THE UNIVERSITY OF CHICAGO

A TRUE HIGHER-ORDER MODULE SYSTEM

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
GEORGE KUAN

CHICAGO, ILLINOIS
DRAFT AS OF APRIL 27, 2010

To my dearest wife

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Acknowledgments. I owe a great debt of gratitude to my advisor, Dave MacQueen, who has guided me and taught me much of what I know about programming language semantics and module system semantics. His detailed feedback very insightful and critical to the writing of this dissertation. I am also very thankful to John Reppy and Matthias Blume for serving on my committee and their helpful feedback.

I am very grateful to my aunt who has raised my brother and me. She is always a source of encouragement and inspiration. Thanks to my uncles who are role models.

I would like to thank my wonderful wife, who is so patient.

ABSTRACT

The ML module system is a powerful and expressive language for modular programming and enforcing data abstraction. Several dialects of ML have extended the module system with support for higher-order modules, which improves support for modular programming and elevates the module system to a full functional language. With the exception of Standard ML of New Jersey and MLton, higher-order modules in all these dialects do not obey natural β-reduction semantics for higher-order functor application (*true higher-order semantics*). The design space and semantics of a true higher-order module system have not been thoroughly explored. Most of the existing formal accounts of module system semantics neglect true higher-order semantics by separating higher-order functors from type generativity, which limits the flexibility of higher-order functors. The accounts which consider higher-order module semantics neglect to account for interactions between higher-order modules and key core language features such as generative datatype declarations. True higher-order semantics also paradoxically complicates true separate compilation.

In this dissertation, I contribute (1) a novel formal account of a module system true higher-order semantics, (2) a static entity calculus that cleanly isolates and expresses the higher-order semantics, (3) an exploration of the design space of higher-order module semantics including true separate compilation and the signature calculus.

CHAPTER 1

INTRODUCTION

This dissertation addresses two main subjects: the formalization of the true higher-order module system in SML/NJ and a formal investigation of the relationship between true higher-order modules and true separate compilation. The implementation of the higher-order module system has evolved since MacQueen and Tofte's paper [45]. This dissertation formalizes, simplifies, extends, and improves 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 studies 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.

1.1 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 [71, 72, 45, 2, 38, 39, 41, 61, 13] which differ mainly on how they handle type sharing in the apply functor (fig. 1.1).

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[71] 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

```
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 1.1: Apply functor example in OCaml

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

Figure 1.2: Signature assigned to Apply functor under Tofte and MacQueen-Tofte semantics

in fig. 1.2. to the apply functor. Notice that the signature for the body does not say anything more about **type** t.

MacQueen-Tofte[45] 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 [44]. 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[38] offered an alternative approach, **applicative functors**, that 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. 1.3.

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

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

Figure 1.3: Signature assigned to Apply functor under applicative functor semantics

generate fresh abstract types to enforce abstraction. To address this shortcoming, Moscow ML and Dreyer's module system[16] 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. 1.4, then applicative functors fail to propagate enough type information. When the typechecker gets to line 3 in fig. 1.4, 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, it can only give the weakest signature, **sig type** t **end** (fig. 1.5). A-normalization gets the program to

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

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

Figure 1.4: 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 1.5: A higher-order ApplyToInt functor fails to properly propagate types under Leroy's applicative functor semantics. Consequently, the last line fails to typecheck.

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. 1.4 and 1.5 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 [45] advocates is a re-elaboration of the functor body given the actual argument module.

In instances such as the SymbolTable functor (fig. 1.6), applicative functors admit too much sharing as noted by Dreyer [13]. Applicative functors would permit the symbols from one SymbolTable ST1 to be used to index another ST2 despite the sealing of the functor 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. 1.7 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 [13] 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 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 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

```
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 1.6: SymbolTable functor example from Dreyer[13]

```
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 1.7: Set functor example from Dreyer[13]

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.

1.2 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 [37, 61, 18] 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, 28] 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 [66]. This suggests that an F_ω-like calculus should be expressive enough to characterize all the static semantic actions of functors. In chapter 7, I describe the entity calculus, a small language that precisely complements the signature calculus and fully describes module types.

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 [38, 2, 61, 16] 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[69]. 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 [31] claim that type sharing constraints can be superseded by definitional type specs in all cases, my study has identified examples (fig. 1.8) 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 [61] and Swasey *et al.* [69] have suggested that the separate compilation

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

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 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.

1.3 Organization

Chapter 2 gives an overview of the evolution of module systems and a discussion of related work including some of the latest variants of the ML module system. Chapter 3 outlines and explores the design space of ML module systems. The focus lies in the variants of higher-order functor semantics including applicative functors, type generativity, first- and second-class modules, separate compilation, and signature calculus design.

Chapter 4 introduces the type system for the surface language. The type system is a first-order calculus supporting type constructors. The chapter covers the kind system and semantics of the type systems. Chapter 5 defines the core and module surface (*i.e.*, syntactic) languages. The discussion covers the a few critical classifications of type constructors in the context of the module language.

The next part of the dissertation formally defines the semantics of the module system in two interweaving modes. Chapter 6 discusses the entity calculus, a small functional language that precisely defines the functor actions needed to fully express true higher-order semantics. The chapter gives a dynamic semantics for entity calculus and establishes some key properties and the semantic and relativized type languages. Chapter 7 introduces the elaboration semantics that both compiles entity declarations/expressions and typechecks programs using the result of the evaluation of the entity declarations/expressions.

Chapter 9 discusses the implications of true higher-order semantics with respect to the other major features of the module system and compares the design with other approaches to higher-order functors. Chapter 10 considers the implication of true higher-order semantics in light of the goal of separate compilation.

Chapter 11 outlines some directions for future work and concludes.

Appendices A and B discuss semantics and implementation of higher-order modules in SML/NJ and contrasts it with that of the present work. Appendix C provides the proofs of the formal properties in above chapters.

CHAPTER 2
BACKGROUND

Modularity in programming dates back at least to Parnas and his information hiding criteria[54]. Linguistic support for program modularity took many forms. There are two main lines in the development of module systems. First, the Modula [75] and Cedar/Mesa [24] line emphasized *ad hoc* and extralinguistic schemes for modeling linking of program modules,*i.e.*, such languages used separate tool, an object file format sensitive link-loader, to compose separately compiled modules. Second, the ML line of modularity [42, 44, 43] 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 chief distinction of the basic ML module system is that modules may include type components. This provision turns out to afford the programmer considerable power and fine-grain control. The ML module system serves three main functions:

1. hierarchically organizes source

```
structure A =
struct
  structure B =
  struct
    structure C = struct type t = int end
  end
  type u = unit
end
```

2. provides a mechanism for generic programming through parameterization

```
functor F(X:sig type t end) =
struct
  type u = X.t
end
```

3. enforces abstraction boundaries

```
structure M = struct type secret = int end :> sig type secret end
```

Landin's Principle of Correspondence

The original motivation for program modularity was highly pragmatic. Computers with very small main memory could not compile a large program efficiently because program source could not fit into memory. Early languages partitioned programs into separate compilation units so that each individual compilation unit can mostly fit into memory and therefore be amenable to efficient compilation. As machine architecture progresses, program modularity is of increasing importance. Because most compiler optimization techniques rely on delicate data and control flow information and thus are sequential, modern compilers are rarely parallelized beyond the compilation of program modules in parallel. Compilers such as distcc can only take advantage of multiple cores during compilation if programs are broken down into finely separated compilation units. Thus, modularity is necessary for efficient compilation even in modern architectures.

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 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[58]. Unfortunately, records normally do not support data abstraction. Pierce's model of object-oriented languages leverages records as its core construct[56]. Lampson and Burstall's Pebble language featured a heterogeneous records that contained both values and types[35]. Cardelli's Quest and Fun languages couple System F-like languages with heterogeneous records[7].

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[75]. 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[52] 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 [5, 4, 19] and traits [64, 21].

Type classes[74] can be modeled using a very stylized use of modules coupled with a dictionary-passing inference[17]. 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.

2.1 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[47] 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 [30]. 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.

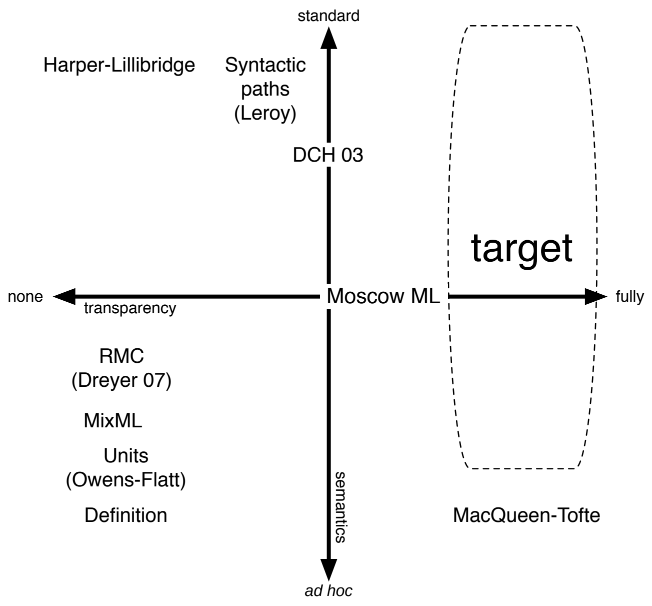
Garcia *et al.*

Garcia *et al.*[23] 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 [57]. 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.

2.2 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 [8] 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 [29] and Dreyer, Crary, and Harper's [16] type-theoretic approach, to the elaboration semantics approach as represented by the Definition[47]. As I have discussed at length in sec. 1.1, 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[75, 24, 71]. Since then, there has been a strong move toward more



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 2.1: The spectrum of major module system families

standardized logical frameworks that are more amenable to encoding in a theorem proving system such as Isabelle/HOL and Coq[29, 16]. Curiously, the most recent accounts have been moving back towards a semantic objects-based approach[14, 18]. Fig. 2.1 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** [30] plays a fundamental role in ensuring that module systems can be typechecked 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 [38, 61]. Others respect the phase distinction in an internal language but not the surface language [45].

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[13] 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* [71, 47]. I will adopt Tofte's terminology.

2.2.1 Abstract Approach

2.2.2 CMU School

Beginning with Harper-Lillibridge's translucent sums module calculus [29], 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.* [9] and later Dreyer [11, 14] (DCH) have explored adding support for recursion. Harper-Lillibridge [29] and then Russo [62] 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 [23] 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 the specialized use case of type classes.

Harper and Stone [32] gave a semantics for ML modeling every core language construct in terms of modules. This semantics was based on the technique of using singleton signatures (and types) to express type definitions [68]. Dreyer's semantics uses a signature synthesis algorithm that constructs a signature directly from a structure expression syntax given the static environment as context. In contrast, THO constructs a signature solely using the static environment.

Syntactic Paths and Applicative Functors

One of the first formal accounts of an ML-like module system is Leroy's manifest types calculus[37] 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[38]. There is a design space for module equivalence, from static equivalence to full observational equivalence. Several designs [67, 16, 61] have tried to incorporate both applicative and generative functors in a single calculus. Cregut [10] 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 [39, 40]. 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[38], claiming that the existence of such an encoding is a strong hint that applicative functors support full transparency. My HO apply functor example in fig. 1.4 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[37]. 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[37]. In the most recent paper in the manifest types series [40], Leroy abstracts away most of the core language details from the manifest calculus to obtain a mostly core language independent module system.

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.

Others

Shao [65, 67] 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.

Govereau [26] system developed an alternative semantics based on the type-theoretic approach. He claims that there is no declaration that produces a generative type that is not abstract, but datatype declarations could potentially be one of these cases. One observation Govereau makes is that the static environment must be ordered because types are dependent. Bindings in the right hand side of the environment may refer to bindings in the left hand side. The combination of applicative and generative types leads to a soundness problem similar to what is found in Russo's Moscow ML system.

The main contribution in Govereau's semantics appears to be a syntactic way to represent generativity that obviates the need for a Dreyer's effect system for managing the interaction between applicative and generative types, or more precisely, functors. The language uses distinct declarations for generative and non-generative abstract types. The type system declaratively partitions the static environment into two corresponding to the lexical scope inside and outside a functor.

The paper claims that if a type is defined "under functor" but does not depend on a generative type defined under a functor, then it can be coerced to non-generative. The claimed win is that although the syntactic purity judgment is not as fine-grain as DCH's effect system, it is simpler to work with. One disclosed limitation to both Govereau and

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

Figure 2.2: 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.

Russo's systems is that a non-generative functor cannot have any generative types that are used in the body of the functor.

2.2.3 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. 2.2 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 [61]. One semantic innovation was the use of existential types to represent abstract types in a module system. The idea of using existential types originated in a paper by Mitchell and Plotkin [48]. MacQueen pointed out how scoping unpacks make conventional existential types ill-suited for modular programming [44]. Montagu and Rémy [49] recently tried to address these concerns by developing a variation of existential types which replaces unpack with modular open and scope restriction constructs.

Moscow ML also adds support for first-class modules[62] and a form of recursion[63]. Moscow ML further attempts to incorporate both non-propagating generative functors and applicative functors. The main limitation, as pointed out of Dreyer [13], 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.

Russo notes the conflict between true separate compilation and first-order module systems. He works around the problem by defining compilation units to be only the sealed modules in an applicative higher-order module system with first-order generative functors. To address the avoidance problem, he introduces a hybrid name and deBruijn-indexing scheme.

Rossberg, Russo, and Dreyer developed a formal semantics based on a translation to System F_ω enriched with records and existential types in a recent variation on Russo's type system [60]. In this account, the type system of the module system was exactly the type system of their variation on System F_ω. All semantic representations of signatures explicitly existentially quantified all primary type constructors. The module system supported type generativity insofar as existentials support it. The module system does not claim to support full transparency, although there is an extension of the system to support applicative functor semantics.

2.2.4 Units and other extralinguistic linking system

Flatt-Felleisen [22] and Owens-Flatt [53] 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[18], 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.* [69] 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.

2.2.5 The Definition and MacQueen-Tofte

The Definition of Standard ML [46, 47] 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[27]. Tofte proves that signature expressions in the first-order, generative semantics have principal signatures in his thesis [70]. He also extended this proof to cover non-propagating higher-order generative functors[72]. Then MacQueen and Tofte introduced true higher-order functor semantics[45].

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

```
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 2.3: 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.

type. In SML90 [46], 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. 2.3). 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[50, 57]. 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 [66] (in a paper unrelated to the one on applicative functor-like module system [67]) 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 [20]. 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. 2.4). 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.

```
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 2.4: 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.

2.2.6 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 any bounded polymorphism features.

2.2.7 MLKit

MLKit's semantics for elaboration calls for what amounts to defunctorization. In Elsman's view, functors are purely static constructs that can be compiled away, generating inlined copies of relevant dynamic content upon functor application.

2.2.8 MixML

Dreyer and Rossberg [18] 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 [18] loosens this restriction by conflating signatures and modules. Signatures can be projected out of an enclosing signature.

2.2.9 Type Generativity

The idea of generative types has a significant history. More recently, Neis *et al* [51], Dreyer and Blume [15], Dreyer [12], and Rossberg [59] have all developed semantics for studying type generativity in the context of a System F$_\omega$-derived language. Although the ML module system utilizes type generativity, it only plays a role within the module system proper. After translation, at which point the modules are eliminated, all the type generation has already occured, having been carried out by the elaborator. Consequently, the target language for translation, a System F$_\omega$ language, need not contain any facilities for generating fresh types.

2.2.10 First-class polymorphism inference and type classes

Jones [34] motivated first-class polymorphism (FCP) by appealing to the constructive logic tautologies for existentials, universals, and implication [34]. I observed that the constructive logic rule $\langle w, \tau_w \rangle \rightarrow \tau' \leftrightarrow w \Rightarrow \tau_w \rightarrow \tau'$ 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[36] and FPH [73] 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.* [17] 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.

2.2.11 FLINT/Typed Cross Compilation

The most closely related work is the formal semantics NRC given by Shao[66]. Like the present semantics, NRC gives a formal semantics based on MacQueen-Tofte's higher-order semantics. Unlike the present account, Shao's semantics combines elaboration and translation into a single interweaving set of judgments whereas the present account cleanly separates them. The separation of elaboration and translation is a conceptual simplification and also beneficial for more modular compiler implementation. Also, Shao's semantics assumes a preprocessing step that eliminates most of the interesting behavior due to coercive signature matching. Although the NRC account alludes to the fact that the functor body does not need to be re-elaborated at each application, the NRC semantics does re-elaborate much like the MacQueen-Tofte account. Finally, the NRC semantics does not directly support datatype declarations, though it is claimed that it is a straightforward extension.

The account of NRC makes a few arguable simplifying assumptions. Value components is a trivial extension that only requires a conversion of source-level types to semantic types. Signature extraction is trivial only requiring the conversion of semantic types back to syntactic types. This conversion is guaranteed by the 1-to-1 mapping of the stamp environment. (This is similar to our entity environment, but as our semantics have demonstrated, this is far from trivial.) Shao says that value components in NRC are only elaborated once, but any type components may be re-elaborated. We simplify this even further. Only functor applications and type generation have to be re-elaborated. Type definitions can be expanded out early on during elaboration. As with some other accounts, NRC entangles type-checking and translation into F_ω. Although translation clearly depends on type-checking, the converse case is not true. In NRC, the assumption is that type-sharing constraints can all be converted to type definitions. As I have shown earlier, this is technically incorrect but practically sufficient. NRC requires A-normalization of the surface syntax SFC so that the only functor applications are from a functor variable (not paths) to structure variable. In THO, the operator position of applications can be a path and the operand can be any structure ex-

System	higher-order	first-class	sep comp	rec	app	gen	phase
HL [29]	✗	✓	✓	✗	✗	✗	✓
Leroy [38]	✗	✗	✓	✗	✓	✗	✓
Russo [63]	✗	✓	✓	✓	✓	✓	✓
DCH [16]	✗	✓	✗	✗	✓	✓	✓
RMC [14]	✗	✗	✗	✓	✗	✓	✓
MT [45]	✓	✗	✗	✗	✗	✓	✓
MixML [18]	✗	✓	✓	✓	✗	✓	✗
Ideal	✓	✓	✓	✓	✗	✓	✓

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

Figure 2.5: A comparison of major ML-like module systems

pression. The α-renaming scheme described in NRC adds new declarations inside of existing structures in order to deal with signature ascription. Signature matching mostly handled during the SFC to NRC translation by adding explicit enrichment coercions. NRC relies on a stamp environment, a finite map from stamp to tycon path. This has turned out to be unnecessary due to the existence of a function to perform the equivalent operation while relying on only the present semantics' structure entity itself. This function is the moral equivalent of a retraction of the realization although the realization does not technically admit a retraction.

2.2.12 Summary

Not all module system designs enjoy the principal signature and phase distinction properties. Fig. 2.5 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.

CHAPTER 3
THE DESIGN SPACE

The simplest ML module system features only hierarchical composition of structures, type components in structures, transparent signature ascription, and first-order functors. Beyond this set of features lies a design space subject to debate. The module system design space is a rich one consisting of several functional dimensions. Much of the design space revolves around the module system's abstraction enforcement through opaque sealing and how that interacts with the other features of the module language.

Many of these dimensions such as applicative versus generative, first- versus second-class, first- versus higher-order, sealing opacity, and amount of signature inference inextricably related. Dreyer frames the design space in terms of phase-separability and purity with respect to static and dynamic effects.

3.1 Applicative Versus Generative Functor Semantics

A chief point of contention is the support of applicative versus generative semantics for functors and the various attempts at supporting both. The central argument of applicative functor semantics is that a functor applied multiple times to the same path should produce results with the same type components. The original motivation was the support of higher-order functors with syntactic signatures. Subsequently, researchers have found few other examples where applicative functors have proven to be useful. One example (typically a set functor) claims that there is a functor such that one would want to apply it multiple times to the same argument, opaquely seal each result, and still expect each result's corresponding types to be equivalent.

```
functor SetFn(K: sig type t ... end) =
struct
  type set = K.t list
  val empty = []
  fun isEmpty = ...
  ...
end :>
```

```
module SetFn(X: sig type t end)
  :
    sig
       type u
       val v : u
       val f : u −> bool
    end
=
struct
   type u = X.t list
   let v = []
   let f x = true
end
module K = struct type t = int end
```

Figure 3.1: fct.ml

```
module I = SetFn(K)
```

Figure 3.2: apply.ml

```
   sig
        type set
        val empty : set
        val isEmpty : set −> bool
        ...
   end

structure IntKey = struct type t = int ... end
structure I0 = SetFn(IntKey)
structure I1 = SetFn(IntKey)

IO.isEmpty(I1.empty)
```

This behavior may be useful in an interactive system where the source containing the functor application may be loaded multiple times and one could reasonably desire the types to be equivalent. Consider the following example session:

```
# #use "fct.ml";;
module SetFn :
   functor (X : sig type t end) −> sig type u val v : u val f : u −> bool end
module K : sig type t = int end
# #use "apply.ml";;
module I : sig type u = SetFn(K).u val v : u val f : u −> bool end
# module I0 = I;;
module I0 : sig type u = SetFn(K).u val v : u val f : u −> bool end
# #use "apply.ml";;
module I : sig type u = SetFn(K).u val v : u val f : u −> bool end
# I.f(I0.v);;
− : bool = true
```

This example is very specific. Most importantly, it assumes that the functor and argument modules are always in the same file. Separating the functor F and module K into different files (a far more common case), loading either of them more than once, and applying the

```
module I0 = F.S(F.K)
module I1 = F.S(F.K)
let _ = I0.f(I1.v)
```

Figure 3.3: apply.ml

functor will cause the types I.t and I0.t to be unequal under applicative semantics because they are either applications of the same syntactic functor to two different modules K (loading K twice gives rise to two different syntactic modules) or different syntactic functors to the same syntactic K.

The above examples also come up in batch compilation. If both functor S and argument module K are defined in a compilation unit F, then applicative semantics deems the following client module well-typed.

Independently developed compilation units may each use an int set which is derived from independent applications of the generic set functor to an int key structure. The programmer might want these compilation units to exchange set values. Unfortunately, applicative functors cannot help in this latter case. Because separate compilation loads the IntKey source twice, I0 and I1 are effectively applied to two different structures from a syntactic perspective.

```
structure IntKey = struct type t = int ... end
structure IntKey' = struct type t = int ... end

structure I0 = SetFn(IntKey)
structure I1 = SetFn(IntKey')
```

Within a single compilation unit, the utility of applicative semantics seems limited. If a functor has already been applied to an argument once, there is no need to apply it again. The result of the original functor application still exists and should be made visible. If the goal is to manage the namespace, then SML's semantics of structure abbreviations sharing types even opaquely sealed ones is sufficient.

The other potential application is in distributed memory systems. Multiple computing nodes may each apply a functor to the same argument and then try to exchange values of a type in the result. Such a scenario will likely require runtime types.

One argument in favor of applicative functors that Dreyer has noted is in the construction of Okasaki's bootstrapped heaps.

3.2 First- Versus Second-Class Modules

In most dialects of ML, the module system is second-class, meaning that there is a clear demarcation between the core and module languages. This arrangement is convenient because it is the foundation of an important property known as phase-distinction, which says that the static and dynamic parts of modules can be extricated such that the static part will not depend on the dynamic part. The phase distinction saves the language designer the complications of dependent types and admits the possibility of type erasure semantics. Nonetheless, there is an argument for dropping the phase distinction in favor of picking up first-class modules, all the while trying not to have to incorporate dependent types in their full generality. The example for first-class modules often involve some runtime condition that determines the optimal abstract data structure to use, made available as modules, as illustrated by the example given by Dreyer:

```
structure M = if n < 20 then LinkedList else HashTable
```

The above is a kind of optimization that does not require first-class modules in their full generality. Neither, I would argue, does it fit with the main roles of a module system. Because the types in the structures LinkedList and HashTable may be completely different, the static description of M cannot be determined at compile-time. It would appear as if first-class module behavior necessitates types at runtime. To avoid all the associated issues, language designers have often decided to limit first-class modules by requiring that such module expressions are opaquely sealed with a single signature.

```
structure M = if n < 20 then LinkedList else HashTable
  : sig type t type item val insert : t * item -> t ... end
```

This turns out to be exactly the route Dreyer and Alice ML have elected to take. Still, even this form of first-class modules is too powerful. The static content of M remains ambiguous. In general, the dynamic part of the expression may raise exceptions, rely on other effects, or be nonterminating. Dreyer distinguishes between dynamically pure and impure module expressions for this very purpose, but dynamic purity does not preclude nontermination.

3.3 Signature Ascription

Where the core language has type ascriptions, existing module systems support ascribing signatures to module expressions. Ascriptions are even more important in the module system than in the core language because module languages are explicitly typed and some forms of ascription serve as the primary means for enforcing data abstraction.

Opaque ascription, sealing, should be construed as a deliberate design choice on the part of the programmer to make a type unique and incomparable. When using opaque ascription, the semantics should be simple. Alternative forms of opaque ascription that makes a type unique in some cases yet simultaneously comparable introduces a new set of issues that greatly complicate the module system.

3.4 OCaml

OCaml's syntax reflects the absence of generativity: there is no functor definition syntax that admits an empty parameter **functor** F() = ... Because the identity of functor body types depends on a syntactic parameter, each functor application must have one.

OCaml also omits major features such as shadowing. Because its notion of type equality is syntactic, OCaml does not support shadowing of type declarations in structures. For example, the following does not elaborate in OCaml:

```
module M = struct type t = A type t = B val x = A end
```

3.5 Definitional Specifications

Definitional specifications such as definitional type specs and definitional structure specs are common in many implementations of the ML module system. For symmetry, it would seem natural that definitional functor specs would also be permitted. However, to the best of the author's knowledge, this kind of definitional spec has not been investigated.

CHAPTER 4

TYPE SYSTEM

The syntactic type system consists of monotypes, type constructors (*tycons*), and type expressions (*aka* types). The monokind Ω and arrow kind $\Omega^n \Rightarrow \Omega$ classify monotypes and tycons respectively. As can be seen from the form of the arrow kind, the syntactic type system does not support higher-kinded tycons but it does support multi-arity tycons where the n is a natural number indicating the arity of the tycon. Furthermore, the core language does not support impredicative polymorphism. Universal quantifiers for polymorphism are added in a semantic type system introduced later in this chapter. In the surface type language, the only type expressions that can occur on the right hand side of type definitions are type variables, arrow types, and type applications. Type functions are only introduced by the type definition construct in the core language.

The kind and tycon language follow a significantly simplified version of Dreyer's language[13]. The tycon language is an impredicative variant of System F$_\omega$ for simplicity, although the core language does not take advantage of impredicativity because the tycon expressions are an elaborated version of the core language types, which do not contain universal quantifiers at all since the quantifiers are added at the module declaration level. Type expressions (T) classify core language terms. Monotypes may be of two forms. Type variables (α) come from a denumerable set. The tycon application form ($p(\vec{C^s})$) requires the type operator to named by the symbolic path p. By letting the argument of the application be an empty vector, the form can represent all simple types as an application of a nullary type operator to an empty

$$\begin{array}{rcll} p & ::= & x \mid p.x & \text{symbolic paths} \\ K & ::= & \Omega \mid \Omega^n \to \Omega & \text{kinds} \\ C^s & ::= & \alpha \mid p(\vec{C^s}) & \text{monotypes} \\ C^\lambda & ::= & \lambda\vec{\alpha}.C^s & \text{closed tycons} \\ T & ::= & \mathsf{typ}(C^s) \mid \forall\vec{\alpha}.C^s & \text{closed types} \\ e & ::= & p \mid \lambda x.e \mid e_1 e_2 & \text{terms} \end{array}$$

Figure 4.1: Surface type system

argument vector.

Wrapping an n-ary λ-abstraction around a monotype produces a tycon. The λ tycon denotes type functions parameterized by tycon variables $\vec{\alpha}$. The vector of tycon parameters may be empty thus permitting nullary type functions, which represent nullary tycons.

In the syntactic tycon language, λ-abstractions are n-ary and not curried, so applications must be saturated. Using the λ-abstraction to represent even nullary tycons ensures that the semantics can distinguish definitional tycons, *i.e.*, they are exactly those wrapped with an outer λ-abstraction. C^λ are the only syntactic tycons, but semantic tycons, described in sec. 4.2, add forms corresponding to datatype tycons. This does not reduce the expressiveness of the tycon language at all. In fact, it is a very close match to the tycon language in ML, considerably closer than more general languages found in previous treatments.

The universal type expression $(\forall\vec{\alpha}.C^s)$ represents polymorphism. Universal quantifiers only occur at the structure declaration level.

4.1 Kind System

The well-kinding judgments are of the form $\Gamma, \Delta \vdash C : K$ where C is C^s, C^λ, and T. The static environment Γ is used to interpret symbolic paths in monotype application forms $p(\vec{C^s})$. At this point, one can think of Γ as a finite partial map from symbolic paths to syntactic tycons. In chapter 7, static environments will be formally defined. $\Gamma(p)$ calculates a tycon, either a defined tycon or a tycon produced by datatype declaration, which is internal, accessible by a type variable. Δs represent the kind environment, which contains all the bound type variables, which by default have kind Ω. Again, only monokinds are necessary here because the syntactic type system does not support higher-kinded tycons. Also noteworthy is that type variables α can only be of the monokind (rule 4.1).

The tycon application kinding rule (4.2) checks that the operator tycon path is well-kinded, the correct number of arguments is supplied and that each of the arguments is well-kinded with kind Ω. The notation C^s_i denotes the ith element in the $\vec{C^s}$ vector. The

$Path$	$=$	set of all symbolic paths	
Tyc_{syn}	$=$	set of all closed syntactic tycons	
Δ	$::=$	$\emptyset_{knds} \mid \Delta[\alpha]$	kind environment
Γ	$:$	$Path \rightharpoonup Tyc_{syn}$	static environment

$\boxed{\Gamma, \Delta \vdash C^s : \Omega}$

$$\frac{\alpha \in \Delta}{\Gamma, \Delta \vdash \alpha : \Omega} \tag{4.1}$$

$$\frac{\Gamma, \Delta \vdash \Gamma(p) : \Omega^n \Rightarrow \Omega \qquad |\vec{C^s}| = n \qquad \Gamma, \Delta \vdash C^s_i : \Omega \ \forall i \in [1, n]}{\Gamma, \Delta \vdash p(\vec{C^s}) : \Omega} \tag{4.2}$$

$\boxed{\Gamma, \Delta \vdash C^\lambda : \Omega^n \to \Omega}$

$$\frac{\Gamma, \Delta[\alpha_1] \ldots [\alpha_n] \vdash C^s : \Omega}{\Gamma, \Delta \vdash \lambda \vec{\alpha}. C^s : \Omega^n \Rightarrow \Omega} \tag{4.3}$$

$\boxed{\Gamma \vdash T : \Omega}$

$$\frac{\Gamma, \emptyset_{knds} \vdash C^s : \Omega}{\Gamma \vdash \mathsf{typ}(C^s) : \Omega} \tag{4.4}$$

$$\frac{\Gamma, [\alpha_1] \ldots [\alpha_n] \vdash C^s : \Omega}{\Gamma \vdash \forall \vec{\alpha}. C^s : \Omega} \tag{4.5}$$

Figure 4.2: Well-kinding of syntactic tycons

τ^n	$\in$	Tycs	n is arity : $\Omega^n \Rightarrow \Omega$
$\mathfrak{C}^s$	$::=$	$\alpha \mid \mathfrak{C}^\lambda(\overrightarrow{\mathfrak{C}^s})$	semantic monotype
$\mathfrak{C}^\lambda$	$::=$	$\lambda\vec{\alpha}.\mathfrak{C}^s \mid \tau^n$	semantic tycon
$\mathfrak{T}$	$::=$	$\mathsf{typ}(\mathfrak{C}^s) \mid \forall\vec{\alpha}.\mathfrak{C}^s$	semantic type expression
$\mathfrak{C}^{nf}$	$::=$	$\alpha \mid \tau^n(\overrightarrow{\mathfrak{C}^{nf}})$	normal form tycons

Figure 4.3: Semantic type system

rest of the rules ensure that the types and tycons are closed with respect to type variables (*i.e.*, all type variables are in Δ).

4.2 Semantic Type System

The foregoing type system is purely syntactic. There is no semantics for evaluating the type expressions and tycons. Syntactic types cannot be evaluated directly due to the presence of the uninterpreted symbolic paths. Fig. 4.3 gives a *semantic* type language that replaces the symbolic paths in the tycon application form with tycons. Semantic monotypes, tycons, and type expressions are distinguished from their syntactic counterparts by the Fraktur script ($\mathfrak{C}$). The semantic type language is first-order and simply-kinded.The Tycs set is the set of all tycon symbols τ^n, distinct from syntactic symbols. Each such tycon has an assigned arity n. The elements in Tycs will be called *atomic tycons.* All primitive tycons such as int, bool, and arrow fall into the Tycs set and all generated datatype tycons will be elements of Tycs.

The semantic type system consists of semantic monotypes ($\mathfrak{C}^s$), semantic tycons ($\mathfrak{C}^\lambda$), and semantic type expressions ($\mathfrak{T}$) each of which corresponds to the respective syntactic type system version. Semantic monotypes differ from syntactic monotypes in that semantic tycons replace symbolic paths in the application form. Semantic tycons consist of both a λ form that represents definitional tycons and a τ^n form, a concrete, atomic tycon from Tycs, which is intended to represent tycons generated by datatype declarations, opaque ascription, and functor applications. The arity of well-formed tycons can always be calculated directly,

$\boxed{\Delta \vdash \mathfrak{C}^s : \Omega}$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha : \Omega} \tag{4.6}$$

$$\frac{\Delta \vdash \mathfrak{C}^\lambda : \Omega^n \Rightarrow \Omega \qquad |\vec{\mathfrak{C}^s}| = n \qquad \Delta \vdash \mathfrak{C}^s_i : \Omega \; \forall i \in [1, n]}{\Delta \vdash \mathfrak{C}^\lambda(\vec{\mathfrak{C}^s}) : \Omega} \tag{4.7}$$

$\boxed{\vdash \mathfrak{C}^\lambda : \Omega^n \to \Omega}$

$$\frac{[\alpha_1] \ldots [\alpha_n] \vdash \mathfrak{C}^s : \Omega}{\vdash \lambda\vec{\alpha}.\mathfrak{C}^s : \Omega^n \to \Omega} \tag{4.8}$$

$$\frac{}{\vdash \tau^n : \Omega^n \to \Omega} \tag{4.9}$$

$\boxed{\vdash \mathfrak{T} : \Omega}$

$$\frac{\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega}{\vdash \mathsf{typ}(\mathfrak{C}^s) : \Omega} \tag{4.10}$$

$$\frac{[\alpha_1] \ldots [\alpha_n] \vdash \mathfrak{C}^s : \Omega}{\vdash \forall\vec{\alpha}.\mathfrak{C}^s : \Omega} \tag{4.11}$$

Figure 4.4: Well-kinding of semantic tycons

$|\tau^n| = n$ and $|\lambda\vec{\alpha}.\mathfrak{C}^s|$.

Fig. 4.4 gives a kind system for semantic monotypes, tycons, and type expressions. This kind system differs from the the previous one in fig. 4.2 in that the application rule 4.7 no longer needs to look up a static environment for a symbolic path, because there are no symbolic paths. The simplifies the kind system in that the static environment can be dropped. To handle atomic tycons, the kind system also adds rule 4.9 that gives it the arrow kind $\Omega^n \Rightarrow \Omega$ to match its arity. Thus, the role of this kind system is solely to verify that arities match up and that the types are closed with respect to type variables.

$$\frac{\overrightarrow{\mathfrak{C}^s} \Downarrow_{mt} \overrightarrow{\mathfrak{C}_1^{nf}} \qquad \mathfrak{C}^s\{\overrightarrow{\mathfrak{C}_1^{nf}}/\vec{\alpha}\} \Downarrow_{mt} \mathfrak{C}_2^{nf}}{(\lambda\vec{\alpha}.\mathfrak{C}^s)(\overrightarrow{\mathfrak{C}^s}) \Downarrow_{mt} \mathfrak{C}_2^{nf}} \tag{4.12}$$

Figure 4.5: Semantic tycon evaluation

4.2.1 Notation

Let $\mathfrak{C}^\lambda$ be a semantic tycon. $|\mathfrak{C}^\lambda|$ refers to the arity of the tycon. If the tycon is an atomic tycon τ^n, then $|\tau^n| = n$. If the tycon is a $\lambda\vec{\alpha}.\mathfrak{C}^s$, then the arity is the length of the $\vec{\alpha}$. The $|\cdot|$ notation is also carried over in the obvious way to relativized tycons. $\mathfrak{C}_1^{nf} \equiv_\alpha \mathfrak{C}_2^{nf}$ is the equivalence of two semantic monotypes module α-renaming of type variables,*i.e.* the two monotypes are isomorphic up to α-renaming of type variables.

4.2.2 Evaluation

Semantic type expressions and monotypes can be evaluated by a β-reduction rule in Fig. 4.5 to a normal form $\mathfrak{C}^{nf}$. Note that the normal form of atomic tycons with 0-arity is $\tau^0()$.

The evaluation rule is the typical big-step call-by-value β-reduction modified to accept a vector of arguments. Since the semantic type language is analogous to a simply-typed λ-calculus (actually a first-order simply-typed λ-calculus enriched with constants), the language is strongly normalizing (lem. 21).

Lemma 1 (Monotypes are strongly normalizing)
If $\mathfrak{C}^s \Downarrow_{mt} \mathfrak{C}_1^s$ and $\mathfrak{C}^s \Downarrow_{mt} \mathfrak{C}_2^s$, then $\mathfrak{C}_1^s = \mathfrak{C}_2^s$ and $\mathfrak{C}_1^s$ is a $\mathfrak{C}^{nf}$.

Not only are the types strongly-normalizing, the semantic type language also has preservation and progress properties.

Lemma 2 (Kind Preservation)
If $\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega$ and $\mathfrak{C}^s \Downarrow_{mt} \mathfrak{C}^{nf}$, then $\emptyset_{knds} \vdash \mathfrak{C}^{nf} : \Omega$.

Lemma 3 (Progress)

If $\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega$, *then* $\mathfrak{C}^s \Downarrow_{mt} \mathfrak{C}^{nf}$.

The connection between the syntactic and semantic type systems will be made clear in chapter 7.

CHAPTER 5
SURFACE LANGUAGE

5.1 Core Language

The surface language (Fig. 5.1) closely follows SML/NJ's syntax with a few exceptions. There are three forms of core level declarations: value, type definition, and datatype. For type definitions, the type parameters are written in an explicit λ-abstraction form $\mathsf{type}\ t = \lambda\vec{\alpha}.C^s$ instead of $\mathsf{type}\ \vec{\alpha}\ t = C^s$. Definitional type specs use a similar notation. For simplicity, the data constructor part of datatype declarations is omitted. The interesting behavior is the datatype declaration's generation of a fresh atomic tycon.

The core language is a simple one comprised of an implicitly typed expression language (e) and a simple declaration language (d^c). There are three forms of declarations in the core language, value declarations, type definitions, and datatype declaration. For simplicity's sake, datatype declarations only declare the name and parameters (arity) of a generative tycon.

5.2 Signatures

A signature is a tree of static information for a structure. This object is a hierarchy because it must reflect the hierarchical nesting of structures. The hierarchy can be considered a tree where the internal nodes are structure specifications (*i.e.*, a signature for a named structure) and the leaves are static entities for tycons and functors.

Signatures are defined as a sequence of comma separated specs. Value specs use the notation $\mathsf{val}\ x : \forall\vec{\alpha}.C^s$ instead of $\mathsf{val}\ \vec{\alpha}\ x = C^s$. Each sequence is terminated by the empty spec $\emptyset_{specs}$. Signature expressions include a variable form, base signature $\mathsf{sig}\ \mathit{spec}\ \mathsf{end}$, and a signature expression modified by a where type clause. Signature declarations are always at top level.

$$
\begin{array}{lrcl}
\text{Core declarations} & d^c & ::= & \mathsf{val}\ x = e \mid \mathsf{type}\ t = C^\lambda \mid \mathsf{datatype}\ \vec{\alpha}\ t \\
 & \mathit{spec} & ::= & \mathsf{structure}\ x : \mathit{sigexp} \mid \mathsf{type}\ \vec{\alpha}\ t \\
 & & \mid & \mathsf{type}\ t = C^\lambda \\
 & & \mid & \mathsf{functor}\ f(x : \mathit{sigexp}_1) : \mathit{sigexp}_2 \\
 & & \mid & \mathsf{val}\ x : T \\
 & \mathit{specs} & ::= & \emptyset_{\mathit{specs}} \mid \mathit{spec}, \mathit{specs} \\
 & \mathit{sigexp} & ::= & x \mid \mathsf{sig}\ \mathit{spec}\ \mathsf{end} \mid \mathit{sigexp}\ \mathsf{where\ type}\ p = C^\lambda \\
 & \mathit{strexp} & ::= & p \mid \mathsf{struct}\ d^m\ \mathsf{end} \mid p(\mathit{strexp}) \\
 & & \mid & \mathit{strexp} : \mathit{sigexp} \mid \mathit{strexp} :> \mathit{sigexp} \\
\text{Module decls} & d^m & ::= & \circ \mid \mathsf{structure}\ x = \mathit{strexp}, d^m \\
 & & \mid & \mathsf{functor}\ f(x : \mathit{sigexp}) = \mathit{strexp}, d^m \mid d^c, d^m \\
\text{Top level decls} & d^t & ::= & \circ \mid \mathbf{signature}\ x = \mathit{sigexp}, d^t \mid d^m, d^t
\end{array}
$$

Figure 5.1: Surface module language

In signatures, there are two kinds of tycon specifications, definitional and open. *Definitional tycon specs* (partially) constrain tycons that can be matched. *Open tycon specs* only declare the name and arity of a tycon that must be present in a matching structure.

```
type ('a, 'b) s = 'a list * 'b (* definitional *)
type ('a,'b) t (* open *)
```

Signature expressions may be the base form that contains specs. The where type form of signature expressions is a generalization of SML's mechanisms. The where type form adds the type definitions on the right hand side to the structure spec (signature expression) identified on the left hand side.

Since signature expressions support the base form of signatures and these expressions are found in signature specs, module declarations, and top level declarations, this amounts to the possibility of inline base signatures throughout the language. The presence of inline signatures must be accounted for during elaboration.

5.3 Module Calculus

Structure expressions can be a path p, a base structure, a functor application (where the functor occurs only as a symbolic path p), transparent signature ascription, and opaque signature ascription. A module declaration is a sequence of structure, functor, and core declarations. $\circ$ denotes the empty structure declaration that terminates every sequence of structure declarations. At the top level, in addition to module declarations, one may also have signature declarations. The top level declaration sequence is terminated by an empty top level declaration $\circ$.

The semantics of transparent and opaque ascription follow that of Standard ML. Unlike some of the more recent module systems, Standard ML's signature ascription is always a coercion on the structure to the form specified in the signature. In the transparent case, this amounts to dropping any value bindings or type declarations omitted in the signature. However, when a signature's open type specification is matched against a structure's type definition, the resulting coerced structure will reveal that type definition. In the opaque case, this kind of type definitions is occluded.

5.4 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 open or definitional. SML/NJ supports a rich notion of symmetric type sharing constraints, that will be deferred for future work. Open type specifications that remain open 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. Even in the absence of type sharing contraints as in the present module system, the distinction of primary and secondary tycons matter due to the presence of type definitions and where type.

```
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.

5.5 Nonvolatile and Volatile Type Constructors

In the module system, tycons can be classified as either nonvolatile or volatile. *Nonvolatile tycons* are atomic tycons in the initial static and entity environments. Open type specs in signatures, datatype declarations, and tycons defined relative to former two give rise to *volatile tycons*,*i.e.*, they may be instantiated to a particular tycon by functor application or signature matching. Moreover, volatile tycons can be instantiated multiple times through different functor applications or signature matching. Although volatile tycons overlap with the notion of abstract types, they are not the same. The defining characteristic of volatile tycons is that their actual instantiation will be supplied at a later point and they can have multiple instantiations within a program. The instantiation used changes depending on the context, thus the tycon is volatile. In cases such as transparent signature ascription, a volatile tycon may be perfectly transparent.

Volatile types themselves can be classified as primary (open tycon specs in functor formal parameters and signatures) and secondary (tycons defined relative to primary volatile tycons).

```
functor F(X: sig
        type a (* primary, volatile *)
        type b = int (* secondary, nonvolatile *)
        structure M0 :
        sig
            type c (* primary, volatile *)
            type d = a (* secondary, volatile *)
        end
        end) =
  struct
      type u = X.a list (* secondary, volatile *)
      structure M1 :
          sig
              type v (* spec, not a tycon *)
          end = struct
          type v = X.M0.c * int (* secondary, volatile *)
      end
  end
```

The above example illustrates some of the complexities of volatile types. Types X.a and X.M0.c are obviously volatile. Definitional types X.M0.d, u, and M1.v are also volatile because they are defined in terms of volatile types.

CHAPTER 6
ENTITY CALCULUS

In this module system, I will use an entity calculus, a refined version of the static representation of functor actions and static module content found in the SML/NJ compiler. The entity calculus provides a natural representation of the static content of modules, which I call *static entities*. The term entity refers to an extended[1] form of type constructors ($\mathbb{C}^{\lambda}$) and a static description of objects that may include type constructors such as structures and functors. In the module system, syntactic signatures alone are insufficient for expressing functor actions. The entity calculus is a small language for expression exactly that, the functor actions for a module. As such, it a a succint language that encompasses the fragment that is needed to supplement the syntactic signature language for supporting separate compilation.

Fig. 6.1 describes the entity calculus, comprised of entity expressions φ, entity declarations η, and static entities υ. The static entities are the entity calculus analogue to values in the lambda calculus. The representation relies on a set of denumerable variables, EntityVars, which represents unique names that can be thought of as the new variable names resulting from alpha-conversion.

Entity environments Υ are sequences of entity variable ρ to static entity υ mappings. The sequence is important because the semantics needs a deterministic way to search through the environment for atomic tycons, and to find the same one each time despite there being multiple occurrences of the same atomic tycon. The role of entity environments is two-fold. They give the current instantiation of a relativized tycon, thus enabling translation of a relativized tycon to an accurate semantic tycon. Furthermore, they permit the translation of a semantic tycon back to an entity path, a principal relativized tycon.

There are four forms of static entities, one corresponding to each of the following: sttructure, functor, atomic tycons, and definitional (derived) tycons. Structure and functor entities

1. Entity calculus tycons differ from syntactic types only in that the tycons may have been relativized, a process described in a later section, and prefixed universal quantifiers.

ρ	$\in$	EntityVars	entity variables
Υ	$::=$	$\emptyset_{ee} \mid \Upsilon[\rho \mapsto \upsilon]$	entity environment
R	$::=$	$\langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle$	structure entity
ψ	$::=$	$\langle \lambda\rho.\varphi; \Upsilon \rangle \mid \langle \lambda\rho.\Sigma; \Upsilon \rangle$	functor entity
υ	$::=$	$\psi \mid R \mid \tau^n \mid \mathbb{C}^\lambda$	static entity

Structure entity expression

φ	$::=$	$\vec{\rho}$	entity path
	$\mid$	$(\!|\eta|\!)$	entity declaration
	$\mid$	$\theta(\varphi)$	functor application
	$\mid$	$\mathbf{let}\ \eta\ \mathbf{in}\ \varphi$	local definition
θ	$::=$	$\vec{\rho} \mid \lambda\rho.\varphi \mid \lambda\rho.\Sigma$	functor entity expression
η	$::=$	$\bullet \mid [\rho =_{str} \varphi]\eta \mid [\rho =_{fct} \theta]\eta \mid [\rho =_{tyc} \mathsf{new}(n)]\eta$	entity declarations
	$\mid$	$[\rho =_{def} \vec{\rho}]\eta \mid [\rho =_{def} \mathbb{C}^\lambda]\eta$	

Figure 6.1: Static entities and entity expressions

are enclosed in $\langle\rangle$ brackets to distinguish them. A structure entity ($\langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle$) is comprised of a local entity environment Υ^{lcl} and a closure environment Υ^{clo} to define any free entity variables in the local one. A structure entity fully encodes the static content of a structure. There are two forms of functor entity. The standard form $\langle \lambda\rho.\varphi; \Upsilon \rangle$ encodes the functor action as a entity function $\lambda\rho.\varphi$ and the closure entity environment Υ for that function. The alternate form $\langle \lambda\rho.\Sigma; \Upsilon \rangle$ encodes functor entities corresponding to functors that are formal functor parameters where the only information known about the functor is the semantic signature Σ, described in chapter 7. Semantic signatures are essentially syntactic signatures where each of the static components (tycons, structures, and functors) are decorated with entity variables. That signature must also be closed by an entity environment Υ. Atomic tycons τ^n and relativized definitional tycons $\mathbb{C}^\lambda$ (defined in sec. 6.4.1) are the tycon entities.

Static entities are the values in the entity calculus. Structure entity expressions and functor entity expressions evaluate to structure entities and functor entities respectively. The entity path form in both structure and functor entity expressions is a reference to an entity corresponding to a previously bound structure entity expression (bound in an entity declaration). Structure entity expressions may also be an entity declaration enclosed by

$(\!|\cdot|\!)$ corresponding to the **struct** ... **end** base structure form, an application of a functor entity expression corresponding to functor application, and a let form. The let form is used to encode coercions induced by signature matching (see sec. 7.12). In addition to an entity path, functor entity expressions may be an entity function or a formal functor $\lambda\rho.\Sigma$.

Entity declarations bind an entity variable to structure entity expressions, functor entity expressions, a special form $\mathsf{new}(n)$ that corresponds to tycons, a type definition declaration, and a tycon alias that will be looked up in an entity environment. The new form produces a new atomic tycon τ^n when evaluated where n corresponds to the arity of the tycon.

Formal tycons from functor parameters are represented by their entity path, $\vec{\rho}$. Datatypes and other generative tycons are represented by the form $\mathsf{new}(n)$ which when evaluated will generate a fresh type constructor unequal to all those generated before it.

The entity calculus semantics is a compile-time semantics. The elaborator evaluates entity expressions to static entities as necessary during elaboration. The judgments for the entity expression evaluation are outlined below:

$\Upsilon \vdash \varphi \Downarrow_{str} \Upsilon'$	structure entity expression evaluation
$\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'$	entity declaration evaluation ($\Upsilon \subseteq \Upsilon'$)
$\Upsilon \vdash \theta \Downarrow_{fct} \psi$	functor entity expression evaluation

Υ, φ, ψ, and υ are the entity environment, resulting entity, and resulting entity environment respectively.

6.1 Entity declarations

The evaluation rules for entity declarations must accumulate the entity environment across declarations in order to support occurrences of entities in value specs within inline open signatures.

```
datatype d = A
structure M : sig type t val x : d * t end =
struct
  datatype t = B
  val x = (C, B)
end
```

Observe that the type name d is a nonlocal name that must be defined in a prior entity declaration. The entity environment for M must contain a binding for d.

6.2 Notation

I indicate extension of entity environments by juxtaposition, $\Upsilon[\rho \mapsto \upsilon]$. Entity environment concatenation of two entity environments Υ and Υ' is represented by juxtaposition, $\Upsilon\Upsilon'$. The notation for entity environment lookup is overloaded. $\Upsilon(\rho)$ looks up the static entity corresponding to ρ. The notation $\Upsilon(\rho_0\vec{\rho})$ looks up ρ_0 in Υ for a structure entity Υ' and then recursively computes $\Upsilon'(\vec{\rho})$. Let $EV(\cdot)$ denote the set of all entity variables in $\cdot$. The notation $dom(\Upsilon)$ denotes the set of entity variables in the domain of all mappings in Υ, including the nested ones.

Lemma 4 (Extended Entity Environment Lookup Terminates)
$\Upsilon(\vec{\rho})$ *terminates.*

Let R be a structure entity $\langle\Upsilon^{lcl}, \Upsilon^{clo}\rangle$ and $\vec{\rho}$ be an entity path defined in either Υ^{lcl} and Υ^{clo}. Then $R(\vec{\rho})$ is the static entity reached when following the entity path $\vec{\rho}$ in the Υ^{lcl} if $\vec{\rho}$ is defined in the local environment, or Υ^{clo} if $\vec{\rho}$ is not in the local environment,*i.e.*, $R(\vec{\rho}) = \Upsilon^{clo}\Upsilon^{lcl}(\vec{\rho})$.

$\Upsilon_1 \subseteq \Upsilon_2$ indicates all the bindings in Υ_1 are in Υ_2. The notation $R \subseteq \Upsilon$ means $\Upsilon^{clo}\Upsilon^{lcl} \subseteq \Upsilon$ such that $R = \langle\Upsilon^{lcl}, \Upsilon^{clo}\rangle$. Similarly, $\Upsilon \subseteq R$ indicates $\Upsilon \subseteq \Upsilon^{clo}\Upsilon^{lcl}$. Sometimes R occurs on the left hand side of a turnstile $\vdash$. This notation means the same as $\Upsilon^{clo}\Upsilon^{lcl} \vdash \ldots$.

6.3 Elaboration Modes

To properly handle the complexities in the module and type system, elaboration occurs in two simultaneous modes: the *direct mode* and the *entity compilation mode.* Direct mode is comprised of core language type checking and evaluation of entity expressions from the static information uncovered during elaboration. The main results of this mode are a static environment mapping symbolic names to static descriptions and an entity environment mapping entity variables to entities. The evaluation of entity expressions may produce new entities which will be folded into static and entity environments. Entity compilation mode constructs entity expressions which includes representations of functor actions.

Elaboration is entirely a compile-time process. The entity language sits on a different level from type, core, and module languages. The language is parallel to the semantic representation and syntactic language. Static entity compilation compiles core and module languages to typed abstract syntax and the entity language. Because the entity language is intended to describe functor actions which are not encoded in the typed abstract syntax, the overlap between the two is superficial. When the elaborator reaches a functor application in the source, the corresponding entity expression for the functor must be evaluated to produce the static information for the functor application result. To be more precise, entity expression evaluation yields realizations.

Static entity compilation cannot be considered a macro expansion phase because the target of compilation, the entity expressions, is in the source language. Whereas macro expansion is principally concerned with a program's dynamic code, static entity compilation encodes only static information such as types and the static content of functors. Neither is static entity compilation a type language macro expansion. Static entity compilation does not suffer from any problems associated with hygiene due to the entity variable mechanism, which will be elaborated in the rest of this chapter.

6.3.1 Direct Mode

Entity expressions evaluate to entity environments. The module system evaluates entity expressions when elaborating functor application (rule 7.39). Each entity expression represents the latent static computation in a structure expression. Once evaluated, the resultant structure entity provides a precise and complete (closed) description of the entities inside a

$\boxed{\Upsilon \vdash \varphi \Downarrow_{str} R}$

$$\frac{}{\Upsilon \vdash \vec{\rho} \Downarrow_{str} \Upsilon(\vec{\rho})} \tag{6.1}$$

$$\frac{\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'}{\Upsilon \vdash (\!|\eta|\!) \Downarrow_{str} \langle \Upsilon', \Upsilon \rangle} \tag{6.2}$$

$$\frac{\begin{array}{c}\Upsilon \vdash \theta \Downarrow_{fct} \langle \lambda \rho_{arg}.\varphi'; \Upsilon^{clo} \rangle \qquad \Upsilon \vdash \varphi \Downarrow_{str} R_{arg} \\ \Upsilon^{clo}[\rho_{arg} \mapsto R_{arg}] \vdash \varphi' \Downarrow_{str} R\end{array}}{\Upsilon \vdash \theta(\varphi) \Downarrow_{str} R} \tag{6.3}$$

$$\frac{\begin{array}{c}\Upsilon \vdash \theta \Downarrow_{fct} \langle \lambda \rho_{arg}.\Sigma; \Upsilon^{clo} \rangle \qquad \Upsilon \vdash \varphi \Downarrow_{str} R_{arg} \\ \Upsilon^{clo}[\rho_{arg} \mapsto R_{arg}] \vdash \Sigma \uparrow \Upsilon_{res}\end{array}}{\Upsilon \vdash \theta(\varphi) \Downarrow_{str} \langle \Upsilon_{res}, \Upsilon^{clo}[\rho_{arg} \mapsto R_{arg}] \rangle} \tag{6.4}$$

$$\frac{\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon' \qquad \Upsilon' \vdash \varphi \Downarrow_{str} R}{\Upsilon \vdash \textbf{let } \eta \textbf{ in } \varphi \Downarrow_{str} R} \tag{6.5}$$

Figure 6.2: Entity expression semantics

structure. Evaluation requires an entity environment as context to interpret entity paths in the structure entity expressions, functor entity expressions, and entity declarations.

Fig. 6.2 lists the rules for evaluating entity expressions. The key rule is the one for application (6.3). The functor entity expression is evaluated to an entity function. The argument structure entity expression is evaluated to an entity environment. The body of the entity function is then evaluated in an entity environment extended with the closure environment and the formal parameter entity variable mapped to the argument structure entity environment. Normal functors are not the only functor entity, so the evaluation rules must also account for the formal functor case. Rule 6.4 evaluates the functor entity expression to a functor entity corresponding to the formal functor case and also the structure entity expression to the structure entity. At that point, there are no more entity expressions to evaluate. The semantic signature Σ (*i.e.* the formal functor body signature) must be instantiated (a process that is covered in sec. 7.8). The rule then forms a structure entity using this instantiation (a structure entity itself) and the extended entity environment as a

$\boxed{\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'}$

$$\frac{}{\Upsilon \vdash \bullet \Downarrow_{decl} \Upsilon} \tag{6.6}$$

$$\frac{\Upsilon \vdash \varphi \Downarrow_{str} R \qquad \Upsilon[\rho \mapsto R] \vdash \eta \Downarrow_{decl} \Upsilon'}{\Upsilon \vdash \rho =_{str} \varphi, \eta \Downarrow_{decl} \Upsilon'} \tag{6.7}$$

$$\frac{\Upsilon \vdash \theta \Downarrow_{fct} \psi \qquad \Upsilon[\rho \mapsto \psi] \vdash \eta \Downarrow_{decl} \Upsilon'}{\Upsilon \vdash \rho =_{fct} \theta, \eta \Downarrow_{decl} \Upsilon'} \tag{6.8}$$

$$\frac{\Upsilon[\rho \mapsto \tau^n] \vdash \eta \Downarrow_{decl} \Upsilon' \qquad (\tau \text{ is fresh in } \Upsilon)}{\Upsilon \vdash \rho =_{tyc} \mathsf{new}(n), \eta \Downarrow_{decl} \Upsilon'} \tag{6.9}$$

$$\frac{\Upsilon[\rho \mapsto \mathbb{C}^\lambda] \vdash \eta \Downarrow_{decl} \Upsilon'}{\Upsilon \vdash [\rho =_{def} \mathbb{C}^\lambda], eta \Downarrow_{decl} \Upsilon'} \tag{6.10}$$

$$\frac{\Upsilon[\rho \mapsto \Upsilon(\vec{\rho})] \vdash \eta \Downarrow_{decl} \Upsilon'}{\Upsilon \vdash [\rho =_{def} \vec{\rho}], \eta \Downarrow_{decl} \Upsilon'} \tag{6.11}$$

Figure 6.3: Entity declaration semantics

closure. The instantiation part of the closure will be replaced by the actual functor closure when it is available.

Entity declarations must be evaluated in sequence. Fig. 6.3 formally defines how to evaluate entity declarations. Each tycon, structure, and functor entity declaration evaluates the respective tycon, structure entity expression, and functor entity expression to the corresponding entity. The subsequent entity declarations are evaluated in the context of an entity environment extended with a mapping for that entity. The entity environment context must be cumulative because entity declarations can mention entity variables declared in earlier declarations.

Rule 6.6 ensures that the resultant entity environment is cumulative. Rules 6.7 and 6.8 evaluate structure and functor entity declarations in the expected way. Rule 6.9 is the key rule that generates fresh atomic tycons with the appropriate arity n. When the $\rho =_{tyc} \mathsf{new}(n)$ entity declaration is embedded in the body of a functor entity, it produces a fresh atomic

$\boxed{\Upsilon \vdash \theta \Downarrow_{fct} \psi}$

$$\frac{}{\Upsilon \vdash \vec{\rho} \Downarrow_{fct} \Upsilon(\vec{\rho})} \tag{6.12}$$

$$\frac{}{\Upsilon \vdash \lambda\rho.\varphi \Downarrow_{fct} \langle \lambda\rho.\varphi; \Upsilon \rangle} \tag{6.13}$$

$$\frac{}{\Upsilon \vdash \lambda\rho.\Sigma \Downarrow_{fct} \langle \lambda\rho.\Sigma; \Upsilon \rangle} \tag{6.14}$$

Figure 6.4: Functor entity evaluation

tycon for ρ each time the functor entity is evaluated,*i.e.* also when this entity declaration is evaluated. This models exactly the behavior of generative datatype declarations in the presence of true higher-order semantics. Rule 6.10 yields a corresponding type definition entity environment binding. Rule 6.11 looks up an entity path (constructed during signature matching coercion rule 7.52) for the corresponding entity mapping.

Fig. 6.4 gives the rules for evaluating a functor entity expression. Rule 6.12 looks up entity paths in the usual way. The entity function is formed by combining an entity λ-abstraction with the current entity environment to form a closure (rule 6.13). Rule 6.14 similarly forms a closure for formal functors.

Lemma 5 (Entity evaluation terminates)

The following evaluation relations terminate (have a finite derivation tree):

1. $\Upsilon \vdash \varphi \Downarrow_{str} R$

2. $\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'$

3. $\Upsilon \vdash \theta \Downarrow_{fct} \psi$

Lemma 6

1. *If* $\Upsilon \vdash \varphi \Downarrow_{str} R$*, then* $\Upsilon \subseteq R$.

2. *If* $\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'$*, then* $\Upsilon \subseteq \Upsilon'$.

$\mathbb{C}^s$	$::=$	$\alpha \mid \vec{\rho}(\overrightarrow{\mathbb{C}^s})$	relativized monotypes
$\mathbb{C}^\lambda$	$::=$	$\lambda\vec{\alpha}.\mathbb{C}^s$	relativized tycons
$\mathbb{T}$	$::=$	$\mathsf{typ}(\mathbb{C}^s) \mid \forall\vec{\alpha}.\mathbb{C}^s$	relativized type expression

Figure 6.5: Relativized type system

3. If $\Upsilon \vdash \theta \Downarrow_{fct} \langle\theta; \Upsilon'\rangle$*, then* $\Upsilon \subseteq \Upsilon'$*.*

6.4 Relativization

Tycon names have a scoping policy much like other program identifiers. Tycon names can be shadowed by later type definitions. A tycon name may refer to a local tycon that is declared in the same scope as the occurrence or to a nonlocal one that is declared as a functor parameter or in another structure. Although signature declarations occur only at the top level, signature expressions may be embedded in functor signatures and signature ascriptions. When signatures occur inline, any type specifications may refer to both local and nonlocal tycons. Some tycon names may refer to volatile tycons which may be instantiated multiple times possibly to a generative type. When elaboration reaches these tycons we need a way to map it back to the corresponding entity path such that we reference the correct "up-to-date" volatile tycon. When the volatile tycon is "instantiated", the tycon must be updated. Relativization replaces fragile symbolic names with robust entity paths to the entity environment mapping the entity paths to corresponding to the current instantiation tycon.

6.4.1 Relativized Type System

The relativized type system ($\mathbb{C}^s, \mathbb{C}^\lambda$, and $\mathbb{T}$) replace all occurrences of concrete, atomic tycons with entity paths. The purpose of relativization and this type system is described in section 5.5. At this point, all three classes of tycons have been introduced: syntactic, semantic, and relativized. The semantics progressively translates syntactic tycons to the

$$\frac{\alpha \in \Delta}{\Upsilon, \Delta \vdash \alpha :: \Omega} \tag{6.15}$$

$$\frac{\Upsilon, \Delta \vdash \Upsilon(\vec{\rho}) :: \Omega^n \to \Omega \qquad \Upsilon, \Delta \vdash \mathbb{C}^s_i \; \forall \mathbb{C}^s_i \in \overrightarrow{\mathbb{C}^s}}{\Upsilon, \Delta \vdash \vec{\rho}(\overrightarrow{\mathbb{C}^s}) :: \Omega} \tag{6.16}$$

$$\frac{n = |\vec{\alpha}| \qquad \alpha_i \in \vec{\alpha} \; i \in [1, n] \qquad \Upsilon, [\alpha_1] \ldots [\alpha_n] \vdash \mathbb{C}^s :: \Omega}{\Upsilon \vdash \lambda \vec{\alpha}.\mathbb{C}^s :: \Omega^n \to \Omega} \tag{6.17}$$

$$\frac{\Upsilon, \emptyset_{knd} \vdash \mathbb{C}^s :: \Omega}{\Upsilon \vdash \mathsf{typ}(\mathbb{C}^s) :: \Omega} \tag{6.18}$$

$$\frac{n = |\vec{\alpha}| \qquad \alpha_i \in \vec{\alpha} \; i \in [1, n] \qquad \Upsilon, [\alpha_1] \ldots [\alpha_n] \vdash \mathbb{C}^s :: \Omega}{\Upsilon \vdash \forall \vec{\alpha}.\mathbb{C}^s :: \Omega} \tag{6.19}$$

Figure 6.6: Well-kinding of relativized monotypes, tycons, and types

latter two. The module system only evaluates semantic tycons. Such tycons exist in the static and entity environments. In contrast, relativized tycons are never evaluated directly. Instead, they must be translated into a semantic tycon for evaluation to take place. Relativized tycons exist in semantic signatures and static environment by way of embedded semantic signatures. Because a relativized tycon persists through multiple instantiations of the semantic tycon, it is more permanent, providing a key to the entity environment which contains the current instantiation, a particular semantic tycon.

Relativized monotypes, tycons, and types must also be well-kinded as defined in Fig. 6.6. The main difference between this well-kinding judgment and the other well-kinding judgments is that rule 6.16 uses the entity environment to obtain a well-kinded **semantic** tycon. The rest of the semantics is standard.

The rules in Figs. 6.7 and 6.8 define monotype and tycon equivalences and type expressions equivalences respectively. The module system semantics will require a notion of relativized monotype equivalence and relativized tycon equivalence, which are given by rules 6.20 and 6.21 respectively. Monotypes and tycons are equivalent under the interpretation of an entity environment when they evaluate to equivalent normal form semantic tycons modulo

$$\frac{\Upsilon(\mathbb{C}^s_1) \Downarrow_{mt} \mathfrak{C}^{nf}_1 \qquad \Upsilon(\mathbb{C}^s_2) \Downarrow_{mt} \mathfrak{C}^{nf}_2 \qquad \mathfrak{C}^{nf}_1 \equiv_\alpha \mathfrak{C}^{nf}_2}{\Upsilon \vdash \mathbb{C}^s_1 \equiv \mathbb{C}^s_2} \tag{6.20}$$

$$\frac{|\vec{\alpha}| = |\vec{\beta}| \qquad \Upsilon \vdash \mathbb{C}^s_1 \equiv \mathbb{C}^s_2}{\Upsilon \vdash \lambda\vec{\alpha}.\mathbb{C}^s_1 \equiv \lambda\vec{\beta}.\mathbb{C}^s_2} \tag{6.21}$$

Figure 6.7: Monotype and tycon equivalence

$\boxed{\Upsilon \vdash \mathbb{T}_1 \equiv \mathbb{T}_2}$

$$\frac{\Upsilon \vdash \mathbb{C}^s_1 \equiv \mathbb{C}^s_2}{\Upsilon \vdash \mathsf{typ}(\mathbb{C}^s_1) \equiv \mathsf{typ}(\mathbb{C}^s_2)} \tag{6.22}$$

$$\frac{|\vec{\alpha}| = |\vec{\beta}| \qquad \Upsilon \vdash \mathbb{C}^s_1 \equiv \mathbb{C}^s_2}{\Upsilon \vdash \forall\vec{\alpha}.\mathbb{C}^s_1 \equiv \forall\vec{\beta}.\mathbb{C}^s_2} \tag{6.23}$$

Figure 6.8: Type expression equivalence

α-renaming. Rules 6.22 and 6.23 describe equivalence for type expressions.

6.4.2 Role of Relativization

Relativization plays in a role in signature extraction:

1. Value bindings in the static environment must be relativized.
2. Definitional type binding in the static environment must be relativized in the inferred signature.

```
sig
  structure M : sig type t end
  type u = M.t
end
```

the type definition for u must be relativized because it refers not to a concrete type, but a type defined relative to a matching structure's realization.

Consider the following example:

```
structure A =
struct
  structure B = struct datatype u end
  functor F(X:sig type t end) =
    struct val a : X.t * B.u = ... end : sig val a : X.t * B.u end
end
```

The following are the semantic signatures for the example:

$$B \mapsto (\rho_B, \{u \mapsto (\rho_u, 0)\})$$

$$X \mapsto (\rho_X, \{t \mapsto (\rho_t, 0)\})$$

In the type for a, the tycon for X.t must be relativized to $[\rho_X, \rho_t]$ and for B.u to $[\rho_B, \rho_u]$.

When inferring the signature from the body of F, we have the formal tycon X.t in the static environment. When inferring the signature from F's body, there are two static environments, the static environment produced by elaborating F's body and the contextual static environment when entering into the body.

Static environment for F's body: $[a \mapsto \tau_t^0 * \tau_u^0]$

Static environment for the context: $[B \mapsto (\rho_B, \{u \mapsto (\rho_u, \tau_u^0)\}, R_B)]$
$[X \mapsto (\rho_X, t \mapsto (\rho_t, \tau_t^0), R_X)]$

R_B and R_X are the structure entities for B and X respectively.

```
structure A =
struct [$\rho_A$]
  structure B = struct [$\rho_B$] datatype t [$\rho_t$] = K end
  functor F(X [$\rho_X$]:sig type u [$\rho_u$] val a : u end) =
  struct
    val b : X.u * B.t = (X.a, B.K)
  end : sig val b : X.u * B.t end
end
```

The relativized tycon, an entity path, can be computed from the static environment at the point of elaboration. For example, when the elaborator reaches the body of the above functor, the static environment is the following:

$[B \mapsto (\rho_B, \langle\{t \mapsto (\rho_t, 0)\}, \emptyset_{ee}\rangle)][X \mapsto (\rho_X, \langle\{u \mapsto (\rho_u, 0), a \mapsto [\rho_u]\}, \emptyset_{ee}\rangle)]$

The tycon for X.u can be relativized by looking up the static environment and accumulating the entity path on the path to the tycon, *i.e.*, $[\rho_X, \rho_u]$.

When elaborating `a` in the functor parameter signature, the type, a generative formal tycon `u` is relativized to an entity path $[\rho_u]$.

The elaborator relativizes two kinds of tycons, generative tycons and definitional tycons.

After instantiation of the parameter signature during functor elaboration, any newly generated tycons, structures, and functors must be mapped back to the corresponding entity variable in the parameter signature. The elaborator uses these reverse mappings when elaborating the functor result signature and functor body.

This after-the-fact entity to entity variable mapping is also needed for structure bindings to create the mappings for coerced entities produced during signature matching. Unlike in the functor case, mappings for the entities in the structure binding and the structure entity itself must be in scope for the rest of the program.

A policy of fully reducing a tycon before relativizing greatly simplifies the process of relativization. Whereas definitional tycons can certainly have multiple distinct declarations, each primary volatile tycon has a unique declaration site and thus entry in the entity environment from which the elaborator can back out the unique entity path leading to that tycon. The definition of relativization uses an operation of the entity environment Υ^{-1} to calculate the entity path. Although Υ technically does not admit a retraction because multiple entity paths may map to the same atomic tycon, this operation is intuitively similar to a retraction of Υ lookup.

Υ is a sequence of bindings of entity variables to entities. Let $(\sqsubseteq_{ee}, \Upsilon)$ be the total ordering of the bindings in Υ according to the ordering in the sequence. For example, for $\Upsilon = [\rho_0 \mapsto \tau^0][\rho_1 \mapsto \tau^0][\rho_2 \mapsto \tau^0]$, $[\rho_0 \mapsto \tau^0] \sqsubseteq_{ee} [\rho_2 \mapsto \tau^0]$.

Definition 1

$\Upsilon^{-1}(\tau^n)$ *is defined to be the entity path associated with the first occurrence (*i.e.*, least binding according to* $\sqsubseteq_{ee}$*) of* τ^n *when treating the entity environment* Υ *as a sequence of bindings.*

$\boxed{\Gamma, \Upsilon \vdash C^s \Rightarrow^{mt}_{rel} \mathbb{C}^s}$

$$\Gamma, \Upsilon \vdash \alpha \Rightarrow^{mt}_{rel} \alpha \tag{6.24}$$

$$\frac{\vec{\rho} = \mathsf{entpath}(\Gamma, \Upsilon, p) \qquad \Gamma, \Upsilon \vdash \overrightarrow{C^s} \Rightarrow^{mt}_{rel} \overrightarrow{\mathbb{C}^s}}{\Gamma, \Upsilon \vdash p(\overrightarrow{C^s}) \Rightarrow^{mt}_{rel} \vec{\rho}(\overrightarrow{\mathbb{C}^s})} \tag{6.25}$$

$\boxed{\Gamma, \Upsilon \vdash C^\lambda \Rightarrow^{tyc}_{rel} \mathbb{C}^\lambda}$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow^{mt}_{rel} \mathbb{C}^s}{\Gamma, \Upsilon \vdash \lambda\vec{\alpha}.C^s \Rightarrow^{tyc}_{rel} \lambda\vec{\alpha}.\mathbb{C}^s} \tag{6.26}$$

$\boxed{\Gamma, \Upsilon \vdash T \Rightarrow^{te}_{rel} \mathbb{T}}$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow^{mt}_{rel} \mathbb{C}^s}{\Gamma, \Upsilon \vdash \mathsf{typ}(C^s) \Rightarrow^{te}_{rel} \mathsf{typ}(\mathbb{C}^s)} \tag{6.27}$$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow^{mt}_{rel} \mathbb{C}^s}{\Gamma, \Upsilon \vdash \forall\vec{\alpha}.C^s \Rightarrow^{te}_{rel} \forall\vec{\alpha}.\mathbb{C}^s} \tag{6.28}$$

$$\mathsf{entpath}(\Gamma, \Upsilon, p) = \begin{cases} \Upsilon^{-1}(\Gamma(x)) & \text{if } p = x \text{ where } x \text{ is singleton} \\ \Gamma_{ep}(p) & \text{o.w.} \end{cases} \tag{6.29}$$

Figure 6.9: Relativization

Example 1

Let $\Upsilon = [\rho_0 \mapsto [\rho_1 \mapsto \tau_1^0]][\rho_2 \mapsto \tau_1^0]$. $\Upsilon^{-1}(\tau_1^0) = \rho_0\rho_1$.

6.4.3 Semantics

Fig. 6.9 formally defines the relativization of syntactic monotypes, tycons, and type expressions. The relativization judgments require three contexts: a static environment Γ to symbolic paths in tycon applications and an entity environment Υ to interpret atomic tycons associated with those symbolic paths.

The entpath auxiliary function first looks up the symbolic path in the static environment and relativize the resultant tycon using the entity environment. Paths in the signature are local whereas paths in the static environment are nonlocal. In either case, an entity path for the tycon is constructed. Otherwise, if p is a singleton x, then $\Gamma(x) = \tau^n$ and the entity path is $\Upsilon^{-1}(\tau^n)$. Finally, if p is nonsingleton, then the entity path can be calculated as $\Gamma_{ep}(p)$ (extracts the entity path for the tycon whose symbolic path is p, formally defined in chapter 7).

Relativization is used in signature expression elaboration of where type (7.7), type definition spec elaboration (7.12), val spec elaboration (7.13), and signature extraction (7.44,7.45).

Lemma 7

If $\Gamma, \Delta \vdash x(\vec{C}^s) : \Omega$ and x is singleton, then $\Gamma(x) = \tau^n$.

The well-kinding of relativized and semantic tycons implies a well-formedness of entity environments.

Definition 2 (Well-formedness of entity environments)

An entity environment Υ is well-formed if for all $\mathbb{C}^\lambda \in \Upsilon$, $\Upsilon \vdash \mathbb{C}^\lambda :: \Omega^n \to \Omega$.

Semantic tycons do not play a role in this definition because the only semantic tycons in Υ are of the form τ^n which is always well-kinded.

CHAPTER 7
ELABORATION SEMANTICS

7.1 Introduction

In a first-order module system, functor application can propagate types in the parameter to the functor result. A liberal type propagation is essential for compilation of efficient module code [66]. More importantly, type propagation is necessary to make some sound programs typecheck. A liberal policy also reflects the dependent type structure of functors. Types in the functor parameter may also be generative, as in the case of datatypes and types in opaquely ascribed substructures. These generative types have to be faithfully propagated during functor application. Higher-order functors complicate type propagation because types in the functor result may be computed from multiple sources. As in the first-order case, types may be locally defined in the result or propagated from the parameter. Applications of formal functors (*i.e.*, functors in the functor parameter) in the functor result should also propagates types. Collectively, generation of fresh instances of generative types and formal functor applications are called *functor actions*, which must be propagated during elaboration of functor application to maximize type propagation.

Similar to other accounts of module system designs, I present the true higher-order module system (THO) as an elaboration semantics. The syntactic module language is elaborated to a semantic representation. Unlike prior accounts, this semantics will use an entity calculus that fully encodes the functor actions in a program. The entity calculus is a third level of representation distinct from the syntactic and semantic representations. It assumes each type is given a unique name, an entity variable. Unlike other internal languages, the entity calculus only plays a role in elaboration. Entity calculus terms are translated to IL types by the time code reaches the optimization stages of a compiler.

Elaboration accomplishes the following:

1. Produces a static environment mapping variables to static descriptions of values, types,

structures, functors, and signatures

2. Produces a typed abstract syntax

3. Produces entity expressions fully describing functor actions

4. Typechecks the program

For simplicity, this chapter's discussion will omit (2), which is irrelevant to the static semantics of the module system.

7.2 Elaboration Representations

Elaboration is mainly concerned with the construction of a static environment. Secondarily, the elaborator produces typed abstract syntax, an entity environment, and an entity expression. The static environment contains the visible static information in the elaborated program. The static information comes in the form of semantic representations of signatures and type information. A semantic representation of a signature is a syntactic signature where the static entities are decorated with entity variables.

```
sig
    type t [$\rho_t$]
    type u = int
    structure M [$\rho_M$] : sig type s [$\rho_s$] end
end
```

The entity environment contains static information that may have been occulted by inexplicitness and the type and functor actions describing the production of new static information during functor application. Much of elaboration is concerned with constructing the correct entity environment and interpreting entity paths using them.

7.3 Semantic Objects

A semantic signature is a sequence of signature elements. Signature elements can be volatile (s^p), definitional ($\mathbb{C}^\lambda$), or a value spec ($\mathbb{T}$). Note that all signature spec elements must be

s^p	$::=$	$arity \mid \Sigma \mid \Sigma^f$	primary spec
Σ	$::=$	$\emptyset_{sig} \mid [x \mapsto (\rho, s^p)]\Sigma \mid [x \mapsto \mathbb{C}^\lambda]\Sigma \mid [x \mapsto \mathbb{T}]\Sigma$	semantic signature
M	$::=$	(Σ, R)	full signature
Σ^f	$::=$	$\Pi\rho : \Sigma.\Sigma$	functor signature
F	$::=$	(Σ^f, ψ)	full functor signature
γ	$::=$	$\mathfrak{T} \mid \mathfrak{C}^\lambda \mid \Sigma \mid \Sigma^f \mid (\rho, M) \mid (\rho, F)$	static binding
Γ	$::=$	$\emptyset_{se} \mid \Gamma[x \mapsto \gamma]$	static type environment

The static bindings for structures and functors include the entity variable to permit direct construction of entity paths during signature extraction, structure path, and functor path elaboration.

Figure 7.1: Semantic representations

relativized. Nonvolatile elements including definitional tycons $\mathbb{C}^\lambda$ and value specs $\mathbb{T}$ need only specify their fixed static description. Volatile primary specs (s^p) such the ones for as open tycons *arity*, structures Σ, and functors Σ^f may be instantiated by signature matching and thus must have a corresponding binding in the entity environment indexed by entity variables.

A *full signature* M gives a full semantic description of a structure. It is comprised of a semantic signature Σ and a structure entity R that interprets all the open specifications in Σ. A semantic functor signature Σ^f binds an entity variable for the functor parameter ρ in the functor result signature. A full functor signature F is comprised a semantic functor signature Σ^f and a functor entity ψ that when evaluated will interpret all the open specifications in the functor result signature.

γ is a static binding, to which the static environment Γ maps identifiers. For value identifiers, the static description is a semantic type expression $\mathfrak{T}$. For tycon definitions, the static description is $\mathfrak{C}^\lambda$. There is no static binding for open tycon because all tycons are either defined or instantiated in the static environment. For signature and functor signature identifiers, the static descriptions are semantic signatures (Σ) and semantic functor signature (Σ^f). For structure and functor identifiers, the static descriptions are full signatures and full functor signatures respectively. Some static binding (namely for structures and functors) consist of the full signature or full functor signature coupled with the entity variable for that

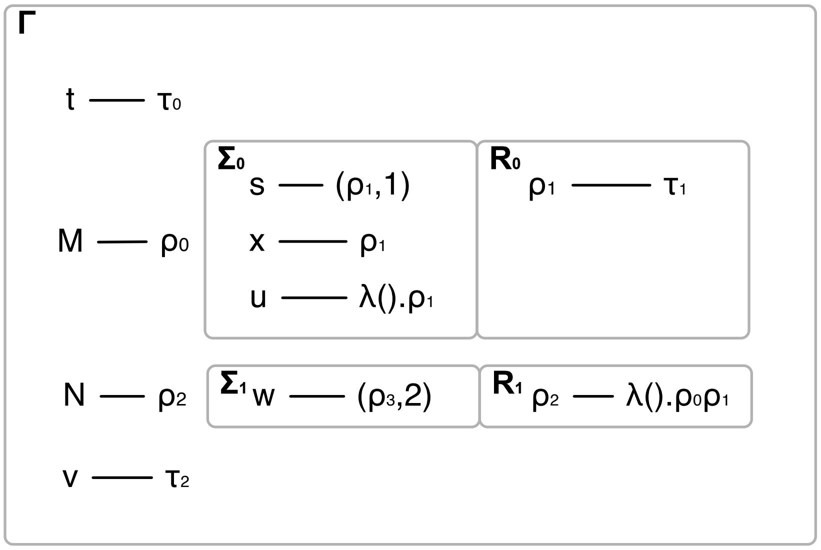


Figure 7.2: Schematic of static environment

structure or functor. The entity variable is used to construct entity paths during signature extraction and elaboration.

Tycons in signatures and in static environments differ in that tycons in the former will have been relativized during signature elaboration or extraction. Unlike static environments that may be extended throughout the elaboration process, signatures do not change. Hence to ensure that volatile tycons in signatures have to an appropriate interpretation, they must be relativized with respect to the entity environment.

In Fig. 7.2, the static environment depicted exhibits the heterogeneous form of the definition. The first layer contains semantic tycons, those types directly bound by the static environment. The second layer consist of the tycons s, u, and w bound in semantic signatures Σ_0 and Σ_1 embedded inside of the static environment. These semantic signatures contain relativized tycons. The full signature for structures also include the structure entity expressions R_0 and R_1 which define the instantiation of the open tycons in the corresponding semantic signature.

7.4 Notation

If p is a symbolic path and Σ is a semantic signature, then $p \in \Sigma$ means that following p in Σ reaches a spec. If p is a singleton, then Σ must contain a binding $p \mapsto \mathit{spec}$. Otherwise, if p is nonsingleton, then $p = xp'$ such that x is a symbol and p' is a symbolic path such that Σ contains a binding $x \mapsto (\rho, \Sigma')$ and $p' \in \Sigma$.

$\Sigma(p)$ is the spec reached by following p in Σ. The notation $\Sigma_{ep}(p)$ denotes the entity path associated with the signature spec referred to by symbolic path p. The entity path is comprised of the entity variables on the path to the element. For example, if the signature is $\Sigma = [A \mapsto (\rho_A, [B \mapsto (\rho_B, [t \mapsto (\rho_t, 0)])])]$, then $\Sigma_{ep}(A.B.t) = \rho_A \rho_B \rho_t$.

Static environment lookup is expressed as $\Gamma(p)$ where p is the symbolic path to the static binding. If p is a singleton, then the static environment lookup would simply return the corresponding static binding. As will be demonstrated below, static environment lookup sometimes requires an entity environment lookup. This is the case when the component being looked up is a definitional tycon in a nonsingleton path. Depending on what kind of entity p refers to, the lookup is handled differently:

full signature or a full functor signature $\Gamma(p)$ denotes a pair $(\vec{\rho}, M)$ such that $\vec{\rho}$ is the entity path to the structure static binding and M is the full signature of that structure. For example, for $\Gamma = [A \mapsto (\rho_A, ([B \mapsto (\rho_B, \Sigma_B)], R_A))]$, $\Gamma(A.B) = (\rho_A \rho_B, (\Sigma_B, R_A(\rho_A \rho_B)))$.

semantic type expression $\Gamma(x.p) = \Sigma(p)$ such that $[x \mapsto (\rho_x, \Sigma)] \in \Gamma$.

definitional tycon $\mathbb{C}^\lambda$ In semantic signatures, definitional tycons are relativized. In order to make $\Gamma(x.p)$ consistent for singleton and nonsingleton path lookup, the relativized tycon must be interpreted using an entity environment. This is because $\Gamma(t)$ for a singleton t results in a semantic tycon $\mathfrak{C}^\lambda$. Let $\Gamma(x.p) = R_x(\Sigma(p))$ such that $[x \mapsto (\rho_x, (\Sigma, R_x))] \in \Gamma$. The notation $R_x(\cdot)$ means use the entity environment $\Upsilon^{clo}\Upsilon^{lcl}$ to interpret any entity paths (*i.e.*, relativized tycons) in some $\cdot$, which could be a type expression, a tycon, *etc*, such that $R_x = \langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle$. The result of this process should be a semantic tycon $\mathfrak{C}^\lambda$. If p is a singleton, it will default to the standard lookup of Γ.

7.5 Elaboration: Entity Compilation Mode

Besides the direct elaboration mode for the evaluation of entity expressions, described in chapter 6, the elaborator must have an entity compilation mode that produces the entity expressions and semantic representations of the modules from the syntax. The process of the compilation mode is the bulk of the elaboration semantics. Entity expressions for both structures and functors are compiled from the raw implicitly typed abstract syntax trees and the contextual environments used during elaboration. The compilation mode elaboration is the subject of the remainder of this chapter.

The compilation mode elaboration typechecks and compiles abstract syntax trees to entity expressions. All of elaboration takes place under a static environment Γ that maps program variables to semantic tycons, type expressions, structure full signatures, full functor signatures, and semantic signatures. Recall that semantic signatures pair all static primary components with entity variables and replace all occurrences of such primaries with a principal entity path to the corresponding static entity in the entity environment, which is calculated by relativization. Values are classified by their type. Structures are described by their full signature consisting of a structure entity and a signature. A functor static description is of a full functor signature, comprised of a functor entity expression (*i.e.*, a λ-expression) and a functor signature. The main elaboration judgments are the following:

$\Gamma \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s$	monotype elaboration
$\Gamma \vdash C^\lambda \Rightarrow_{tyc} \mathfrak{C}^\lambda$	tycon elaboration
$\Gamma \vdash T \Rightarrow_{te} \mathfrak{T}$	type elaboration

The above judgments produce semantic monotypes/tycons/type expressions from syntactic ones. Tycon elaboration's sole role is during elaboration of tycon declarations (rule 7.33).

$\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow^{mt}_{rel} \mathbb{C}^s$	monotype relativization
$\Gamma, \Upsilon, \Sigma \vdash C^\lambda \Rightarrow^{tyc}_{rel} \mathbb{C}^\lambda$	tycon relativization
$\Gamma, \Upsilon, \Sigma \vdash T \Rightarrow^{te}_{rel} \mathbb{T}$	type expression relativization

The relativization judgment produce relativized tycons from syntactic ones. The entity environment Υ is used to relativize the nonlocally defined tycons. The signature Σ is used to construct relativized entity paths for locally defined tycons. Signature extraction and elaboration rely on relativization.

$\Gamma, \Upsilon, \Sigma \vdash sigexp \Rightarrow_{sig} \Sigma'$	signature elaboration
$\Gamma, \Upsilon, \Sigma \vdash fsgexp \Rightarrow_{fsg} \Sigma^f$	functor signature elaboration
$\Upsilon^{clo}, \Upsilon^{lcl} \vdash \Sigma \uparrow \Upsilon^{lcl}$	signature instantiation
$\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$	module declaration elaboration
$\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} (M, \varphi)$	structure expression elaboration
$\Upsilon \vdash \Gamma \hookrightarrow \Sigma$	signature extraction
$\Upsilon \vdash (M, \varphi) : \Sigma \Rightarrow_{match} (M_c, \varphi_c)$	signature matching
$\Upsilon \vdash F \preceq \Sigma^f \Rightarrow_{fsgmtch} (\psi_c, \theta_c)$	functor signature matching

7.6 Type Constructors

The syntactic type language is a subset of the semantic tycon language. Before proceeding, the elaborator must first elaborate the syntactic types to semantic tycons. The elaboration is simply a syntax-directed recursive translation.

The syntactic type language elaborates to the semantic type language. Elaboration interprets the symbolic paths in tycon application $p(\vec{C^s})$ and evaluates the application by standard β-reduction. Note that tycon elaboration $\Rightarrow_{tyc}$ elaborates and therefore reduces under λ-abstraction. This ensures that tycon definitions are fully reduced. Because λs cannot be nested, this reduction is well-defined and normalizing. The normal form is defined as $\mathfrak{C}^{nf}$ in fig. 4.3.

It is noteworthy that the monotype, tycon, and type expression elaboration judgments

$\boxed{\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s}$

$$\Gamma, \Upsilon \vdash \alpha \Rightarrow_{mt} \alpha \tag{7.1}$$

$$\frac{\Gamma, \Upsilon \vdash C^s_i \Rightarrow_{mt} \mathfrak{C}^s_i \forall i \in [1, |\vec{C^s}|] \qquad ((\Gamma, \Upsilon)(p))(\vec{\mathfrak{C}^s}) \Downarrow_{tyc} \mathfrak{C}'^s}{\Gamma, \Upsilon \vdash p(\vec{C^s}) \Rightarrow_{mt} \mathfrak{C}'^s} \tag{7.2}$$

$\boxed{\Gamma, \Upsilon \vdash C^\lambda \Rightarrow_{tyc} \mathfrak{C}^\lambda}$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s}{\Gamma, \Upsilon \vdash \lambda\vec{\alpha}.C^s \Rightarrow_{tyc} \lambda\vec{\alpha}.\mathfrak{C}^s} \tag{7.3}$$

$\boxed{\Gamma, \Upsilon \vdash T \Rightarrow_{te} \mathfrak{T}}$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s}{\Gamma, \Upsilon \vdash \mathsf{typ}(C^s) \Rightarrow_{te} \mathsf{typ}(\mathfrak{C}^s)} \tag{7.4}$$

$$\frac{\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s}{\Gamma, \Upsilon \vdash \forall\vec{\alpha}.C^s \Rightarrow_{te} \forall\vec{\alpha}.\mathfrak{C}^s} \tag{7.5}$$

Figure 7.3: Monotype and tycon elaboration

all require an entity environment Υ as context. This is because the monotype elaboration rule 7.2 attempts to lookup a symbolic path in a static environment. If the symbolic path were singleton, then the entity environment is unnecessary. If, however, the path were nonsingleton, then the target tycon must be embedded in some semantic signature, and all tycons in semantic signatures are relativized. Therefore, an entity environment is needed to interpret the relativized tycons.

The type elaboration judgments preserve the well-kinding defined in chapter 4.

Lemma 8 (Monotype Elaboration Preserves Kinding)

If $\Gamma, \emptyset_{knds} \vdash C^s : \Omega$ *and* $\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s$*, then* $\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega$.

Lemma 9 (Tycon Elaboration Preserves Kinding)

If $\Gamma, \emptyset_{knds} \vdash C^\lambda : \Omega^n \Rightarrow \Omega$ *and* $\Gamma, \Upsilon \vdash C^\lambda \Rightarrow_{tyc} \mathfrak{C}^\lambda$*, then* $\emptyset_{knds} \vdash \mathfrak{C}^\lambda : \Omega^n \Rightarrow \Omega$.

$\boxed{\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp} \Rightarrow_{sig} \Sigma'}$

$$\frac{}{\Gamma, \Upsilon, \Sigma \vdash x \Rightarrow_{sig} \Gamma(x)} \tag{7.6}$$

$$\frac{\begin{array}{c}\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp} \Rightarrow_{sig} \Sigma' \qquad \Sigma'(p) = (\rho, n) \\ \Gamma, \Upsilon, \Sigma\Sigma' \vdash C^{\lambda} \Rightarrow_{rel}^{tyc} \mathbb{C}^{\lambda} \qquad |\mathbb{C}^{\lambda}| = n\end{array}}{\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp}\ \textbf{where type}\ p = C^{\lambda} \Rightarrow_{sig} \mathsf{rebind}(p, \mathbb{C}^{\lambda}, \Sigma')} \tag{7.7}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash \mathit{specs} \Rightarrow_{specs} \Sigma'}{\Gamma, \Upsilon, \Sigma \vdash \textbf{sig}\ \mathit{specs}\ \textbf{end} \Rightarrow_{sig} \Sigma'} \tag{7.8}$$

$\mathsf{rebind}(p, \mathbb{C}^{\lambda}, \Sigma)$ replaces the binding $[x \mapsto (\rho, n)]$ in Σ with $[x \mapsto \mathbb{C}^{\lambda}]$ where p ends in x.

$\boxed{\Gamma, \Upsilon, \Sigma \vdash \mathit{specs} \Rightarrow_{specs} \Sigma}$

$$\frac{}{\Gamma, \Upsilon, \Sigma \vdash \emptyset_{specs} \Rightarrow_{specs} \emptyset_{sig}} \tag{7.9}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash \mathit{spec} \Rightarrow_{spec} \Sigma' \qquad \Gamma, \Upsilon, \Sigma\Sigma' \vdash \mathit{specs} \Rightarrow_{specs} \Sigma''}{\Gamma, \Upsilon, \Sigma \vdash \mathit{spec}, \mathit{specs} \Rightarrow_{specs} \Sigma'\Sigma''} \tag{7.10}$$

Figure 7.4: Signature elaboration

7.7 Signature Elaboration

The role of signature elaboration is to relativize tycons, to reduce where type clauses to type definitions, and to decorate all specs corresponding to static entities with entity variables. The elaboration judgment uses static and entity environment to interpret symbolic paths in tycon definitions. However, not all symbolic paths are bound in the static environment. For example, in the following signature, the type definition for u mentions tycon t which does not have a corresponding structure entity at this point, hence tycon t will not be in static and entity environment.

sig type t **type** u = t **end**

Rule 7.10 composes the result of elaborating the individual specs. Elaborating subsequent specs in the same signature will require a context (a signature) that includes all the previously elaborated specs Σ and the newly elaborated spec Σ'.

$\boxed{\Gamma, \Upsilon, \Sigma \vdash \mathit{spec} \Rightarrow_{\mathit{spec}} \Sigma'}$

$$\frac{(\rho \text{ fresh in } \Gamma \text{ and } \Upsilon)}{\Gamma, \Upsilon, \Sigma \vdash \textbf{type}\ \vec{\alpha}\ t \Rightarrow_{\mathit{spec}} [t \mapsto (\rho, |\vec{\alpha}|)]} \tag{7.11}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash C^{\lambda} \Rightarrow_{\mathit{rel}}^{\mathit{tyc}} \mathbb{C}^{\lambda}}{\Gamma, \Upsilon, \Sigma \vdash \textbf{type}\ t = C^{\lambda} \Rightarrow_{\mathit{spec}} [t \mapsto \mathbb{C}^{\lambda}]} \tag{7.12}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash T \Rightarrow_{\mathit{rel}}^{\mathit{tyc}} \mathbb{T}}{\Gamma, \Upsilon, \Sigma \vdash \textbf{val}\ x : T \Rightarrow_{\mathit{spec}} [x \mapsto \mathbb{T}]} \tag{7.13}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp} \Rightarrow_{\mathit{sig}} \Sigma' \qquad (\rho \text{ fresh in } \Gamma \text{ and } \Upsilon)}{\Gamma, \Upsilon, \Sigma \vdash \textbf{structure}\ x : \mathit{sigexp} \Rightarrow_{\mathit{spec}} [x \mapsto (\rho, \Sigma')]} \tag{7.14}$$

$$\frac{\begin{array}{c}\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp}_1 \Rightarrow_{\mathit{sig}} \Sigma_1 \qquad (\rho_x \text{ and } \rho \text{ fresh in } \Gamma \text{ and } \Upsilon) \\ \Gamma, \Upsilon, \Sigma[X \mapsto (\rho_x, \Sigma_1)] \vdash \mathit{sigexp}_2 \Rightarrow_{\mathit{sig}} \Sigma_2\end{array}}{\Gamma, \Upsilon, \Sigma \vdash \textbf{functor}\ f(X : \mathit{sigexp}_1) : \mathit{sigexp}_2 \Rightarrow_{\mathit{spec}} [f \mapsto (\rho, \Pi\rho_x : \Sigma_1.\Sigma_2)]} \tag{7.15}$$

Figure 7.5: Signature spec elaboration

Signature spec elaboration produces entity variables for each static component (*i.e.*, open type specs, structure spec, and functor spec) and puts them in the resultant semantic signature. Rules 7.12 and 7.13 relativize syntactic tycons and type expressions respectively. Note that signature specs may contain relativized tycons, entity paths, referencing entities declared earlier in the same semantic signature. In order to fully relativize such a symbolic path, the entpath metafunction needs to first lookup the symbolic path in the preceding signature specs. These preceding signature specs (a semantic signature Σ itself) help construct an entity path for symbolic paths not defined in the static environment. Fig. 7.6 shows the revised relativization judgment that threads the semantic signature Σ through and looks it up in rule 7.21. The rule first looks up symbolic path in the signature Σ which contains all the preceding specs in the current signature and only if that fails does it look up in the static environment.

In particular, the signature Σ is needed when relativizing specs in a signature Σ' that contain symbolic paths defined within the same Σ. Latter specs can contain symbolic paths declared in earlier specs within the same signature. Because the signature is not yet fully

$\boxed{\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow_{rel}^{mt} \mathbb{C}^s}$

$$\Gamma, \Upsilon, \Sigma \vdash \alpha \Rightarrow_{rel}^{mt} \alpha \tag{7.16}$$

$$\frac{\vec{\rho} = \mathsf{entpath}(\Gamma, \Upsilon, \Sigma, p) \qquad \Gamma, \Upsilon, \Sigma \vdash \overrightarrow{C^s} \Rightarrow_{rel}^{mt} \overrightarrow{\mathbb{C}^s}}{\Gamma, \Upsilon, \Sigma \vdash p(\overrightarrow{C^s}) \Rightarrow_{rel}^{mt} \vec{\rho}(\overrightarrow{\mathbb{C}^s})} \tag{7.17}$$

$\boxed{\Gamma, \Upsilon, \Sigma \vdash C^\lambda \Rightarrow_{rel}^{tyc} \mathbb{C}^\lambda}$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow_{rel}^{mt} \mathbb{C}^s}{\Gamma, \Upsilon, \Sigma \vdash \lambda\vec{\alpha}.C^s \Rightarrow_{rel}^{tyc} \lambda\vec{\alpha}.\mathbb{C}^s} \tag{7.18}$$

$\boxed{\Gamma, \Upsilon, \Sigma \vdash T \Rightarrow_{rel}^{te} \mathbb{T}}$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow_{rel}^{mt} \mathbb{C}^s}{\Gamma, \Upsilon, \Sigma \vdash \mathsf{typ}(C^s) \Rightarrow_{rel}^{te} \mathsf{typ}(\mathbb{C}^s)} \tag{7.19}$$

$$\frac{\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow_{rel}^{mt} \mathbb{C}^s}{\Gamma, \Upsilon, \Sigma \vdash \forall\vec{\alpha}.C^s \Rightarrow_{rel}^{te} \forall\vec{\alpha}.\mathbb{C}^s} \tag{7.20}$$

$$\mathsf{entpath}(\Gamma, \Upsilon, \Sigma, p) = \begin{cases} \Sigma_{ep}(p) & \text{if } p \in \Sigma \\ \Upsilon^{-1}(\Gamma(x)) & \text{if } p = x \text{ where } x \text{ is singleton} \\ \Gamma_{ep}(p) & \text{o.w.} \end{cases} \tag{7.21}$$

Figure 7.6: Relativization with signature spec lookup

$\boxed{\Upsilon^{clo}, \Upsilon_1^{lcl} \vdash \Sigma \uparrow \Upsilon_2^{lcl}}$

$$\frac{}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash \emptyset_{sig} \uparrow \emptyset_{ee}} \tag{7.22}$$

$$\frac{\Upsilon^{clo}, \Upsilon^{lcl} \vdash \Sigma \uparrow \Upsilon'}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash [x \mapsto \mathbb{C}^\lambda]\Sigma \uparrow \Upsilon'} \tag{7.23}$$

$$\frac{\Upsilon^{clo}, \Upsilon^{lcl} \vdash \Sigma \uparrow \Upsilon'}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash [x \mapsto \mathbb{T}]\Sigma \uparrow \Upsilon'} \tag{7.24}$$

$$\frac{\Upsilon^{clo}, \Upsilon^{lcl}[\rho \mapsto \tau^n] \vdash \Sigma \uparrow \Upsilon' \qquad (\tau \text{ is fresh in } \Upsilon^{clo} \text{ and } \Upsilon^{lcl})}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash [x \mapsto (\rho, n)]\Sigma \uparrow [\rho \mapsto \tau^n]\Upsilon'} \tag{7.25}$$

$$\frac{\Upsilon^{clo}, \Upsilon^{lcl} \vdash \Sigma' \uparrow \Upsilon' \qquad \Upsilon^{clo}, \Upsilon^{lcl}[\rho \mapsto \langle \Upsilon', \Upsilon^{clo}\Upsilon^{lcl}\rangle] \vdash \Sigma \uparrow \Upsilon''}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash [x \mapsto (\rho, \Sigma')]\Sigma \uparrow [\rho \mapsto \langle \Upsilon', \Upsilon^{clo}\Upsilon^{lcl}\rangle]\Upsilon''} \tag{7.26}$$

$$\frac{\Upsilon^{clo}, \Upsilon^{lcl}[\rho \mapsto \langle \lambda\rho_x.\Sigma_r; \Upsilon^{clo}\Upsilon^{lcl}\rangle] \vdash \Sigma \uparrow \Upsilon'}{\Upsilon^{clo}, \Upsilon^{lcl} \vdash [x \mapsto (\rho, \Pi\rho_x : \Sigma_x.\Sigma_r)]\Sigma \uparrow [\rho \mapsto \langle \lambda\rho_x.\Sigma_r; \Upsilon^{clo}\Upsilon^{lcl}\rangle]\Upsilon'} \tag{7.27}$$

Figure 7.7: Signature instantiation

elaborated and does not correspond to any full signature, it is neither defined in the static environment nor the entity environment, each of which only mapping paths defined outside of the signature currently being elaborated.

Rule 7.15 elaborate the formal parameter signature $sigexp_1$ and then the functor body signature $sigexp_2$ with the signature context extended with a spec corresponding to the formal parameter.

The following lemma guarantees that all tycons are fully relativized by the relativization judgment.

Lemma 10

If $AT(\Sigma) = \emptyset$, for all $\Sigma_i \in \Gamma.AT(\Sigma_i) = \emptyset$, and $\Gamma, \Upsilon, \Sigma \vdash sigexp \Rightarrow_{sig} \Sigma'$, then $AT(\Sigma') = \emptyset$.

7.8 Signature Instantiation

Signature instantiation produces a free instantiation of a given semantic signature, where free is in the context of both a closure and a local entity environment. Instantiation produces an entity environment that is local, meaning exclusive of closure entity environment bindings (see rule 7.22).

Most of the judgments ignore the signature element. The elements that matter are the those corresponding to static entities. Rule 7.25 produces a fresh atomic tycon τ^n of the appropriate arity n. The tycon must be fresh in both closure and local entity environments. Subsequent signature elements must be instantiated in the context of the local entity environment extended with the binding to the new tycon $[\rho \mapsto \tau^n]$. This new binding serves two roles. First, it ensures that τ^n is not reused when instantiating the rest of the signature elements. Second, the new binding may be included as part of the closure entity environment when instantiating a structure and functor specs.

Rule 7.26 instantiates a structure spec by recursively instantiating its semantic signature. The structure's semantic signature may mention either entities in Υ^{clo} and Υ^{lcl}, that is, preceding entities in the same signature. The resulting entity environment is combined with a closure $\Upsilon^{clo}\Upsilon^{lcl}$ to form a structure entity. Rule 7.27 instantiates a functor spec by producing a functor entity contains a formal functor functor entity expression $\lambda\rho_x.\Sigma_r$ and a closure entity environment.

Lemma 11 (Signature Instantiation Terminates)
$\Upsilon \vdash \Sigma \uparrow \Upsilon'$ *terminates.*

7.9 Top Level Declaration Elaborations

Elaboration of top level declaration is straightforward. The semantics is given in Fig. 7.8. Rule 7.29 elaborates a signature expression and extends the static environment with a binding of the signature name to the semantic signature. Rule 7.30 elaborates module declarations

$\boxed{\Gamma, \Upsilon \vdash^{top} d^t \ \mathsf{ok}}$

$$\frac{}{\Gamma, \Upsilon \vdash^{top} \circ \ \mathsf{ok}} \tag{7.28}$$

$$\frac{\Gamma, \Upsilon, \emptyset_{sig} \vdash sigexp \Rightarrow_{sig} \Sigma \qquad \Gamma[x \mapsto \Sigma], \Upsilon \vdash^{top} d^t \ \mathsf{ok}}{\Gamma, \Upsilon \vdash^{top} \textbf{signature } x = sigexp, d^t \ \mathsf{ok}} \tag{7.29}$$

$$\frac{\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \qquad \Gamma', \Upsilon' \vdash^{top} d^t \ \mathsf{ok}}{\Gamma, \Upsilon \vdash^{top} d^m, d^t \ \mathsf{ok}} \tag{7.30}$$

Figure 7.8: Top level declarations

using the $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$ judgment. The judgment can safely discard the entity declarations from the module declarations because top level declarations will not be wrapped in a functor and therefore lead to functor actions.

7.10 Module Elaboration

The module elaboration judgment calls for a static environment and an entity environment as its context. The static environment serves to elaborate tycons and structures, by looking up symbolic paths. The entity environment plays a role in signature elaboration and instantiation of functor parameters.

Fig. 7.9 gives the rules for elaborating module declarations. The judgment computes an entity declaration, a static environment, and an entity environment for the module declarations. The two environments will only contain bindings representing the information in the module declarations and not the closure. Each module declaration elaboration rule calculates a semantic representation of the declaration value (*i.e.*, type expressions, (definitional) tycons, atomic tycons, full signatures, and full functor signatures) and binds that representation in the static environment (and the entity environment if the entity in question is primary).

$\boxed{\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')}$

$$\frac{}{\Gamma, \Upsilon \vdash \circ \Rightarrow_{decl} (\bullet, \emptyset_{se}, \emptyset_{ee})} \tag{7.31}$$

$$\frac{\Gamma, \Upsilon \vdash e \Rightarrow_{core} \mathfrak{T} \qquad \Gamma[x \mapsto \mathfrak{T}], \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')}{\Gamma, \Upsilon \vdash \mathbf{val}\ x = e, d^m \Rightarrow_{decl} (\eta, [x \mapsto \mathfrak{T}]\Gamma', \Upsilon')} \tag{7.32}$$

$$\frac{\Gamma, \Upsilon \vdash C^\lambda \Rightarrow_{tyc} \mathfrak{C}^\lambda \qquad \Gamma[t \mapsto \mathfrak{C}^\lambda], \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')}{\Gamma, \Upsilon \vdash \mathbf{type}\ t = C^\lambda, d^m \Rightarrow_{decl} (\eta, [t \mapsto \mathfrak{C}^\lambda]\Gamma', \Upsilon')} \tag{7.33}$$

$$\frac{\begin{array}{c} n = |\vec{\alpha}| \qquad \Gamma[t \mapsto \tau^n], \Upsilon[\rho_t \mapsto \tau^n] \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \\ (\rho_t \text{ and } \tau \text{ are fresh}) \end{array}}{\Gamma, \Upsilon \vdash \mathbf{datatype}\ \vec{\alpha}\ t, d^m \Rightarrow_{decl} ([\rho_t =_{tyc} \mathsf{new}(n)]\eta, [t \mapsto \tau^n]\Gamma', [\rho_t \mapsto \tau^n]\Upsilon')} \tag{7.34}$$

$$\frac{\begin{array}{c} \Gamma, \Upsilon \vdash strexp \Rightarrow_{str} (M, \varphi) \qquad M = (\Sigma, R) \qquad (\rho \text{ fresh}) \\ \Gamma[X \mapsto (\rho, M)], \Upsilon[\rho \mapsto R] \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \end{array}}{\Gamma, \Upsilon \vdash \mathbf{structure}\ X = strexp, d^m \Rightarrow_{decl} ([\rho =_{str} \varphi]\eta, [X \mapsto (\rho, M)]\Gamma', [\rho \mapsto R]\Upsilon')} \tag{7.35}$$

$$\frac{\begin{array}{c} \Gamma, \Upsilon, \emptyset_{sig} \vdash sigexp \Rightarrow_{sig} \Sigma_x \qquad \Upsilon, \emptyset_{ee} \vdash \Sigma_x \uparrow \Upsilon_x \qquad R_x = \langle \Upsilon_x, \Upsilon \rangle \\ \Gamma[X \mapsto (\rho_x, (\Sigma_x, R_x))], \Upsilon[\rho_x \mapsto R_x] \vdash strexp \Rightarrow_{str} ((\Sigma_{res}, \_), \varphi) \\ \theta = \lambda \rho_x.\varphi \qquad \psi = \langle \theta; \Upsilon \rangle \\ \Gamma[f \mapsto (\rho, (\Pi \rho_x : \Sigma_x.\Sigma_{res}, \psi))], \Upsilon[\rho \mapsto \psi] \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \\ (\rho_x, \rho \text{ fresh}) \end{array}}{\begin{array}{c} \Gamma, \Upsilon \vdash \mathbf{functor}\ f(X : sigexp) = strexp, d^m \\ \Rightarrow_{decl} ([\rho =_{tyc} \theta]\eta, [f \mapsto (\rho, (\Pi \rho_x : \Sigma_x.\Sigma_{res}, \psi))]\Gamma', [\rho \mapsto \psi]\Upsilon') \end{array}} \tag{7.36}$$

The resultant Υ must be the local entity environment in order for the structure expression judgment for struct d^m end to properly construct a structure realization.

Figure 7.9: Module declaration elaboration

$$\boxed{\Gamma, \Upsilon \vdash \mathit{strexp} \Rightarrow_{str} (M, \varphi)}$$

$$\frac{\Gamma(p) = (\vec{\rho}, M)}{\Gamma, \Upsilon \vdash p \Rightarrow_{str} (M, \vec{\rho})} \tag{7.37}$$

$$\frac{\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon^{lcl}) \qquad \Upsilon \vdash \Gamma' \hookrightarrow \Sigma}{\Gamma, \Upsilon \vdash \textbf{struct}\ d^m\ \textbf{end} \Rightarrow_{str} ((\Sigma, \langle \Upsilon^{lcl}, \Upsilon \rangle), (\!|\eta|\!))} \tag{7.38}$$

$$\frac{\begin{array}{c}\Gamma(p) = (\vec{\rho}, (\Pi X : \Sigma_{par}.\Sigma_{body}, \langle \theta; \Upsilon' \rangle)) \\ \Gamma, \Upsilon \vdash \mathit{strexp} \Rightarrow_{str} (M, \varphi) \\ \Upsilon \vdash (M, \varphi) : \Sigma_{par} \Rightarrow_{match} (R_c, \varphi_c) \\ \varphi_{app} = \vec{\rho}(\varphi_c) \qquad \Upsilon \vdash \varphi_{app} \Downarrow_{str} R_{app}\end{array}}{\Gamma, \Upsilon \vdash p(\mathit{strexp}) \Rightarrow_{str} ((\Sigma_{body}, R_{app}), \varphi_{app})} \tag{7.39}$$

$$\frac{\begin{array}{c}\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta_{def}, \Gamma_{def}, \Upsilon_{def}) \\ \Gamma_{def}, \Upsilon_{def} \vdash \mathit{strexp} \Rightarrow_{str} (M, \varphi)\end{array}}{\Gamma, \Upsilon \vdash \textbf{let}\ d^m\ \textbf{in}\ \mathit{strexp} \Rightarrow_{str} (M, \textbf{let}\ \eta_{def}\ \textbf{in}\ \varphi)} \tag{7.40}$$

$$\frac{\begin{array}{c}\Gamma, \Upsilon, \emptyset_{sig} \vdash \mathit{sigexp} \Rightarrow_{sig} \Sigma_{spec} \qquad \Gamma, \Upsilon \vdash \mathit{strexp} \Rightarrow_{str} (M_u, \varphi_u) \\ \Upsilon \vdash (M_u, \varphi_u) : \Sigma_{spec} \Rightarrow_{match} (R_c, \varphi_c)\end{array}}{\Gamma, \Upsilon \vdash \mathit{strexp} : \mathit{sigexp} \Rightarrow_{str} ((\Sigma_{spec}, R_c), \varphi_c)} \tag{7.41}$$

$$\frac{\begin{array}{c}\Gamma, \Upsilon, \emptyset_{sig} \vdash \mathit{sigexp} \Rightarrow_{sig} \Sigma_{spec} \qquad \Gamma, \Upsilon \vdash \mathit{strexp} \Rightarrow_{str} (M_u, \varphi_u) \\ \Upsilon \vdash (M_u, \varphi_u) : \Sigma_{spec} \Rightarrow_{match} (R_c, \varphi_c) \qquad \Upsilon, \emptyset_{ee} \vdash \Sigma_{spec} \uparrow \Upsilon_{spec}\end{array}}{\Gamma, \Upsilon \vdash \mathit{strexp} :> \mathit{sigexp} \Rightarrow_{str} ((\Sigma_{spec}, \langle \Upsilon_{spec}, \Upsilon \rangle), \varphi_c)} \tag{7.42}$$

Figure 7.10: Structure expression elaboration

Rules 7.32 and 7.33 elaborates values to a semantic type expression and syntactic tycon to semantic tycon respectively. Neither of these produce any entity declarations and entity environment because they do not contain static entities. Rule 7.34 generates a fresh atomic tycon τ^n for a datatype. This atomic tycon is bound in both the static and entity environments. The elaborator further produces an entity declaration $\rho_t =_{tyc} \mathsf{new}(n)$ that memoizes the functor action attributed to the datatype declaration. The entity declaration is reserved for evaluation upon functor application of the possible enclosing functor. When evaluated, it will produce another atomic tycon distinct from the τ^n generated here. Rule 7.35 elaborates the structure expression to a full signature and a structure entity expression. Rule 7.36 elaborates a functor declaration via several steps:

1. elaborate the functor parameter signature
2. instantiate it
3. form a full signature using the result of the previous two steps
4. elaborate the functor body using a static environment extended with this full signature and the entity environment extended with the structure entity
5. form a full functor signature, a functor entity, and a functor entity expression using the structure entity expression from above

Fig. 7.10 elaborates a structure expression into a full signature and structure entity expression. Rule 7.37 elaborates a structure path. The structure entity expression is an entity path, calculable by accumulating the entity variables in Γ leading up to the element denoted by p. Rule 7.38 must extract a semantic signature from the static environment produced by elaborating the base structure declarations. Rule 7.39 elaborates a functor application in several steps:

1. looks up the entity path and full functor signature of the functor denoted by symbolic path p

2. elaborates the argument structure

3. coerces the argument structure and structure entity expression using the functor parameter signature by signature matching

4. forms the structure entity expression for the application

5. evaluates the structure entity expression using the current entity environment

Rule 7.40 elaborates structure let expressions in the expected way. Rule 7.41 elaborates transparent ascription by signature matching. Rule 7.42 elaborates opaque ascription by signature matching and then instantiating the spec signature to get fresh types for the open type specs.

An important invariant that must be maintained for module declaration elaboration is that all the atomic tycons in the static environment are also in the entity environment. Let $AT(\cdot)$ denote the set of all atomic tycons (τ^n) in $\cdot$. This property guarantees that all atomic tycons can be relativized by the entity environment.

Lemma 12

If $AT(\Gamma) \subseteq AT(\Upsilon)$ *and* $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$, *then* $AT(\Gamma') \subseteq AT(\Upsilon')$.

The relationship between the semantic signature Σ and structure entity R produced by elaboration is precisely related. R is said to *interpret* Σ, which is defined as follows:

Definition 3

Υ *interprets* Σ *if for all specs in* Σ, *one of the following must be true:*

1. *If the spec is an open tycon* (ρ, n), *then* $\Upsilon(\rho) = \tau^n$ *or* $\Upsilon(\rho) = \mathbb{C}^\lambda$ *such that* $\Upsilon \vdash \mathbb{C}^\lambda :: \Omega^n \rightarrow \Omega$.

2. *If the spec is a structure* (ρ, Σ), *then* $\Upsilon(\rho) = R'$.

3. *If the spec is a functor* (ρ, Σ^f), *then* $\Upsilon(\rho) = \psi$.

$\boxed{\Upsilon \vdash \Gamma \hookrightarrow \Sigma}$

$$\frac{}{\Upsilon \vdash \emptyset_{se} \hookrightarrow \emptyset_{sig}} \tag{7.43}$$

$$\frac{\Upsilon \vdash \mathfrak{T} \Rightarrow^{te}_{rel} \mathbb{T} \qquad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r}{\Upsilon \vdash [x \mapsto \mathfrak{T}]\Gamma \hookrightarrow [x \mapsto \mathbb{T}]\Sigma_r} \tag{7.44}$$

$$\frac{\Upsilon \vdash \mathfrak{C}^\lambda \Rightarrow^{tyc}_{rel} \mathbb{C}^\lambda \qquad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r}{\Upsilon \vdash [t \mapsto \mathfrak{C}^\lambda]\Gamma \hookrightarrow [t \mapsto \mathbb{C}^\lambda]\Sigma_r} \tag{7.45}$$

$$\frac{\Upsilon^{-1}(\tau^n) = \rho \qquad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r}{\Upsilon \vdash [t \mapsto \tau^n]\Gamma \hookrightarrow [t \mapsto (\rho, n)]\Sigma_r} \tag{7.46}$$

$$\frac{\Upsilon \vdash \Gamma_r \hookrightarrow \Sigma_r}{\Upsilon \vdash [x \mapsto (\rho, (\Sigma_1, R_1))]\Gamma_r \hookrightarrow [x \mapsto (\rho, \Sigma_1)]\Sigma_r} \tag{7.47}$$

$$\frac{\Upsilon \vdash \Gamma_r \hookrightarrow \Sigma_r}{\Upsilon \vdash [f \mapsto (\rho, (\Sigma_1^f, \psi))]\Gamma_r \hookrightarrow [f \mapsto (\rho, \Sigma_1^f)]\Sigma_r} \tag{7.48}$$

Figure 7.11: Signature extraction

Lemma 13

If Υ interprets the extracted signature of Γ and $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi)$, then R interprets Σ.

7.11 Signature Extraction

Fig. 7.11 gives the rules for signature extraction, the process of synthesizing a semantic signature for a structure given only the static environment produced by elaborating that structure and an entity environment. Because signature must contain only relativized type expressions and tycons, rules 7.44 and 7.45 must relativize the semantic type expression and tycon in the static environment. Rule 7.46 produces a spec from an atomic tycon binding, which specifies the arity n of the tycon. The rule looks up the atomic tycon in the retract of Υ. This rule assumes that Υ and Γ are synchronized. In particular, since t is a local name in

$\boxed{\Upsilon \vdash (M, \varphi) : \Sigma \Rightarrow_{match} (R_c, \varphi_c)}$

$$\frac{(\rho_u \text{ is fresh in } \Upsilon) \qquad \Upsilon_a \Upsilon^{clo} \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)}{\Upsilon \vdash ((\Sigma_a, \langle \Upsilon_a, \Upsilon^{clo} \rangle), \varphi) : \Sigma_s \Rightarrow_{match} (\langle \Upsilon', \Upsilon \rangle, \textbf{let } \rho_u = \varphi \textbf{ in } (\!|\eta|\!))} \tag{7.49}$$

Figure 7.12: Signature matching elaboration

the static environment (*i.e.*, its binding is not nested inside some signature in Γ), τ^n should also be local in Υ and therefore have a singleton entity path.

Lemma 14 (Synchronization of Static and Entity Environments)
If $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$ *and* $EV(\Gamma) \subseteq dom(\Upsilon)$, *then* $EV(\Gamma') \subseteq dom(\Upsilon')$.

For full signatures and full functor signatures, signature extraction is simple. Rules 7.47 and 7.48 forms structure and functor specs by projecting out the entity variable and semantic signature (or semantic functor signature) from the static environment mapping for the structure and functor respectively.

7.12 Signature Matching

Signature matching plays three principal roles during elaboration. It coerces functor arguments to the form dictated by the functor parameter signature in rule 7.39. Rules 7.41 and 7.42 use signature matching to constrain the full signature of the structure for transparent and opaque ascription respectively. Figs. 7.12 and 7.13 describe the signature matching judgment and its subsidiary judgment respectively. The main judgment for signature matching $\Upsilon \vdash (M, \varphi) : \Sigma \Rightarrow_{match} (R_c, \varphi_c)$ forms the *coercion structure entity expression* φ_c which binds the uncoerced, original entity expression to a fresh entity variable ρ_u for use by the coercion entity declarations η. In general, η may alias components of φ as is in the case in rule 7.52. The coercion structure entity expression needs both φ and η because the elaborator needs to evaluate φ at least for its functor actions and η are the actual exported entity variables. The entity environment for signature matching is a composite of the context Υ,

$$\boxed{\Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)}$$

$$\frac{}{\Upsilon, \Sigma_a, \rho_a \vdash \emptyset_{sig} \Rightarrow_{coerce} (\emptyset_{ee}, \bullet)} \tag{7.50}$$

$$\frac{\begin{array}{c}\Upsilon \vdash \Sigma_a(x) \equiv \mathbb{T} \\ \Upsilon, \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)\end{array}}{\Upsilon, \Sigma_a, \rho_a \vdash [x \mapsto \mathbb{T}]\Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)} \tag{7.51}$$

$$\frac{\Sigma_a(t) = (\rho_a, n) \qquad \Upsilon, \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)}{\Upsilon, \Sigma_a, \rho_u \vdash [t \mapsto (\rho, n)]\Sigma_s \Rightarrow_{coerce} ([\rho \mapsto \Upsilon(\rho_a)]\Upsilon', [\rho =_{def} \rho_u\rho_a]\eta)} \tag{7.52}$$

$$\frac{\begin{array}{c}\Sigma_a(t) = \mathbb{C}^\lambda \qquad |\mathbb{C}^\lambda| = n \\ \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)\end{array}}{\Upsilon, \Sigma_a, \rho_u \vdash [t \mapsto (\rho, n)]\Sigma_s \Rightarrow_{coerce} ([\rho \mapsto \mathbb{C}^\lambda]\Upsilon', [\rho =_{def} \mathbb{C}^\lambda]\eta)} \tag{7.53}$$

$$\frac{\begin{array}{c}\Sigma_a(t) = \mathbb{C}_a^\lambda \qquad \Upsilon \vdash \mathbb{C}_a^\lambda \equiv \mathbb{C}_s^\lambda \\ \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)\end{array}}{\Upsilon, \Sigma_a, \rho_u \vdash [t \mapsto \mathbb{C}_s^\lambda]\Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)} \tag{7.54}$$

$$\frac{\begin{array}{c}\Sigma_a(t) = (\rho, n) \qquad n = |\mathbb{C}_s^\lambda| \qquad \Upsilon \vdash \mathbb{C}_s^\lambda \equiv \Upsilon(\Sigma_{a_{ep}}(t)) \\ \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)\end{array}}{\Upsilon, \Sigma_a, \rho_u \vdash [t \mapsto \mathbb{C}_s^\lambda]\Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)} \tag{7.55}$$

$$\frac{\begin{array}{c}\Sigma_a(x) = (\rho_x, \Sigma_x) \qquad \Upsilon \vdash ((\Sigma_x, \Upsilon(\rho_x)), \rho_u\rho_x) : \Sigma_s \Rightarrow_{match} ((\Sigma_c, R_c), \varphi_c) \\ \Upsilon[\rho_s \mapsto R_c], \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta')\end{array}}{\Upsilon, \Sigma_a, \rho_u \vdash [x \mapsto (\rho_s, \Sigma_s)]\Sigma_s \Rightarrow_{coerce} (\Upsilon', [\rho_s = \varphi_c]\eta')} \tag{7.56}$$

$$\frac{\begin{array}{c}\Sigma_a(f) = (\rho_a, \Sigma_a^f) \qquad \Upsilon \vdash ((\Sigma_a^f, \Upsilon(\rho_a)), \rho_u\rho_a) : \Sigma_s^f \Rightarrow_{fsgmtch} (\psi_c, \theta_c) \\ \Upsilon[\rho_s \mapsto \psi_c], \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)\end{array}}{\Upsilon, \Sigma_a, \rho_u \vdash [f \mapsto (\rho_s, \Sigma_s^f)]\Sigma_s \Rightarrow_{coerce} ([\rho_s \mapsto \psi_c]\Upsilon', [\rho_s = \theta_c]\eta)} \tag{7.57}$$

Figure 7.13: Structure signature matching

actual structure local environment Υ_a, and actual structure closure environment Υ^{clo}. All three is necessary because Υ closes the spec signature Σ_s, Υ_a the actual Σ_a, and Υ^{clo} the local environment Υ_a.

A subsidiary judgment $\Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)$ matches actual structure components to individual specs. The actual structure signature Σ_a is used to match against each spec. The entity variable ρ_u provides access to the uncoerced structure entity expression in case the elaborator needs to alias components. The judgment will produce both a semantic signature and a local entity environment, used by the match judgment to produce a coerced full signature.

Rule 7.50 matches against an empty spec signature, in which case the coercion signature, entity environment, and entity declarations are empty. Rule 7.51 matches value specs only when the value's type expressions are equivalent when interpreted under the entity environment. For open tycon specs, there are two cases. Rule 7.52 matches an open tycon spec against an open tycon spec in actual. In that case, the rule will use the structure entity for the actual and an alias to the entity path for the actual $\rho_u\rho_a$ for the entity declaration. Rule 7.53 matches an open tycon spec against a definitional tycon spec in the actual. The arities of the definitional tycon and the open tycon spec must match up. Rule 7.54 matches a definitional tycon spec in the spec signature to a definitional tycon spec in the actual. Again, this match requires the equivalence of the spec and actual definitional tycons interpreted under the entity environment. Rule 7.55 matches a definition tycon spec in the spec signature to an open tycon in the actual, which must correspond to an atomic tycon in entity environment $\Upsilon(\Sigma_{ep}(t))$. Rules 7.56 and 7.57 match structure and functor specs respectively. Being specs for entities, these rules must produce a new entity environment binding and entity declaration. Rule 7.56 forms the bindings using the full signature and structure entity expression from performing signature matching on the signature in the structure spec. The full signature the rule will match against is formed by the spec signature Σ_x and the corresponding structure entity $\Upsilon(\rho_x)$. Rule 7.57 works similarly.

$\boxed{\Upsilon \vdash F : \Sigma^f \Rightarrow_{fsgmtch} (\psi_c, \theta_c)}$

$$\frac{\begin{array}{c}\Upsilon^{clo}, \Upsilon \vdash \Sigma_{apar} \uparrow \Upsilon_{apar} \qquad \Upsilon \vdash (\Sigma_{apar}, \langle \Upsilon_{apar}, \Upsilon^{clo}\Upsilon\rangle) : \Sigma_{spar} \Rightarrow_{match} (M_c, \varphi_c) \\ M_c = (\Sigma_c, \Upsilon_c) \qquad \Upsilon^{clo}, \Upsilon[\rho' \mapsto \langle \Upsilon_c, \Upsilon^{clo}\Upsilon\rangle] \vdash \Sigma_{sres} \uparrow \Upsilon_{sres} \\ \Upsilon \vdash (\Sigma_{sres}, \langle \Upsilon_{sres}, \Upsilon^{clo}\Upsilon\rangle) : \Sigma_{ares} \Rightarrow_{match} (M'_c, \varphi'_c) \qquad \theta_c = \lambda\rho'.\varphi'_c\end{array}}{\begin{array}{c}\Upsilon \vdash (\rho_f, \Pi\rho : \Sigma_{apar}.\Sigma_{ares}, \langle \lambda\rho.\varphi; \Upsilon^{clo}\rangle) : (\Pi\rho' : \Sigma_{spar}.\Sigma_{sres}) \\ \Rightarrow_{fsgmtch} (\langle \theta_c; \Upsilon\rangle, \theta_c)\end{array}} \tag{7.58}$$

Figure 7.14: Functor signature matching

The relationship between the resultant structure entity, the spec signature, and actual signature is precise:

Lemma 15

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{match} (R_c, \varphi_c)$*, then for all* $[x \mapsto s] \in \Sigma_s$*,* $[x \mapsto s'] \in \Sigma_a$ *such that* $R_c(s) = R_a(s')$*.*

An important property is that the structure entity expression constructed by signature matching should evaluate to the coercion entity environment. After all, the role of the structure entity expression is to evaluate to an entity environment that instantiates the spec signature to an actual argument structure in the current entity environment.

Lemma 16

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{match} (R_c, \varphi_c)$*, then* $\Upsilon \vdash \varphi_c \Downarrow_{str} R_c$*.*

7.13 Functor Signature Matching

Fig. 7.14 defines the functor signature matching judgment fsgmtch which is composed of a single rule, rule 7.58. The rule calculates a coerced functor entity ψ_c and functor entity expression θ_c. To match the functor spec, the functor spec's parameter signature Σ_{spar} is matched against a full signature formed by the actual parameter signature Σ_{apar} and the instantiation of Σ_{apar}. The functor spec's result signature Σ_{sres} is then instantiated and that instantiation is used to form a full signature for the functor result in order to match against the actual functor spec's result signature Σ_{ares}. The resultant structure entity expression can be used to form the final coerced ψ_c and θ_c.

CHAPTER 8

TRANSLATION

One way to show the soundness of the module system is to provide a translation to an applied F_ω language. What this translation amounts to is a decoupling of the static content from the dynamic content of modules. Module systems that permit such at decouple are said to admit *phase separation*.

The elaboration semantics given in the previous chapter is sufficient and self-contained. However, to support translation into F_ω, the elaboration must be instrumented to annotate module language terms with their full signatures, full functor signature, signature matching coercions, and free instantiation of the functor parameter.

8.1 Annotated Module Language

Fig. 8.1 gives the grammar for the module language annotated with semantic objects from elaboration. The semantic objects are enclosed by angle brackets $\langle\langle\rangle\rangle$. Structure paths must be annotated with the entity path corresponding to that symbolic path. The functor path in a functor application should be annotated with its entity path and the entity variable for the formal parameter. The entity variable for the formal parameter is necessary for producing the necessary coercions to get the type and value content of an argument structure into the proper form. The argument structure should be annotated with its coercion full signature and coercion structure entity expression. The associated entity variable should annotate both structure and functor declarations. The functor declaration further requires the annotation of a full functor signature and free instantiation entity environment. All this information is readily available during the elaboration process. Thus, annotating is straightforward. Below, I show a select modified rule:

$$
\begin{array}{lrcl}
\text{Core declarations} & d^c & ::= & \mathsf{val}\ x = e \mid \mathsf{type}\ t = C^\lambda \mid \mathsf{datatype}\ \vec{\alpha}\ t \\
 & spec & ::= & \mathsf{structure}\ x : sigexp \mid \mathsf{type}\ \vec{\alpha}\ t \\
 & & \mid & \mathsf{type}\ t = C^\lambda \\
 & & \mid & \mathsf{functor}\ f(x : sigexp_1) : sigexp_2 \\
 & & \mid & \mathsf{val}\ x : T \\
 & specs & ::= & \emptyset_{specs} \mid spec, specs \\
 & sigexp & ::= & x \mid \mathsf{sig}\ spec\ \mathsf{end} \mid sigexp\ \mathsf{where\ type}\ p = C^\lambda \\
 & strexp & ::= & p\langle\langle\vec{\rho}\rangle\rangle \mid \mathsf{struct}\ d^m\ \mathsf{end} \mid p\langle\langle\vec{\rho}, \rho_{par}\rangle\rangle(strexp\langle\langle M, \rho_u\rangle\rangle) \\
 & & \mid & strexp : sigexp \mid strexp :> sigexp \\
\text{Module decls} & d^m & ::= & \circ \mid \mathsf{structure}\ x\langle\langle\rho\rangle\rangle = strexp, d^m \\
 & & \mid & \mathsf{functor}\langle\langle\rho, F, \Upsilon\rangle\rangle\ f(x : sigexp) = strexp, d^m \mid d^c, d^m \\
\text{Top level decls} & d^t & ::= & \circ \mid \textbf{signature}\ x = sigexp, d^t \mid d^m, d^t
\end{array}
$$

Figure 8.1: Annotated module language

$$
\frac{\begin{array}{c}\Gamma(p) = (\vec{\rho}, (\Pi\rho_{par} : \Sigma_{par}.\Sigma_{body}, \langle\theta; \Upsilon'\rangle)) \qquad \Gamma, \Upsilon \vdash strexp \Rightarrow_{str} (M, \varphi) \\ \Upsilon \vdash (M, \varphi) : \Sigma_{par} \Rightarrow_{match} (M_c, \textbf{let}\ \rho_u = \varphi\ \textbf{in}\ (\!|\eta|\!)) \\ \varphi_{app} = \vec{\rho}(\varphi_c) \qquad \Upsilon \vdash \varphi_{app} \Downarrow_{str} R_{app}\end{array}}{\Gamma, \Upsilon \vdash p(strexp) \Rightarrow_{str} (p\langle\langle\vec{\rho}, \rho_{par}\rangle\rangle(strexp\langle\langle M_c, \rho_u\rangle\rangle), (\Sigma_{body}, R_{app}), \varphi_{app})} \qquad (8.1)
$$

As shown in the above rule, the entity path for the functor and the functor parameter entity variable are in the static environment at elaboration time. The coercion full signature M_c and entity variable ρ_u for the uncoerced argument structure.

8.2 Target Language: F_ω

Fig. 8.2 gives the grammar for the applied F_ω. The language is nearly identical to the one Shao used [66]. It is an F_ω core enriched with record types and declarations.

An F_ω kind k is either a monokind or an arrow kind. A tycon c can be a type variable, an atomic tycon $\mathsf{inj}(\tau^n)$, an explicitly kinded tycon-level λ abstraction, and a tycon application. The tycon for a record is represented as $\{l_0 : c_0, \ldots, l_n : c_n\}$ where l_i's are record field

k	$::=$	$\Omega \mid k \to k \mid \{l_0 :: k_0, \ldots, l_n :: k_n\}$	kinds
c	$::=$	$\alpha \mid \mathsf{inj}(\tau^n) \mid \mathsf{new}(n) \mid \lambda\alpha : k.c \mid c(c) \mid \{l_0 = c_0, \ldots, l_n = c_n\} \mid c.l$	tycons
t	$::=$	$\mathsf{ty}(c) \mid \forall\alpha :: k.t \mid \{l_0 : t_0, \ldots, l_n : t_n\}$	types
e_ω	$::=$	$x \mid \lambda x : t.e_\omega \mid e_\omega(e_\omega) \mid \Lambda\alpha :: k.e_\omega \mid e_\omega[c] \mid \{l_0 = e_\omega, \ldots, l_n = e_\omega\} \mid e_\omega.l$	
	$\mid$	let d in e_ω	exps
v	$::=$	$\lambda x : t.e_\omega \mid \Lambda\alpha :: k.e_\omega$	values
d	$::=$	$\emptyset_{\omega dec} \mid x = e_\omega; d$	decl
E	$::=$	$\emptyset_{\omega env} \mid E, \alpha :: k \mid E, x : t$	environs

Let $t \to t'$ be shorthand for $\to^2 (t, t')$.

Figure 8.2: System F_ω

labels and c_i's are the tycons for the fields corresponding to those labels. The tycon form $c.l$ can project out the field type of field l assuming that c is a record type containing a field of that label. Type expressions t are either a tycon or an explicitly quantified and kinded polymorphic type $\forall\alpha :: k.t$. Unlike the version of the F_ω type system given by Shao, this type system includes atomic tycons. Also, this type system simplifies Shao's by subsuming base types (int) and arrow types $\tau \to \tau'$ using an atomic tycon and type application $\to^2 (\tau)(\tau')$. Types t omit arrow and record types from Shao's version.

F_ω expressions include variables, an explicitly-typed λ-abstractions, applications of expressions, an explicitly-kinded type abstraction $\Lambda\alpha :: k.e_\omega$, and type application $e_\omega[t]$. An expression can also be a record expression $\{l_0 = e_{\omega_0}, \ldots, l_n = e_{\omega_n}\}$, a record field projection $e_\omega.l$, or a let expression. Within let expressions are declarations, which are sequences of variables bound to expressions terminated by $\cdot$, the empty declaration. The same environment is used for both kinding $\alpha :: k$ and typing $x : t$. Note that the abstraction $(\lambda x : t.e_\omega)$ and type application $e_\omega[t]$ both permit application to polymorphic types $\forall\alpha :: k.t$. This is necessary because functors and functor applications in the module language support abstraction over polymorphic types ($\mathsf{functor}\ F(X : \mathsf{sig\ val}\ id : \forall\alpha.\alpha \to \alpha\ \mathsf{end}) = \mathsf{struct\ end}$) and application to such types. When these are translated to F_ω, the polymorphism must be maintained.

$\boxed{E \vdash c :: k}$

$$\frac{}{E \vdash \alpha :: E(\alpha)} \tag{8.2}$$

$$\frac{}{E \vdash \mathsf{inj}(\tau^n) :: \Omega_1 \to \dots \Omega_n \to \Omega} \tag{8.3}$$

$$\frac{}{E \vdash \mathsf{new}(n) :: \Omega_1 \to \dots \Omega_n to \Omega} \tag{8.4}$$

$$\frac{E[\alpha :: k] \vdash c :: k'}{E \vdash (\lambda\alpha :: k.c) :: k \to k'} \tag{8.5}$$

$$\frac{E \vdash c_1 :: k_1 \to k_2 \qquad E \vdash c_2 :: k_1}{E \vdash c_1(c_2) :: k_2} \tag{8.6}$$

$$\frac{E \vdash c_1 :: k_1 \dots E \vdash c_n :: k_n}{E \vdash \{l_1 = c_1, \dots, l_n = c_n\} :: \{l_1 :: k_1, \dots, l_n :: k_n\}} \tag{8.7}$$

$$\frac{E \vdash c :: \{lks\} \qquad l :: k \in \{lks\}}{E \vdash c.l :: k} \tag{8.8}$$

Figure 8.3: F_ω well-kinding

8.2.1 Kind System

Fig. 8.3 describes well-kinding of F_ω tycons. The kind system is standard. The judgment relies only on the kind part of the environment. Rule 8.3 gives the curried kind for the n-ary atomic tycon. The resultant kind is curried n times. Rules 8.7 and 8.8 kind record tycons and record projection tycons respectively in the usual way.

8.2.2 Type System

Fig. 8.4 gives the type system for F_ω. $\vdash t$ type is an axiom. In particular $\vdash k$ type is not true. Type application (rule 8.13) assumes that the left-hand side has a type $\lambda\alpha.k.t'$. The type of the type application is $t'\{t/\alpha\}$. Rule 8.14 gives the record tycon $\{l_0 : c_0, \ldots, l_n : c_n\}$ for a record expression $\{l_0 = e_0, \ldots, l_n = e_n\}$ as long as all its field expressions have the correct type,*i.e.*, $e_i : c_i$.

The notation $\{c/\alpha\}_{tyc}$ denotes tycon substitution. This kind of substitution does not act on values that may coincidentally share the same name α. This distinct form of substitution is necessary because tycons and values will have distinct namespaces.

8.3 Translation Semantics

The translation rules all assume an annotated module language that was produced by an error-free elaboration. Since the program was successfully elaborated, the semantics can make some assumptions about the program, especially what variables will or will not be accessed, and thereby simplify the rules.

The notation $dom(d)$ here denotes the set of all the bound names,*i.e.*, the ρ in $\rho = e$.

Fig. 8.5 gives the rules for translating module declarations to F_ω declarations. Rule 8.19 is the empty declaration case. Rule 8.20 translates the core language expression e to a corresponding e_ω. Unlike the static entity case, shadowing a previous variable x is harmless so x will be used as the variable to bind. Rules 8.21 and 8.22 do not produce F_ω expressions

$\boxed{E \vdash e_\omega : t}$

$$\frac{\vdash E(p) \ \mathsf{type}}{E \vdash p : E(p)} \tag{8.9}$$

$$\frac{E[x:t] \vdash e_\omega : t'}{E \vdash (\lambda x : t.e_\omega) : t \to t'} \tag{8.10}$$

$$\frac{E \vdash e_{\omega_1} : t_1 \to t_2 \qquad E \vdash e_{\omega_2} : t_1}{E \vdash e_{\omega_1}(e_{\omega_2}) : t_2} \tag{8.11}$$

$$\frac{E[\alpha : k] \vdash e_\omega : t}{E \vdash (\Lambda\alpha :: k.e_\omega) : (\forall\alpha :: k.t)} \tag{8.12}$$

$$\frac{E \vdash e_\omega : \forall\alpha :: k.t' \qquad E \vdash c :: k}{E \vdash e_\omega[c] : t'\{c/\alpha\}} \tag{8.13}$$

$$\frac{E \vdash e_{\omega_0} : t_0 \ldots E \vdash e_{\omega_n} : t_n}{E \vdash \{l_0 = e_{\omega_0}, \ldots, l_n = e_{\omega_n}\} : \{l_0 : t_0, \ldots, l_n : t_n\}} \tag{8.14}$$

$$\frac{E \vdash e_\omega : \{\ldots, l_i : t_i, \ldots\}}{E \vdash e_\omega.l_i : t_i} \tag{8.15}$$

$$\frac{E \vdash d :_{dec} E' \qquad E' \vdash e_\omega : t}{E \vdash \text{let } d \text{ in } e_\omega : t} \tag{8.16}$$

$\boxed{E \vdash d :_{dec} E'}$

$$\frac{}{E \vdash \emptyset_{\omega dec} :_{dec} E} \tag{8.17}$$

$$\frac{E \vdash e : t \qquad E[x:t] \vdash d :_{dec} E'}{E \vdash x = e; d :_{dec} E'} \tag{8.18}$$

Figure 8.4: F$_\omega$ Type System

$$\boxed{d^m \rightsquigarrow_{dec} d}$$

$$\frac{}{\circ \rightsquigarrow_{dec} \emptyset_{\omega dec}} \tag{8.19}$$

$$\frac{e \rightsquigarrow_{exp} e_\omega}{\textbf{val}\ x = e \rightsquigarrow_{dec} x = e_\omega} \tag{8.20}$$

$$\frac{}{\textbf{type}\ t = C^\lambda \rightsquigarrow_{dec} \emptyset_{\omega dec}} \tag{8.21}$$

$$\frac{}{\textbf{datatype}\ \vec{\alpha}\ x \rightsquigarrow_{dec} \emptyset_{\omega dec}} \tag{8.22}$$

$$\frac{strexp \rightsquigarrow_{exp} e_\omega}{\textbf{structure}\ x\langle\langle\rho\rangle\rangle = strexp \rightsquigarrow_{dec} \rho = e_\omega} \tag{8.23}$$

$$\frac{F = (\Pi\rho_x : \Sigma_{par}.\Sigma', \psi) \qquad \vdash \Sigma_{par} \rightsquigarrow_{knd} k \qquad \Upsilon_{ins} \vdash \Sigma_{par} \rightsquigarrow_{type} t \\ strexp \rightsquigarrow_{exp} e_\omega}{\textbf{functor}\ f\langle\langle\rho, F, \Upsilon_{ins}\rangle\rangle(X : sigexp) = strexp \rightsquigarrow_{dec} \rho = \Lambda\rho_x :: k.\lambda\rho_x : t.e_\omega} \tag{8.24}$$

$$\frac{decl \rightsquigarrow_{dec} d_1 \qquad d^m \rightsquigarrow_{dec} d_2}{decl, d^m \rightsquigarrow_{dec} d_1; d_2} \tag{8.25}$$

$$\begin{array}{rcl} decl & ::= & \textbf{val}\ x = e \mid \textbf{type}\ t = C^\lambda \mid \textbf{datatype}\ \vec{\alpha}\ x \mid \textbf{structure}\ x\langle\langle\rho\rangle\rangle = strexp \\ & \mid & \textbf{functor}\ f\langle\langle\ldots\rangle\rangle(X : sigexp) = strexp \end{array}$$

Figure 8.5: Translation for declarations

$\boxed{strexp \rightsquigarrow_{exp} e_\omega}$

$$\frac{}{p\langle\langle\vec{\rho}\rangle\rangle \rightsquigarrow_{exp} \vec{\rho}} \tag{8.26}$$

$$\frac{d^m \rightsquigarrow_{dec} d \qquad e_\omega = \{\rho_i = \rho_i, \ldots\} \forall \rho_i \in dom(d) \text{ and } \rho_i \text{ is not a type}}{\mathsf{struct}\ d^m\ \mathsf{end} \rightsquigarrow_{exp} \mathbf{let}\ d\ \mathbf{in}\ e_\omega} \tag{8.27}$$

$$\frac{\begin{array}{c} strexp \rightsquigarrow_{exp} e_\omega \qquad M_c \rightsquigarrow_{tyc} c \qquad M_c = (\Sigma, R) \\ R \vdash \Sigma \rightsquigarrow_{type} lts_c \qquad e_{\omega_c} = \mathsf{coerce}(\rho_u, c, \{lts_c\}) \end{array}}{p\langle\langle\vec{\rho}, \rho_{par}\rangle\rangle(strexp\langle\langle M_c, \rho_u\rangle\rangle) \rightsquigarrow_{exp} \mathbf{let}\ \rho_u = e_{\omega_u}\ \mathbf{in}\ \vec{\rho}[c](e_{\omega_c})} \tag{8.28}$$

$$\frac{strexp \rightsquigarrow_{exp} e_\omega}{strexp : sigexp \rightsquigarrow_{exp} e_\omega} \tag{8.29}$$

$$\frac{strexp \rightsquigarrow_{exp} e_\omega}{strexp :> sigexp \rightsquigarrow_{exp} e_\omega} \tag{8.30}$$

Figure 8.6: Translation of structure expressions to System F_ω

because type definitions and datatype declarations play no role in the dynamic semantics. Rules 8.23 and 8.24 must produce declarations binding the entity variable of the structure or functor instead of the syntactic variable. The rule 8.24 will use the judgment $\vdash \Sigma \rightsquigarrow_{sig} k$ defined in Fig. 8.7 to extract the kind information from the semantic signature for the functor parameter. The type for the F_ω λ abstraction is calculated separately from the same semantic signature.

Fig. 8.6 gives the translation of the module language to F_ω. Rule 8.28 translates structure applications. The functor being applied can be translated by translating its entity path to the corresponding F_ω record projection form. The actual argument structure expression is translated to e_ω, but the functor is expecting a parameter of the form defined in full signature M. The associated coercion structure expression $\mathbf{let}\ \rho_u = \varphi_u\ \mathbf{in}\ (\!|\eta|\!)$ can help produce coerced type and value arguments. For the type argument, the rule translates the coercion entity declaration to F_ω via the tycon function defined in Fig. 8.11. For the value argument, reusing the uncoerced expression by aliasing $\rho_{par} = \rho_u$ is sufficient because the

$\boxed{\vdash \Sigma \rightsquigarrow_{knd} k}$

$$\frac{}{\vdash \emptyset_{sig} \rightsquigarrow_{knd} \{\}} \tag{8.31}$$

$$\frac{\vdash \Sigma \rightsquigarrow_{knd} k}{\vdash [x \mapsto \mathbb{T}]\Sigma \rightsquigarrow_{knd} k} \tag{8.32}$$

$$\frac{\vdash \Sigma \rightsquigarrow_{knd} k}{\vdash [x \mapsto \mathbb{C}^{\lambda}]\Sigma \rightsquigarrow_{knd} k} \tag{8.33}$$

$$\frac{\vdash \Sigma \rightsquigarrow_{knd} k}{\vdash [x \mapsto (\rho, n)]\Sigma \rightsquigarrow_{knd} \{\rho :: \Omega^n \to \Omega\} \uplus k} \tag{8.34}$$

$$\frac{\vdash \Sigma \rightsquigarrow_{knd} k \qquad \vdash \Sigma' \rightsquigarrow_{knd} k'}{\vdash [x \mapsto (\rho, \Sigma)]\Sigma' \rightsquigarrow_{knd} \{\rho :: k\} \uplus k'} \tag{8.35}$$

$$\frac{\vdash \Sigma_x \rightsquigarrow_{knd} k_x \qquad \vdash \Sigma \rightsquigarrow_{knd} k \qquad \vdash \Sigma' \rightsquigarrow_{knd} k'}{\vdash [x \mapsto (\rho, \Pi\rho_x : \Sigma_x.\Sigma)]\Sigma' \rightsquigarrow_{knd} \{\rho :: k_x \to k\} \uplus k'} \tag{8.36}$$

Figure 8.7: F_ω kind synthesis

prior elaboration already guarantees that none of components in ρ_u that is not specified in the formal parameter signature will be accessed. Furthermore, none of the variables are being renamed.

The F_ω kinds, tycons, and types for tycon content of structures, type content of structures, and structures themselves are record kinds, record tycons, and record types respectively. Each label-field pair is a pair of an entity variable and the kind, tycon, and type of the entity referred to by that entity variable. Let lks, lcs, and lts be a sequence of record label-field for kinds, tycons, and types respectively. The notation $lks \uplus lks'$ appends two record kinds. The notation is extended in the usual way to append tycons and types. No shadowing is expected because all records field names will be entity variables.

Fig. 8.7 gives the rules for producing F_ω kinds from semantic signatures. Value specs and type definition specs do not contribute to the kind (rules 8.32 and 8.33). Open tycon specs do contribute. Rule 8.34 produces the kind field $\rho :: \Omega^n \Rightarrow \Omega$ such that n is the arity of the open tycon spec. Rules 8.35 and 8.36 construct kinds for structure and functor specs

$\boxed{\vdash M \rightsquigarrow_{tyc} c}$

$$\frac{\Upsilon^{clo}\Upsilon^{lcl} \vdash \Sigma \rightsquigarrow_{spec} lcs}{\vdash (\Sigma, \langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle) \rightsquigarrow_{tyc} \{lcs\}} \tag{8.37}$$

$\boxed{\Upsilon \vdash \Sigma \rightsquigarrow_{spec} lcs}$

$$\frac{}{\Upsilon \vdash \emptyset_{sig} \rightsquigarrow_{spec} \{\}} \tag{8.38}$$

$$\frac{\Upsilon \vdash \Sigma \rightsquigarrow_{spec} lcs}{\Upsilon \vdash [x \mapsto \mathfrak{T}]\Sigma \rightsquigarrow_{spec} lcs} \tag{8.39}$$

$$\frac{\Upsilon \vdash \Sigma \rightsquigarrow_{spec} lcs}{\Upsilon \vdash [x \mapsto \mathfrak{C}^{\lambda}]\Sigma \rightsquigarrow_{spec} lcs} \tag{8.40}$$

$$\frac{\Upsilon \vdash \Sigma \rightsquigarrow_{spec} lcs}{\Upsilon \vdash [x \mapsto (\rho, n)]\Sigma \rightsquigarrow_{spec} \rho = \mathsf{inj}(\Upsilon(\rho)), lcs} \tag{8.41}$$

$$\frac{\Upsilon(\rho) \vdash \Sigma \rightsquigarrow_{spec} lcs \qquad \Upsilon \vdash \Sigma' \rightsquigarrow_{spec} lcs'}{\Upsilon \vdash [x \mapsto (\rho, \Sigma)]\Sigma' \rightsquigarrow_{spec} \rho = \{lcs\}, lcs'} \tag{8.42}$$

$$\frac{\vdash (\Sigma^f, \Upsilon(\rho)) \rightsquigarrow_{fctspec} c \qquad \Upsilon \vdash \Sigma \rightsquigarrow_{spec} lcs'}{\Upsilon \vdash [x \mapsto (\rho, \Sigma^f)]\Sigma \rightsquigarrow_{spec} \rho = c, lcs'} \tag{8.43}$$

$\boxed{\vdash F \rightsquigarrow_{fctspec} c}$

$$\frac{\vdash \Sigma \rightsquigarrow_{knd} k \qquad \vdash \varphi \rightsquigarrow_{tyc}^{strexp} c}{\vdash (\Pi\rho : \Sigma.\Sigma', \langle \lambda\rho.\varphi; \Upsilon \rangle) \rightsquigarrow_{fctspec} \lambda\rho :: k.c} \tag{8.44}$$

Figure 8.8: F_ω tycon synthesis

$\boxed{\vdash \mathfrak{C}^{nf} \leadsto^{nf}_{type} t}$

$$\frac{}{\vdash \alpha \leadsto^{nf}_{type} \alpha} \tag{8.45}$$

$$\frac{\vdash \mathfrak{C}^{nf}_1 \leadsto^{nf}_{type} t_1 \ldots \vdash \mathfrak{C}^{nf}_2 \leadsto^{nf}_{type} t_n}{\vdash \tau^n \left(\overrightarrow{\mathfrak{C}^{nf}}\right) \leadsto^{nf}_{type} \mathsf{inj}(\tau^n)(t_1)\ldots(t_n)} \tag{8.46}$$

$\boxed{\Delta, \Upsilon \vdash \mathbb{C}^s \leadsto^{tyc}_{type} t}$

$$\frac{\alpha \in \Delta}{\Delta, \Upsilon \vdash \alpha \leadsto^{tyc}_{type} \alpha} \tag{8.47}$$

$$\frac{\begin{array}{c}\overrightarrow{\mathfrak{C}^s}\ s.t.\ \Delta, \Upsilon \vdash \mathbb{C}^s_i \forall \mathbb{C}^s_i \in \overrightarrow{\mathbb{C}^s} \\ \Upsilon(\vec{\rho})(\overrightarrow{\mathfrak{C}^s}) \Downarrow_{mt} \mathfrak{C}^{nf} \qquad \vdash \mathfrak{C}^{nf} \leadsto^{nf}_{type} t\end{array}}{\Delta, \Upsilon \vdash \vec{\rho}\left(\overrightarrow{\mathbb{C}^s}\right) \leadsto^{tyc}_{type} t} \tag{8.48}$$

$\boxed{\Upsilon \vdash \mathbb{T} \leadsto^{t}_{type} t}$

$$\frac{\emptyset_{knd}, \Upsilon \vdash \mathbb{C}^s \leadsto^{tyc}_{type} t}{\Upsilon \vdash \mathsf{typ}(\mathbb{C}^s) \leadsto^{t}_{type} t} \tag{8.49}$$

$$\frac{[\vec{\alpha}], \Upsilon \vdash \mathbb{C}^s \leadsto^{t}_{type} t}{\Upsilon \vdash \forall \vec{\alpha}.\mathbb{C}^s \leadsto^{t}_{type} \forall \vec{\alpha}.t} \tag{8.50}$$

Figure 8.9: F$_\omega$ synthesis of types from normal form semantic tycons, relativized monotypes, and type expressions

$$\boxed{\Upsilon \vdash \Sigma \rightsquigarrow_{type} lts}$$

$$\frac{}{\Upsilon \vdash \emptyset_{sig} \rightsquigarrow_{type} \{\}} \tag{8.51}$$

$$\frac{\Upsilon \vdash \mathbb{T} \rightsquigarrow^{t}_{type} t \qquad \Upsilon \vdash \Sigma' \rightsquigarrow_{type} lts}{\Upsilon \vdash [x \mapsto \mathbb{T}]\Sigma' \rightsquigarrow_{type} x : t, lts} \tag{8.52}$$

$$\frac{\Upsilon \vdash \Sigma' \rightsquigarrow_{type} lts}{\Upsilon \vdash [x \mapsto \mathbb{C}^{\lambda}]\Sigma' \rightsquigarrow_{type} lts} \tag{8.53}$$

$$\frac{\Upsilon \vdash \Sigma' \rightsquigarrow_{type} lts}{\Upsilon \vdash [x \mapsto (\rho, n)]\Sigma' \rightsquigarrow_{type} lts} \tag{8.54}$$

$$\frac{\Upsilon(\rho) \vdash \Sigma \rightsquigarrow_{type} lts \qquad \Upsilon \vdash \Sigma' \rightsquigarrow_{type} lts'}{\Upsilon \vdash [x \mapsto (\rho, \Sigma)]\Sigma' \rightsquigarrow_{type} \rho : \{lts\}, lts'} \tag{8.55}$$

$$\frac{\begin{array}{c}\vdash \Sigma_x \rightsquigarrow_{knd} k \qquad \Upsilon \vdash \Sigma_x \rightsquigarrow_{type} lts_x \\ \Upsilon \vdash \Sigma \rightsquigarrow_{type} lts \qquad \Upsilon \vdash \Sigma' \rightsquigarrow_{type} lts'\end{array}}{\Upsilon \vdash [x \mapsto (\rho, \Pi\rho_x : \Sigma_x.\Sigma)]\Sigma' \rightsquigarrow_{type} \rho : (\forall \rho_x :: k.\{lts_x\} \rightarrow \{lts\}), lts'} \tag{8.56}$$

Figure 8.10: F_ω type synthesis

$$\mathsf{coerce}(\rho_u, lts) = \begin{cases} \{\} & lts = \{\} \\ \{l_0 = \rho_u.l_0\} \uplus \mathsf{coerce}(\rho_u, lts') & lts = \{l_0 : t_0\} \uplus lts' \end{cases} \tag{8.57}$$

Figure 8.11: Coercion of dynamic components

from the semantic signature and functor signature respectively.

Fig. 8.8 defines the judgment that calculates the tycons from the full signature of a structure. Again, value specs, and type definition specs do not contribute to the tycon (rules 8.39 and 8.40). Rule 8.41 translates an open spec by looking up the entity environment for the tycon entity associated with the open tycon spec. The rule 8.42 translates the structure spec signature in the context of the entity environment calculated from the structure entity ($\Upsilon(\rho) = \langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle = \Upsilon^{clo}\Upsilon^{lcl}$). To calculate the tycon for a functor component, rule 8.43 must supply rule 8.44 the functor signature and entity from looking up the entity environment. Rule 8.44 calculates the kind for the functor parameter using the parameter signature and also the tycon for the functor body from the functor body signature and closure entity environment. The judgment relies on $\vdash \varphi \rightsquigarrow^{strexp}_{tyc} c$ which translates the structure entity expression to the corresponding F_ω tycon expression, which is the same because there is an embedding of the structure entity expression language to the F_ω tycon expression language. The above statement fully defines $\vdash \varphi \rightsquigarrow^{strexp}_{tyc}$.

Fig. 8.10 gives the rules for producing F_ω types from semantic signatures. These two judgments decompose semantic signatures into kinds (for classifying the static components) and types (for classifying the dynamic components). Kind synthesis is used in rule 8.56 to produce the kind annotation for the polymorphic quantifier.

Let the $\mathsf{env}(lts) = [\rho : t]$ where $\rho_i : t_i \in lts$.

Fig. 8.11 defines the function coerce that coerces an F_ω expression e_ω to a record type lts. It thins out the value part by dropping components not in the specified type. The resultant expression should have the target type $\{lts\}\{c/\rho\}_{tyc}$ where lts is parameterized by ρ. The type-level coercion was already taken care of by F_ω tycon synthesis.

Lemma 17 (Correctness of Value-Level Coercion)

If $E \vdash \{lts_u\} :: k$, $E \vdash \rho_u : \{lts_u\}$, *for all* $l \in dom(lts_u) \cap dom(lts_c)$ $t_c\{c/\alpha\} = t_u$ *such that* $l : t_c \in lts_c$ *and* $l : t_u \in lts_u$, *and* $\mathsf{coerce}(\rho_u, c, lts_c) = e_\omega$, *then* $E \vdash e_\omega : \{lts_c\{c/\alpha\}\}$.

8.4 Soundness

Lemma 18

If a structure expression strexp elaborates to $(M, \_)$ *and strexp translates to* e_ω, *then* $\vdash e_\omega : t$ *and* $\vdash M \rightsquigarrow_{type} t$.

The proof of the above lemma is sketched out in more detail in the proof appendix. It relies on much of the same machinery as Shao's proof [66]. The proof diverges in Shao's memoization of pre-translated F_ω tycon and types in the functor representation during elaboration of the functor. Furthermore, this proof must deal with the coercive signature matching semantics which is not present in Shao's main elaboration-translation semantics.

CHAPTER 9

FULLY TRANSPARENT GENERATIVE FUNCTORS

There are implicitly two types of functors, those that have static effects, *generative* functors, and those that do not, *pure* functors. In the higher-order case, pure functors can be fully described using applicative functor-like syntactic extensions. The application of a functor to the same argument twice should give the same result because pure functor arguments are truly referentially transparent. This observation is certainly false when considering generative functors which have static effects.

Type generativity is typically modeled using existential types. Each time an existential type is unpacked, a fresh abstract type is created. Dreyer, Crary, and Harper [16] advocated treating opaque sealing as a computational effect, a runtime fresh type generation. Dreyer [12] reconciles this account of generative types with recursion via a backpatching type semantics.

9.1 Type Sharing

Type sharing is necessary to support type-safe composition of modules. When a functor is parameterized over two modules which each declare an abstract type, the elaborator assumes that the two types are different. Consider the following example:

```
signature S0 = sig
  type t
  val a : t
end

signature S1 = sig
  type t
  val f : t -> int
end

functor F(structure X:S0
              structure Y:S1) = struct
  val y = Y.f(X.a)
end
```

The above functor will not type check because there is no reason to believe *a priori* that X.t and Y.t are equivalent. F can certainly be applied to X and Y such that X.t $\neq$ Y.t.

If F is applied to X and Y such that X.t = Y.t, then this is an instance of the *diamond import problem.* Simply put, when a single component, which may be a type or a structure, is exported in two different structures, it must be reconciled, retaining its identity and thus reflexive equivalence. The ML solution to this problem is type sharing. *Type sharing* can be expressed in four forms. **sharing type** constraints specifies equivalences on symbolic paths to types. The right hand side of **sharing type** must consist of only symbolic paths to types.

```
functor F(structure X:S0
              structure Y:S1
              sharing type X.t = Y.t) =
      ...
```

where **type**, which is derived from the categorical notion of fibrations, defines abstract type specs *post hoc.* The left hand side of the where **type** should be a signature S1 containing the type to be constrained. The right hand side consists of the name of the type t in S1 and an arbitrary type expression such as X.t or X.t list.

```
functor F(structure X:S0
              structure Y:S1 where type t = X.t) =
      ...
```

Definitional type specs constrains type specs inside of signatures. Consequently, they only apply if the signature is expanded out in the functor formal parameter. From the perspective of implementation, where **type** elaborates to definitional type specs by pushing down the where **type** definition to the relevant type spec.

```
functor F(structure X:S0
              structure Y:sig
                 type t = X.t
                 val f : t -> int
              end) = ...
```

The type sharing can also be expressed in terms of signatures parameterized on structures. This alternative was considered and rejected during the initial development of the ML module system in favor of where **type** and **sharing type** because parameterized signatures require programmers to anticipate all such sharing when writing the signatures.

```
signature S2(Z) = sig
   type t = Z.t
   val f : t -> int
end

functor F(structure X:S0
              structure Y:S2(X)) =
```

...

CHAPTER 10

SEPARATE COMPILATION

Early module systems had two main guiding principles. First, modules should be partitioned into interface and implementation. The interface describes the form of a module, the implementation the details and mechanisms. Second, one should be able to compile modules independently from one another with only the aid of interfaces to resolve intermodule dependencies. In languages with simple type systems, reconciling these two goals is usually quite straightforward.

The fundamental tension between separate compilation and an expressive, flexible module system is this: separate compilation requires the language to expose as much as possible in the interface or signature language but the goal of modularity is to keep a distinction between interface and implementation. Reconciling these two goals is a challenge in the ML language because of the presence of static effects. These effects, embodied in type generativity, are computed during the compiler elaboration process. Normally, a signature language should only contain information about the form of the module being described. Inclusion of static effect information would amount to adding information about the implementation. I contend that including static effects would run against the purpose of signatures. Moreover, it would entail significant code duplication and encourage further breakdown of the distinction between interface and implementation.

As Russo noted because ML type information is spread between signature and realization, module clients that depend on the realization cannot be typechecked separately from the module.

Syntactic signatures cannot even distinguish between functors that contain static effects and those that do not much less the nature of the static effect. For example, a functor body signature may contain a datatype declaration. Naively, an elaborator might consider this an indication that the functor body has a static effect, generating a fresh datatype upon each application. However, this assumption is incorrect because this specification matches both datatype replication declarations, which do not generate a fresh type, *bona fide* generative datatype declarations.

```
datatype t = K
structure M : sig datatype t = K end =
struct
   datatype t = datatype t (* pure *)
end
```

Furthermore, the signature spec could have been **type** t or **datatype** t = **datatype** t. Consequently, an elaborator cannot handle the special case of pure functors by looking at the functor signature because they are indistinguishable from the signature.

10.1 Target Calculi and Type Sharing

ML compilers often use enriched System F_ω-like intermediate languages. Unlike the module system, System F_ω does not have any native support for type sharing. Naively compiling the type sharing example in sec. 9 produces $\Lambda t.\lambda a : t\Lambda s.\lambda f : s \to int.f(a)$, which is ill-typed. There are a number of solutions to this problem.

1. Singleton kinds express the type sharing as a kinding of type s: $\Lambda.\lambda a : t.\Lambda s : \{t\}.\lambda f : s \to int.f(a)$. This technique is used in the TIL/TILT compilers and the Harper-Stone semantics.

2. The s dependency can be abstracted and expressed in terms of System F_ω's type constructors and expressions: $\lambda t.\lambda a : t.\lambda f : (\lambda s.s \to int)\ t.f(a)$

3. FLINT preprocesses type sharing by replacing defined type names with their definitions: $\Lambda t.\lambda a : t.\lambda f : t \to int.f(a)$

10.2 Alternatives

Despite its difficulty, true separate compilation is still desirable. As others have noted, one solution is to distinguish between compilation units and modules. Indeed, OCaml, Moscow ML, and Swasey's SMLSC take exactly this approach. In OCaml, each separate OCaml source file is a compilation unit which is a kind of quasi-module which must be coupled with an mli interface file. Because there is no compilation unit-level parameterization, the question of higher-order functors does not apply. Moscow ML only considers an opaquely seal module to be a compilation unit.

10.3 Relationship to Type Classes

As Wehr notes, module encodings in terms of type classes also conflict with separate compilation. Because type classes are ordinarily transparent, they pose lead to type propagation problems similar to module systems.

10.4 Comparison to Shao

Shao's calculus KMC aims to support both separate compilation and full transparency in higher-order modules by means of higher-order type constructors. The calculus is based on Harper-Lillibridge and Leroy's abstract approach, utilizing existentials for abstract types. The general idea is to factor out all the volatile components to a single higher-order type constructor. Inside of functor signatures, one can use selectors to project out the volatiles from this type constructor. Signatures are then parameterized by this higher-order type constructor.

The Apply functor's signature would be $\lambda u_1 : K.\Pi X : (\exists u_2.SIG[u_2]).(SIG[u_1[\overline{X}]])$. The higher-order type constructor u_1 plays a role similar to structure realizations and functor realization parameter. However, whereas structure realizations are completely distinct and nonoverlapping with signatures in M, the higher-order tycon is mixed in with syntactic signatures. This affords Shao's calculus the possibility of separate compilation.

KMC's main limitation, however, is that it is still not flexible enough to express generative types and abstract types. Though KMC can model opaque sealing via existential types, it still cannot type check programs such as the following:

```
signature SIG = sig type t val x : t end
```

```
funsig FSIG(X: SIG) = SIG

structure S = struct type t = int val x = 1 end

functor APPS (F:FSIG) = F(S)

functor G2(X: SIG) =
struct
   datatype t = A
   val x = A
end
```

CHAPTER 11
CONCLUSION

The design of higher-order module systems is indeed rich and complex. Much of the richness is due to the introduction of functor actions and transparent signature matching semantics. While the β-reduction semantics for functor applications is both natural and intuitive, supporting such semantics in the presence of higher-order functors requires the richer static representation that the entity calculus affords. This present account improves the state-of-the-art in module system semantics by providing a formal account of true higher-order semantics and type generativity without resorting to re-elaboration of functor bodies.

11.1 Future Work

The entity calculus is a simple and natural way to represent module computation. It remains a question whether it can be easily adapted to support recursive modules while maintaining the strong normalization property.

The main argument that the design of the entity calculus makes is that functor actions should be calculated by the elaborator and not written down in a syntactic signatures. This kind of functor action inference suggests that inference may be able to play a larger role in module system semantics.

APPENDIX A
SML/NJ ELABORATION SEMANTICS

A.1 Signatures

$\boxed{\Gamma, \Upsilon, C_{ep}, C_{sig} \vdash \mathit{sigexp} \Rightarrow_{sig} \Sigma}$

Signatures elaborate in the context of the following semantic objects:

static environment (Γ) The static environment interprets any nonlocal symbolic paths mapping them to semantic representations.

entity environment (Υ) The entity environment interprets all external entity paths in datatype and datatype replication specifications and any other relativized type expressions.

entity path context (C_{ep}) The entity path context helps relativize types in where clauses, type, exception, and datatype/replication specifications. The elaborator also looks up the entity path context to obtain the entity path for structure definition specs. [so structure references are also relativized in certain circumstances] This is an auxiliary implementation mechanism used only in the process of building static representations – not part of the representations themselves.

signature context (C_{sig}) A signature context is a list of spec elements, specifically, structure spec elements. These spec elements are semantic representations, the product of elaborating earlier or outer scope syntactic specs. It plays a very specialized role. When elaborating datatype replication specs, the tycon of the datatype being replicated may be either relativized or unrelativized. If the tycon has been relativized, then the elaborator needs to look up the original tycon spec. The tycon spec must be either in the same or an outer context, both of which are recorded in the signature context. [seems sightly unexpected that the static environment can't play this role.]

The signature elaboration process produces a semantic representation of a signature (Σ in fig. 7.1).

The main steps of signature elaboration are:

1. Elaborate outer where definitions – the ones applying directly to the current signature

2. Check for duplicate names

3. Relativize types for value, type, and data constructor specifications

4. Decorate static component specs with fresh entity variables

5. Push where definitions down to relevant type specification [does this process involve adjusting any relative references, i.e., entity paths?]

Signature elaboration proceeds by first eliminating where type constraints by translating the constraints into a list of relativized semantic definitional type constructors. This list of type constructors is put aside until the very end of signature elaboration. At that point, the elaborator goes through all the specs in the signature and applies all applicable definitional constraints. In SML/NJ, definitional constraints only apply to open types and datatypes. The precludes the possibility of overconstraining a tycon with multiple potentially contradictory definitions such as the following:

```
signature S = sig type t = int end where type t = bool
```

However, observe that not all overconstraining by type definitions and where type results in contradiction. I describe a policy that relaxes the semantics of where type to permit definitional type names on the left hand side in a later section.

For functor signatures, the elaborator first processes the parameter signature and then elaborates the functor body signature using a static environment extended with a binding mapping the parameter's symbolic name to a full signature where the free instantiation realization which will be subsequently ignored. The resultant body signature and parameter signature with parameter entity variable are put together as the semantic representation of the functor signature.

When elaborating structure specs, we have a special case. In order to elaborate any specs that may follow a structure spec, we need to extend the static environment with a binding for the structure spec. Unfortunately, we do not have a full signature for that structure spec because there is no realization. Hence, the elaborator creates a placeholder free instantiation as in the case above.

During signature elaboration, the elaboration of a spec may require the result of the elaboration of previous specs. The notation $\Gamma[\Sigma_1]$ puts the result of the elaboration of the preceding specs in the static environment by dropping the entity variables because static environment bindings do not include entity variables.

The rebind function rebinds q to a new spec in the semantic signature.

A.2 Functor Signatures

$\boxed{\Gamma, \Upsilon, C_{ep}, C_{sig} \vdash fsgexp \Rightarrow_{fsg} S^f}$

Functor signatures elaborate in a context of:

static environment

entity environment

signature context

entity path context

Elaboration produces a semantic representation of a functor signature.

The main steps are:

1. Elaborate the parameter signature expression

2. Extend the static environment with a binding from parameter name to parameter signature. This binding is the same as the special binding form used for structure specs.

3. Extend the signature context with a binding from parameter name to decorated structure spec (with a fresh entity variable for the functor parameter) corresponding to the parameter structure. The elaborator needs this decorated structure spec because the body of the functor may contain datatype replication specs that replicate some datatype whose tycon is declared in the parameter.

4. Elaborate the body signature expression

A.2.1 Signature Instantiation

$\boxed{\Upsilon \vdash \Sigma \uparrow \Upsilon'}$

Signature instantiation produces a free instantiation of the given signature Σ with the least amount of type sharing necessary to satisfy the signature.

A.3 Declarations

$\boxed{\Gamma, \Upsilon, C_{ep}, C_{elab} \vdash \mathit{decl} \Rightarrow_{\mathit{decl}} (\eta, \Gamma', \Upsilon')}$

Elaboration of declaration uses the following context:

static environment Elaboration augments static environments with new bindings corresponding to the declarations. It also uses them to interpret external symbolic paths.

entity environment The elaborator builds up an entity environment for structures inside of functors

elaboration context (C_{elab}) The elaboration context distinguishes between top level, in structure, in functor, and in signature contexts. The primary distinction is whether elaboration is occurring inside a functor or not. When inside of a functor, entity declarations must be produced whereas they are unnecessary for other contexts. The point when the elaborator first enters into a functor also partitions externally-defined type constructors which are rigid and locally-defined type constructors which are flexible.

entity path context The entity path context is used to relativize entity paths.

The elaboration process yields the following principal results:

typed abstract syntax

entity declaration

static environment

entity environment

A.3.1 Structure Declaration

The main steps of structure declaration elaborations are:

1. elaborate signature expressions if any
2. elaborate structure definiens expression
3. signature matching
4. build entity environment from entity environments produced by first two steps
5. build typed abstract syntax from coercion
6. add binding to full signature to static environment

A structure declaration consists of a name, constraining signature expression, and definiens expression. The elaborator first elaborates both the signature expression and the structure definition (a structure expression). Then the elaborator does signature matching on the resultant semantic signature, full signature, typed abstract syntax, and entity expression for the structure to produce a typed abstract syntax declaration, full signature, and entity expression for the coerced structure. The entity environment delta from elaborating the structure does not play a role in signature matching.

If a signature constraint is present, the entity environment delta from the structure will need to be extended with a binding that maps an entity variable for the short-lived temporary binding of the uncoerced structure. This new entity environment will be combined with a binding for entity variable for the coerced realization and the contextual entity environment to yield the final result entity environment.

The full signature of the possibly coerced structure is combined with the typed abstract syntax to produce a structure binding in typed abstract syntax. The principal results of structure declaration elaboration include this typed abstract syntax, a static environment extended with a binding for the relativized coerced full signature, the final result entity environment, and an entity declaration mapping the entity variable for the coerced structure to the entity expression for the same.

[Idea: The only difference between the bindStr in the STRB and the resStr in the LETstr part of the STRB is the dynamic access. Couldn't bindStr be used in both places?] [Idea: The cruft of ascription handling in elabStrbs is likely due to the lingering presence of AbsDec along with the currently used forms of Transparent and Opaque constraints. Eliminate AbsDec and things may simplify.] [Idea: The elaboration code for signature constraints is duplicated in the elabStr[ConstrainedStr] and elabStrbs case. This duplication does not appear necessary. Can't ConstrainedStr be expanded into a structure binding or vice versa?]

A.3.2 Signature Declaration

The main steps of signature declaration elaboration are:

1. elaborate signature expression

2. instantiate resultant signature

3. add signature to static environment

For signature declarations which can only occur at the top level, the elaborator may attempt to instantiate the elaborated signature to serve as a sanity check. This check is not necessary for correctness, but it does eliminate certain uninhabitable signatures at elaboration-time. [Idea: Why don't we do something similar for functor signatures?]

A.3.3 Functor Declaration

1. elaborate functor

2. build typed abstract syntax

3. extend entity environment with entity declaration from functor elaboration if in functor according to elaboration context

4. add full functor signature to the static environment

Upon elaboration of a functor declaration, the elaborator must extend the entity environment with the delta produced during elaboration of the functor and a binding for the functor entity. Furthermore, the functor entity declaration must be produced. If this functor declaration is not nested, the extended entity environment and entity declaration are unnecessary.

[Idea: Currying value-level functions has definite benefits in terms of partial evaluation. Is currying functor parameters as prevalent and if so does it carry the same benefits?]

A.4 Structures

$\boxed{\Gamma, \Upsilon, C_{ep}, C_{elab} \vdash \mathit{strexp} \Rightarrow_{str} (M, \varphi, \Upsilon')}$

Structures elaborate in the following context:

static environment

entity environment

elaboration context The explanation is in the above section.

entity path context The elaborator will look up functors (as in functor application) in the entity path context. The resultant entity path will be used to fill out the entity expression for the functor application. Structure variables will also be looked up in the entity path context to translate it into a corresponding entity expression.

The principal results are:

typed abstract syntax Binds a symbolic name to the possibly coerced full signature and a body comprised of a let-expression with the typed abstract syntax from the uncoerced structure and the bindings extracted from the static environment

full signature (M) The full signature is the principal product of structure expression elaboration. It gives the static description of the structure expression.

entity expression This entity expression is used to form the entity declaration in the case where the above structure expression is found inside of another functor's body, thus necessitating recording of the entity function.

entity environment

A.4.1 Base Form

The main steps of base structure expression elaboration are the following:

1. elaborate declarations

2. extract full signature from resultant static environment

3. combine entity environments from first two steps

4. build typed abstract syntax

The elaborator deals with the basic form of a structure by elaborating the constituent declarations and extracting the signature the resultant static environment. The elaborator can infer or extract a full signature from the static environment derived during elaboration of a structure. Extraction forms a list of specifications such that each value, data constructor, structure, functor, and type constructor binding in the static environment has a corresponding specification. Entity variables for each specification is obtained by looking up the entity path context.

Extraction computes the specifications from the full signatures of the structures and functors. Type constructor specifications are minimal. They only mention the name and arity of the said type constructor, thus are opaque.

More interestingly, extraction must also build entity declarations and entity environments to go along with each specification. The entity declarations are simply declarations that bind the a fresh entity variable to the entity path of the static entity. The result entity environments bind this fresh entity variable to the realization. All structure, functor, and value bindings are also returned as part of the result.

The entity environments and declarations produced by declaration elaboration and signature extraction are combined with the contextual entity environment and used to form the final structure realization. The elaborator must concatenate the entity environment from declaration elaboration onto the result entity environment because of possible inexplicitness. The extracted signature and realization are combined to form the full signature. The entity declarations from elaboration and extraction are sequenced together to form the final structure entity expression. The typed abstract syntax from declaration elaboration and the bindings from extraction and the static environment bindings returned from extraction are combined to form the typed abstract syntax for this structure.

A.4.2 Signature Extraction

$\boxed{C_{ep}, C_{elab} \vdash \Gamma \hookrightarrow (M, \eta)}$

Signature extraction requires the following environments:

static environment The static environment is decomposed into its bindings from which the elaborator builds a signature.

entity path context The entity path context is used to look up the entity paths (*i.e.*, the relativized entity paths) for each binding in the static environment. It will also be used to relativize entity paths.

elaboration context (C_{elab}) Types will be relativized only if elaboration context is in functor.

The process produces the following:

spec elements (elems) specs produced by relativizing types in static environment bindings and reverse lookup of structures in the entity path context

static environment bindings (bindings) the list keeps only the structure, functor, tycon, value, and data constructor bindings. These bindings will become the let-body in the typed abstract syntax.

entity declaration (η) This entity declaration contains an entity declaration for each static (*i.e.*, non-value) spec. The entity declaration is either a variable entity or a constant entity depending on whether there is a corresponding entity path in the entity path context. They are propagated directly through elaboration.

entity environment (Υ') The entity environment, which will be the extracted (inferred) structure realization, is extended with the entities from structure, functor, and tycon bindings.

When signature extraction looks at an entity, there are two main cases:

tycon Volatile tycons have stamps which uniquely identify them in the static environment. By exhaustively scanning through the static environment, signature extraction builds up the entity path leading up to the binding occurrence (not applied occurrence) of that entity. The binding occurrence of the entity is unique in a module system without sharing type constraints. In the presence of sharing type constraints, the canonical entity path for an entity can be defined as the path leading up to the first binding occurrence where first is first in terms of syntactic ordering.

structure and functor Structure and functor entities must be extended with its binding occurrence entity variable. This entity variable will identify the associated spec in the signature and by extension help calculate the entity path.

When signature extraction encounters an entity that entity may or may not be already in the entity environment. If it is already there, then

Note that entity environments are not injective during entity evaluation because entity paths will be looked up and another entity variable may now be mapped to that entity. Because these entity environments are ultimately used during elaboration when propagated through by functor application elaboration, this means that the entity environment may be non-injective throughout elaboration.

```
functor F(X:sig type t end) =
struct
  type u = X.t
end
```

Lemma 19

The entity environment is injective during elaboration.

Proof: Entity environments are constructed by signature instantiation, structure expression elaboration, signature extraction, functor elaboration, signature matching, and entity evaluation.

During entity

entity evaluation Entity evaluation constructs new entity environments by binding fresh entity variables to the result of entity evaluation. Will an entity declaration or expression ever evaluate to an entity that already occurs in the entity environment? Yes, evaluating an entity path will look it up in the entity environment thereby yielding an entity that already exists in the environment.

■

Lemma 20

There exists a retraction of the entity environment lookup function.

Proof: This follows from lem. 19. ■

Static environment bindings contain tycons, signatures, and functor signatures necessary to reconstitute the specs in a signature. Given a static environment binding for a static entity, the corresponding entry in the entity path context gives the relativized entity path. If the relativized entity path is a single entity variable, that entity variable is the one for this signature spec. Otherwise, a fresh entity variable is used for the spec. In this latter case, the entity declaration will either be equal to the realization of the entity or an entity path depending on whether the entity is in the entity path context.

Signature and functor signature bindings are not permitted inside of structures so signature synthesis ignores them.

A.4.3 Functor Application

$\boxed{\Gamma, C_{ep} \vdash {}^{\theta}F({}^{\varphi}M) \rightsquigarrow (M_{res}, \varphi_{res})}$

Functor application requires the following:

static environment (Γ) The static environment is used to signature match the argument full signature to the formal parameter signature.

entity path context (C_{ep}) The entity path context is used as the evaluation entity environment when evaluating the application of the entity function.

full functor signature (F) This is the full functor signature for the functor in the application.

functor entity expression (θ) This is the functor entity expression for the functor in the application. This will be used to constructor the application structure entity expression.

full signature (M) This is the full signature for the argument structure.

structure entity expression (φ) This is the structure entity expression for the argument structure. This entity expression will be coerced by signature matching, the result of which will be used to construct the application structure entity expression.

The process results in:

full signature (M_{res}) This is the full signature of the functor application result.

entity expression (φ_{res}) This is the application structure entity expression, memoizing all latent functor actions.

The main steps of functor application elaboration are:

1. look up functor in static environment
2. elaborate argument structure expression
3. extend entity environment with uncoerced structure realization
4. signature match uncoerced argument with formal functor parameter signature
5. evaluate functor body entity expression with coerced argument realization

After elaborating the argument structure expression and functor, the elaborator looks up the entity path for the functor to form either a functor variable or constant entity expression. [Can constant be eliminated?] Then, the elaborator does signature matching on the functor formal parameter signature and the actual argument structure using the closure entity environment in the functor realization. The elaborator then evaluates the functor body replacing the formal parameter with the coerced argument realization. The resultant evaluated functor body realization is combined with the functor body signature to form the full signature of the result. The coerced argument full signature is also used to produce the usual typed abstract syntax and structure entity expression, an apply.

The semantics for static functor application varies considerable among ML variants. Leroy's semantics only permits applying functors to paths, thus requiring all functor arguments to be A-normalized. DCH does not impose coercive signature matching in the functor argument. The key distinction of GEN's semantics is the judgment $\vdash \varphi_{app} \Downarrow \Upsilon_{app}$. This judgment evaluates only the static content of the functor application. In DCH, the argument structure is substituted directly into the result signature, thus requiring a complex theory of projectibility.

A.5 Functors

$\boxed{\Gamma, \Upsilon, C_{ep}, C_{elab} \vdash fctexp \Rightarrow_{fct} (F, \theta, \delta\Upsilon')}$

Functor elaboration assumes a context consisting of:

static environment

entity environment

elaboration context

entity path context

and produces four principal results

typed abstract syntax comprised of a functor binding containing a copy of the full functor signature and and the typed abstract syntax for a functor containing the full signature for the free instantiation of the parameter, the full signature of the functor body, and the typed abstract syntax for the body

full functor signature (F)

functor entity expression

delta entity environment

A.5.1 Functor Variable

The main steps of functor variable elaboration are:

1. look up symbolic path to functor in static environment to get full functor signature
2. look up functor in entity path context to get relativized entity path
3. if there is an ascribed functor signature, elaborate the functor signature and signature match full functor signature and ascribed functor signature to yield the coercion

A.5.2 Base Functor

The main steps of functor elaboration are:

1. elaborate the formal parameter signature expression
2. instantiate formal parameter signature
3. extend entity environment and static environment by binding parameter expression and parameter's full signature
4. elaborate functor result signature

5. elaborate functor body using the environments from step 3

6. signature match functor body with functor body signature

7. build functor entity expression, full functor signature, and typed abstract syntax

Functors elaborate by elaborating the parameter signature expression and instantiating the resultant semantic parameter signature. The elaborator then elaborates the functor body using an entity environment extended with the resultant free instantiation of the parameter signature, a static environment extended with the full signature, an elaboration context that differentiates between type constructors from inside and outside of any/all functors marking those type constructors as volatile, and an entity path context that is extended with the entity variable for the functor parameter. The entity expression from the functor body is used to form the functor entity expression ($\lambda\rho_{param}.bodyExp$). The full signature from the functor body is used to form the full functor signature. The functor realization's closure environment is the current context entity environment.

Some functors are themselves a result of functor application. The elaborator treats these by expanding the syntax tree to a let-expression where the only definition is a structure application and the body is a functor variable that extracts out the functor component of the definition.

For functor-level let-expressions, the typed abstract syntax of the elaborated definition (a declaration) and body (a functor) are sequenced. The entity declarations are collected in a let entity expression. The resultant entity environments are also concatenated.

Functor variables may include a transparent ascription in which case the functor signature must be elaborated and the elaborator must match the functor with the functor signature possibly yielding a coercion expression.

A.6 Signature Matching

A.6.1 Structure Signature Matching

$\boxed{\Gamma, \Upsilon \vdash {}^{\varphi}M \preceq \Sigma \rightrightarrows (M_{coerced}, \varphi_{coerced})}$

Structure signature matching involves the following:

semantic signature **(Σ)** the ascribed signature to be matched against

full signature **(M)** the full signature of the structure to be matched

structure entity expression **(φ)**

entity environment **(Υ)**

static environment **(Γ)**

The process produces the following results:

full signature **(M')** the full signature of the coerced structure

structure entity expression **(φ')** an entity expression representing the coerced structure.

The steps to structure signature matching is quite substantial:

1. match the elements in the ascribed signature
   - (a) get the element's entity variable from the signature
   - (b) look up that entity variable in the full signature of the uncoerced structure to get the realization for that element
   - (c) match the spec with the realization
   - (d) produce the thinned representations
2. check that sharing constraints are followed in the result entity environment

3. build the full signature for the coerced structure (signature is the ascribed signature and realization is the result entity environment)
4. build typed abstract syntax that binds symbolic structure name to full signature and a let-expression with coerced typed abstract syntax declarations and bindings from matching the elements

For structure specs, the matching requires the following steps:

1. look up the element's entity variable and the uncoerced's structure's full signature as above
2. look up the realization for any structure definitional spec and match those with the uncoerced structure
3. match the uncoerced structure's full signature with the structure spec
4. extend the entity environment with the coerced realization
5. produce an entity declaration with the entity expression from the coercion
6. return this extended entity environment, this entity declaration, the binding from the coercion, and the typed abstract syntax from the coercion

For functor specs:

1. look up the element's entity variable and full functor signature
2. match the functor spec with the full functor signature noting the entity path to the functor element

For value specs:

1. look up the symbolic name of the spec in the signature part of the full signature to get the corresponding uncoerced spec (*i.e.*, the actual type)

2. match the types according to core language typing discipline

3. record any instantiations due to the coercion from the core language type match in the typed abstract syntax and bindings

A.6.2 Functor signature matching

$\boxed{\Gamma, \Upsilon \vdash F \preceq S^f \rightrightarrows (F_{coerced}, \theta_{coerced})}$

Functor signature matching requires:

functor signature (S^f) the ascribed functor signature (the "spec")

full functor signature (F) the full functor signature for the functor to be coerced

entity environment (Υ)

static environment (Γ)

The process produces the following results:

full functor signature ($F_{coerced}$) the coerced full functor signature

functor entity expression ($\theta_{coerced}$) the coerced functor entity expression

1. Instantiate the formal parameter signature from the spec and given entity environment

2. Take that realization and form a full signature for the spec formal parameter

3. Evaluate the functor application of the actual functor to the spec full signature

4. Match the full signature from the application result to the specified functor result signature using the entity expression from the evaluation and entity environment extended with the spec formal parameter signature's full signature (from 1)

5. build the coerced functor realization using the given entity environment (not the extended one) a let-expression with the uncoerced functor realization as the definiens and entity expression from the coercion in the previous step as the body.

$$
\begin{array}{rcl}
p_s & ::= & s_i \mid p_s.s_i \\
p_f & ::= & f_i \mid p_s.f_i \\
p_t & ::= & t_i \mid p_s.t_i \\
D & ::= & \epsilon \mid D\ D' \mid \mathit{type}\ t_i :: \kappa_c \\
 & \mid & \mathit{type}\ t_i :: \kappa_c = \mu_c \mid \mathit{structure}\ s_i : M_s \\
 & \mid & \mathit{functor}\ f_i : M_f \\
M_s & ::= & \mathit{sig}\ D\ \mathit{end} \\
M_f & ::= & \mathit{fsig}(s_i : M_s) M'_s \\
\kappa_c & ::= & \Omega \mid \Omega \to \kappa_c \\
\mu_c & ::= & p_t \mid \mathit{int} \mid \mu_c \to \mu'_c \mid \lambda t_i :: \Omega.\mu_c \mid \mu_c[\mu'_c] \\
d & ::= & \epsilon \mid dd' \mid \mathit{local}\ d\ \mathit{in}\ d'\ \mathit{end} \mid \mathit{type}\ t_i :: \kappa_c = \mu_c \\
 & \mid & \mathit{structure}\ s_i = m_s \mid \mathit{functor}\ f_i = m_f \\
m_s & ::= & p_s \mid f_i(s_i) \mid (s_i : M_s) \mid m_b \\
m_b & ::= & \mathit{struct}\ d\ \mathit{end} \\
m_f & ::= & p_f \mid \mathit{funct}(s_i : M_s) m_b
\end{array}
$$

Figure A.1: Normalized module calculus NRC

$$
\begin{array}{rcl}
\kappa_t & ::= & \Omega \mid \kappa_t \to \kappa'_t \mid \{l :: \kappa_t, \ldots, l' :: \kappa'_t\} \\
\mu_t & ::= & \alpha \mid \mathit{Int} \mid \mu_t \to \mu'_t \mid \lambda\alpha :: \kappa_t.\mu_t \mid \mu_t[\mu'_t] \\
 & \mid & \{l = \mu_t, \ldots, l' = \mu'_t\} \mid \mu_t.l \\
\sigma_t & ::= & T(\mu_t) \mid \sigma_t \to \mu'_t \mid \{l : \sigma_t, \ldots, l' : \sigma'_t\} \mid \forall\alpha :: \kappa_t.\sigma_t \\
e_t & ::= & x \mid i \mid \lambda x : \sigma_t.e_t \mid @e_t e'_t \mid \Lambda\alpha :: \kappa_t.e_t \mid e_t[\mu_t] \\
 & \mid & \{l = e_t, \ldots, l' = e'_t\} \mid e_t.l \mid \mathit{let}\ d_t\ \mathit{in}\ e_t] \\
d_t & ::= & \epsilon \mid (x = e_t); d_t
\end{array}
$$

Figure A.2: F_ω-based target calculus TGC

6. return the coerced full functor signature

A.7 Comparisons to Existing Accounts

A.7.1 Abstract Approach

The abstract approach relies on signature subsumption and strengthening to propagate types for functor application.

A.7.2 Shao

A.7.3 Parameterized Signatures (Jones)

Jones proposed using signatures parameterized on type constructors to express type propagation[33]. Under this system, the apply functor is expressed as the following:

```
signature S t = sig ... end
functor apply(functor f(x:S t):S u
                        structure a:S t): S u
                    = f a
```

This proposal argues removing type components from structures and wholly relying on type constructor parameters in these parameterized signatures to express type relationships. The resulting system only requires higher-order and first-class polymorphism and records, which encode the value-only structures. First-class polymorphism permits what amounts to functor application over first-class structures. Jones' approach is similar to Shao's in that it uses higher-order type constructors to encode type sharing. Under this scheme, all types must be declared at the top level. This idea follows Harper, Mitchell, and Moggi's observation of phase splitting[30]. The difference, however, is that Jones proposes such a separation of type and value modules in the surface language.

The drawback in this approach is that the programmer will have to know *a priori* the types that must be propagated. Jones' approach separates type abstraction concerns from the module system. However, this means that the scope of such type parameters are no longer restricted. Even when type components in modules are once again permitted, expressing the correct type sharing will force the programmer to lift type components to the top level, thus negating the advantage of the module system in favor of a simpler type-theoretic explanation.

This technique does not support more general forms of functor actions such as generative datatypes though Jones suggests using core language existentials or abstypes.

A.7.4 Dynamic ML

Dynamic ML suggests the addition of a where type definition (where datatype) that shares a datatype and its constructors such that no new datatype is generated [25]. The where datatype definition can be derived from the composition of datatype replication and a regular where type definition both at the signature level.

```
functor F(structure M : SIG where datatype t = A of int | B of bool)
functor F(datatype t = datatype t structure M : SIG where type t = t)
```

This encoding is incomplete. The replicated datatype in the functor parameter must be hidden so that only M.t remains. Furthermore, the replicated data constructors are in the functor parameter but not M.

A.8 Comparison with SML/NJ

Currying adds a minor complication that can be encoded by expanding the functor out to a nested functor signature.

Datatype Replication Specs

Datatype replication is treated specially. The tycon which is being replicated may occur in a few contexts:

functor parameter

```
functor F(X:sig datatype t = A end) =
struct
   structure M : sig datatype t = datatype X.t end
end
```

earlier spec

```
sig
  datatype t = A
  datatype u = datatype t
end
```

outer scope

```
sig
  structure M :
      sig
         datatype t = A
         structure N :
            sig
               datatype u = datatype t
            end
        end
 end
```

The source tycon, the tycon that is being replicated, must be either a generative tycon of kind datatype or a relativized tycon. In the generative tycon case, the elaborator will relativize the tycon, wrap the resulting relativized tycon in a definitional tycon, and add the data constructor specs to the cumulative elements list. To elaborate the remainder of the signature specs, the static environment is extended with a new relativized tycon for this new tycon.

In the relativized case (that is the target tycon was relativized), the elaborator looks up the entity path in the signature context to obtain the datatype family. From the datatype family, the elaborator extracts the member of interest and constructs a definitional tycon around the original relativized target tycon. This definitional tycon is added to the elaborated elements list. A new relativized tycon (for this replication) is added to the static environment. The elaborator then adds the data constructor specs, checking that each relativized parameter tycon is bound in the entity environment.

APPENDIX B
IMPLEMENTATION IN SML/NJ

The ML module system has several independent implementations in the compilers Moscow ML, TIL/TILT, MLton, MLKit, SML.NET, SML#, AliceML, HaMLet, OCaml, and SML/NJ. All of these compilers have extended the module language, departing from the module system as specified by the Definition considerably. Of these compilers, OCaml and SML/NJ stand out as the two that are the most robust and flexible, compiling the widest range of ML code. SML/NJ's longevity is due in large part to its module system.

The SML/NJ module language builds on the SML97 Definition. Much of the evolution of the Standard ML language from SML93 to SML97 was first prototyped in the compiler. Beyond SML97, SML/NJ incorporates higher-order modules, signature inheritance via include, and a more stringent signature instantiation check via the instantiation algorithm.

B.0.1 SML97 Module System

The ML module system provides a set of constructs for expressing large-scale program architecture. The module system is also the means for defining and enforcing abstractions. Basic modules are called *structures*, and are collections of types, values, and nested modules. They can be hierarchically nested with no restriction on depth of nesting. *Signatures* express static interfaces of structures and are the analogue of types for structures. A signature is comprised of a collection of type, value, and module specifications, specifying the kind, type, and signature respectively. A *functor* is a module-level function formed by parameterizing a structure (the functor body) with respect to a structure variable constrained by a signature. Signatures and structures have a many-to-many relationship. Multiple structures can match a single signature, and multiple signatures can be ascribed to a single structure.

Having functors leads to an interesting issue called the diamond import problem. Simply put, the diamond import problem refers to the scenario where a functor parameter signature

```
1 signature SIG = sig type s end
2 signature SIG' =
3   sig
4     type t
5     type u = t
6     structure M : SIG where type s = t
7     structure N : SIG
8     sharing type t = N.s
9   end
```

Figure B.1: The sharing mechanisms in SML

contains two types possibly nested inside substructures that must be equal for the functor body to typecheck. The example in (fig. B.1) illustrates each of SML's three mechanisms for constraining types to ensure required *type sharing*, namely:

1. definitional specifications: two types specified in the same signature scope (line 5).
2. where clauses in signature expressions: two types in different signatures where the latter signature's type depends on the former's (line 6).
3. sharing type constraints: two types in different signatures without any order restriction (line 7).

All three of these methods for constraining types represent a form of type sharing which must be resolved by the compiler for typechecking.

B.0.2 Higher-order Functors

Higher-order functors arise naturally from a module system with functors. Since any compilation can reference an external functor (*i.e.*, external to the compilation unit), it is natural to abstract over that external functor to make the import explicit. Higher-order functors pose a number of interesting engineering challenges foremost of which is the need to fully capture the range of functor actions of a formal functor. A *functor action* includes *generative static effects* and nested formal functor applications in the body of a functor. Generative static effects refer to the generation of fresh type identities[1] for ML datatypes and types made

1. In SML97, structures do not have generative identities, unlike in SML90, which is the semantics assumed in Crégut and MacQueen[10]. This simplification in the semantics allows simplification of the

```
functor F(X:sig functor G(): sig end end) =
struct
  functor H() = struct end
end
```

Figure B.2: Higher-order functor example

abstract by opaque signature ascription. In contrast, OCaml's applicative functors[38] only aim to capture nested formal functor applications in functor signatures. In order to support formal functor parameters, SML/NJ's syntax is extended with functor signatures. Syntactic functor signatures are always named unlike structure signatures which may be anonymous. Functor signatures consist of a named formal parameter signature and a body signature. Components in the body signature may depend on the formal parameter signature.

B.1 Semantic Objects

Syntactic signatures, structures, and functors translate into internal representations called semantic objects. These semantic objects make certain implicit properties of syntactic objects explicit for typechecking and translation into the intermediate language. We bind the names of signatures, structures, and functors to their respective semantic objects in a *static environment*.

During elaboration, we are principally concerned with the static (type-level) aspects of various elements of programs such as values, types, and modules. In the case of modules, we adopt the strategy of partitioning this static information into a stable part (the signature) and a volatile part (the realization). In this view, signatures may be the internal representation of a source level signature, or they may be inferred from a structure. From this point on, we will refer to the internal representation of a signature as "signature" and the surface level signature as "syntactic signature".

Realizations consist of *static entities* (*entity* for short), one for each possibly type-containing structure binding. In fig. B.3, the stable part of `MO` is represented in the signature

compiler.

```
signature SIG =
sig
   type t
   val x : t
end
```

```
structure M0 : SIG =
struct
   type t = int
   val x = 1
end
```

```
structure M1 : SIG =
struct
   type t = bool
   val x = true
end
```

Figure B.3: A simple example of two structures sharing the same signature

```
functor F(type t val x : t) =
struct
   type t
   val y = x
end
```

Figure B.4: t in the body shadows t in the (anonymous) parameter so y in the result of functor application will have an orphaned type.

`SIG`. The shape of `SIG` characterizes both `M0` and `M1`. In contrast, the volatile part, *i.e.*, the definition of `t`, varies between the two structures. In this example, the entities corresponding to `t` for `M0` and `M1` are `int` and `bool`, respectively.

B.1.1 Entity Variables and Paths

When a visible value binding contains a type constructor that is shadowed, we say that that binding has an *orphaned type*. This phenomenon of orphaned types occurs quite frequently even throughout the SML Basis library due to how bootstrapping works. Fig. B.4 gives a simple example of how type definitions in a functor body can shadow those in a functor parameter. Because the type of values can be orphaned, the internal representation of modules cannot refer to types simply by their symbolic paths (henceforth referred to as "path"), which would be the natural naive representation. The alternative adopted by most module systems, called the "label-variable distinction" [29], assigns each structure component an internal name (variable) in addition to the external-facing label that comprises symbolic paths.

SML/NJ assigns *entity variables* to module components which are similar to Harper-Lillibridge's internal names[29]. Unlike internal names which depend on assumed alpha-conversion to avoid shadowing, entity variables are made globally unique by being generated from a global counter. A list of entity variables constitutes an *entity path* which uniquely references a component in a module. Entity variables, unlike symbolic names, never shadow. We call them “entity” variables because they index into an environment of entities.

```
signature ENTPATHSIG =
sig
    type entVar
    type entPath = entVar list

    val eqEntVar : entVar * entVar -> bool
    val mkEntVar : unit -> entVar
end
```

The phenomenon of type shadowing complicates the implementation of higher-order functors. We use entity paths to provide a more robust way of naming types in modules.

B.1.2 Types

The SML/NJ compiler elaborates syntactic types to internal representations, which contain stamps to distinguish generative types. Here we give a highly simplified presentation of the syntactic and corresponding internal type representation, where `tyckind` provides descriptive information for primitive, abstract, datatype, and formal constructors, and `entPath` is the type of entity paths.

```
datatype ty
    = CONty of tycon * ty list
    | ...

datatype tycon
    = GENtyc of {arity: int, stamp: stamp, kind : tyckind}
    | PATHtyc of entPath
    | ...
```

The SML type language is comprised of type constructors and types. GENtyc (generative type constructor) is the internal representation for abstract types, datatypes, and formal types in functor parameters. Instances of each of these type constructors are considered unique and therefore only equal to themselves. For example, in **functor** F(X:**sig type** t **type** u **end**) = ... the compiler must conservatively assume that `t` and `u` are not equal to each other or any other type for the purpose of typechecking the body of `F`. A `GENtyc`'s stamp is the unique identifier that distinguishes each `GENtyc`. These stamps are a variation on the stamps in the MacQueen and Tofte semantics[45]. `PATHtycs` come from syntactic signature specs that refer to a type component elsewhere in the signature or in the context. The lookup path to that type component is encoded in the entity path.

B.1.3 Signature and Realization

The internal signature differs from syntactic signatures in that all static signature components are assigned entity variables whereas syntactic signatures make no reference to entity variables. A signature component is *static* when it is a type or potentially contains a type (such as structure or functor signatures). Structurse are assigned entity variables because they can potentially contain types that can be referenced. Functors are assigned entity variables because they encode type functions. A *signature* serves as a skeleton specifying the components of the structure in most general terms without constraining the contents beyond what is explicitly specified. The signature part of the representation maps symbolic names to entity variables. A structure having such a signature supplies additional static information such as type definitions in the realization part, which maps entity variables and paths to types. The combination of signature and realization is called the *full signature*, and it serves as the complete static description of a structure.

A signature (see fig. B.5) consists principally of its *elements*, a list of symbol and spec pairs, two lists of symbolic paths recording type and structure sharing constraints and an optional name for the signature. Specs contain entity variable, type, and dynamic access slot information. The elaborator will use the entity variables to construct an entity path to lookup types in the realization in an appropriate entity environment. The dynamic access slot is an index into the dynamic part of structures.

In the example in fig. B.6, the content of structure `A` is factored into an inferred internal signature and a realization. **`type`** `t` is assigned the entity variable e_t. Substructure B and type u are assigned entity variables e_B and e_u respectively. The realization for A maps e_t to `int` and e_B to a realization for B, which maps e_u to `bool`. We can visualize a realization as a tree of entity variables and entities.

```
datatype tycSpec
   = InferredTycSpec of {name : symbol, arity : int}
   | RegTycSpec of {tycon: tycon}

datatype spec
   = TYCspec of {entVar : entVar, tycSpec : tycSpec}
   | STRspec of {entVar : entVar, sign : Signature}
   | VALspec of {spec : ty, slot : int}
   | ...
and Signature
   = SIG of {name : symbol option,
                  elements : spec list,
                  typsharing : path list,
                  strsharing : path list,
                  ...}
```

Figure B.5: Definition of signature and spec in a module language with no higher-order functors

```
structure A =                              signature ASIG =
struct                                     sig
   type t = int (* declarations *)            type t (e_t) (* specifications *)
   structure B =                              structure B (e_B) :
   struct                                     sig
      type u = bool                              type u (e_u)
      val x : t * u = (3, true)                  val x : int * bool
   end                                        end
   val y : t * B.u = B.x                      val y : int * bool
end                                        end
```

Figure B.6: Schematic example of an inferred internal signature where e_t, e_B, and e_u are fresh entity variables

```
structure A' :
  sig
    type t
    structure B :
      sig
        type u
        val x : t * u
      end
    val y : t * B.u
  end
  = A
```

```
signature A'SIG =
sig
  type t ($e_t$)
  structure B ($e_B$) :
  sig
    type u ($e_u$)
    val x : PATHtyc[$e_t$] * PATHtyc[$e_u$]
  end
  val y : PATHtyc[$e_t$] * PATHtyc[$e_B$,$e_u$]
end
```

where e_t, e_B, and e_u are fresh

Figure B.7: The signature derived from the ascribed signature of a structure A'

Figure B.8: The realization derived for structure A'

Where available, the compiler will use an ascribed syntactic signature to produce an internal signature for a structure[2]. One such example of this is given in fig. B.7. Because types `t` and `u` are abstract, the value specs for `x` and `y` refer to them by entity paths because their meanings may (potentially) vary beteen different realizations of the signature.[3] The entity paths should be *relativized* (*i.e.*, given relative) to the scope of the spec as shown in fig. B.8.

The construction of the realization for A' is somewhat subtle as shown in fig. B.8. The value spec B.x refers to e_t relative to the scope of the signature of B which happens to be written inline with respect to ASIG. If the substructure B were selected out by structure abbreviation (*e.g.*, **structure** B = A'.B), then the signature and the realization for B have to be able to stand alone, hence the realization for B must also interpret e_t despite the absence of t in the signature for B. We do this by repeating the e_t entity in the realization for e_B with exactly the same entity variable e_t.

This dichotomy between signatures and realizations is useful for distinguishing the essential shape of a structure as represented in the signature from the independent static content

2. In the elaborator, we always first infer a signature for a structure even if it has an ascription, but we immediately try to match the inferred and ascribed signature ("signature matching"). The resultant full signature then includes the ascribed signature and not the inferred one.

3. Because the signature of `A'` is given inline here, there will in fact be only one realization for this signature.

in the realization. The compiler leverages this dichotomy when it elaborates functor application by only mapping from the realization part of the parameter to the realization part of the result, instead of having to recreate the signature part of both parameter and functor result. The ability to do functor application by only operating on volatile components is one of the main reasons for the entity paths approach. It is also more parsimonious to have multiple structures share the same signature representation. The static environment, however, maps structure names to the full signatures of the corresponding structures, defined as follows:

```
datatype Structure = STR of {sign : Signature, rlzn : strEntity, ...}
```

The structure realization itself is considered a *structure entity* (type `strEntity`), whose operative content is an entity environment, mapping the entity variables of static components to appropriate kinds of entities:

```
type strEntity = {entities : entityEnv, stamp: stamp, ...}
```

So a structure entity may be thought of as a tree with entity environments at the internal nodes. The leaves of the tree are always core language type constructors (or possibly functor entites (`FCTent`) in the higher-order case).

```
datatype entity = TYCent of tycon | STRent of strEntity | FCTent of fctEntity
```

To look up a symbolic path using these semantic objects, the elaborator would first lookup the initial symbol x_0 in the symbolic path $x_0.x_1.\ldots.x_n$, which should be a name of a structure x_0, in the static environment to obtain the internal signature for x_0. Then, the elaborator interprets the suffix of the path $x_1.\ldots.x_n$ using the internal signature to obtain the corresponding entity path. The realization is then used to interpret the entity path, which may involve a series of lookups in the nested structure entities.

Not all full signatures can be faithfully simulated by a syntactic signature. In particular, generative datatypes have unique identities in the full signature, indicated by their stamps, whereas there is no mechanism for expressing such a unique identity in a syntactic signature.

B.1.4 Static functor representation

The representation of the static description of functors follows the same technique used for structures. The static description is factored into functor signature and functor realization. A functor signature consists of a parameter name and its associated signature, and a body or result signature that may depend on (*i.e.* mention) the parameter. The internal representation adds an entity variable for the functor parameter for the functor body to reference.

```
datatype fctSig
  = FSIG of {paramsym : symbol,
             paramvar : entVar,
             paramsig : Signature,
             bodysig : Signature}
```

Whereas the realization for a structure simply fills in types specified in the signature with appropriate entities, a functor realization must express how the entities in the result structure depend on the those in the parameter structure. So it must express a mapping from a parameter realization to the corresponding result realization. We call this kind of mapping an entity expression or strExp.

```
functor F(X: sig type t val x: t end) =
  struct
    datatype u = A of X.t
    type v = X.t * u list
    fun f(x: X.t) : u = A x
  end
```

The above functor is represented by a functor signature and a functor realization. The inferred functor signature is:

```
paramsym = X
paramvar = $e_X$

paramsig = sig
               type t ($e_t$)
               val x : [$e_t$]
             end

bodysig = sig
               type u ($e_u$)
               type v ($e_v$)
               val f : [$e_X$, $e_t$] −> [$e_u$]
             end
```

where e_X, e_t, e_u, and e_v are fresh entity variables. From this point on, entity paths in value specs should be understood as PATHtycs on the given entity path, *i.e.*, **val** x : [e_t] is read as **val** x : PATHtyc[e_t].

The realization for the functor has to specify how realizations for the static components of the result (entities for the types u and v) are constructed given a realization for X, and hence a type entity for X.t. This is accomplished using a “static lambda calculus”, which provides expressions, declarations, and functions on entities. The realization for F is a lambda term of the form:

```
LAMBDA(e_X, STRUCTURE{stamp = NEW,
                      entDec = [TYCdec(e_u, FORMtyc(tc_u*)),
                                TYCdec(e_v, FORMtyc(tc_v*))]})
```

where tc_u* and tc_v* are internal representations of the definitions of the datatype u and the type abbreviation v, relativized by replacing type constructor references by entity paths.

```
tc_u* = data [A of [e_X, e_t]]
tc_v* = deftyc([e_X, e_t] * [e_u])
```

When this functor is applied to an argument structure, the argument structure is coerced by signature matching (described in sec. B.2) with the parameter signature, paramsig, yielding a realization for paramsig. This parameter realization is bound to the entity variable e_X and the body of the LAMBDA expression is evaluated in the resulting entity environment.

The body specifies that a structure realization is to be constructed, whose contents will be defined by a sequence of two entity variable declarations. e_u will be bound to a new datatype generated from the datatype specification, with the associated entity paths referencing imported types being evaluated relative to the evaluation entity environment. Similarly, the definition of type v will be instantiated by evaluating its embedded entity paths in that same entity environment extended with the binding of e_u.

In particular, in the application

```
F(struct type t = int end)
```

The realization bound to e_X will be the entity environment $\{e_t \mapsto \mathsf{int}\}$, and the evaluation environment is $\{e_X \mapsto \{e_t \mapsto \mathsf{int}\}\}$. Evaluating FORMtyc($tc_u*$) in this environment yields a fresh datatype corresponding to the definition

```
datatype u = A of int
```

and FORMtyc(tc_v*) yields an instantiated abbreviation type [type abbreviation?] corresponding to the definition

type v = int ∗ u.

The stamp field of the STRUCTURE argument is a special expression NEW, which is evaluated by generating a new stamp for the new structure being constructed. This stamp is used for administrative purposes and is not involved in type checking.

B.1.5 Higher-order functors

The preceding example involves the classic case of a first-order functor defined at top level, or, more particularly one not defined within another functor. Supporting higher-order functors introduces an additional requirement on the functor representation.

Consider the following example:

```
functor F(X: sig type t end) =
  struct
    datatype u = C of X.t
    functor G(Y : sig type v val x : v * u * X.t end) =
      struct
        datatype s = B of X.t * u -> Y.v
      end
  end
```

We can see that all the type constructors X.t, u, Y.v, and s are "volatile" in the sense that their actual bindings are to be determined later, when F is applied. When we relativize the specification of datatype s with respect to these volatile type constructors, we get something like:

tc_s* = data (B **of** [e_X, e_t] ∗ e_u –> [e_Y, e_v])

Now consider an application of F

structure A = F(**struct type** t = int **end**)

When this is evaluated, we will develop an entity environment that binds e_X and its extension [e_X, e_t] as before, and the definition of u will give rise to a new datatype that will be bound to e_u. As before, the realization of functor G will take the form of a lambda term in our entity calculus:

```
LAMBDA(e_Y, STRUCTURE{stamp = NEW,
                      entDec = [TYCdec(e_s, FORMtyc(tc_s*))]})
```

But now we note that this term binds only the entity variable e_Y, leaving e_X and e_u free. So the lambda term is not closed. As usual, we need to close it by supplying a closure environment, namely the environment mentioned above that binds e_X and e_u.

Thus the realization for functors has to be augmented to include a closure of a lambda term with respect to an entity environment.

```
type fctEntity =
  {exp : fctExp,
   closureEnv: entityEnv,
   ...}
```

[Need to define fctExp and strExp earlier]

Now if we apply A.G we will add a binding of e_Y to the closure environment and use this when evaluating the body of the lambda term for G. For instance, after

```
structure B = A.G(struct type = bool val x = (true, A.C 3, 1) end)
```

the datatype B.s will be given by

```
B.s = data (B of int * A.u -> bool)
```

B.2 Elaboration

Elaboration is the process of translating the simple syntax trees (Ast) produced by the parser into explicitly typed abstract syntax and other static semantic information. This involves performing type checking and type inference at the core ML level, and corresponding processes at the module level. For modules, elaboration means translating the syntactic forms for signature, structure, and functor expressions and declarations into their internal representations. A simplified general scheme for elaboration functions (for declarations) is

```
elaborate_decl : syntax_tree * static_environment -> abstract_syntax * static_environment
```

The abstract syntax product is ultimately used to generate executable code, while the static_environment records type information for bound identifiers that is used in type checking code that is in the scope of the declaration. A static environment is thus a mapping from symbols to their static descriptions. In the case of module level symbols naming signatures, structures, and functors, the static descriptions are the corresponding static representations described above. Static environments also map type names to their representations as type constructors, values to their types, and data constructors (and exceptions) to their types. The initial static environment is the pervasive environment which contains core types and operators.

When elaborating the declarations in the body of a functor, this scheme needs to be augmented. Each functor application produces a new realization of the functor body possibly with new entities while leaving the functor body signature intact. Not only do we need to elaborate and type check the body directly, we need to capture an *entity expression* that can be evaluated at functor applications to carry out the "static action" of the functor, *i.e.*, the type propagation and generation performed by the functor. So in this context, elaboration must produce *entity declarations* in parallel with the typed abstract syntax and the static environment. These entity declarations will be evaluated to produce the entity environment that is the realization of the functor result signature.

Constructing entity expressions and declarations requires some extra contextual information, because external references to types defined within a functor body will be converted to entity path form (PATHtyc) in the entity expressions. We need to pass a sort of inverse entity environment that maps such types to their entity paths. This inverse environment is (part of) an extra argument called an entity path context (epcontext). For technical reasons related to type shadowing and creating functor realization closures we also have to thread an entity environment through the elaboration. So a more complete elaborate_decl scheme is:

```
elaborate_decl : syntax_tree * static_environment * entity_environment * entity_path_context
             -> abstract_syntax * static_environment * entity_environment * entity_declaration
```

This scheme works for both elaborating within a functor body and without, in which case epcontext will be empty and the resulting entity_declaration would be ignored.

Signature elaboration . The specifications in the body of the signature are translated into a mapping from component names to internal specs in the form of formal type constructors for types, types for values, and internal signatures for structures and functors. Static components (types and modules) are assigned fresh entity variables. The types of value components and types occurring in definitional type specs are relativized by replacing local type constructor references with PATHtycs containing entity paths. Sharing constraints are recorded in a normalized form.

Structure elaboration . There are several cases for structure expression elaboration, corresponding to the syntactic forms for such expressions (*e.g.*, structures declared inline **struct** ... **end**, structure variables A, functor application). The function header for the main structure elaboration function is:

```
fun elabStr
      (strexp: Ast.strexp, (* the syntax tree *)
       name: symbol option,
       env: staticEnv,
       entEnv: entityEnv,
       epContext: EntityPathContext.context,
       entVarOp: entVar option, (* entVar this structure is bound to *)
       ...)
      : Absyn.dec * Modules.Structure * Modules.strExp * EntityEnv.entityEnv
```

The basic inline structure expression "struct ... end" form is elaborated using the following steps

1. elaborate the body declaration, yielding a static environment envBody containing the internal component bindings and an entity declaration entDecl.
2. infer a signature and construct a realization from envBody and an entity declaration entDecl'.
3. build a STR (i.e. a static structure) using the signature and realization from step 2, and return it together with a structure entity expression of the form
   `STRUCTURE{stamp=NEW, entDec=entDecl'}`

Functor elaboration . Functor elaboration involves several new problems. One issue is how to deal with references to the formal parameter structure in the body, both during elaboration of the body and later during application of the functor. As we have seen in the earlier example, at application time the parameter will be represented by an entity variable that serves as the bound variable of a lambda-abstraction entity expression.

During elaboration of the functor body, we need to bind the parameter name to a full static representation (full signature) of the parameter structure – one that can serve as a formal representative of all possible actual arguments. We are given an explicit signature for the parameter in the form of the functor parameter signature, and so we only need to create a representative realization for that signature. The process of creating this realization is called *signature instantiation*.

Signature instantiation is the process of creating a “free” realization of the signature. This realization includes fresh formal type constructors for each type component, but chosen to satisfy the signature's sharing constraints, and only those sharing constraints (*i.e.*, no sharing not forced by the specifications). The algorithm used for signature instantiation is adapted from the Patterson-Wegman linear unification algorithm [55].

When instantiating a functor specification in a signature, we must create a corresponding functor realization. This will be, as usual, a closed entity lambda expression, but the body of this lambda expression is a special internal form, FORMstr(fctSig), containing the functor signature. The use of this special form will be explained below in the paragraph describing functor application.

Now having bound the formal parameter symbol to the instantiation of the parameter signature in the static environment, the body of the functor is elaborated. This produces a full signature for the body and a body entity expression of type strExp. A functor signature is created by combining the parameter signature and signature part of the body full signature. The realization of the functor is created by wrapping a lambda abstraction around the body entity expression, and closing it with respect to the entity environment in which the functor is elaborated.

Signature matching . When a signature is ascribed to a structure in a structure declaration, or when a functor is applied to a structure, implicitly ascribing the parameter signature to the argument, we must verify that the structure in question *matches* the signature. This is a kind of module-level type checking, but it also has a coercive effect, producing a modified structure that is exactly conforming to the ascribed signature. Signature matching involves scanning the specifications in the signature and verifying that the matching structure satisfies these specifications, but it also produces an abstract syntax expression that performs the coercive *thinning* of the value record for the structure to drop extraneous value components.

Functor Application . When a functor is applied, the argument structure expression is elaborated, and then signature matching is used to verify that it matches the parameter specification. The result of this signature match is a full signature for the coerced argument. The functor realization, which is a closed entity lambda expression, is then applied to the realization from the argument full signature.

If the functor realization was created by instantiating a functor component of a formal parameter signature (as when elaborating the body of a functor), then the body of the lambda expression is of the form FORMstr(fctSig). The result of the application of the functor is computed by instantiating the body signature from the fctSig relative to an entity environment containing the binding of the argument realization to the parameter entity variable.

B.3 Discussion

The type information generated during elaboration in ML can grow quite large, and experience with early, relatively naive versions of the elaborator demonstrated that the size of static data structures can become a real resource bottleneck. Although we do not have systematic experimental data comparing the efficiency of the current implementation with simpler versions, it is certainly the case that the current implementation has shown that it scales very well and can easily cope with large and complex programs. Sharing signatures is conjectured to be a considerable win. We believe that the factorization of modules into signatures and realizations is a key part of the scalability of the SML/NJ's elaborator. Although hash-consing of type information turned out to be necessary in the FLINT intermediate language, this technique does not seem to be required in the front end, and this is probably partly due to the sharing of signature information.

APPENDIX C
SEMANTICS PROOFS

C.1 Type System

Lemma 21 (Monotypes are strongly normalizing)

If $\mathfrak{C}^s \Downarrow \mathfrak{C}^s_1$ and $\mathfrak{C}^s \Downarrow \mathfrak{C}^s_2$, then $\mathfrak{C}^s_1 = \mathfrak{C}^s_2$ and $\mathfrak{C}^s_1$ is a $\mathfrak{C}^{nf}$.

Proof: There is a precise embedding of the monotype language in the simply-typed λ-calculus. Because the simply-typed λ-calculus is strongly normalizing, so this the monotype language. ■

Lemma 22 (Inversion)

1. *If $\Delta \vdash \alpha : \Omega$, then $\alpha \in \Delta$.*
2. *If $\Delta \vdash \mathfrak{C}^\lambda(\vec{\mathfrak{C}^s}) : \Omega$, then $\Delta \vdash \mathfrak{C}^\lambda : \Omega^n \Rightarrow \Omega$, $|\vec{\mathfrak{C}^s}| = n$, and $\Delta \vdash \mathfrak{C}^s_i : \Omega$ for all $i \in [1, n]$.*
3. *If $\Delta \vdash \lambda\vec{\alpha}.\mathfrak{C}^s : \Omega^n \Rightarrow \Omega$, then $\Delta[\alpha_1] \ldots [\alpha_n] \vdash \mathfrak{C}^s : \Omega$.*
4. *If $\Delta \vdash \mathsf{typ}(\mathfrak{C}^s) : \Omega$, then $\Delta \vdash \mathfrak{C}^s : \Omega$.*
5. *If $\Delta \vdash \forall\vec{\alpha}.\mathfrak{C}^s : \Omega$, then $\Delta[\alpha_1] \ldots [\alpha_n] \vdash \mathfrak{C}^s : \Omega$.*

Proof: By syntax-directedness of kind system and inspection ■

Lemma 23 (Kind Preservation)

If $\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega$ and $\mathfrak{C}^s \Downarrow_{tyc} \mathfrak{C}^{nf}$, then $\emptyset_{knds} \vdash \mathfrak{C}^{nf} : \Omega$.

Proof: [by induction on the typing derivation] The interesting case is the application rule (4.7). By inversion $\emptyset_{knds} \vdash \mathfrak{C}^\lambda : \Omega^n \Rightarrow \Omega$ and $\emptyset_{knds} \vdash \mathfrak{C}^s_i : \Omega \forall i \in [1, n]$ (0). By definition, $\mathfrak{C}^\lambda$ must be following two cases:

$\lambda\vec{\alpha}.\mathfrak{C}'^s_1$ Then by the β-reduction rule, $\mathfrak{C}^s_i \Downarrow_{tyc} \mathfrak{C}^{nf}_i$ (1) and $\mathfrak{C}^s \Downarrow_{tyc} \mathfrak{C}'^s\{\vec{\mathfrak{C}^{nf}}/\vec{\alpha}\}$ (2). By induction for (0) and (1), $\emptyset_{knds} \vdash \mathfrak{C}^{nf}_i : \Omega$ for all $i \in [1, n]$ (3). If $\mathfrak{C}'^s = \alpha$, then done by (3). If $\mathfrak{C}'^s$ is an application, then done by induction.

τ^n Vacuously true

■

Lemma 24 (Progress)

If $\emptyset_{knds} \vdash \mathfrak{C}^s : \Omega$, *then* $\mathfrak{C}^s \Downarrow \mathfrak{C}^{nf}$.

Proof: [by induction on the typing derivation] ■

C.2 Entity Calculus

Lemma 25 (Extended Entity Environment Lookup Terminates)

$\Upsilon(\vec{\rho})$ *terminates.*

Proof: [by induction on the length of $\vec{\rho}$] ■

Lemma 26 (Entity evaluation terminates)

The following evaluation derivations terminate (has a finite derivation):

1. $\Upsilon \vdash \varphi \Downarrow_{str} R$
2. $\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'$
3. $\Upsilon \vdash \theta \Downarrow_{fct} \psi$

Proof: [by the size of the entity expression (excluding the closure environments) and entity environment]

1. **Rule 6.1** This follows from lem. 25.

Rule 6.2 This follows from (2) by induction.

Rule 6.3 The entity expression calculus is a simply-typed λ (which known to be strongly normalizing) where types are implicit.

By induction (3), the functor entity evaluation terminates. By induction (1), the argument structure entity evaluation terminates. The functor entity $\psi = \langle \rho_{arg}.\varphi'; \Upsilon^{clo} \rangle$ must have either been in the entity environment Υ or be $\langle \theta; \Upsilon \rangle$.

Because either $|\Upsilon^{clo}| < |\Upsilon|$ or $|\Upsilon^{clo}| = |\Upsilon|$ for the respective cases above, $|\Upsilon^{clo}| \leq |\Upsilon|$. By assumption, $|R_{arg}| < |\varphi| + |\Upsilon|$.

If the former, then $|\psi| < |\Upsilon| < |\theta(\varphi)| + |\Upsilon|$. By induction (1), the evaluation of φ' must terminate.

If the latter, then $|\psi| = |\theta| + |\Upsilon| < |\theta| + |\Upsilon| + |\varphi|$. By induction (1), the evaluation of φ' must terminate.

Rule 6.4 By induction (3), the functor entity evaluation terminates. By induction (1), the argument structure entity evaluation terminates. By lem. 41, instantiation terminates.

Rule 6.5 By induction (2), entity declaration terminates. By induction (1), the evaluation of the body of the let terminates. Therefore the whole rule terminates.

2. **Rule 6.6** Trivial because no premises

   Rule 6.9 By induction (2)

   Rule 6.7 By induction (1) and (2)

   Rule 6.8 By induction (1) and (2)

3. For functor entity expressions, this is trivial. By inspection, each of the rules does not depend on any premise with the sole exception of entity environment lookup, which is known to terminate by lem. 25. They are single-step rules that always apply and will terminate.

■

Lemma 27

If $\Upsilon \vdash \varphi \Downarrow_{str} R$, *then* $\Upsilon \subseteq R$.

If $\Upsilon \vdash \eta \Downarrow_{decl} \Upsilon'$, *then* $\Upsilon \subseteq \Upsilon'$.

If $\Upsilon \vdash \theta \Downarrow_{fct} \langle \theta; \Upsilon' \rangle$, *then* $\Upsilon \subseteq \Upsilon'$.

Proof: [by inspection of rules] ■

C.3 Elaboration

C.3.1 Type Elaboration

Lemma 28 (Monotype Elaboration Preserves Kinding)

If $\Gamma, \Delta \vdash C^s : \Omega$ *and* $\Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathfrak{C}^s$, *then* $\Delta \vdash \mathfrak{C}^s : \Omega$.

Proof: [by induction on the derivation of $\Gamma, \Delta \vdash C^s : \Omega$]

Rule 4.1 (tyvar) $C^s = \alpha$, therefore rule 7.1 is the only applicable elaboration rule. By inversion of the syntactic type kinding, $\alpha \in \Delta$ so $\Delta \vdash \alpha : \Omega$ by rule 4.6.

Rule 4.2 (app) $C^s = p(\vec{C^s})$, therefore rule 7.2 is the only applicable elaboration rule. Let $n = |\vec{C^s}|$. By inversion of the syntactic type kinding, $\Gamma, \Delta \vdash \vec{C^s}_i : \Omega \forall i \in [1, n]$. By inversion of rule 4.2, $\Gamma, \Upsilon \vdash C^s_i \Rightarrow_{mt} \mathfrak{C}^s_i \forall i \in [1, n]$. By induction, $\Delta \vdash \mathfrak{C}^s_i : \Omega \forall i \in [1, n]$.

■

Lemma 29 (Tycon Elaboration Preserves Kinding)

If $\Gamma, \Delta \vdash C^\lambda : \Omega^n \Rightarrow \Omega$ *and* $\Gamma, \Upsilon \vdash C^\lambda \Rightarrow_{tyc} \mathfrak{C}^\lambda$, $\Delta \vdash \mathfrak{C}^\lambda : \Omega^n \Rightarrow \Omega$.

Proof: This follows directly from lem. 28. ■

C.3.2 Signature Elaboration

Lemma 30

If $AT(\Sigma) = \emptyset$, for all $\Sigma_i \in \Gamma . AT(\Sigma_i) = \emptyset$, and $\Gamma, \Upsilon, \Sigma \vdash \mathit{sigexp} \Rightarrow_{\mathit{sig}} \Sigma'$, then $AT(\Sigma') = \emptyset$.

Proof: [by induction on the derivation]

Rule 7.6 By assumption, $AT(\Gamma(x)) = \emptyset$.

Rule 7.7 By induction, $AT(\Sigma') = \emptyset$. By definition of $\mathbb{C}^{\lambda}$, $AT(\mathbb{C}^{\lambda}) = \emptyset$.

Rule 7.8 This depends on the above property holding for $\Gamma, \Upsilon, \Sigma \vdash \mathit{spec} \Rightarrow_{\mathit{spec}} \Sigma'$. The open tycon case is trivial. The type and val spec cases are true by definition. The structure and functor cases are by induction.

■

Lemma 31

If $\Gamma \vdash C^{\lambda} \Rightarrow_{\mathit{tyc}} \mathfrak{C}^{\lambda}$, then $AT(\mathfrak{C}^{\lambda}) \subseteq AT(\Gamma)$.

Lemma 32

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon \vdash \mathit{strexp} \Rightarrow_{\mathit{str}} ((\Sigma, R), \varphi)$, then $AT(M) \subseteq AT(R)$.

Lemma 33

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon, \emptyset_{\mathit{sig}} \vdash \mathit{sigexp} \Rightarrow_{\mathit{sig}} \Sigma_x$, then $AT(\Sigma_x) \subseteq AT(\Upsilon)$.

Lemma 34

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon \vdash d^m \Rightarrow_{\mathit{decl}} (\eta, \Gamma', \Upsilon')$, then $AT(\Gamma') \subseteq AT(\Upsilon')$.

Proof: [by induction on the derivation of $\Gamma, \Upsilon \vdash d^m \Rightarrow_{\mathit{decl}} (\eta, \Gamma', \Upsilon')$]

Rule 7.31 (empty) $AT(\emptyset_{se}) \subseteq AT(\emptyset_{ee})$.

Rule 7.32 (val) Requires $AT(\mathfrak{T}) \subseteq AT(\Upsilon)$. Because $\Rightarrow_{\mathit{core}}$ cannot generate new atomic tycons, $AT(\mathfrak{T}) \subseteq AT(\Gamma) \cup AT(\Upsilon) \subseteq AT(\Upsilon)$ by assumption. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.33 (typedef) By lem. 31, $AT(\mathfrak{C}^\lambda \subseteq AT(\Gamma)$. Therefore $AT(\mathfrak{C}^\lambda) \subseteq AT(\Upsilon)$. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.34 datatype By assumption $AT(\Gamma) \subseteq AT(\Upsilon)$. By extension, $AT(\Gamma[t \mapsto \tau^n]) \subseteq AT(\Upsilon[\rho_t \mapsto \tau^n])$. By induction $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.35 structure By lem. 32, $AT(M) \subseteq AT(R)$. Thus, $AT(\Gamma[X \mapsto (\rho, M)]) \subseteq AT(\Upsilon[\rho \mapsto R])$. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.36 functor By lem. 33, $AT(\Sigma_x) \subseteq AT(\Upsilon)$. Certainly, $AT(\Gamma[X \mapsto (\rho, (\Sigma_x, R_x))]) \subseteq AT(\Upsilon[\rho_x \mapsto R_x])$. All tycons in Σ_{res} are relativized so $AT(\Sigma_{res}) = \emptyset$. Therefore $AT(\Gamma[f \mapsto (\rho, (\Pi\rho_x : \Sigma_x.\Sigma_{res}, \psi))]) \subseteq AT(\Upsilon[\rho \mapsto \psi])$. By induction, $AT(\Gamma'') \subseteq AT(\Upsilon'')$.

■

Lemma 35

If $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi)$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Sigma, R) \subseteq dom(R)$ and $EV(\varphi) \subseteq dom(R)$.

Lemma 36

If $\Gamma, \Upsilon, \emptyset_{sig} \vdash sigexp \Rightarrow_{sig} \Sigma$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Sigma) \subseteq dom(\Upsilon)$.

Definition 4

Υ interprets Σ if for all specs in Σ, one of the following must be true:

1. *If the spec is an open tycon (ρ, n), then $\Upsilon(\rho) = \tau^n$ or $\Upsilon(\rho) = \mathbb{C}^\lambda$ such that $\Upsilon \vdash \mathbb{C}^\lambda :: \Omega^n \to \Omega$.*

2. *If the spec is a structure (ρ, Σ), then $\Upsilon(\rho) = R'$.*

3. *If the spec is a functor (ρ, Σ^f), then $\Upsilon(\rho) = \psi$.*

Lemma 37

If Υ interprets the extracted signature of Γ and $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi)$, then R interprets Σ.

Proof: The proof for module declaration elaboration is by inspection. For structure expressions, the base structure, and let expression rely on the proof for module declaration. The functor application and two ascription cases rely on signature matching preserving the *interprets* relationship. For signature matching, we can see that the resultant entity environment does indeed interpret the spec signature by inspection. ■

C.3.3 Module Declaration Elaboration

Lemma 38 (Synchronization of Static and Entity Environments)

If $\Gamma, \Upsilon \vdash\!\vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Gamma') \subseteq dom(\Upsilon')$.

Proof: [by induction on the derivations]

Rules 7.32, 7.33, and 7.34 $\mathfrak{T}$, $\mathfrak{C}^{\lambda}$, τ^n are semantic and therefore contain no entity variables.

Rule 7.35 By lem. 35, $EV(M) \subseteq dom(\Upsilon) \cup dom(R)$. By induction, $EV(\Gamma') \subseteq dom(\Upsilon')$. It follows that $EV([X \mapsto (\rho, M)]\Gamma') \subseteq dom([\rho \mapsto R]\Upsilon')$.

Rule 7.36 By lem. 42, $EV(\Gamma[X \mapsto (\rho_x(\Sigma_x, R_x))]) \subseteq dom(\Upsilon[\rho_x \mapsto R_x])$. By lem. 35, $EV(\Sigma_{res}, R_{res}) \subseteq dom(\Upsilon) \cup dom(R_{res})$.

■

The following lemmas guarantee that semantic signatures and semantic functor signatures in full signatures and full functor signatures are closed by the structure entity and functor entity respectively.

Lemma 39

If $\Gamma, \Upsilon \vdash$ **structure** $X = strexp, d^m \Rightarrow_{decl} (\_, [X \mapsto (\rho, (\Sigma, R))]\Gamma', \_)$, *then for all relativized paths* $\vec{\rho}$ *in* Σ, *there exists an entity* υ *such that* $R(\vec{\rho}) = \upsilon$.

Lemma 40

If $\Gamma, \Upsilon \vdash$ **functor** $f(X : sigexp) = strexp, d^m \Rightarrow_{decl} (\_, [f \mapsto (\rho, (\Pi\rho_x : \Sigma_x, \Sigma_{res}, \psi))]\Gamma'', \_)$ *and* $\Upsilon \vdash \Sigma_x \uparrow \Upsilon_x$, *then for all entity paths* $\vec{\rho}$ *in* Σ_{res}, $\langle \Upsilon_x, \Upsilon \rangle(\vec{\rho})$ *is defined.*

C.3.4 Signature Instantiation

Lemma 41 (Signature Instantiation Terminates)

$\Upsilon \vdash \Sigma \uparrow \Upsilon'$ *terminates.*

Proof: [by induction on length of semantic signature] All the rules work on a strict subsequence of the semantic signature Σ except rule 7.26 (the structure spec instantiation rule). Even in that case, the structure spec signature Σ' is clearly less than the length of $[x \mapsto (\rho, \Sigma')]\Sigma$. ■

Lemma 42

If $\Upsilon^{clo}, \Upsilon^{lcl} \vdash \Sigma \uparrow \Upsilon$, *then* $EV(\Sigma) \subseteq dom(\Upsilon) \cup dom(\Upsilon^{clo}) \cup dom(\Upsilon^{lcl})$

Signature Matching

Lemma 43

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{match} (R_c, \varphi_c)$, *then for all* $[x \mapsto s] \in \Sigma_s$, $[x \mapsto s'] \in \Sigma_a$ *such that* $R_c(s) = s'$.

Proof: [by straightforward induction on the derivation] ■

The induction on the two lemmas, lem. 44 and 45 require that they be proved simultaneously.

Lemma 44

If $\Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta)$*,* $EV(\Sigma_s) \subseteq dom(\Upsilon)$*,* $R_u \subseteq \Upsilon$*, and* $\forall \vec{\rho} \in dom(R_u).R_u(\vec{\rho}) = \Upsilon(\vec{\rho})$*, then* $\Upsilon[\rho_u \mapsto R_u] \vdash \eta \Downarrow_{decl} \Upsilon'$

Proof: [by simultaneous induction on the derivation of $\Rightarrow_{coerce}$ and the lemma below]

Rule 7.52 $\Upsilon[\rho_u \mapsto R_u] \vdash [\rho =_{def} \rho_u\rho_a] \Downarrow_{decl} [\rho \mapsto \upsilon]$ such that $\upsilon = \Upsilon[\rho_u \mapsto R_u](\rho_u\rho_a) = R_u(\rho_a) = \Upsilon(\rho_a)$. By induction, we are done.

Rule 7.56 $\Upsilon[\rho_u \mapsto R_u] \vdash \rho_u\rho_x \Downarrow_{str} \Upsilon[\rho_u \mapsto R_u](\rho_u\rho_x) = \Upsilon(\rho_x)$. $\Upsilon[\rho_u \mapsto R_u] \vdash [\rho_s =_{str} \varphi_c] \Downarrow_{decl} [\rho_s \mapsto \upsilon]$ such that $\upsilon = R_c$ because $\Upsilon \vdash \varphi_c \Downarrow_{str} R_c$ by lem. 45.

All the other cases are essentially the same as the above two cases. ■

Lemma 45

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{match} (R_c, \varphi_c)$*,* $\Upsilon \vdash \varphi \Downarrow_{str} R_a$*, and* $EV(\Sigma_a) \subseteq dom(R_a)$*, then* $\Upsilon \vdash \varphi_c \Downarrow_{str} R_c$*.*

Proof: This lemma follows from lem. 44. ■

C.3.5 Translation

Lemma 46 (Correctness of Coercion)

If $E \vdash \{lts_u\} :: k$*,* $E \vdash \rho_u : \{lts_u\}$*,* $E \vdash \alpha :: k'$*,* $E \vdash c :: k'$*, for all* $l \in dom(lts)$ $l \in dom(lts_u)$ *and* $t\{c/\alpha\} = t_u$ *such that* $l : t \in lts$ *and* $l : t_u \in lts_u$*, and* $\mathsf{coerce}(\rho_u, c, lts) = e_\omega$*, then* $E \vdash e_\omega : \{lts\{c/\alpha\}\}$*.*

Proof: [by induction on the structure of lts] If $lts = \{\}$, then $e_\omega = \{\}$. $E \vdash \{\} : \{\}$ by rule 8.14. Otherwise, $lts = \{l_0 : t_0\} \uplus lts'$ in which case $\mathsf{coerce}(\rho_u, c, \{l_0 : t_0\} \uplus lts') = \{l_0 = \rho_u.l_0\} \uplus les$ such that $les = \mathsf{coerce}(\rho_u, c, lts')$. By induction, $E \vdash \mathsf{coerce}(\rho_u, c, lts') : \{lts'\{c/\alpha\}\}$. By assumption, $l_0 \in dom(lts_u)$. Therefore, $l_0 : t_{u_0} \in lts_u$. By assumption, $t_{c_0}\{c/\alpha\} = t_{u_0}$ where $l_0 : t_{c_0} \in lts_c$. Certainly, $\{l_0 = \rho_u.l_0\} : t_{u_0} = t_{c_0}\{c/\alpha\}$. It follows that

$E \vdash e_\omega : \{l_0 : t_{c_0}, lts'\}\{c/\alpha\} = \{lts\{c/\alpha\}\}$. ■

Lemma 47

For all Σ, there exists a k such that $\vdash \Sigma \rightsquigarrow_{knd} k$.

Proof: [by induction on the structure of Σ] ■

Lemma 48

For all $\mathfrak{C}^{nf}$, there exists a t such that $\vdash \mathfrak{C}^{nf} \rightsquigarrow_{type}^{nf} t$.

Proof: [by induction on the structure of $\mathfrak{C}^{nf}$] ■

Lemma 49 (Closed semantic types translate)

1. *If $EV(\mathbb{T}) \subseteq dom(\Upsilon)$, then $\Upsilon \vdash \mathbb{T} \rightsquigarrow_{type}^{t} t$*

2. *If $EV(\Sigma) \subseteq dom(\Upsilon)$, then $\Upsilon \vdash \Sigma \rightsquigarrow_{type} t$.*

3. *If $EV(\mathbb{C}^s) \subseteq dom(\Upsilon)$ and $TV(\mathbb{C}^s \subseteq \Delta$, then $\Delta, \Upsilon \vdash \mathbb{C}^s \rightsquigarrow_{type}^{tyc} t$.*

Proof: [by induction on the size of the term]

1. The proof for this derivation follows directly from the proof of (3)

2. We consider the cases.

   $\Sigma = \emptyset_{sig}$ By rule 8.51

   $\Sigma = spec\Sigma'$ Consider the cases for spec.

   $\mathbb{T}$ By assumption $EV(\mathbb{T}) \subseteq dom(\Upsilon)$. By induction (1), $\Upsilon \vdash \mathbb{T} \rightsquigarrow_{type}^{t} t$. By induction (2), $\Upsilon \vdash \Sigma' \rightsquigarrow_{type} t'$. By rule 8.52, $\Upsilon \vdash [x \mapsto \mathbb{T}]\Sigma' \rightsquigarrow_{type} \{x : t\} \uplus t'$.

   $\mathbb{C}^\lambda$ By induction (2), $\Upsilon \vdash \Sigma' \rightsquigarrow_{type} t'$. By rule 8.53, we are done.

   (ρ, n) Similar reasoning as above

$(\rho, (\Sigma_0, R))$ By induction (2), $\Upsilon \vdash \Sigma_0 \rightsquigarrow_{type} t$ and $\Upsilon \vdash \Sigma' \rightsquigarrow_{type} t'$. By rule 8.55, we are done.

$(\rho, (\Pi\rho_x : \Sigma_x.\Sigma, \psi))$ By induction (2), $\Upsilon \vdash \Sigma_x \rightsquigarrow_{type} t_x$, $\Upsilon \vdash \Sigma \rightsquigarrow_{type} t$, and $\Upsilon \vdash \Sigma' \rightsquigarrow_{type} t'$. By rule 8.56, we are done.

3. Rule 8.47 is by assumption. Rule 8.48 relies on normalization of semantic tycon evaluation and lem. 48.

■

Lemma 50 (Tycon Substitution)

If $E \vdash e_\omega \rightsquigarrow_{type} t$, $E(\alpha) = k$, and $E \vdash c :: k$, then $E \vdash e_\omega\{c/\alpha\} \rightsquigarrow_{type} t\{c/\alpha\}$.

Proof: [by induction on the derivation of $\rightsquigarrow_{type}$] ■

Lemma 51 (Correctness of Type Synthesis)

If R is closed, $R \vdash \Sigma \rightsquigarrow_{type} lts$, $l : \mathbb{T} \in \Sigma$, $R \vdash \mathbb{T} \rightsquigarrow^{t}_{type} t$, and $l : t' \in lts$ such that t' is not a record type, then $t = t'$.

Lemma 52 (Correctness of Relativized Type Translation)

If $R(\mathbb{T}) = R'(\mathbb{T}')$ and $R \vdash \mathbb{T} \rightsquigarrow^{t}_{type} t$, then $R' \vdash \mathbb{T}' \rightsquigarrow^{t}_{type} t$.

Proof: ■

Lemma 53

If R interprets Σ, $\vdash \Sigma \rightsquigarrow_{knd} k$, and $R \vdash \Sigma \rightsquigarrow_{spec} lcs$, then $R \vdash \{lcs\} :: k$.

Proof: [by induction on the derivation of $R \vdash \Sigma \rightsquigarrow_{spec} lcs$] Both $\vdash \Sigma \rightsquigarrow_{knd} k$ and $R \vdash \Sigma \rightsquigarrow_{spec} lcs$ ignore $[x \mapsto \mathbb{T}]$ and $[x \mapsto \mathbb{C}^\lambda]$ specs. For open tycon specs, $[x \mapsto (\rho, n)]\Sigma' \rightsquigarrow_{knd} \{\rho :: \Omega^n \to \Omega\} \uplus k'$ by rule 8.34. By rule 8.41, $lcsis\rho = \mathsf{inj}(R(\rho)), lcs'$ such that $R \vdash \Sigma' \rightsquigarrow_{spec} lcs'$ for $\Sigma = [x \mapsto (\rho, n)]\Sigma'$. By def. 4, $R(\rho) = \tau^n$. Therefore

$\mathsf{inj}(R(\rho)) :: \Omega^n \to \Omega$. ■

Lemma 54

If R interprets Σ, $\vdash \Sigma \leadsto_{knd} k$, $\vdash (\Sigma, R) \leadsto_{tyc} c$, then $R \vdash c :: k$.

Proof: This proof follows directly from lem. 53. ■

Theorem 1 (Translation Preserves Well-Typing)

If $EV(\Gamma) \subseteq dom(\Upsilon)$, $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} (\widehat{strexp}, (\Sigma, R), \_)$, $\Upsilon \vdash \Gamma \hookrightarrow \Sigma'$, $\Upsilon \vdash \Sigma' \leadsto_{type} \{lts\}$, and $\widehat{strexp} \leadsto_{exp} e_\omega$, then $\mathsf{env}(lts) \vdash e_\omega : t$ and $R \vdash \Sigma \leadsto_{type} t$.

If $EV(\Gamma) \subseteq dom(\Upsilon)$, $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\widehat{d^m}, \_, \Gamma', \Upsilon')$, $\Upsilon\Upsilon' \vdash \Gamma' \hookrightarrow \Sigma'$, $\widehat{d^m} \leadsto_{dec} d$, then $\mathsf{env}(lts) \vdash d :_{dec} E'$ where $\Upsilon \vdash \Sigma \leadsto_{type} \{lts\}$, $\Upsilon\Upsilon' \vdash \Sigma' \leadsto_{type} \{lts'\}$, and $\mathsf{env}(lts') = E'$.

Proof: [By induction on the derivation of $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} (\widehat{strexp}, (\Sigma, R), \varphi)$]

Structure expressions

Rule 7.37 (structure path) The rule says $\Gamma, \Upsilon \vdash p \Rightarrow_{str} (p\langle\langle\vec{\rho}\rangle\rangle, (\Sigma, R), \vec{\rho})$. By inversion of $\Rightarrow_{str}$, $\Gamma(p) = (\vec{\rho}, M)$ where $M = (\Sigma, R)$. By definition, $[x \mapsto (\rho, M)] \in \Gamma$ such p ends in a singleton x. By definition of signature extraction (rule 7.47), $[x \mapsto (\rho, \Sigma)] \in \Sigma'$.

By rule 8.26, $p\langle\langle\vec{\rho}\rangle\rangle \leadsto_{exp} \vec{\rho}$. By lem. 39, $EV(\Sigma) \subseteq dom(R)$. By lem. 49, $R \vdash \Sigma \leadsto_{type} t$ and $\Upsilon \vdash \Sigma' \leadsto_{type} t'$. By rule 8.55, $\Upsilon \vdash [x \mapsto (\rho, M)]\Sigma' \leadsto_{type} \{\rho : t\} \uplus t' = \{lts\}$. By rule 8.15, $\mathsf{env}(lts) \vdash \vec{\rho} : t$.

Rule 7.38 (base structure) By induction and reasoning similar to the above.

Rule 7.39 (application) The nontrivial case is the application rule.

$\widehat{strexp} = p\langle\langle\vec{\rho}, \rho_{par}\rangle\rangle(\widehat{strexp}_x\langle\langle M_c, \rho_u\rangle\rangle)$.

By inversion of rule 7.39, $\Gamma(p) = (\vec{\rho}, (\Pi\rho_{par} : \Sigma_{par}.\Sigma_{body}, \psi))$ (b.1), $\Gamma, \Upsilon \vdash \widehat{strexp}_x \Rightarrow_{str} (M_x, \varphi_x)$ (b.2), $\Upsilon \vdash (M_x, \varphi_x) : \Sigma_{par} \Rightarrow_{match} (R_c, \varphi_c)$ (b.3), $\varphi_{app} = \vec{\rho}(\varphi_c)$ (b.4), and $\Upsilon \vdash \varphi_{app} \Downarrow_{str} R_{app}$ (b.5).

This is a caveat for (b.1). $\Gamma(p)$ contains a ψ if p is a real functor. If it is a formal functor, $\Gamma(p)$ will only have the entity variable and functor signature. The ψ will have to be produced from $\Upsilon(\vec{\rho})$. In either case, we can get a full functor signature.

By assumption, $\Upsilon \vdash \Gamma \hookrightarrow \Sigma'$ and $\Upsilon \vdash \Sigma' \rightsquigarrow_{type} \{lts\}$. Let ρ be the last entity variable in $\vec{\rho}$ and f be the final symbol in p. Then by definition of signature extraction, $[f \mapsto (\rho, \Pi\rho_{par} : \Sigma_{par}.\Sigma_{body})] \in \Sigma'$. Thus, $\Upsilon \vdash [f \mapsto (\rho, \Pi\rho_{par} : \Sigma_{par}.\Sigma_{body})]\Sigma'' \rightsquigarrow_{type} \{\rho : \forall\rho_{par} :: k.t_{par} \to t_{body}\} \uplus t''$ By inversion, $\vdash \Sigma_{par} \rightsquigarrow_{knd} k$ (c.1), $\Upsilon \vdash \Sigma_{par} \rightsquigarrow_{type} t_{par}$ (c.2), $\Upsilon \vdash \Sigma_{body} \rightsquigarrow_{type} t_{body}$ (c.3), and $\Upsilon \vdash \Sigma'' \rightsquigarrow_{type} t''$ (c.4).

Because $\{\rho : \forall\rho_{par} :: k.t_{par} \to t_{body}\} \in lts$, $E(\vec{\rho}) = \forall\rho_{par} :: k.t_{par} \to t_{body}$.

By assumption and rule 8.28, $\Gamma, \Upsilon \vdash p\langle\langle\vec{\rho}, \rho_{par}\rangle\rangle(\widehat{strexp}_x\langle\langle M_c, \rho_u\rangle\rangle) \rightsquigarrow_{exp}$ $\textbf{let}\ \rho_u = e_{\omega_x}\ \textbf{in}\ \vec{\rho}[c](e_{\omega_c})$ where $\vdash M_c \rightsquigarrow_{tyc} c$, $\widehat{strexp}_x \rightsquigarrow_{exp} e_{\omega_x}$ (a.0), $R_c \vdash \Sigma_{par} \rightsquigarrow_{type} \{lts_c\}$ (a.1), and $e_{\omega_c} = \mathsf{coerce}(\rho_u, c)$ (a.2). Let $M_c = (\Sigma_{par}, R_c)$ and $t_{par} = \{lts_c\}$.

By rule 8.9, $E \vdash \vec{\rho} : \forall\rho_{par} :: k.t_{par} \to t_{body}$. By lem. 54, $E \vdash c :: k$. By rule 8.13, $E \vdash \vec{\rho}[c] : t_{par}\{c/\rho_{par}\}_{tyc} \to t_{body}\{c/\rho_{par}\}_{tyc}$.

Let $M_x = (\Sigma_x, R_x)$. By (b.2), (a.0), and the original assumptions, $E \vdash e_{\omega_x} : t_x$ and $R_x \vdash \Sigma_x \rightsquigarrow_{type} \{lts_x\}$.

By definition, $\mathsf{coerce}(\rho_u, \{lts_{par}\}) = \{x_1 = \rho_u.x)_1, \ldots, x_n = \rho_u.x_n\}$ where $x_i : t_i \in lts_{par}$.

$E[\rho_u = e_{\omega_x}] \vdash x_1 = \rho_u.x_1 :_{dec} x_1 : t'_1$ such that $\forall i \in [1,n] x_i : t'_i \in lts_x$.

By definition of tycon synthesis, $c = \{lcs\}$ where $R_c \vdash \Sigma_{par} \rightsquigarrow_{spec} lcs$ such that:

1. $\forall[x \mapsto (\rho_i, n)] \in \Sigma_{par}\ \rho_i = c_i \in lcs$ such that $c_i = \mathsf{inj}(R_c(\rho_i))$. By lem. 43, for all $[x \mapsto (\rho_i, n)] \in \Sigma_{par}$, $[x \mapsto s'] \in \Sigma_x$ such that $R_c(\rho_i) = R_x(s')$. Therefore, $t_i\{\{\rho_i = \mathsf{inj}(R_c(\rho_i))\}/\rho\}_{tyc} = t_i\{\{\rho_i = R_x(s')\}/\rho\}_{tyc} = t'_i$.
2. $\forall[x \mapsto (\rho_i, \Sigma)] \in \Sigma_{par}\ \rho_i = \{lcs_i\}$ such that $R(\rho_i) \vdash \Sigma \rightsquigarrow_{spec} lcs_i$. This case goes through by induction.

3. $\forall[x \mapsto (\rho_i, \Pi\rho'_{par} : \Sigma'_{par}.\Sigma'_{body})] \in \Sigma_{par}\ \rho_i = \lambda\rho'_{par} :: k.c'$ such that $\vdash \Sigma'_{par} \rightsquigarrow_{knd} k$ and $\vdash R_c(\rho_i) \rightsquigarrow^{strexp}_{tyc} c'$.

(d.1)

Thus, $E[\rho_u = e_{\omega_x}] \vdash \mathsf{coerce}(\rho_u, \{lts_{par}\}) : \{lts_{par}\}\{c/\rho\}_{tyc}$. Recall that $e_{\omega_c} = \mathsf{coerce}(\rho_u, \{lts_{par}\})$. Let $E' = E[\rho_u = e_{\omega_x}]$. By rule 8.11, $E' \vdash \vec{\rho}[c](e_{\omega_c}) : t_{body}\{c/\rho_{par}\}$.

Let $t_{body} = \{lts_{body}\}$. Since $\Upsilon \subseteq R_{app}$, by (c.3) and reasoning similar to (d.1), $R_{app} \vdash \Sigma_{body} \rightsquigarrow_{type} lts_{body}\{c/\rho_{par}\}$.

Module declarations These cases are straightforward given the structure expression proof.

$\blacksquare$

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. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[8] 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.

[9] 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.

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

[11] 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.

[12] Derek Dreyer. Recursive type generativity. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 41–53, New York, NY, USA, 2005. ACM.

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

[14] 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.

[15] Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 441–457, 2007.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, University of Edinburgh, December 1997.

[26] Paul Govereau. Type generativity in higher-order module systems. Technical report, Harvard University, 2005.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] Robert Harper and Christopher Stone. *A type-theoretic interpretation of Standard ML*, pages 341–387. MIT Press, Cambridge, MA, USA, 2000.

[33] 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.

[34] 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.

[35] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Inf. Comput.*, 76(2-3):278–346, 1988.

[36] 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.

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 37–51, New York, NY, USA, 1985. ACM.

[49] Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 63–74, Savannah, Georgia, USA, January 2009.

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

[51] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 135–148, New York, NY, USA, 2009. ACM.

[52] 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.

[53] 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.

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

[55] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the 8th annual ACM Symposium on Theory of Computing*, pages 181–186, New York, NY, USA, 1976. ACM.

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

[57] 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.

[58] 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.

[59] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 241–252, New York, NY, USA, 2003. ACM.

[60] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 89–102, New York, NY, USA, 2010. ACM.

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

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

[63] 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.

[64] 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.

[65] Zhong Shao. Parameterized signatures and higher-order modules. Technical Report YALEU/DCS/TR-1161, Yale University, August 1998.

[66] 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.

[67] 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.

[68] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 2000. ACM.

[69] 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.

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

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.