelli's linkset approach to separate compilation. SMLSC introduces a compilation unit that sits on top of the module system that can be separately compiled from unimplemented dependencies by means of a handoff units whose role resembles that of header files in C.

4.5 The Definition and MacQueen-Tofte

The Definition of Standard ML [39, 40] semantics for the module system evolved throughout the late 1980s and early 1990s. Early on, Harper *et al.* gave a fairly complete account of the static semantics of the first-order ML module system in terms of an operational stamp-based semantics [22]. Tofte proves that signature expressions in the first-order, generative semantics have principal signatures in his thesis [55]. He also extended this proof to cover non-propagating higher-order generative functors [57]. Then MacQueen and Tofte introduced true higher-order functor semantics [38].

Apart from type propagation transparency issues, the evolution of the Definition also addressed other key issues type sharing issues. The role of **type sharing constraints** has evolved through the development of the Standard ML semantics and SML/NJ implementation. Type sharing constraints solve two problems in ML, type specification refinement and coherence. Originally, type specifications only declared the name of an expected type. It is quite useful to be able to refine type specifications to restrict it to particular definite types. The coherence problem is the challenge of constraining type components of two structures which may or may not be identical to be equivalent regardless of the actual identity of that

type. In SML90 [39], explicit sharing equations among visible abstract types and generative structure sharing served as the sole means for constraining type specifications. Under generative structure sharing semantics, each structure had a unique identity. Thus, two structures shared only when they were identical in the sense that they were defined at the same point in the program and are merely aliases (fig. 15). Structures that shared in this sense were equivalent both statically and dynamically. This kind of rich sharing semantics turned out to be quite complicated and was soon abandoned in favor of a structure sharing that simply reduced to type sharing constraints on the type specifications inside the signature.

SML93 introduced definitional type specifications, giving programmers two ways for constraining types to definite ones. SML97 added where type and definitional type specifications completely replaced the definitional type sharing found in SML90. Definitional type specifications and type sharing were finally disentangled. Generative structure sharing was eliminated in favor of the simpler semantics of a structural sharing that amounted to a type sharing equation for each common type specification, no matter how deeply nested. The semantics of type sharing and related mechanisms such as where **type** are still somewhat problematical and unsettled [41, 46]. Type sharing as it stands gives rise to a delicate and non-obvious resolution algorithm, **instantiation**. Ramsey *et al.* has argued that the scoping of where **type** definitions should be more symmetric thereby permitting more flexible type specification refinements.

Shao [52] (in a paper unrelated to the one on applicative functor-like module system [53]) extends MacQueen-Tofte fully transparent modules with support for type definitions, type sharing (normalized into type definitions), and hidden module components. This treatment of higher-order modules is a more recent form of what is currently in the SML/NJ compiler. Elsman presents a module system compilation technique used in the ML Kit compiler [17]. The semantics follows the style of the Definition. The compilation technique is comparable to Shao's FLINT compilation scheme.

The SML/NJ compiler implements a version of the module system that departs from the Definition in a number of aspects. Some of these extensions have not been formalized as of yet. In particular, the compiler has a richer semantics for **include** and the elaborator now compiles a functor body to a static lambda calculus which is what is used in place of the actual functor body during the re-elaboration at application. The scoping of sharing constraints has also changed. In the current implementation, SML/NJ no longer permits nonlocal forms of sharing of the flavor illustrated in the example in MT (fig. 16). Instead, structure definitions express the same kind of sharing. The module system also has some significant limitations such as recursion, the tension between separate compilation and fully transparent generative functors, and the limited signature language.

4.6 Alice ML

Alice ML provides a number of the features mentioned in this proposal especially in the area of signature language enrichments. The language supports nested signatures of both varieties: those that must be repeated in the signature for the enveloping structure and those that are abstract. The abstract signatures do not, however, appear to be complemented with

```
signature S0 =
sig
  type t
  val f : t -> t
  val state : t ref
end

functor F(X: sig
               structure A : S0
               structure B : S0
               sharing A = B
             end) = ...

functor G() =
struct
  type t = unit
  val f = ...
  val state = ref 0
end

structure M0 = G()
structure M1 = G()
structure M2 = M0

structure M3 = F(struct
                   structure A=M0
                   structure B=M2
                 end)

structure M4 = F(struct
                   structure A=M0
                   structure B=M1
                 end)
```

Figure 15: Under SML90 identity-based structure sharing, A and B have to be aliases, so the functor application at M4 fails to typecheck. Under SML97, the sharing constraint merely rewrites to sharing type A.t = B.t, thus both functor applications typecheck.

```
structure S = struct end;
signature SIG =
sig
  structure A : sig end
  structure B : sig end
  structure C : sig end
  sharing A = S
  sharing B = C
end
```

Figure 16: In the current implementation of SML/NJ, the first kind of sharing constraint is no longer permitted. Both sides of the constraint must be in local scope as is in the second sharing constraint.

any bounded polymorphism features.

4.7 MixML

Dreyer and Rossberg [15] show how to encode ML signatures, structures, and functors in a mixin module calculus that appeals to something similar to Bracha's merge [4] as its only linking mechanism. When linking a module A and B, the semantics tries to satisfy the imports of A using the exports of B and vice versa. The mixin merging syntax is **link** x=M0 **with** M1 in the surface language. It binds the name x to a module M0 and concatenates it with M1 merging components where appropriate. The scope of x is the body of M1. This mixin merging semantics supports recursion and separate compilation. Modules in this language consist of atomic modules that only contain values, types, type constructors, *etc.*, labeled modules ($\{\ell = M\}$), and the merging form link ... with ....

The peculiarity in this language is that modules and indeed anything that can be encoded in these modules are stateful. For example, signatures in MixML are fundamentally stateful. Linking against a signature S mutates it. Consequently, the typechecker rejects the following program.

```
module U = link x={module S = {type t}}
           with {module A=(link x=x.S with {type t = int}),
                 module b=(link x=x.S with {type t = bool})};
```

Signatures must be suspended and then new'ed in order to be matched multiple times. This suspension is called a unit in Dreyer and Rossberg's terminology. Functors are also represented by suspending modules with unsatisfied imports. Although units have full support for hierarchical composition, MixML's design still retains the problem of stratifying modules and units, a problem inherited from the Flatt-Felleisen units that inspired it.

As it stands, the part of the MixML language that encodes the ML module system is but a small fraction of the whole. The question remains what implications the rest of the language has. Part of the language is obviously semantically meaningless such as **link** X =

[int] **with** [3], which is a well-formed program. The approach that the proposed dissertation's semantics will take is to extricate the part of MixML that encodes the ML module system and hopefully simplify the semantics by omitting the rest of features. One feature I think is worth pursuing is the fact that in MixML signatures can be composed together. In ML, one can only project on modules. MixML [15] loosens this restriction by conflating signatures and modules. Signatures can be projected out of an enclosing signature.

4.8 First-class polymorphism inference and type classes

Jones [28] motivated first-class polymorphism (FCP) by appealing to the constructive logic tautologies for existentials, universals, and implication [28]. I observed that the constructive logic rule $\langle w, \tau_w \rangle \to \tau' \leftrightarrow w \Rightarrow \tau_w \to \tau'$ corresponds to the FLINT transformation (currying) in the forward direction and an uncurrying operation, perhaps a kind of module inference. At any rate, I would like to investigate the use cases for FCP where modules would be sufficient. More recent first-class polymorphic calculi such MLF [29] and FPH [58] add some limited type inference. Although inference in general may be undecidable, these limited inferencers still go a long way to make programming in these first-class polymorphic calculi more practical. If some of the ideas for inference for FCP calculi can be transferred over to a module calculus, one might also address the syntactic overhead of ML module systems. Adding FCP to the core introduces a certain amount of redundancy with respect to the FCP afforded by the module system. It would be useful to consider what exactly is redundant and whether that can be minimized.

Another language construct related to module systems that enjoys type inference is the type class. Type classes are a special case of modular programming where a kind of automatic deduction would be useful. Unfortunately, the scope of class instances are global. In Modular Type Classes, Dreyer *et al.* [14] develop semantics and a translation from type classes to a stylized use of the ML module system. I hypothesize that the type inference features of type classes might be able to be "back-ported" to module systems.

4.9 Summary

Not all module system designs enjoy the principal signature and phase distinction properties. Fig. 17 summarizes the key features of the main ML-like module system families. Note that many module systems have various combinations of these features, but none are complete. Ideally, a module system would have true higher-order semantics and all the other features except for applicative functors which would be redundant.

5 Methodology

Informed by MixML, MacQueen-Tofte semantics, and FLINT semantics as the main sources of inspiration, my dissertation research will define a new formal semantics (including a dynamic semantics) and type system for true higher-order module system based on the current

System	higher-order	first-class	sep comp	rec	app	gen	phase
HL [24]	✗	✓	✓	✗	✗	✗	✓
Leroy [31]	✗	✗	✓	✗	✓	✗	✓
Russo [50]	✗	✓	✓	✓	✓	✓	✓
DCH [13]	✗	✓	✗	✗	✓	✓	✓
RMC [12]	✗	✗	✗	✓	✗	✓	✓
MT [38]	✓	✗	✗	✗	✗	✓	✓
MixML [15]	✗	✓	✓	✓	✗	✓	✗
Ideal	✓	✓	✓	✓	✗	✓	✓

higher-order = true higher-order
rec = recursive modules
app = applicative functors
gen = non-propagating generative functors
phase = respects the phase distinction

Figure 17: A comparison of major ML-like module systems

module system design in the SML/NJ compiler. The semantics will clarify and extend the implicit compiler semantics. Part of this study will include experimental prototypes evaluated according to the design criteria outlined above. This prototype module system will validate the practicality of the formal design. The prototype will include a module language elaborator including typechecker and basic compilation into a suitable typed intermediate language.

Through the course of formalizing the module system, the dissertation will establish type soundness of the module language for the dynamic semantics and type system in the style of Owens-Flatt but for the more powerful ML module system. Moreover, it will precisely define full transparency, separate compilation, and the relationship between the two. The hypothesis is that these two features are mutually exclusive because I conjecture that typechecking a signature language powerful enough to encode all possible relationships between functor parameter and body would be undecidable. If the hypothesis turns out to be false, then the dissertation should develop a signature calculus powerful enough to represent all possible static semantic actions of higher-order functor application. The final component of the dissertation will be the decidability and soundness of the signature calculus typechecking.

5.1 Primary and secondary components

SML/NJ's implementation of elaboration and translation into the FLINT language offers some unique insight into the semantics of the module system. These insights are not merely implementation choices or details. They reflect novel fundamental issues in the design and understanding of module systems. In studying translation, it has become clear that not all abstract type components are equal. The form of a functor argument is constrained by the functor parameter signature possibly modified by a where type definition. In the parameter signature, there can be structure specifications, formal functor specifications,

structure/type sharing constraints, and two classes of type specifications. Type specifications may be abstract or definitional. Abstract type specifications that remain abstract after the elaborator resolves all sharing and where type constraints are called flexible or primary components. These primary type components are those essential components that must be kept to maintain the semantics of functor application (*i.e.*, the type application associated with the functor application). The specific function of primary type components is to capture a canonical representative of an equivalence class of abstract types induced by type sharing constraints. Each equivalence class has exactly one primary type component that serves as a representative element. References to all other members of the equivalence class should be redirected to the associated primary type component. The remaining type components are secondary and therefore should be fully derivable from the primary components and externally defined types. Secondary types do not have to be explicitly represented in the parameter signature because all occurrences of these secondary types can be expanded out according to their definitions.

```
functor F(type s
          type t
          type u = s * t
          sharing type t = s) = ...
```

In the above example, `s` can be primary, representative for the equivalence class containing both `s` and `t`, and `u` is secondary.

6 Conclusion

ML module systems have evolved throughout the last decade by avoiding the implications of true higher-order functors and adopting a combination of applicative and opaque generative functors. At the cost of the complexity of two coexisting kinds of functors and the loss of some abstraction power, recent module systems gained true separate compilation. This dissertation will revisit true higher-order functors, motivated by the search for the exact nature of the relationship between true higher-order functors and true separate compilation. Because the problem of true separate compilation concerns the expressiveness of the signature calculus, I will also take this opportunity to revisit the ML signature calculus and extend it to support more flexible modular software composition.

References

[1] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 13–23, New York, NY, USA, 1994. ACM.

[2] Sandip K. Biswas. Higher-order functors with transparent signatures. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–163, New York, NY, USA, 1995. ACM.

[3] Matthias Blume. *Standard ML of New Jersey compilation manager*. Manual accompanying SML/NJ software, 1995.

[4] Gilad Bracha. *The programming language Jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 1992.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[6] Gary Bray. Implementation implications of Ada generics. *Ada Lett.*, III(2):62–71, 1983.

[7] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM.

[8] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 50–63, New York, NY, USA, 1999. ACM.

[9] P. Cregut and D. MacQueen. An implementation of higher-order functors. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 1994.

[10] Derek Dreyer. A type system for well-founded recursion. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–305, New York, NY, USA, 2004. ACM.

[11] Derek Dreyer. Understanding and evolving the ML module system. Technical report, School of Computer Science, Carnegie Mellon University, 2005.

[12] Derek Dreyer. A type system for recursive modules. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 289–302, New York, NY, USA, 2007. ACM.

[13] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–249, New York, NY, USA, 2003. ACM.

[14] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM.

[15] Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 307–320, New York, NY, USA, 2008. ACM.

[16] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 262–273, New York, NY, USA, 1996. ACM.

[17] Martin Elsman. Static interpretation of modules. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 208–219, New York, NY, USA, 1999. ACM.

[18] Kathleen Fisher and John Reppy. Statically typed traits. Technical Report TR-2003-13, Department of Computer Science, University of Chicago, Chicago, IL, December 2003.

[19] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248, New York, NY, USA, 1998. ACM.

[20] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, March 2007.

[21] Charles M. Geschke, Jr. James H. Morris, and Edwin H. Satterthwaite. Early experience with Mesa. *Commun. ACM*, 20(8):540–553, 1977.

[22] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *2nd Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development*, pages 308–319, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[23] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[24] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM.

[25] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354, New York, NY, USA, 1990. ACM.

[26] Robert Harper and Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*, chapter Design Considerations for ML-Style Module Systems. MIT Press, 2005.

[27] Mark P. Jones. Using parameterized signatures to express modular structure. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 68–78, New York, NY, USA, 1996. ACM.

[28] Mark P. Jones. First-class polymorphism with type inference. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 483–496, New York, NY, USA, 1997. ACM.

[29] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, August 2003.

[30] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, New York, NY, USA, 1994. ACM.

[31] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–153, New York, NY, USA, 1995. ACM.

[32] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[33] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[34] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. Available as Technical Report CMU-CS-97-122.

[35] David MacQueen. Modules for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

[36] David MacQueen. An implementation of Standard ML modules. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 212–223, New York, NY, USA, 1988. ACM.

[37] David B. MacQueen. Using dependent types to express modular structure. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 277–286, New York, NY, USA, 1986. ACM.

[38] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 409–423, London, UK, 1994. Springer-Verlag.

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

[40] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[41] Philippe Narbel. Type sharing constraints and undecidability. *J. Funct. Program.*, 17(2):207–214, 2007.

[42] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.

[43] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 87–98, New York, NY, USA, 2006. ACM.

[44] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[45] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[46] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An expressive language of signatures. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 27–40, New York, NY, USA, 2005. ACM.

[47] D. Rémy. Type checking records and variants in a natural extension of ML. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 1989. ACM.

[48] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, 1998.

[49] Claudio V. Russo. First-class structures for Standard ML. *Nordic J. of Computing*, 7(4):348–374, 2000.

[50] Claudio V. Russo. Recursive structures for Standard ML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2001. ACM.

[51] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *In Proc. European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[52] Zhong Shao. Typed cross-module compilation. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 141–152, New York, NY, USA, 1998. ACM.

[53] Zhong Shao. Transparent modules with fully syntactic signatures. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 220–232, New York, NY, USA, 1999. ACM.

[54] David Swasey, VII Tom Murphy, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. In *ML '06: Proceedings of the 2006 Workshop on ML*, pages 32–42, New York, NY, USA, 2006. ACM.

[55] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, May 1988.

[56] Mads Tofte. Principal signatures for higher-order program modules. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 189–199, New York, NY, USA, 1992. ACM.

[57] Mads Tofte. Principal signatures for higher-order program modules. *Journal of Functional Programming*, 4(03):285–335, 1994.

[58] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: first-class polymorphism for Haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 295–306, New York, NY, USA, 2008. ACM.

[59] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[60] Phil Wadler. The expression problem. Email to the Java Genericity mailing list, December 1998.

[61] Niklaus Wirth. The module: A system structuring facility in high-level programming languages. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 1–24, London, UK, 1980. Springer-Verlag.

A Initial progress on formal semantics

The following is a summary of an initial progress on the formalization of the true higher-order module semantics found in SML/NJ 110.60+. It is a work-in-progress. Fig. 18 summarizes the surface module calculus. It supports higher-order functors, sharing constraints, eqtype, and exception specifications. Figs. 19, 22, and 23 give the first few rules of the elaboration

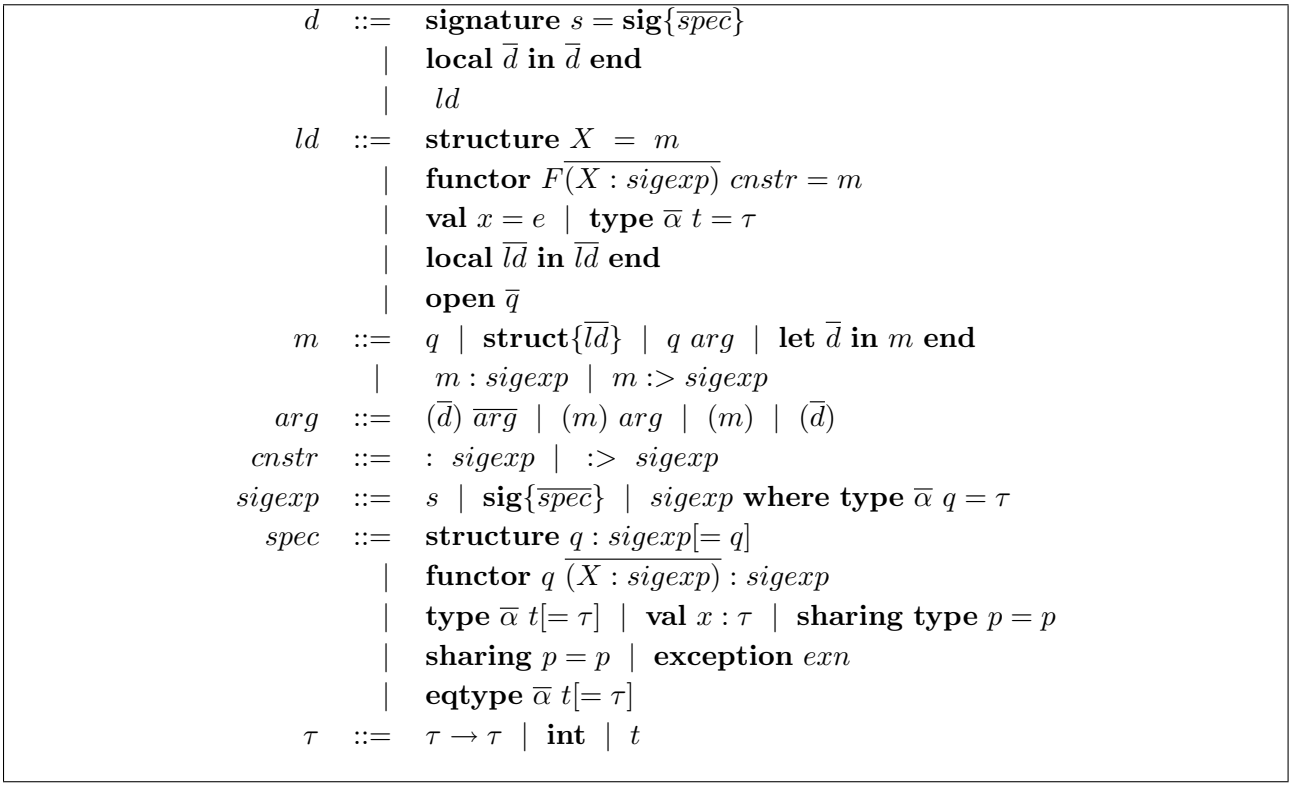


Figure 18: Module surface language: Design caveat – In SML/NJ the AST permits signature declarations within structures. The parser, however, does not support this. The surface language follows the parser. The implementation AST has an Abstract Structure form, but the parser does not seem to ever produce it.

semantics. More importantly, fig. 20 gives the internal module representation including for higher-order functors and the static lambda calculus of entities. Fig. 21 describe how entities are evaluated.

$$\frac{\begin{array}{l}\Gamma \vdash \overline{d_1} \Rightarrow (\overline{\underline{d_1}}, entdec_1, \Gamma_1, \Delta_1) \\ \Gamma \oplus \Gamma_1 \vdash \overline{d_2} \Rightarrow (\overline{\underline{d_2}}, entdec_2, \Gamma_2, \Delta_2)\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{local } \overline{d_1} \textbf{ in } \overline{d_2} \textbf{ end} \\ \Rightarrow (\underline{\textbf{local}}\ \overline{\underline{d_1}}\ \textbf{in}\ \overline{\underline{d_2}}\ \textbf{end}, entdec, \Gamma_2, \Delta)\end{array}}\ \textbf{(local)}$$

$$\frac{}{\Gamma \vdash \textbf{type } \overline{\alpha}\ t = \tau \Rightarrow (, entdec, \Gamma, \Delta)}\ \textbf{(type)}$$

$$\frac{}{\Gamma \vdash \textbf{sig}\{\overline{spec}\} \Rightarrow (\textbf{sig}\{\overline{spec}\}, \bullet, \Gamma', \emptyset)}\ \textbf{(sig)}$$

$$\frac{}{\Gamma \vdash \textbf{struct}\{\overline{ld}\} \Rightarrow \textbf{struct}\{\overline{ld}\}}\ \textbf{(structure)}$$

Figure 19: Static Semantics

Realization expression

φ	::=	$\overline{\rho}$	entity path
	|	χ	structure entity
	|	(μ, η)	stamping
	|	$\theta(\varphi)$	functor application
	|	$\varphi \downarrow \Sigma$	abstraction matching
	|	**let** η **in** φ	local definition
	|	Ξ	formal functor body
	|	$\rho.\varphi \curlyvee \varphi$	constraint
η	::=	$\rho, t \mid \rho, strid = \varphi \mid \rho, \theta$	entity declarations
θ	::=	$\lambda\rho.\varphi \mid \rho \mid$ **let** η **in** $\theta \mid \psi$	functor expression
Σ	::=	$\langle m, x, \overline{x = spec}, typs, strs\rangle$	
$spec$	::=	$\rho : \Sigma$	str no def
	|	$\rho : \Xi$	functor
	|	τ	semantic tycon
	|	$datacon$	
	|	$\rho : \Sigma =_{\overline{\rho}} \Sigma$	str const def
	|	$\rho : \Sigma = \Sigma; \varphi$	str relative def
υ	::=	$\psi \mid \chi$	entity
ψ	::=	$[\![\lambda\rho.\varphi; \Upsilon]\!]$	functor entity
χ	::=	$[\![\Upsilon]\!]$	structure entity
	|	$[\![\Sigma; \overline{\rho}]\!]$	strsig
Ξ	::=	$\langle \Sigma_p, \rho, x, \Sigma_r\rangle$	functor signature
Θ	::=	$\bullet \mid (\Psi, \overline{\rho}, \overline{\rho}, \Theta)$	entity path context
t	::=	$\overline{\rho} \mid$ Const $\tau \mid$ Formal τ	tyc expression
μ	::=	**new** $\mid$ **get** φ	stamp expression

$\Theta \oplus \rho \triangleq (\Psi, \overline{\rho_1}\rho, \overline{\rho_2}, \Theta)$ $\quad \Psi : \ \rightarrow \overline{\rho}$

$\sigma : sp \rightarrow \mu$ $\quad$ Entity Env $\Upsilon : \rho \rightarrow \upsilon$

Figure 20: Module Representation

$$\frac{\Upsilon;\Theta \vdash \varphi \Downarrow (\varphi, \Delta\Upsilon) \qquad \psi = [\![m, \lambda\rho.\varphi, \Upsilon, rp]\!] \qquad \Upsilon[\rho:\chi];\Theta \vdash_\bullet \varphi \Downarrow (\varphi', \Delta\Upsilon)}{\Theta \vdash \psi(\chi) \Downarrow \varphi'} \textbf{ (evalapp)}$$

$$\frac{}{\Upsilon;\Theta \vdash \rho \Downarrow (\Upsilon(\rho), \Upsilon)} \textbf{ (fctvar)} \qquad \frac{}{\Upsilon;\Theta \vdash \psi \Downarrow (\psi, \Upsilon)} \textbf{ (fctconst)}$$

$$\frac{(m \text{ new})}{\Upsilon;\Theta \vdash \lambda\rho.\varphi \Downarrow ([\![m, \lambda\rho.\varphi, rp]\!], \Upsilon)} \ (fct\lambda)$$

$$\frac{\Upsilon;\Theta \vdash \eta \Downarrow \Upsilon' \qquad \Upsilon';\Theta \vdash \theta \Downarrow (\psi, \Upsilon'')}{\Upsilon;\Theta \vdash \textbf{let } \eta \textbf{ in } \theta \Downarrow (\psi, \Upsilon'')} \textbf{ (fctlet)}$$

$$\frac{\Upsilon;\Theta \vdash_\rho \varphi \Downarrow (\chi, \Upsilon')}{\Upsilon;\Theta \vdash \rho, strid = \varphi \Downarrow \Upsilon'[\rho:\chi]} \textbf{ (decstr)}$$

$$\frac{\Upsilon;\Theta \vdash_{tyc} \rho, t \Downarrow \tau}{\Upsilon;\Theta \vdash \rho, t \Downarrow \Upsilon[\rho:\tau]} \textbf{ (dectyc)} \qquad \frac{\Upsilon;\Theta \vdash \theta \Downarrow (\psi, \Upsilon')}{\Upsilon;\Theta \vdash \rho, \theta \Downarrow \Upsilon'[\rho:\psi]} \textbf{ (decfct)}$$

$$\frac{}{\Upsilon;\Theta \vdash_\iota \overline{\rho} \Downarrow (\Upsilon(\overline{\rho}), \Upsilon))} \textbf{ (strvar)} \quad \frac{}{\Upsilon;\Theta \vdash_\iota \chi \Downarrow (\chi, \Upsilon)} \textbf{ (strconst)}$$

$$\frac{\Theta' = \Theta \oplus \iota \qquad \Upsilon;\Theta' \vdash \mu \Downarrow m \qquad \Upsilon;\Theta' \vdash \eta \Downarrow \Upsilon'}{\Upsilon;\Theta \vdash_\iota (\mu, \eta) \Downarrow ([\![m, \Upsilon', rp]\!], \Upsilon)} \textbf{ (strstp)}$$

$$\frac{\Upsilon;\Theta \vdash \theta \Downarrow (\psi, \Upsilon') \qquad \Upsilon';\Theta \vdash_\iota \varphi \Downarrow (\chi, \Upsilon'') \qquad \Theta' \vdash \psi(\chi) \Downarrow \varphi'}{\Upsilon;\Theta \vdash_\iota \theta(\varphi) \Downarrow \varphi'} \textbf{ (strapp)}$$

$$\frac{\Upsilon;\Theta \vdash \eta \Downarrow \Upsilon' \qquad \Upsilon';\Theta \vdash_\iota \varphi \Downarrow (\chi, \Upsilon'')}{\Upsilon;\Theta \vdash_\iota \textbf{let } \eta \textbf{ in } \varphi \Downarrow (\chi, \Upsilon'')} \textbf{ (strlet)}$$

$$\frac{\Upsilon;\Theta \vdash_\iota \varphi \Downarrow (\chi, \Upsilon') \qquad \ldots}{\Upsilon;\Theta \vdash_\iota \varphi \downarrow \Sigma \Downarrow (\chi', \Upsilon')} \textbf{ (strabs)}$$

$$\frac{\Upsilon;\Theta \vdash_\rho \varphi \Downarrow (\chi, \Upsilon') \qquad \Upsilon;\Theta' \oplus \rho:\chi \vdash \varphi' \Downarrow (\chi', \Upsilon'')}{\Upsilon;\Theta \vdash_\iota \rho.\varphi \curlyvee \varphi' \Downarrow (\chi, \Upsilon'')} \textbf{ (strcstr)}$$

$$\iota ::= \bullet \mid \rho$$

Figure 21: Entity semantics

$$\frac{\rho \text{ fresh}}{\Gamma;\Theta \vdash_{spc} X : \Sigma \Rightarrow (\Gamma[X : \Sigma; \rho], \rho : \Sigma)} \textbf{ (strspec)}$$

$$\frac{\rho \text{ fresh}}{\begin{array}{l}\Gamma;\Theta \vdash_{spc} X : \Sigma = spath \\ \Rightarrow (\Gamma[X : \Sigma; \rho], \rho : \Sigma = \Theta(\Gamma(spath)))\end{array}} \textbf{ (strspec-def)}$$

$$\frac{\begin{array}{l}\Gamma;\Theta \vdash_{sig} sigexp_p \Rightarrow \Sigma_p \qquad \rho_p \text{ fresh} \\ \Gamma[X : \Sigma_p; [\rho_p]];\Theta \vdash_{sig} sigexp_r \Rightarrow \Sigma_r\end{array}}{\Gamma;\Theta \vdash_{spc} (X : sigexp_p) : sigexp_r} \textbf{ (fctspec)}$$

$$\frac{\Gamma;\Theta \vdash_{spc} \overline{spec} \Rightarrow \overline{\Sigma} \qquad \overline{X} = \text{SpecNames}(\overline{spec})}{\Gamma;\Theta \vdash_{sig} \textbf{sig}\{\overline{spec}\} \Rightarrow \langle \overline{X : \Sigma}, typs, strs\rangle} \textbf{ (sig)}$$

Figure 22: Signature elaboration

$$\frac{\begin{array}{l}\Theta' = \Theta \oplus \iota \qquad \Gamma;\Upsilon;\Theta' \vdash_{dcl} \overline{ld} \Rightarrow (td, entdcl, \Gamma', \Upsilon') \\ \Sigma \qquad\qquad = \qquad\qquad \langle m, elems, [], []\rangle\end{array}}{\begin{array}{l}\Gamma;\Upsilon;\Theta \vdash^{\iota}_{str} \textbf{struct}\{\overline{ld}\} \\ \Rightarrow (\Sigma = \textbf{let } \Sigma; \varphi \textbf{ in } locs, \Sigma, (\textbf{new}, \eta))\end{array}} \textbf{ (strdec)}$$

$$\frac{\begin{array}{l}\rho \text{ fresh} \qquad \psi \qquad = \qquad \Gamma(sp) \\ \Gamma;\Upsilon;\Theta \vdash_{\rho} strexp \Rightarrow (argDec, argStr, argExp, \Delta\Upsilon)\end{array}}{\Gamma;\Upsilon;\Theta \vdash_{\iota} sp(strexp)} \textbf{ (strapp)}$$

Figure 23: Module elaboration