What is a Recursive Module?

Karl Crary Robert Harper Sidd Puri
October 1998
CMU-CS-98-168

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A hierarchical module system is an effective tool for structuring large programs. Strictly hierarchical module systems impose an acyclic ordering on import dependencies among program units. This can impede modular programming by forcing mutually-dependent components to be consolidated into a single module. Recently there have been several proposals for module systems that admit cyclic dependencies, but it is not clear how these proposals relate to one another, nor how one might integrate them into an expressive module system such as that of Standard ML or O'Caml. To address this question we provide a type-theoretic analysis of the notion of a recursive module in the context of the "phase-distinction" formalism for higher-order module systems. We extend this calculus with a recursive module mechanism and a new form of signature, called a *recursively-dependent signature*, to support the definition of recursive modules. These extensions are justified by an interpretation in terms of more primitive language constructs. This interpretation may also serve as a guide for implementation.

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: Type systems, module systems, functional programming, phase splitting.

1 Introduction

Hierarchical decomposition is a fundamental design principle for controlling the complexity of large programs. According to this principle a software system is to be decomposed into a collection of modules whose dependency relationships form a directed, acyclic graph. Most modern programming languages include module systems that support hierarchical decomposition. Many, such as Standard ML [13] and O'Caml [12], also support parameterized, or generic, modules to better support code re-use.

There is no question that hierarchical design is an important tool for structuring large systems. It has often been noted, however, that strict adherence to a hierarchical architecture can preclude the decomposition of a system into "mind-sized" components. In some situations the natural decomposition of a system into modules introduces cyclic dependencies, which cannot be expressed in a purely hierarchical formalism. The only solution is to consolidate mutually-dependent fragments into a single module, which partially undermines the very idea of modular organization.

In response several authors have proposed linguistic mechanisms to support non-hierarchical modular decomposition. Recent examples include: Sirer, *et al.'s* extension of Modula-3 with a "cross-linking" mechanism [17]; Flatt and Felleisen's extension of their MzScheme language with cyclically-dependent "units" [6]; Duggan and Sourelis's "mixin modules" that extend the Standard ML module system with a special "mixlink" construct for integrating mutually-dependent structures [4, 5]; and Ancona and Zucca's algebraic formalism for mixin modules [2]. Each of these proposals seeks to address the problem of cyclic dependencies in a module system, but each does so in a slightly different way. For example, Flatt and Felleisen's formalism does not address the critical issue of controlling propagation of type information across module boundaries. Duggan and Sourelis's framework relies on a syntactic transformation that, in effect, coalesces the code of mutually-dependent modules into a single module. It is not clear what are the fundamental ideas, nor is it clear how to integrate the various aspects of these proposals into a full-featured module system.

It is natural to ask: what is a recursive module? We propose to address this question in the framework of type theory, which has proved to be a powerful tool for both the design and implementation of module systems. We conduct our analysis in the context of the "phase distinction" module formalism introduced by Harper, Mitchell, and Moggi [9] (hereafter, HMM). The phase distinction calculus provides a rigorous account of higher-order modules (supporting hierarchy and parameterization) in a framework that makes explicit the critical distinction between the static, or compile-time, part of a module and the dynamic, or run-time, part. This calculus has proved to be of fundamental importance to the implementation of higher-order modules, as evidenced by its use in Shao's FLINT formalism used in the SML/NJ compiler [15] and in the TIL/ML compiler [18].

Our analysis proceeds in two stages. First we consider a straightforward extension of the phase distinction calculus with a notion of recursive (self-referential) module. An interpretation of this new construct is provided by an interpretation of it into the primitive module formalism of the phase distinction calculus. This interpretation renders the compile-time part as a recursive type and the run-time part as a recursive function, as might be expected. In essence a recursive module is just a convenient way of introducing recursive types and functions.

Unfortunately this simple-minded extension does not go far enough to be of much practical use. As Duggan and Sourelis have observed [5], it is of critical importance for most practical examples that the type equations that hold of a recursive module be propagated into the definition of the recursive module itself. In essence the definitions of the type components of a recursive module must be taken to be the types that they will eventually turn out to be (!) once the recursive declaration has been processed. Accounting for this "forward reference" is the core contribution of our work. We introduce a new form of signature (interface) for recursive modules, called a *recursively-dependent signature*, that allows us to capture the required type identities during type checking of a recursive module binding. This significantly increases the expressive power of the recursive module formalism, and is, we assert, of fundamental importance to the very idea of recursive modules.

2 Type-Theoretic Framework

We begin by presenting the framework in which we conduct our analysis. We will conduct our examples using an informal external language closely modeled after the syntax of Standard ML. The external language is then elaborated into the type-theoretic internal language that we describe below. We will treat the elaboration process informally, illustrating it by examples. Details of how elaboration may be formalized appear in Harper and Stone [10].

Our internal language is an extension of the *phase distinction calculus* of Harper, Mitchell, and Moggi [9]. The language consists of two main components: a *core calculus*, a predicative variant of Girard's F_ω, and a *structure calculus*, extending the core language with a primitive module construct without explicit mechanisms for hierarchy (*e.g.*, substructures) or parameterization (*e.g.*, functors). Primitive modules consist of a static, or compile-time, part containing the type constructors of the module, together with a dynamic, or run-time, part containing the executable code of the module. This separation is known as the *phase distinction*. An important property of the formalism is that the phase distinction is maintained, even in the presence of higher-order (and, as we shall see, recursive) module constructs.

The main result of HMM is that higher-order module constructs are a definitional extension of the primitive structure calculus. In other words higher-order constructs are *already present* in the primitive structure calculus in the sense that they may be defined in terms of existing constructs. (This interpretation may be thought of as a compilation strategy for higher-order modules, and indeed this fact has been exploited in the FLINT [15] and TIL [18] compilers.) This means that we need not explicitly discuss higher-order module constructs in this paper, but rather appeal to HMM for a detailed discussion of their implicit presence.

To support the extension with recursive modules we enrich the core phase distinction calculus with these additional constructs:

1. Singleton and dependent kinds to allow expression of type sharing information in signatures. Related formalisms for expressing type sharing information are given by Harper and Lillibridge [7] and Leroy [11].

2. A fixed point operation for building collections of mutually-recursive type constructors. These recursive constructors are definitionally equal to their unrollings. We term such constructors *equi-recursive*, to distinguish them from the more conventional *iso-recursive* constructors, in which conversions between the constructors and their unrollings must be mediated by the explicit use of an isomorphism. We discuss the interplay of equi- and iso-recursive constructors in Section 5.

3. A fixed point operation for building collections of mutually-recursive functions. As will become apparent later on, we cannot (as in SML) limit this operation to collections of explicit lambda abstractions. Instead we formalize a notion of *valuability* (indicating terminating expressions) and a corresponding notion of *total function,* essentially as in Harper and Stone [10], but with the additional idea that recursively-defined variables are not considered valuable within the body of their definitions, but are considered valuable in their subsequent scope.

In subsequent sections of this paper, we will further augment our structure calculus with various constructs for recursive modules, and then show how those constructs can be reduced to the elementary constructs discussed in this section.

2.1 The Core Calculus

The core phase distinction calculus contains four syntactic classes: kinds, type constructors (or just "constructors"), types, and terms. As usual, types classify terms and kinds classify constructors. The constructors provide a lambda calculus for constructing types. The syntax of the core calculus appears in Figure 1. We shall consider expressions that differ only in the names of bound variables to be identical, and write capture-avoiding substitution of E for X in E' as $E'[E/X]$.

The kinds include the kind T of all monotypes; the trivial kind 1, containing only the constructor $\star$; dependent products $\Pi\alpha{:}\kappa_1.\kappa$, containing constructor functions from κ_1 to κ_2 where α stands for the

$$
\begin{array}{llcl}
\textit{kinds} & \kappa & ::= & T \mid 1 \mid \mathcal{Q}(c) \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \Sigma\alpha{:}\kappa_1.\kappa_2 \\
\textit{constructors} & c & ::= & \alpha \mid \star \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \langle c_1, c_2\rangle \mid \pi_i(c) \mid 1 \mid c_1 \rightharpoonup c_2 \mid \\
 & & & c_1 \times c_2 \mid \mu\alpha{:}\kappa.c \\
\textit{types} & \sigma & ::= & c \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \rightharpoonup \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \\
\textit{terms} & e & ::= & x \mid \star \mid \lambda x{:}\sigma.e \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid \pi_i(e) \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \mathit{fix}(x{:}\sigma.e) \\
\textit{contexts} & \Gamma & ::= & \epsilon \mid \Gamma[\alpha : \kappa] \mid \Gamma[x : \sigma] \mid \Gamma[\alpha \uparrow \kappa] \mid \Gamma[x \uparrow \sigma]
\end{array}
$$

Figure 1: The Core Calculus

$$
\begin{array}{rcll}
\mathcal{Q}(c : T) & \stackrel{\text{def}}{=} & \mathcal{Q}(c) & \\
\mathcal{Q}(c : \Pi\alpha{:}\kappa_1.\kappa_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\kappa_1.\, \mathcal{Q}(c\,\alpha : \kappa_2) & \quad (\text{for } \alpha \text{ not free in } c) \\
\mathcal{Q}(c : \kappa_1 \times \kappa_2) & \stackrel{\text{def}}{=} & \mathcal{Q}(\pi_1(c) : \kappa_1) \times \mathcal{Q}(\pi_2(c) : \kappa_2) & \\
\mathcal{Q}(c : 1) & \stackrel{\text{def}}{=} & 1 &
\end{array}
$$

Figure 2: Higher-Order Singletons

argument and may appear free in κ_2; and dependent sums $\Sigma\alpha{:}\kappa_1.\kappa_2$, containing constructor pairs built from κ_1 and κ_2 (respectively) where α stands for the left-hand member and may appear free in κ_2. As usual, if α does not appear free in κ_2, we write $\kappa_1 \to \kappa_2$ for $\Pi\alpha{:}\kappa_1.\kappa_2$ and $\kappa_1 \times \kappa_2$ for $\Sigma\alpha{:}\kappa_2.\kappa_2$.

Finally, for any constructor c with kind T, the *singleton kind* $\mathcal{Q}(c)$ contains monotypes definitionally equal to c. Thus, if c has kind $\mathcal{Q}(c')$, the calculus permits the deduction of the equation $c = c' : T$. Singleton kinds provide a mechanism for expressing type sharing information [7, 11]. Although singleton kinds exist only for monotypes, they may be used in conjunction with higher dependent kinds to express higher-order sharing information. For instance, if c has kind $\Pi\alpha{:}T.\,\mathcal{Q}(\mathit{list}(\alpha))$, it follows that $c = \mathit{list} : T \to T$. The definition in Figure 2 generalizes this idea.[1]

The type constructors are largely standard. The trivial type 1 contains the trivial term $\star$. The types $c_1 \to c_2$ and $c_1 \rightharpoonup c_2$ are the types of total and partial functions from c_1 to c_2 and are discussed in more detail below. The *equi-recursive* constructor $\mu\alpha{:}\kappa.c[\alpha]$ is is a fixed point of the equation $\alpha = c[\alpha]$. Thus $\mu\alpha{:}\kappa.c[\alpha]$ is equal to its unrolling $c[\mu\alpha{:}\kappa.c[\alpha]]$. This is in contrast to the somewhat more conventional *iso-recursive* formulation, where conversions between the two must be mediated by explicit operations. In Section 5 we discuss how to simplify the type theory to use only iso-recursive constructors.

The final construct, $\mathit{fix}(x{:}\sigma.e)$ at the term level, allows the definition of recursive values. However, to achieve conservativity over ML, we wish to prevent the definition of cyclic data structures such as $\mathit{fix}(x : \texttt{int list}.\texttt{1::}x)$, which cannot be defined in ML.[2] We do this by imposing a value restriction on the bodies of recursive definitions. The calculus contains judgements $\Gamma \vdash e \downarrow \sigma$ asserting that e has type σ and terminates without computational effects. (In the present setting, the only computational effect is nontermination.) With this so-called value restriction in place, the formation rule for recursive values is:

$$
\frac{\Gamma[x \uparrow \sigma] \vdash e \downarrow \sigma \qquad \Gamma \vdash \sigma \ \mathit{type}}{\Gamma \vdash \mathit{fix}(x{:}\sigma.e) : \sigma} \ (x \notin \mathrm{Dom}(\Gamma))
$$

This rule is read: $\mathit{fix}(x{:}\sigma.e)$ has type σ if e terminates with type σ under the assumption that x has type σ *but cannot be taken as valuable.* This rules out the cyclic list proposed above, since $\texttt{1::}x$ is not valuable unless x is valuable. The value restriction implies that all appearances of x must be guarded by (*i.e.*, within the body of) a lambda abstraction; lambda abstractions are always valuable, regardless of the state of their

[1] Note that $\mathcal{Q}(c : \kappa)$ is not defined when κ is a singleton or dependent sum kind. This does not change the expressive power of this construct and is necessary to obtain some desirable properties, for instance that $\mathcal{Q}(c : \kappa)$ is a subkind of κ, and that κ is uniquely determined by $\mathcal{Q}(c : \kappa)$.

[2] To achieve conservativity over ML, we will also need to eliminate equi-recursive types in favor of iso-recursive types. We discuss this in Section 5.

$$
\begin{array}{lrcl}
\textit{constructors} & c & ::= & \cdots \mid Fst(s) \\
\textit{terms} & e & ::= & \cdots \mid snd(s) \\
\textit{signatures} & S & ::= & [\alpha{:}\kappa.\sigma] \\
\textit{modules} & M & ::= & [c, e] \\
\textit{contexts} & \Gamma & ::= & \cdots \mid \Gamma[s : S] \mid \Gamma[s \uparrow S]
\end{array}
$$

Figure 3: The Structure Calculus

free variables. As in Harper and Stone [10], the collection of valuable expression is enlarged by including a type for total (pure) functions, such as cons (`::`). The application of a valuable total function to a valuable argument is considered valuable. Total functions are considered to be types, but not type constructors, in order to prevent their erroneous use in conjunction with recursive types.

A similar restriction is made on the formation on equi-recursive type constructors. In order to show that $\mu\alpha{:}\kappa.c$ is well-formed, one must show that c is *contractive* in α [1]. This is in contrast to iso-recursive types, which require no such condition. Informally, contractiveness means that the infinite tree specified by iterating the body c is actually infinite, or, in other words, that iterating the body "goes somewhere." Thus, $\mu\alpha{:}T.\texttt{int} \times \alpha$ is legal, but $\mu\alpha{:}T.\alpha$ is not. Contractiveness is formalized in the type theory by a judgement $[\alpha \uparrow \kappa] \vdash c \downarrow \kappa$, which is read: c is contractive in kind κ under the assumption that α has kind κ but cannot be taken as contractive. The rules for contractiveness of constructors are similar to those for valuability of terms; they ensure that all occurrences of the recursive variable are guarded by a type construction (such as $\texttt{int} \times \alpha$ in the above example).

2.2 The Structure Calculus

Atop the core calculus we erect a structure calculus, exactly as in HMM. To review, we add two syntactic classes, one for flat signatures and one for flat structures (Figure 3). Structures are pairs $[c, e]$ of constructors and terms. The left-hand component is referred to as the compile-time (or, static) component, and the right-hand component is referred to as the run-time (or, dynamic) component. Signatures, which classify structures, have the form $[\alpha{:}\kappa, \sigma]$, where α stands for the compile-time component and may appear free in σ. The structure, $[c, e]$ has kind $[\alpha{:}\kappa, \sigma]$ if c has kind κ and e has type $\sigma[c/\alpha]$. Often we will write $[\alpha = c, e]$ as shorthand for $[c, e[c/\alpha]]$. We also add constructor and term constructs $Fst(s)$ and $snd(s)$ for extracting the first and second components out of structures named by variables. We will occasionally treat these constructs as variables and allow substitution for them.

The structure calculus shows an explicit *phase distinction* between compile-time and run-time expressions [9, 3]. Static expressions may be separated from dynamic ones, and static ones will never depend on dynamic ones. This ensures that programs may be typechecked without the need to execute any run-time code.

HMM show that higher-order modules can be reduced to the simple structure calculus given here. Therefore we will omit explicit discussion of higher-order modules, without any loss of generality. In this paper, we show how recursive modules may similarly be reduced to the structure calculus given here. In so doing, we will show that despite the apparent intertwining of static and dynamic expressions in recursive modules, that the phase distinction can be preserved, just as HMM showed for higher-order modules.

3 Opaque Recursive Modules

We begin our examination by considering what we call "opaque" recursive modules. These will prove to insufficiently expressive for most applications, but they will serve to illustrate the main ideas and motivate the more complex machinery in the next section.

In the (informal) external language, we write an opaque recursive module definition as:

```
structure rec S :> SIG = struct ... end
```

$$\frac{\Gamma \vdash \kappa\ kind \quad \Gamma[\alpha:\kappa] \vdash \sigma\ type \quad \Gamma[\alpha:\kappa] \vdash c:\kappa \quad \Gamma[\alpha:\kappa][x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(s{:}[\alpha{:}\kappa.\sigma].[c[\mathit{Fst}(s)/\alpha], e[\mathit{Fst}(s), \mathit{snd}(s)/\alpha, x]]) = [\alpha = \mu\alpha{:}\kappa.c, \mathit{fix}(x{:}\sigma.e)] : [\alpha{:}\kappa.\sigma]}\ (\alpha, x \notin \mathrm{Dom}(\Gamma))$$

Figure 4: Phase-Splitting Recursive Modules

The structure variable `S` is, of course, permitted to appear free within the structure's body. The signature `SIG` then expresses all the information that is known about `S` in the body or in the subsequent code. (We borrow the "`:>`" symbol from Standard ML 1997 [13] to suggest this opacity.) In particular, the opaque signature obscures the fact that the types in `S` are recursively defined.

This declaration construct corresponds to a module fixed point operation in the internal language, written $\mathit{fix}(s{:}S.M)$. For reasons similar to those in the previous section, we must impose a value restriction on M, resulting in the following typing rule:

$$\frac{\Gamma[s \uparrow S] \vdash M \downarrow S \quad \Gamma \vdash S\ sig}{\Gamma \vdash \mathit{fix}(s{:}S.M) : S}\ (s \notin \mathrm{Dom}(\Gamma))$$

Thus, a recursive module is valid if its body (M) is valuable without assuming the recursive variable (s) to be valuable. If a module M is $[c, e]$, then M will be valuable exactly when e is valuable (*i.e.*, constructors are always valuable).

Following HMM, we wish to reduce recursive modules to the primitive structure formalism by defining $\mathit{fix}(s{:}S.M)$ in terms of primitive constructs. We will do this by phase-splitting recursive modules into run-time and compile-time components. Suppose S is the signature $[\alpha{:}\kappa.\sigma]$ and M is the structure $[c(\mathit{Fst}(s)), e(\mathit{Fst}(s), \mathit{snd}(s))]$. Then we can interpret $\mathit{fix}(s{:}S.M)$ by wrapping the static and dynamic components in fixed point expressions:

$$\mathit{fix}(s{:}S.M) = [\alpha = \mu\alpha{:}\kappa.c(\alpha), \mathit{fix}(x:\sigma.e(\alpha, x))]$$

This definition is formalized in the type theory by the equational rule in Figure 4. This rule parallels the non-standard equational rules from HMM, and illustrates that recursive modules are *already present* in the underlying calculus. In particular, the formation rule for recursive modules given above follows from the definition and need not appear as a primitive rule.

3.1 Trouble with Opacity

The opaque interpretation of recursive modules is pleasantly simple, but unfortunately, it is not sufficiently expressive to support some desired programming idioms. One common application of recursive modules is to break up mutually recursive data types. As a particularly simple (though somewhat contrived) example, consider an implementation of integer lists as a recursive module that defers recursively to itself for an implementation of the tail:

```
signature LIST =
   sig
      type t
      val nil : t
      val null : t -> bool
      val cons : int * t -> t
      val uncons : t -> int * t
   end
```

```
structure rec List :> LIST =
   struct
      datatype t = NIL | CONS of int * List.t
      val nil = NIL
      fun null NIL = true
        | null (CONS _) = false
      fun cons (n : int, l : t) =
         case l of
            NIL => CONS (n, List.nil)
          | CONS (n' : int, l' : List.t) =>
               CONS (n, List.cons (n', l'))
      fun uncons NIL = raise Fail
        | uncons (CONS (n : int, l : List.t)) =
             if List.null l then
                (n, NIL)
             else
                (n, CONS (List.uncons l))
   end
```

This implementation typechecks properly, and it is observationally equivalent to a conventional implementation. However, intensionally it is very different, because each use of `cons` and `uncons` must traverse the entire list, leading to poor behavior in practice. A more direct implementation is impossible because the opacity of `List.t` precludes any knowledge that `List.t` is the same as `t`.

Some other examples cannot be written in the opaque case at all. For example, consider an implementation of abstract syntax trees using mutually dependent modules for expressions and declarations. These modules interact with each other through the `let` expression, which contains a declaration, and the `val` declaration, which contains an expression. To optimize a common case, the expression code includes a function for `let val` expressions that defers to the declaration code to build a declaration:

```
signature EXPR =
   sig
      type exp
      type dec
      val make_let : dec * exp -> exp  (* let DEC in EXP end *)
      val make_let_val : identifier * exp * exp -> exp
                                (* let val ID = EXP in EXP end *)
      ⋮
   end

signature DECL =
   sig
      type dec
      type exp
      val make_val : identifier * exp -> dec  (* val ID = EXP *)
      ⋮
   end
```

```
structure rec Expr :> EXPR =
   struct
      datatype exp = LET of Decl.dec * exp | ...
      type dec = Decl.dec

      fun make_let (d : dec, e : exp) = LET (d, e)
      fun make_let_val (id : identifier, e1 : exp, e2 : exp) =
         let val d = Decl.make_val (id, e1)  (* type error! e1 : exp ≠ Decl.exp *)
         in
            LET (d, e2)
      ⋮
   end
and Decl :> DECL =
   struct
      datatype dec = ...
      type exp = Expr.exp
      ⋮
   end
```

Unfortunately, this code does not typecheck. The call to `make_val` within `make_let_val` expects an argument with type `Decl.exp`, which, because of the opacity of `Decl`, is not known to be the same type as `exp`, the type of its actual argument `e1`. The type error occurs because the type system cannot tell that `exp` is equal to `Decl.exp`, even though an examination of the recursive definition reveals that it is actually true.

4 Transparent Recursive Modules

The difficulties described in the previous section can be traced to the inability to track sufficient type information in the context of a recursive structure binding. In the abstract syntax example the proposed binding fails to typecheck because within the definition of `Expr` it is not apparent that the type `exp` is equivalent to the type `Decl.exp`, *even though* this equation will be valid once the recursive binding is in force. Similarly, within the definition of `Decl` it is not apparent that the type `dec` is equivalent to the type `Expr.dec`, which will turn out to be true once the binding is in force. Were this equation available while the definitions of `Expr` and `Decl` are being typechecked, the entire declaration would be seen to be valid, and these very equations would hold true afterwards. Similarly, the inefficiency of the suggested implementation of lists may be traced to the failure to identify the types `List.t` and `t` inside the definition of `List`.

What is needed is a means of propagating the type equations that will, upon completion of the recursive binding, turn out to be true of the recursively-defined structures, into the scope of the recursive definition itself. This makes it possible to exploit the recursive definitions of the types involved during typechecking of the dynamic part of the recursively-defined modules, leading to a much more flexible and useful notion of recursive module. In effect we are exploiting the phase distinction by solving the *static* recursion equations prior to checking the *dynamic* typing conditions of the module, but we are achieving this using a one-pass algorithm.

How is this additional type sharing information to be propagated? The obvious solution is to add the appropriate equations to the signatures of the modules involved. In the case of the abstract syntax example this may be achieved as follows:

```
structure rec Expr : EXPR where type dec = Decl.dec = ...
          and Decl : DECL where type exp = Expr.exp = ...
```

Similarly, in the list example we may propagate the required information as follows:

```
structure rec List : sig
                        datatype t = NIL | CONS of int * List.t
                        ⋮
                        val cons : int * t -> t
                        val uncons : t -> int * t
                     end = ...
```

The underlined phrases indicate free occurrences of structure variables that are introduced by the recursive structure binding. Since the signatures of the recursively-defined structure variables depend on the structures themselves, we call these signatures *recursively-dependent signatures*, or *rds's* for short.

The purpose of a recursively-dependent signature is to express the sorts of recursive type equations that are required to recover the ill-formed examples of the preceding section. Let us now revisit those examples to see how rds's are used to resolve the difficulties those examples raise. With the addition of the type definitions given above, the abstract syntax example from Section 3 is now type correct since the equations `Expr.dec` = `Decl.dec` and `Decl.exp` = `Expr.exp` are propagated into the bindings of the structures `Expr` and `Decl`. This is all that is required for the code given in Section 3 to be type correct.

The list example is handled similarly, but also raises a delicate point about recursive datatypes in the context of a recursive structure binding. Using a recursively-dependent signature it is possible to give an implementation of lists with constant-time primitive operations as follows:

```
structure rec List : sig
                        datatype t = NIL | CONS of int * List.t
                        ⋮
                        val cons : int * t -> t
                        val uncons : t -> int * t
                     end =
  struct
     datatype t = NIL | CONS of int * List.t
     ⋮
     fun cons (n : int, l : t) = CONS (n, l)
     fun uncons NIL = raise Fail
       | uncons (CONS (n, l)) = (n, l)
  end
```

The effect of the recursively-dependent signature in this example is to ensure that the *implementation type* of the recursive datatype `List.t` coincides with the *implementation type* of the type `t` within the body of the definition. In other words we impose a *structural*, or *transparent*, interpretation of recursive datatypes within the scope of a recursive structure binding, rather than the more familiar *nominal*, or *opaque*, interpretation used in Standard ML. In type-theoretic terms the rds ascribed to `List` is tantamount to a signature that *transparently* defines the type `t` to be the underlying iso-recursive type of the recursive datatype. We note, however, that this interpretation can be limited to the recursive structure binding itself, and need not propagated into the subsequent scope of the binding. In type-theoretic terms the elaborator must "seal" the structure with an opaque signature hiding the implementation type of `List.t` after the binding has been processed.

In order to maximize the propagation of type information we will assume that the elaborator implicitly renders every recursively dependent signature to be fully transparent in the sense that every type component is given by an explicit type definition. In the present case of recursive module definitions, the elaborator can produce the needed fully transparent signature by inspection of the module being defined. (This is always possible since we are assuming a transparent interpretation of datatypes.)

As emphasized by Duggan and Sourelis [5], it is important in practice to consider recursive structure bindings whose right-hand sides are applications of previously-defined functors (parameterized modules). A naïve attempt to do so runs afoul of the opacity problem once again, as demonstrated by the following "functorized" version of the list example. Specifically, we wish to define the `List` structure as follows:

```
structure rec List : sig
                        datatype t = NIL | CONS of int * List.t
                        ⋮
                     end =
   BuildList (structure List = List)
```

where the functor **BuildList** abstracts the efficient implementation of lists as follows:

```
functor BuildList (structure List : LIST) = ... as above ...
```

However, the efficient implementation of lists no longer typechecks since the assumption governing the parameter **List** of **BuildList** does not propagate the critical recursive type equation, as was observed by Duggan and Sourelis (for an essentially similar example). The solution is to use a recursively-dependent signature for the functor parameter, as follows:

```
functor BuildList (structure rec List : sig
                                           datatype t = NIL | CONS of int * List.t
                                           ⋮
                                        end) =
  ... as above ...
```

Here again we assume a structural interpretation of datatypes that occur within an rds, which is consistent with the structural signature matching rule for functor applications in Standard ML.

Again, the elaborator must render rds's fully transparent. For recursive module definitions that was easily done by inspection of the definition. However, when an rds is the signature of a functor argument, the argument is unavailable for such inspection. In order to render such rds's fully transparent, the elaborator must name any abstract types within the signature and pull them out. (A similar device is used by the generative stamps in the Definition of Standard ML [13].) For example, the signature

```
rec S : sig
           type t
           type u = S.u -> t
        end
```

is rewritten by introducing a type definition for `t` setting it equal to an abstract type that is defined outside the rds. The resulting signature is acceptable because the rds, which now lies within an outer signature, is fully transparent:

```
sig
   type t'
   structure rec S :
      sig
         type t = t'
         type u = S.u -> t
      end
end
```

In order to allow the phase-splitting of rds's, we must also require that the static component of rds's be contractive (recall Section 2.1). The elaborator can usually rewrite signatures to achieve this condition in the same way it rewrites them to achieve full transparency. For example, the signature

```
rec S : sig
           type t = S.t
           type u = S.u -> t
        end
```

is noncontractive because it does not provide any information as to the identity of `t`. The elaborator can correct this by naming and hoisting `t` exactly as before, in fact producing the same legal rds as before.

$$\frac{\Gamma \vdash \kappa\ kind \quad \Gamma[\alpha:\kappa] \vdash \sigma[\alpha/Fst(s)]\ type \quad \Gamma[\beta:\kappa] \vdash \mathcal{Q}(c:\kappa)\ kind \quad \Gamma[\beta \uparrow \kappa] \vdash c \downarrow \kappa}{\rho s.[\alpha{:}\mathcal{Q}(c[Fst(s)/\beta]:\kappa), \sigma] = [\alpha{:}\mathcal{Q}((\mu\beta{:}\kappa.c):\kappa), \sigma[\alpha/Fst(s)]]}$$

Figure 5: Phase-Splitting Recursively-Dependent Signatures

4.1 Formalization of Recursively-Dependent Signatures

The addition of recursively-dependent signatures to the phase distinction calculus is performed in two stages. First, we extend the syntax of signatures with the recursively dependent form, which we write $\rho s.S$, and extend the signature formation and equivalence rules with rules governing this new form. We also extend the module formation rules to include introductory and eliminatory rules for recursively-dependent signatures. Second, we show that this enrichment of the structure calculus may be interpreted into the original structure calculus (over the extended core language described in Section 2) by exhibiting an equation between rds's and ordinary signatures.

Informally, the recursively-dependent signature $\rho s.S$ contains those modules M that belong to S where s may appear free in S and stands for M. In other words, M belongs to $\rho s.S$ when M belongs to $S[M/s]$. Formally, rds's adhere to the following introductory and eliminatory rules:

$$\frac{\Gamma \vdash M : S[M/s] \quad \Gamma \vdash \rho s.S\ sig}{\Gamma \vdash M : \rho s.S} \qquad \frac{\Gamma \vdash M : \rho s.S}{\Gamma \vdash M : S[M/s]}$$

As discussed previously, in the rds $\rho s.S$ we require that the static component of S be fully transparent, that is, that it completely specify the identity of its static component using singleton kinds. Thus, in order for an rds $\rho s.S$ to be well-formed, S must be fully transparent and well-formed under the assumption that s has signature S', where S' is obtained from S by stripping out the singleton kinds specifying the identity of the static component. Formally, rds's have the following formation rule:

$$\frac{\Gamma \vdash S\ sig \quad \Gamma[s:S] \vdash [\alpha{:}\mathcal{Q}(c:\kappa), \sigma]\ sig \quad \Gamma[s \uparrow S] \vdash c \downarrow \kappa}{\Gamma \vdash \rho s.[\alpha{:}\mathcal{Q}(c:\kappa), \sigma]\ sig}\ (S \equiv [\alpha{:}\kappa, \sigma[\alpha/Fst(s)]])$$

As with the recursive modules of Section 3, we wish to reduce recursively-dependent signatures to primitive constructs of the structure formalism. We do this by wrapping the compile-time component of the rds in a fixed point expression, and by redirecting recursive references in the run-time component:

$$\rho s.[\alpha{:}\mathcal{Q}(c(Fst(s)):\kappa), \sigma(\alpha, Fst(s))] = [\alpha{:}\mathcal{Q}((\underline{\mu\beta{:}\kappa.c(\beta)}):\kappa), \sigma(\alpha, \underline{\alpha})]$$

In the second underlined fragment, recursive references using $Fst(s)$ are redirected to use α. The interesting part is the first underlined fragment: Suppose $[c', e]$ is a prospective member of the rds. The rds dictates that $c' : \mathcal{Q}(c(c'):\kappa)$ and consequently that $c' = c(c') : \kappa$. Therefore, c' may be taken to be $\mu\beta{:}\kappa.c(\beta)$ as provided by the first underlined fragment.

This definition is formalized in the type theory by the equational rule in Figure 5. As in Section 3, this rule illustrates that recursively-dependent signatures are already present in the underlying calculus. In particular, the introductory and eliminatory rules given above follow from the definition and need not appear as primitive rules.

5 Future Work

Purely hierarchical module systems, such as the Standard ML module system, may be criticized on the grounds that they lack adequate support for cyclic dependencies among components. In such languages interdependent components must be consolidated into a single module, which can prevent decomposition of a system into "mind-sized" fragments. Several authors (including Duggan and Sourelis [4, 5] and Flatt and Felleisen [6]) have proposed module systems that better support such cyclic dependencies among units.

With at least two different proposals for recursive modules in hand, it is natural to ask "what is a recursive module?" We provide an answer to this question in the form of a type-theoretic analysis of recursive modules based on the "phase distinction" calculus of higher-order modules [7].

Specifically, we propose an extension of the phase distinction calculus with a new form of recursive module and a new form of signature, called a recursively-dependent signature. Following the paradigm of the phase distinction interpretation of higher-order modules, we demonstrate the sensibility of this extension by giving an interpretation of it into a pure calculus of structures (without explicit recursive module constructs). This interpretation demonstrates that in a precise sense, recursive modules are already present in the pure structure calculus. As in the case of higher-order modules, this is the key to implementing recursive modules in a type-passing compiler such as Shao's FLINT-based implementation of Standard ML [15] or Morrisett, *et al.'s* TIL compiler [18] — simply translate them into the pure structure formalism using the interpretation given here.

To make these ideas practical more work remains to be done. It is important to demonstrate that type-checking remains decidable in the presence of recursively-dependent signatures. The central issue for decidability is decidability of equivalence for equi-recursive constructors of higher kind. Amadio and Cardelli [1] provide an algorithm for checking equality of equi-recursive types; it is not clear at present whether their work extends to higher kinds. It is also important to consider a dynamic semantics for the extended language and to demonstrate the soundness of the type system for this dynamic semantics. We do not expect any difficulties with this extension.

A natural question is whether the reliance on equi-recursive constructors is essential for supporting recursive modules. (For example, Duggan and Sourelis's formalism does not rely on this form of recursive types.) We conjecture that it is not essential, based on the following observations. Under the standard type-theoretic interpretation of ML (see, for example, Harper and Mitchell [8]), the implementation of a recursive datatype is an iso-recursive type. (We will write iso-recursive types as $\mu_{\cong}\alpha.c$.) If we restrict recursive modules to datatypes (as in Duggan and Sourelis' formalism), and adopt the "transparent" interpretation outlined in Section 4, then equi-recursive types are completely eliminable by the translation into the underlying structure calculus, provided that we adopt *Shao's equation* for iso-recursive types:[3]

$$\mu_{\cong}\alpha.c(\alpha) = \mu_{\cong}\alpha.c(\mu_{\cong}\alpha.c(\alpha))$$

The relevance of Shao's equation to the elimination of equi-recursive types is based on the following observation. After translation into the pure structure calculus, datatypes in the body of a recursive module definition have implementation types of the form

$$\mu\alpha.\mu_{\cong}\beta.c(\alpha, \beta)$$

for some constructor c, where α results from recursion via recursive modules and β results from ordinary datatype recursion. By invoking a bisimilarity rule for equi-recursive types (Figure 6) and applying Shao's equation, as shown in Figure 7, we may prove that this type is equivalent to the type

$$\mu_{\cong}\beta.c(\beta, \beta)$$

which is a purely iso-recursive type. This observation sheds light on the nature of Duggan and Sourelis's restriction on the recursively defined type components of a mixin module to datatypes, which are implicitly iso-recursive. Strictly speaking, this restriction is not necessary, but if it were to be adopted, it would, by the observation above, allow the elimination of equi-recursive types from the internal language of a type-based compiler for ML.

[3] This equation was introduced by Shao [16] in his FLINT formalism in order to support the compilation of Standard ML. Shao observed that this equation is *essential* for efficiently compiling Standard ML, even in the absence of recursive modules.

$$\frac{\Gamma \vdash c = c'[c/\alpha] : \kappa \quad \Gamma[\alpha \uparrow \kappa] \vdash c' \downarrow \kappa}{\Gamma \vdash c = \mu\alpha{:}\kappa.c' : \kappa} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

Figure 6: Bisimilarity

$$\frac{\dfrac{\text{Shao's equation}}{\Gamma \vdash \mu_{\cong}\beta.c(\beta,\beta) = \mu_{\cong}\beta.\, c(\,\underline{\mu_{\cong}\beta.c(\beta,\beta)}\,,\beta)} \quad \dfrac{\mu_{\cong}\text{ contractiveness}}{\Gamma[\alpha \uparrow T] \vdash \mu_{\cong}\beta.c(\alpha,\beta)\downarrow}}{\Gamma \vdash \mu_{\cong}\beta.c(\beta,\beta) = \mu\alpha{:}T.\mu_{\cong}\beta.c(\alpha,\beta) : T} \; (\text{bisimilarity})$$

Figure 7: Elimination of Equi-Recursive Types

A Type Theory

A.1 Core Calculus

$\boxed{\Gamma \vdash \kappa \;\; kind}$

$$\frac{}{\Gamma \vdash T \;\; kind}$$

$$\frac{}{\Gamma \vdash 1 \;\; kind}$$

$$\frac{\Gamma \vdash c : Type}{\Gamma \vdash \mathcal{Q}(c) \;\; kind}$$

$$\frac{\Gamma \vdash \kappa_1 \;\; kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 \;\; kind}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \;\; kind} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \kappa_1 \;\; kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 \;\; kind}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \;\; kind} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash \kappa_1 = \kappa_2 \;\; kind}$

$$\frac{}{\Gamma \vdash T = T \;\; kind}$$

$$\frac{}{\Gamma \vdash 1 = 1 \;\; kind}$$

$$\frac{\Gamma \vdash c_1 = c_2 : Type}{\Gamma \vdash \mathcal{Q}(c_1) = \mathcal{Q}(c_2) \;\; kind}$$

$$\frac{\Gamma \vdash \kappa_1 = \kappa_1' \;\; kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 = \kappa_2' \;\; kind}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 = \Pi\alpha{:}\kappa_1'.\kappa_2' \;\; kind} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \kappa_1 = \kappa_1' \;\; kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 = \kappa_2' \;\; kind}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 = \Sigma\alpha{:}\kappa_1'.\kappa_2' \;\; kind} \; (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash \kappa_1 \leq \kappa_2 \;\; kind}$

$$\frac{}{\Gamma \vdash T \leq T \;\; kind}$$

$$\frac{}{\Gamma \vdash 1 \leq 1 \;\; kind}$$

$$\frac{\Gamma \vdash c_1 = c_2 : Type}{\Gamma \vdash \mathcal{Q}(c_1) \leq \mathcal{Q}(c_2)\ \ kind}$$

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash \mathcal{Q}(c) \leq T\ \ kind}$$

$$\frac{\Gamma \vdash \kappa_1' \leq \kappa_1\ \ kind \quad \Gamma[\alpha : \kappa_1'] \vdash \kappa_2 \leq \kappa_2'\ \ kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2\ \ kind}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2'\ \ kind}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \kappa_1 \leq \kappa_1'\ \ kind \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 \leq \kappa_2'\ \ kind \quad \Gamma[\alpha : \kappa_1'] \vdash \kappa_2'\ \ kind}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \leq \Sigma\alpha{:}\kappa_1'.\kappa_2'\ \ kind}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{}{\Gamma \vdash \alpha : \kappa}\ (\alpha : \kappa \in \Gamma)$$

$$\frac{}{\Gamma \vdash \alpha : \kappa}\ (\alpha \uparrow \kappa \in \Gamma)$$

$$\frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\Gamma \vdash \kappa_1\ \ kind \quad \Gamma[\alpha : \kappa_1] \vdash c : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c : \Pi\alpha{:}\kappa_1.\kappa_2}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c_1 : \Pi\alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1 c_2 : \kappa_2[c_2/\alpha]}$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_2 : \kappa_2[c_1/\alpha] \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2\ \ kind}{\Gamma \vdash \langle c_1, c_2 \rangle : \Sigma\alpha{:}\kappa_1.\kappa_2}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_1(c) : \kappa_1}$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_2(c) : \kappa_2[\pi_1(c)/\alpha]}$$

$$\frac{}{\Gamma \vdash 1 : T}$$

$$\frac{\Gamma \vdash c_1 : T \quad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \rightharpoonup c_2 : T}$$

$$\frac{\Gamma \vdash c_1 : T \quad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \times c_2 : T}$$

$$\frac{\Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \mu\alpha{:}\kappa.c : \kappa}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c : \kappa' \quad \Gamma \vdash \kappa' \leq \kappa\ \ kind}{\Gamma \vdash c : \kappa}$$

$$\frac{\Gamma \vdash \lambda\alpha{:}\kappa_1.c\,\alpha : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c : \Pi\alpha{:}\kappa_1.\kappa_2}\ (\alpha \text{ not free in } c)$$

$$\frac{\Gamma \vdash \langle \pi_1(c), \pi_2(c) \rangle : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$\boxed{\Gamma \vdash c \downarrow \kappa}$

$$\frac{}{\Gamma \vdash \alpha \downarrow \kappa} \ (\alpha : \kappa \in \Gamma)$$

$$\frac{}{\Gamma \vdash \star \downarrow 1}$$

$$\frac{\Gamma \vdash \kappa_1 \ \mathit{kind} \quad \Gamma[\alpha : \kappa_1] \vdash c \downarrow \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c \downarrow \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c_1 \downarrow \Pi\alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash c_2 \downarrow \kappa_1}{\Gamma \vdash c_1 c_2 \downarrow \kappa_2[c_1/\alpha]}$$

$$\frac{\Gamma \vdash c_1 \downarrow \kappa_1 \quad \Gamma \vdash c_2 \downarrow \kappa_2[c_1/\alpha]\Gamma[\alpha : \kappa_1] \vdash \kappa_2 \ \mathit{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle \downarrow \Sigma\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c \downarrow \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_1(c) \downarrow \kappa_1}$$

$$\frac{\Gamma \vdash c \downarrow \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_2(c) \downarrow \kappa_2[\pi_1(c)/\alpha]}$$

$$\frac{}{\Gamma \vdash 1 \downarrow T}$$

$$\frac{\Gamma \vdash c_1 : T \quad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \rightharpoonup c_2 \downarrow T}$$

$$\frac{\Gamma \vdash c_1 : T \quad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \times c_2 \downarrow T}$$

$$\frac{\Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \mu\alpha{:}\kappa.c \downarrow \kappa} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c \downarrow \kappa' \quad \Gamma \vdash \kappa' \leq \kappa \ \mathit{kind}}{\Gamma \vdash c \downarrow \kappa}$$

$$\frac{\Gamma \vdash c = c' : \kappa \quad \Gamma \vdash c' \downarrow \kappa}{\Gamma \vdash c \downarrow \kappa}$$

$$\frac{\Gamma \vdash \lambda\alpha{:}\kappa_1.c\,\alpha \downarrow \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c \downarrow \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \text{ not free in } c)$$

$$\frac{\Gamma \vdash \langle \pi_1(c), \pi_2(c) \rangle \downarrow \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c \downarrow \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$\boxed{\Gamma \vdash c_1 = c_2 : \kappa}$

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c = c : \kappa}$$

$$\frac{\Gamma \vdash c_2 = c_1 : \kappa}{\Gamma \vdash c_1 = c_2 : \kappa}$$

$$\frac{\Gamma \vdash c_1 = c_2 : \kappa \quad \Gamma \vdash c_2 = c_3 : \kappa}{\Gamma \vdash c_1 = c_3 : \kappa}$$

$$\frac{\Gamma \vdash \kappa_1 = \kappa_1' \ \mathit{kind} \quad \Gamma[\alpha : \kappa_1] \vdash c = c' : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c = \lambda\alpha{:}\kappa_1'.c' : \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c_1 = c_1' : \Pi\alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash c_2 = c_2' : \kappa_1}{\Gamma \vdash c_1 c_2 = c_1' c_2' : \kappa_2[c_2/\alpha]}$$

$$\frac{\Gamma \vdash c_1 = c_1' : \kappa_1 \quad \Gamma \vdash c_2 = c_2' : \kappa_2[c_1/\alpha] \quad \Gamma[\alpha : \kappa_1] \vdash \kappa_2 \ \mathit{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle = \langle c_1', c_2' \rangle : \Sigma\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c = c' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_1(c) = \pi_1(c') : \kappa_1}$$

$$\frac{\Gamma \vdash c = c' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \pi_2(c) = \pi_2(c') : \kappa_2[\pi_1(c)/\alpha]}$$

$$\frac{\Gamma \vdash c_1 = c_1' : T \quad \Gamma \vdash c_2 = c_2' : T}{\Gamma \vdash c_1 \rightharpoonup c_2 = c_1' \rightharpoonup c_2' : T}$$

$$\frac{\Gamma \vdash c_1 = c_1' : T \quad \Gamma \vdash c_2 = c_2' : T}{\Gamma \vdash c_1 \times c_2 = c_1' \times c_2' : T}$$

$$\frac{\Gamma \vdash \kappa = \kappa' \quad \Gamma[\alpha : \kappa] \vdash c = c' : \kappa \quad \Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \mu\alpha{:}\kappa.c = \mu\alpha{:}\kappa'.c' : \kappa} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c_1 = c_2 : \kappa' \quad \Gamma \vdash \kappa' \leq \kappa \ \mathit{kind}}{\Gamma \vdash c_1 = c_2 : \kappa}$$

$$\frac{\Gamma \vdash c : \mathcal{Q}(c')}{\Gamma \vdash c = c' : T}$$

$$\frac{\Gamma \vdash c : 1}{\Gamma \vdash c = \star : 1}$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma[\alpha : \kappa_1] \vdash c_2 : \kappa_2}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.c_2)c_1 = c_2[c_1/\alpha] : \kappa_2[c_1/\alpha]} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.c\,\alpha) = c : \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \text{ not free in } c)$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_2 : \kappa_2}{\Gamma \vdash \pi_i(\langle c_1, c_2 \rangle) = c_i : \kappa_i} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \langle \pi_1(c), \pi_2(c) \rangle = c : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \mu\alpha{:}\kappa.c = c[(\mu\alpha{:}\kappa.c)/\alpha] : \kappa}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash c = c'[c/\alpha] : \kappa \quad \Gamma[\alpha \uparrow \kappa] \vdash c' \downarrow \kappa}{\Gamma \vdash c = \mu\alpha{:}\kappa.c' : \kappa}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash \sigma\ \ type}$

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash c\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1\ \ type \quad \Gamma \vdash \sigma_2\ \ type}{\Gamma \vdash \sigma_1 \to \sigma_2\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1\ \ type \quad \Gamma \vdash \sigma_2\ \ type}{\Gamma \vdash \sigma_1 \rightharpoonup \sigma_2\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1\ \ type \quad \Gamma \vdash \sigma_2\ \ type}{\Gamma \vdash \sigma_1 \times \sigma_2\ \ type}$$

$$\frac{\Gamma[\alpha : \kappa] \vdash \sigma\ \ type}{\Gamma \vdash \forall\alpha{:}\kappa.\sigma\ \ type}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash \sigma_1 = \sigma_2\ \ type}$

$$\frac{\Gamma \vdash \sigma\ \ type}{\Gamma \vdash \sigma = \sigma\ \ type}$$

$$\frac{\Gamma \vdash \sigma_2 = \sigma_1\ \ type}{\Gamma \vdash \sigma_1 = \sigma_2\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_2\ \ type \quad \Gamma \vdash \sigma_2 = \sigma_3\ \ type}{\Gamma \vdash \sigma_1 = \sigma_3\ \ type}$$

$$\frac{\Gamma \vdash c = c' : T}{\Gamma \vdash c = c'\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1'\ \ type \quad \Gamma \vdash \sigma_2 = \sigma_2'\ \ type}{\Gamma \vdash \sigma_1 \to \sigma_2 = \sigma_1' \to \sigma_2'\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1'\ \ type \quad \Gamma \vdash \sigma_2 = \sigma_2'\ \ type}{\Gamma \vdash \sigma_1 \rightharpoonup \sigma_2 = \sigma_1' \rightharpoonup \sigma_2'\ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1'\ \ type \quad \Gamma \vdash \sigma_2 = \sigma_2'\ \ type}{\Gamma \vdash \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'\ \ type}$$

$$\frac{\Gamma \vdash \kappa = \kappa' \quad \Gamma[\alpha : \kappa] \vdash \sigma = \sigma'\ \ type}{\Gamma \vdash \forall\alpha{:}\kappa.\sigma = \forall\alpha{:}\kappa'.\sigma'\ \ type}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{}{\Gamma \vdash x : \sigma}\ (x : \sigma \in \Gamma)$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (x \uparrow \sigma \in \Gamma)$$

$$\frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type} \quad \Gamma[x : \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type} \quad \Gamma[x : \sigma] \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\sigma.e : \sigma \rightharpoonup \sigma'} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \sigma'}$$

$$\frac{\Gamma \vdash e_1 : \sigma \rightharpoonup \sigma' \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \sigma'}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_i(e) : \sigma_i} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash \kappa \ \ \mathit{kind} \quad \Gamma[\alpha : \kappa] \vdash e \downarrow \sigma}{\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\sigma} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e : \forall\alpha{:}\kappa.\sigma \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c] : \sigma[c/\alpha]}$$

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type} \quad \Gamma[x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(x{:}\sigma.e) : \sigma} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e : \sigma' \quad \Gamma \vdash \sigma = \sigma' \ \ \mathit{type}}{\Gamma \vdash e : \sigma}$$

$\boxed{\Gamma \vdash e \downarrow \sigma}$

$$\frac{}{\Gamma \vdash x \downarrow \sigma} \ (x : \sigma \in \Gamma)$$

$$\frac{}{\Gamma \vdash \star \downarrow 1}$$

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type} \quad \Gamma[x : \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \lambda x{:}\sigma.e \downarrow \sigma \to \sigma'} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type} \quad \Gamma[x : \sigma] \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\sigma.e \downarrow \sigma \rightharpoonup \sigma'} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 \downarrow \sigma \to \sigma' \quad \Gamma \vdash e_2 \downarrow \sigma}{\Gamma \vdash e_1 e_2 \downarrow \sigma'}$$

$$\frac{\Gamma \vdash e_1 \downarrow \sigma_1 \quad \Gamma \vdash e_2 \downarrow \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle \downarrow \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash e \downarrow \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_i(e) \downarrow \sigma_i}\ (i = 1, 2)$$

$$\frac{\Gamma \vdash \kappa\ \mathit{kind} \quad \Gamma[\alpha : \kappa] \vdash e \downarrow \sigma}{\Gamma \vdash \Lambda\alpha{:}\kappa.e \downarrow \forall\alpha{:}\kappa.\sigma}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e \downarrow \forall\alpha{:}\kappa.\sigma \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c] \downarrow \sigma[c/\alpha]}$$

$$\frac{\Gamma \vdash \sigma\ \mathit{type} \quad \Gamma[x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(x{:}\sigma.e) \downarrow \sigma}\ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e \downarrow \sigma' \quad \Gamma \vdash \sigma = \sigma'\ \mathit{type}}{\Gamma \vdash e \downarrow \sigma}$$

$$\frac{\Gamma \vdash e' \downarrow \sigma \quad \Gamma \vdash e = e' : \sigma}{\Gamma \vdash e \downarrow \sigma}$$

$\boxed{\Gamma \vdash e = e' : \sigma}$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e = e : \sigma}$$

$$\frac{\Gamma \vdash e_2 = e_1 : \sigma}{\Gamma \vdash e_1 = e_2 : \sigma}$$

$$\frac{\Gamma \vdash e_1 = e_2 : \sigma \quad \Gamma \vdash e_2 = e_3 : \sigma}{\Gamma \vdash e_1 = e_3 : \sigma}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_2\ \mathit{type} \quad \Gamma[x : \sigma_1] \vdash e = e' : \sigma_2 \quad \Gamma[x : \sigma_1] \vdash e \downarrow \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_1.e = \lambda x{:}\sigma_1'.e' : \sigma_1 \to \sigma_2}\ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_2\ \mathit{type} \quad \Gamma[x : \sigma_1] \vdash e = e' : \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_1.e = \lambda x{:}\sigma_1'.e' : \sigma_1 \rightharpoonup \sigma_2}\ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 = e_1' : \sigma \to \sigma' \quad \Gamma \vdash e_2 = e_2' : \sigma}{\Gamma \vdash e_1 e_2 = e_1' e_2' : \sigma'}$$

$$\frac{\Gamma \vdash e_1 = e_1' : \sigma \rightharpoonup \sigma' \quad \Gamma \vdash e_2 = e_2' : \sigma}{\Gamma \vdash e_1 e_2 = e_1' e_2' : \sigma'}$$

$$\frac{\Gamma \vdash e_1 = e_1' : \sigma_1 \quad \Gamma \vdash e_2 = e_2' : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle = \langle e_1', e_2' \rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash e = e' : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_i(e) = \pi_i(e') : \sigma_i}\ (i = 1, 2)$$

$$\frac{\Gamma \vdash \kappa = \kappa'\ \mathit{kind} \quad \Gamma[\alpha : \kappa] \vdash e = e' : \sigma \quad \Gamma[\alpha : \kappa] \vdash e \downarrow \sigma}{\Gamma \vdash \Lambda\alpha{:}\kappa.e = \Lambda\alpha{:}\kappa'.e' : \forall\alpha{:}\kappa.\sigma}\ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e = e' : \forall \alpha{:}\kappa.\sigma \quad \Gamma \vdash c = c' : \kappa}{\Gamma \vdash e[c] = e'[c'] : \sigma[c/\alpha]}$$

$$\frac{\Gamma \vdash \sigma = \sigma' \ \ type \quad \Gamma[x \uparrow \sigma] \vdash e = e' : \sigma \quad \Gamma[x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(x{:}\sigma.e) = \mathit{fix}(x{:}\sigma'.e') : \sigma} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e = e' : \sigma' \quad \Gamma \vdash \sigma = \sigma' \ \ type}{\Gamma \vdash e = e' : \sigma}$$

$$\frac{\Gamma \vdash e \downarrow 1}{\Gamma \vdash e = \star : 1}$$

$$\frac{\Gamma \vdash e_1 \downarrow \sigma_1 \quad \Gamma[x : \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash (\lambda x{:}\sigma_1.e_2)e_1 = e_2[e_1/x] : \sigma_2} \ (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e \downarrow \sigma_1 \to \sigma_2}{\Gamma \vdash (\lambda x{:}\sigma_1.e\, x) = e : \sigma_1 \to \sigma_2} \ (x \text{ not free in } e)$$

$$\frac{\Gamma \vdash e \downarrow \sigma_1 \rightharpoonup \sigma_2}{\Gamma \vdash (\lambda x{:}\sigma_1.e\, x) = e : \sigma_1 \rightharpoonup \sigma_2} \ (x \text{ not free in } e)$$

$$\frac{\Gamma \vdash e_1 \downarrow \sigma_1 \quad \Gamma \vdash e_2 \downarrow \sigma_2}{\Gamma \vdash \pi_i(\langle e_1, e_2 \rangle) = e_i : \sigma_i} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \langle \pi_1(e), \pi_2(e) \rangle = e : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash c : \kappa \quad \Gamma[\alpha : \kappa] \vdash e : \sigma}{\Gamma \vdash (\Lambda \alpha{:}\kappa.e)[c] = e[c/\alpha] : \sigma[c/\alpha]} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e \downarrow \forall \alpha{:}\kappa.\sigma}{\Gamma \vdash (\Lambda \alpha{:}\kappa.e[\alpha]) = e : \forall \alpha{:}\kappa.\sigma} \ (\alpha \text{ not free in } e)$$

$$\frac{\Gamma \vdash \sigma \ \ type \quad \Gamma[x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(x{:}\sigma.e) = e[\mathit{fix}(x{:}\sigma.e)/x] : \sigma} \ (x \notin \mathrm{Dom}(\Gamma))$$

A.2 Structure Calculus

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{}{\Gamma \vdash \mathit{Fst}(s) : \kappa} \ (s : [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$$\frac{}{\Gamma \vdash \mathit{Fst}(s) : \kappa} \ (s \uparrow [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$\boxed{\Gamma \vdash c \downarrow \kappa}$

$$\frac{}{\Gamma \vdash \mathit{Fst}(s) \downarrow \kappa} \ (s : [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{}{\Gamma \vdash \mathit{snd}(s) : \sigma[\mathit{Fst}(s)/\alpha]} \ (s : [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$$\frac{}{\Gamma \vdash snd(s) : \sigma[Fst(s)/\alpha]} \quad (s \uparrow [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$\boxed{\Gamma \vdash e \downarrow \sigma}$

$$\frac{}{\Gamma \vdash snd(s) \downarrow \sigma[Fst(s)/\alpha]} \quad (s : [\alpha{:}\kappa.\sigma] \in \Gamma)$$

$\boxed{\Gamma \vdash S \;\; sig}$

$$\frac{\Gamma \vdash \kappa \;\; kind \quad \Gamma[\alpha : \kappa] \vdash \sigma \;\; type}{\Gamma \vdash [\alpha{:}\kappa.\sigma] \;\; sig} \quad (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash S_1 = S_2 \;\; sig}$

$$\frac{\Gamma \vdash S \;\; sig}{\Gamma \vdash S = S \;\; sig}$$

$$\frac{\Gamma \vdash S_2 = S_1 \;\; sig}{\Gamma \vdash S_1 = S_2 \;\; sig}$$

$$\frac{\Gamma \vdash S_1 = S_2 \;\; sig \quad \Gamma \vdash S_2 = S_3 \;\; sig}{\Gamma \vdash S_1 = S_3 \;\; sig}$$

$$\frac{\Gamma \vdash \kappa = \kappa' \;\; kind \quad \Gamma[\alpha : \kappa] \vdash \sigma = \sigma' \;\; type}{\Gamma \vdash [\alpha{:}\kappa.\sigma] = [\alpha{:}\kappa'.\sigma'] \;\; sig}$$

$\boxed{\Gamma \vdash S_1 \leq S_2 \;\; sig}$

$$\frac{\Gamma \vdash S_1 = S_2 \;\; sig}{\Gamma \vdash S_1 \leq S_2 \;\; sig}$$

$$\frac{\Gamma \vdash S_1 \leq S_2 \;\; sig \quad \Gamma \vdash S_2 \leq S_3 \;\; sig}{\Gamma \vdash S_1 \leq S_3 \;\; sig}$$

$$\frac{\Gamma \vdash \kappa \leq \kappa' \;\; kind \quad \Gamma[\alpha : \kappa] \vdash \sigma = \sigma' \;\; type \quad \Gamma[\alpha : \kappa'] \vdash \sigma' \;\; type}{\Gamma \vdash [\alpha{:}\kappa.\sigma] \leq [\alpha{:}\kappa'.\sigma'] \;\; sig} \quad (\alpha \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash M : S}$

$$\frac{\Gamma \vdash c : \kappa \quad \Gamma \vdash e : \sigma[c/\alpha] \quad \Gamma[\alpha : \kappa] \vdash \sigma \;\; type}{\Gamma \vdash [c, e] : [\alpha{:}\kappa.\sigma]} \quad (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash M : S' \quad \Gamma \vdash S \leq S' \;\; sig}{\Gamma \vdash M : S}$$

$\boxed{\Gamma \vdash M \downarrow S}$

$$\frac{\Gamma \vdash c \downarrow \kappa \quad \Gamma \vdash e \downarrow \sigma[c/\alpha] \quad \Gamma[\alpha : \kappa] \vdash \sigma \;\; type}{\Gamma \vdash [c, e] \downarrow [\alpha{:}\kappa.\sigma]} \quad (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash M \downarrow S' \quad \Gamma \vdash S \leq S' \;\; sig}{\Gamma \vdash M \downarrow S}$$

$$\frac{\Gamma \vdash M' \downarrow S \quad \Gamma \vdash M = M' : S}{\Gamma \vdash M \downarrow S}$$

$\boxed{\Gamma \vdash M_1 = M_2 : S}$

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash M = M : S}$$

$$\frac{\Gamma \vdash M_2 = M_1 : S}{\Gamma \vdash M_1 = M_2 : S}$$

$$\frac{\Gamma \vdash M_1 = M_2 : S \quad \Gamma \vdash M_2 = M_3 : S}{\Gamma \vdash M_1 = M_3 : S}$$

$$\frac{\Gamma \vdash c = c' : \kappa \quad \Gamma \vdash e = e' : \sigma[c/\alpha] \quad \Gamma[\alpha : \kappa] \vdash \sigma \ \mathit{type}}{\Gamma \vdash [c, e] = [c', e'] : [\alpha{:}\kappa.\sigma]} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash M_1 = M_2 : S' \quad \Gamma \vdash S \leq S' \ \mathit{sig}}{\Gamma \vdash M_1 = M_2 : S}$$

A.3 Recursive Module Calculus

$\boxed{\Gamma \vdash M : S}$

$$\frac{\Gamma[s \uparrow S] \vdash M \downarrow S \quad \Gamma \vdash S \ \mathit{sig}}{\Gamma \vdash \mathit{fix}(s{:}S.M) : S} \ (s \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash M_1 = M_2 : S}$

$$\frac{\Gamma \vdash \kappa \ \mathit{kind} \quad \Gamma[\alpha : \kappa] \vdash \sigma \ \mathit{type} \quad \Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa \quad \Gamma[\alpha : \kappa][x \uparrow \sigma] \vdash e \downarrow \sigma}{\Gamma \vdash \mathit{fix}(s{:}[\alpha{:}\kappa.\sigma].[c[\mathit{Fst}(s)/\alpha], e[\mathit{Fst}(s), \mathit{snd}(s)/\alpha, x]]) = [\alpha = \mu\alpha{:}\kappa.c, \mathit{fix}(x{:}\sigma.e)] : [\alpha{:}\kappa.\sigma]} \ (\alpha, x \notin \mathrm{Dom}(\Gamma))$$

$\boxed{\Gamma \vdash S \ \mathit{sig}}$

$$\frac{\Gamma \vdash S \ \mathit{sig} \quad \Gamma[s : S] \vdash [\alpha{:}\mathcal{Q}(c : \kappa), \sigma] \ \mathit{sig}}{\Gamma \vdash \rho s.[\alpha{:}\mathcal{Q}(c : \kappa), \sigma] \ \mathit{sig}} \ (S \equiv [\alpha{:}\kappa, \sigma[\alpha/\mathit{Fst}(s)]])$$

$\boxed{\Gamma \vdash S_1 = S_2 \ \mathit{sig}}$

$$\frac{\Gamma \vdash \kappa \ \mathit{kind} \quad \Gamma[\alpha : \kappa] \vdash \sigma[\alpha/\mathit{Fst}(s)] \ \mathit{type} \quad \Gamma[\beta : \kappa] \vdash \mathcal{Q}(c : \kappa) \ \mathit{kind} \quad \Gamma[\beta \uparrow \kappa] \vdash c \downarrow \kappa}{\rho s.[\alpha{:}\mathcal{Q}(c[\mathit{Fst}(s)/\beta] : \kappa), \sigma] = [\alpha{:}\mathcal{Q}((\mu\beta{:}\kappa.c) : \kappa), \sigma[\alpha/\mathit{Fst}(s)]]} \ (\alpha \notin \mathrm{Dom}(\Gamma))$$

References

[1] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.

[2] Davide Ancona and Elena Zucca. An algebra of mixin modules. In F. Parisi-Presicce, editor, *WADT '97 12th Workshop on Algebraic Development Techniques – Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 92–106, Berlin, 1997. Springer Verlag.

[3] Luca Cardelli. Phase distinctions in type theory. unpublished manuscript.

[4] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM International Conference on Functional Programming*, pages 262–273, Philadelphia, PA, June 1996.

[5] Dominic Duggan and Constantinos Sourelis. Parameterized modules, recursive modules, and mixin modules. In *1998 ACM SIGPLAN Workshop on ML*, pages 87–96, Baltimore, Maryland, September 1998. ACM SIGPLAN.

[6] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, June 1998.

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

[8] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. (See also [14].).

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

[10] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrifft.* MIT Press, 1998. (To appear).

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

[12] Xavier Leroy. The Objective Caml system: Documentation and user's guide. Available at `http://pauillac.inria.fr/ocaml/htmlman/`., 1996.

[13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

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

[15] Zhong Shao. An overview of the FLINT/ML compiler. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation*, Kyoto, Japan, June 1997.

[16] Zhong Shao. Equality of recursive types. (Private communication), September 1998.

[17] Emin Gün Sirer, Marc E. Fiucynski, Przemysław Pardyak, and Brian N. Bershad. Safe dynamic linking in an extensible operating system. In *Workshop on Compiler Support for System Software*, Tucson, AZ, February 1996.

[18] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.