# Toward a Practical Type Theory for Recursive Modules


Derek R. Dreyer Robert Harper Karl Crary



March 2001

CMU-CS-01-112



School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213



**Abstract**

Module systems for languages with complex type systems, such as Standard ML, often lack the ability to express mutually recursive type and function dependencies across module boundaries. Previous work by Crary, Harper and Puri [5] set out a type-theoretic foundation for recursive modules in the context of a phase-distinction calculus for higher-order modules. Two constructs were introduced for encoding recursive modules: a *fixed-point module* and a *recursively dependent signature*. Unfortunately, the implementations of both constructs involve the use of equi-recursive type constructors at higher-order kinds, the equivalence of which is not known to be decidable.

In this paper, we show that the practicality of recursive modules is not contingent upon that of equi-recursive constructors. We begin with the theoretical infrastructure described above and study precisely how equi-recursiveness is used in the recursive module constructs, resulting in a clarification and generalization of the underlying ideas. We then examine in depth how the recursive module constructs in the revised type system can serve as the target of elaboration for a recursive module extension to Standard ML.



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

Modular programming is critical to the development of large-scale software systems. By enabling programs to be broken into relatively independent components that interact with one another through statically-defined interfaces, modularization provides a way for multiple programmers to work on a project simultaneously and coherently. The power of a module system lies in the flexibility of its facility for expressing dependencies between modular components. Some languages (such as Java, Modula-3 and Ada95) only allow inter-module dependencies to be hardwired. Yet frequently in such languages those dependencies are permitted to be mutually recursive. The module system of Standard ML [16] is much more expressive. Its *functor* construct encourages generic programming in the style of Modula's *generics* or C++'s *templates*, but also allows a modular component of a large program to be written and compiled separately by abstracting the component over its import dependencies. However, Standard ML does not support any kind of mutually recursive type or function definitions across module boundaries. In other words, dependencies between Standard ML modules, separately compiled or not, must be strictly hierarchical.

A mechanism for writing recursive modules is one of the most requested extensions for Standard ML. One obvious reason is that the need to write mutually recursive datatypes and functions in inseparable bundles inhibits modularity. Yet even forgetting the goal of separate compilation, recursive modules seem to the typical ML programmer like a natural generalization of recursive constructs in the core language. Consider a simple syntax. If one can write mutually recursive datatypes separated by the keyword "`and`" and mutually recursive functions separated by the keyword "`and`", why can't one write mutually recursive modules separated by the keyword "`and`"? (*Note:* ML actually has an "`and`" syntax for *simultaneous* module declarations, but such declarations are not mutually recursive.) Consider a simple semantics. Suppose that these mutually recursive modules are restricted to contain only datatypes and functions. A mutually recursive bundle of modules could be compiled into a single non-recursive module by joining the datatypes in all the modules into one recursive bundle of datatypes and joining the functions in all the modules into one recursive bundle of functions.

The simplicity of this "naive" programmer's point of view is a dual-edged sword. To the compiler hacker, it gives the impression that recursive modules, perhaps in a limited form, would be an easy extension to design and implement. To the type theorist, it makes them appear like so much *ad hoc* syntactic sugar. From either angle, it does not suggest a fruitful line of research, which is unfortunate considering the perennial demand for recursive modules.

## 1.1 Workarounds

In the absence of recursive modules, there are two major techniques that programmers typically employ to work around them. The first, more primitive, technique is based on the philosophy that mutually recursive types and terms ought to be written together to begin with and "modularized after the fact" by copying them into separate modules. For example, if we wanted to separate two mutually recursive functions `f` and `g` into separate modules `A` and `B`, we would have to first define them together in one module with a `fun` declaration, then copy the functions into `A` and `B` as follows:

```
structure AandB = struct
  fun f x = ... g(...) ...
  and g y = ... f(...) ...
end
```

```
structure A = struct val f = AandB.f end
structure B = struct val g = AandB.g end
```

The same technique can be applied to datatypes by means of ML's "datatype copying" mechanism.

While this first workaround is certainly straightforward, it is unsatisfying for two reasons. First, it is no more nor less than a namespace management trick and does not address the problem of modularizing the actual recursive code. Second, if the module is large, as is typical for the kinds of modules one would like to use recursive modules to break up, copying each type or value declaration into a separate module after the fact can be quite cumbersome.

The second common approach, known often as the "parameterization" technique, allows for real separate compilation of mutually recursive datatypes and functions, but is in turn more complex. There are a number of variations on this approach, but the basic idea is to define the recursive modules hierarchically so that they can be written separately. In order to do that, the functions in the first module in the hierarchy, which is defined in the absence of the others, must be parameterized over the functions they refer to from the other modules. When referring to the first module, the functions in the other modules must supply themselves (recursively) as arguments to instantiate the import dependencies of the functions in the first module. The first module can then be defined "for real" by applying its original parameterized functions to the functions defined in the other modules, thereby tying the recursive knot manually. Continuing the simple example from the first technique, we could do the following:

```
structure preA =
  struct fun f g x = ... g(...) ... end
structure B =
  struct fun g y = let val f = preA.f g in ... f(...) ... end
structure A =
  struct val f = preA.f B.g end
```

A similar technique can be applied to datatypes by means of polymorphic types (which are really $\lambda$-expressions at the type level).

While this kind of approach is undeniably effective at breaking up recursive code, it also forces considerable changes at the level of individual functions and types, not to mention the overall program structure. Every function in `A` that needs to refer to `B` must explicitly convert its import dependencies (such as `B.g` above) into function arguments. Code transformations like this that affect all the code in a module are what modules were invented to simplify. For instance, it is not necessary to program with functors in ML, but they simplify and clarify programming tremendously in the situation where all the code in a module is parameterized over the code in another module. Unfortunately, there is no way to generalize the recursive parameterization trick to the module level, e.g.:

```
functor preA (B : BSIG) =
  struct fun f x = ... B.g(...) ... end
structure B =
  let structure A = preA(B) in     (* Type Error: B not defined here *)
    struct fun g y = ... A.f(...) ... end
```

In addition, the complexity of the parameterization technique introduces inefficiency due to function applications that is avoided by simply writing the functions in all the modules together under one fixed-point.

## 1.2 Type Theory for Recursive Modules

In response to the inadequacies of the recursive module workarounds, there have been a few attempts to extend ML-like languages with real recursive modules, most notably by Flatt and Felleisen [8] and Duggan and Sourelis [6, 7]. However, the former's extension is bound up with dynamic linking and first-class modules in the concept of a *unit*. The latter's extension is tied to *mixin* modules, which can be used to achieve the effect of virtual types in a functional language. Both extensions are interesting (and we will return to them in Section 6 on related work), but neither explains what a recursive module *is*, in a fundamental sense independent of the specific language extension in which it appears.

Toward this end, Crary, Harper and Puri [5] (hereafter, CHP) have given an analysis of recursive modules in the setting of type theory, specifically in a *phase-distinction* calculus in the style of Harper, Mitchell and Moggi [9] (hereafter, HMM). The main reason for working in such a calculus is to show how module constructs can be split into separate compile-time (type-level) and run-time (term-level) core constructs. This ability to "phase-split" a module is important for several reasons. Modern type-directed compilers track type information throughout compilation to enable certain type-based optimizations. Phase-splitting enables them to compile module code into core language code without discarding the type part of the module. In addition, working in the style of HMM provides compatibility with fully transparent higher-order modules, including functors and nested structures. Although higher-order modules are not the concern of this paper, we will see in Section 5.2 that the ability to phase-split functors and functor applications leads to increased expressiveness for recursive modules.

As the recursive core constructs at the type and term levels of the core calculus take the form of a fixed-point, the analogous module construct that CHP study is a *fixed-point* module. This fixed-point module has a straightforward typechecking rule, and an intuitive phase-splitting interpretation that separates the module into a fixed-point type constructor and a fixed-point term. However, simple examples exhibit serious limitations in the new module construct. To remedy the problem, CHP introduce a new signature construct called a *recursively dependent signature*, or *rds*, that more accurately models types inside a recursive module. Rds's are required by their phase-splitting interpretation to be fully transparent, and hence the combination of fixed-point modules with rds's is termed "transparent recursive module" programming, in contrast to the "opaque" programming with the fixed-point construct alone.

While transparent recursive modules constitute a useful and expressive extension to the module system, there is a serious problem with the entire analysis. The phase-splitting rules for both fixed-point modules and rds's depend on the assumption that the recursive core construct at the type level is *equi-recursive*, i.e. a fixed-point over type constructors that is *equal* to its unrolling, not merely isomorphic. This is an odd assumption since recursive datatypes in Standard ML are *iso-recursive*, requiring explicit rolls (and unrolls) into (and out of) the fixed-point. Moreover, checking type equivalence in the presence of equi-recursive type constructors of higher kind is not known to be decidable, thus casting doubt on the practicality of the whole type system.

## 1.3 Overview

In this paper, we will begin with the rigorous analysis and technical infrastructure afforded by CHP and show what is needed to make recursive modules useful and practical. Our approach to the theory involves determining which constructs are fundamentally equi-recursive and which are not, with the ulterior goal of developing a practical design for recursive modules that depends only on iso-recursive type constructors.

| | | | |
|---|---|---|---|
| *kinds* | $\kappa$ | ::= | $T \mid 1 \mid \mathfrak{S}(c) \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \Sigma\alpha{:}\kappa_1.\kappa_2$ |
| *constructors* | $c$ | ::= | $\alpha \mid \star \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \langle c_1, c_2\rangle \mid c.1 \mid c.2 \mid 1 \mid c_1 \rightharpoonup c_2 \mid c_1 \times c_2 \mid \mu_{\equiv}\alpha{:}\kappa.c$ |
| *types* | $\sigma$ | ::= | $T(c) \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \rightharpoonup \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall\alpha{:}\kappa.\sigma$ |
| *terms* | $e$ | ::= | $x \mid \star \mid \lambda x{:}\sigma.e \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \mathit{fix}(x{:}\sigma.e)$ |
| *contexts* | $\Gamma$ | ::= | $\epsilon \mid \Gamma[\alpha : \kappa] \mid \Gamma[x : \sigma] \mid \Gamma[\alpha \uparrow \kappa] \mid \Gamma[x \uparrow \sigma]$ |

Figure 1: The Core Calculus

First, in Section 2, we review the definitions and motivations for the fixed-point and rds constructs introduced by CHP, as well as the type-theoretic framework in which they are situated. Then, in Section 3, we split fixed-points and rds's into opaque and transparent forms in an attempt to isolate the uses of equi-recursive constructors and to clarify the similarities and differences between opaque and transparent recursive modules. In Section 4, we examine how well the theoretical constructs defined in Section 3 fare in practice, culminating in a proposal for a modest recursive module extension to Standard ML. Finally, Section 5 explores some directions for future work related to the interaction of recursive modules and abstraction, and Section 6 provides a brief survey of related work.

# 2 Summary of CHP Analysis

In this section, we will briefly summarize the type-theoretic framework of CHP, the new constructs they introduce and the problems that arise.

## 2.1 Framework

The analysis is conducted in a phase-distinction calculus, consisting of a core calculus and a primitive structure calculus. The core calculus (Figure 1) is a fairly standard higher-order predicative polymorphic lambda calculus with a few extensions, namely dependent and singleton kinds, equi-recursive type constructors and fixed-point expressions.

Dependent and singleton kinds have proven to be a convenient and expressive mechanism for fine-grained control of type information in module signatures. Specifically, the kind $\mathfrak{S}(c)$ classifies all types (actually, type constructors of base kind $T$) that are equivalent to $c$. The kind $\Sigma\alpha{:}\kappa_1.\kappa_2$ is used to classify pairs of constructors where the kind of the second constructor may refer to the name of the first constructor. The kind $\Pi\alpha{:}\kappa_1.\kappa_2$ is used to classify constructor functions where the result kind may refer to the name of the argument. To illustrate all three, the following faux-ML functor signature

```
(X : sig type t end) -> sig type 'a u; type v = X.t * int u end
```

could be translated into the kind

$$\Pi\alpha{:}T.\,\Sigma\beta{:}T{\to}T.\,\mathfrak{S}(\alpha \times \beta(\mathit{int}))$$

As shown in Figure 2, one can define singletons at higher kind, written $\mathfrak{S}(c : \kappa)$, inductively in terms of singletons at kind $T$. The thesis work of Chris Stone [26] explores the issue of singleton kinds in great depth, including a detailed proof that type equivalence remains decidable in a language containing them. Dependent and singleton kinds are currently implemented in the MIL intermediate

$$
\begin{aligned}
\mathfrak{S}(c:T) &\stackrel{\text{def}}{=} \mathfrak{S}(c)\\
\mathfrak{S}(c:\mathfrak{S}(c')) &\stackrel{\text{def}}{=} \mathfrak{S}(c)\\
\mathfrak{S}(c:\Pi\alpha{:}\kappa_1.\kappa_2) &\stackrel{\text{def}}{=} \Pi\alpha{:}\kappa_1.\,\mathfrak{S}(c\,\alpha:\kappa_2)\\
\mathfrak{S}(c:1) &\stackrel{\text{def}}{=} 1\\
\mathfrak{S}(c:\Sigma\alpha{:}\kappa_1.\kappa_2) &\stackrel{\text{def}}{=} \mathfrak{S}(c.1:\kappa_1)\times\mathfrak{S}(c.2:\kappa_2[c.1/\alpha])
\end{aligned}
$$

Figure 2: Higher-Order Singletons

language of the TILT compiler for Standard ML [19]. See Stone and Harper [25] for a more concise account of the motivation and practicality of singleton kinds.

The recursive operators at the constructor and term levels take the form $\mu_{\equiv}\alpha{:}\kappa.c$ and $\mathit{fix}(x{:}\sigma.e)$, respectively. The type constructor $\mu_{\equiv}\alpha{:}\kappa.c$ denotes the unique fixed-point of the constructor function $\lambda\alpha{:}\kappa.c$ and is subject to the following well-formedness rule:

$$
\frac{\alpha\notin\mathrm{Dom}(\Gamma)\qquad \Gamma\vdash\kappa\ \mathit{kind}\qquad \Gamma[\alpha\uparrow\kappa]\vdash c\downarrow\kappa}{\Gamma\vdash\mu_{\equiv}\alpha{:}\kappa.c:\kappa}
$$

The second premise checks that $c$ has the right kind and that it is *contractive* in a context where $\alpha$ is not. In the context of this rule, this means that all references in $c$ to $\alpha$ must occur under a type construction operation like $\alpha\times\alpha$. In general, contractiveness can be viewed as a way of ensuring that there exists a unique fixed-point for $\lambda\alpha{:}\kappa.c$. For instance, the non-contractive identity function $\lambda\alpha{:}\kappa.\alpha$ has no unique fixed-point because every constructor of kind $\kappa$ is a fixed-point of the identity.

The notation $\equiv$ in $\mu_{\equiv}$ indicates that the recursive type constructor is *equi-recursive*, as opposed to iso-recursive constructors, which we will denote with $\mu_{\cong}$. Equi-recursive constructors obey the following "equational unrolling" rule, which exhibits the fact that $\mu_{\equiv}$ is a fixed-point (here, as in the rest of this paper, we write $E'[E_1,\cdots,E_n/X_1,\cdots,X_n]$ to indicate the simultaneous capture-avoiding substitution of $E_1,\cdots,E_n$ for $X_1,\cdots,X_n$ in $E'$):

$$
\frac{\alpha\notin\mathrm{Dom}(\Gamma)\qquad \Gamma\vdash\kappa\ \mathit{kind}\qquad \Gamma[\alpha\uparrow\kappa]\vdash c\downarrow\kappa}{\Gamma\vdash\mu_{\equiv}\alpha{:}\kappa.c\equiv c\,[\mu\alpha{:}\kappa.c/\alpha]:\kappa}
$$

Uniqueness is guaranteed by the following "bisimilarity" rule, which says that any constructor that is a fixed-point of $\lambda\alpha{:}\kappa.c$ is equivalent to the $\mu_{\equiv}$:

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

The fixed-point equations induced by unrolling and bisimilarity make $\mu_{\equiv}$ apposite for CHP's and our theoretical development of recursive modules, as we will see in the next section. However, having a true fixed-point operator in our core calculus complicates the procedure for deciding constructor equivalence to the point that it is not known to be decidable. We will study iso-recursive constructors, which complicate the theory but are more practical, in Section 4.2.1.

The term-level construct $\mathit{fix}(x{:}\sigma.e)$ enables the definition of recursive values. The well-formedness rule for fixed-point expressions is similar to the rule for equi-recursive constructors:

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

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

Figure 3: The Structure Calculus

The last premise involves a *valuability* check analogous to the *contractiveness* check. Essentially, it must be statically checkable that the evaluation of $e$ will terminate without side-effects, and references in $e$ to $x$ must occur inside $\lambda$-abstractions (i.e. functions). The *fix* construct is basically a generalization of ML's `fun` declarations to allow valuable expressions to be included in the recursive definition.

The primitive structure calculus (Figure 3) consists of two new syntactic classes: modules $M$ and signatures $S$. The module class is inhabited by a primitive *phase-split* module construct and the signature class by a primitive *phase-split* signature construct. Each phase-split construct is composed directly from core constructs. A phase-split module $[c, e]$ consists of a type constructor $c$ and a term $e$. Sometimes we will use the notation $[\alpha = c, e]$ as shorthand for $[c, e[c/\alpha]]$. A phase-split signature $[\alpha{:}\kappa.\sigma]$ classifies a phase-split module, under the following typing rule:

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

The "static" (or "compile-time" or "constructor") part of a phase-split module can be extracted through a new form of type constructor $\mathit{Fst}\, M$, and the dynamic part (or "run-time" or "term") part can be extracted through a new form of expression $\mathit{Snd}\, M$. They are subject to the following typing rules:

$$\frac{\Gamma \vdash M : [\alpha{:}\kappa.\sigma]}{\Gamma \vdash \mathit{Fst}\, M : \kappa} \qquad\qquad \frac{\Gamma \vdash M : [\alpha{:}\kappa.\sigma]}{\Gamma \vdash \mathit{Snd}\, M : \sigma[\mathit{Fst}\, M/\alpha]}$$

Valuability for modules is embodied in the judgement $\Gamma \vdash M \downarrow S$, which ensures that the static part of the module is contractive and the dynamic part is valuable:

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

HMM show that higher-order modules, such as functors and nested structures, are *already present* in the primitive structure calculus described here. We may thus assume compatibility with higher-order modules while working within a simpler system. The method HMM employ is to define typing rules for higher-order module and signature constructs, then to *equate* those constructs with primitive forms through *non-standard equational rules* on modules and signatures. As we shall see below, CHP use the same technique to define the *phase-splitting interpretation* of the recursive module and signature constructs they introduce in terms of core constructs.

The full set of typing rules for the core and structure calculi can be found in Appendices A.1 and A.2, respectively.

## 2.2 Fixed-Point Modules

CHP study two styles of recursive module programming: *opaque* and *transparent*. Both utilize a simple recursive module construct analogous to the recursive constructs for constructors and terms in the core calculus: the fixed-point. CHP's fixed-point module takes the form $\mathit{fix}(s{:}S.M)$, and obeys the following typing rule, typical for a fixed-point:

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

The module body is required to be valuable in a context where the recursive module variable is not. This valuability restriction is essentially the same as the restriction on polymorphic bindings imposed in the revised 1997 definition of Standard ML[16], but here it is applied to *all* the bindings in the body of a fixed-point, not just the polymorphic ones. Note that this restriction is rather conservative in the sense that there are perfectly acceptable recursive modules with non-valuable bodies, e.g. those containing the binding `val x = ref 2`. The practical implications of this conservativity are not really the focus of CHP *or* this paper, but we will return to the issue briefly in Section 4.4.

Phase-splitting for fixed-point modules works as follows. Note that $s^{\mathsf{c}}$ and $s^{\mathsf{r}}$ are used here to signify the compile-time and run-time components, respectively, of primitive structure variable $s$ after phase-splitting, but they are merely specially-named variables, unrelated to $s$.

$$\frac{\begin{array}{c} \alpha, s, s^{\mathsf{c}}, s^{\mathsf{r}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[\alpha : \kappa] \vdash \sigma \ \mathit{type} \\ \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[s^{\mathsf{c}} : \kappa][s^{\mathsf{r}} \uparrow \sigma[s^{\mathsf{c}}/\alpha]] \vdash e \downarrow \sigma[c/\alpha] \end{array}}{\begin{array}{c} \Gamma \vdash \mathit{fix}(s{:}[\alpha{:}\kappa.\sigma].[c[\mathit{Fst}\ s/s^{\mathsf{c}}], e[\mathit{Fst}\ s, \mathit{Snd}\ s/s^{\mathsf{c}}, s^{\mathsf{r}}]]) \equiv \\ [s^{\mathsf{c}} = \mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c, \mathit{fix}(s^{\mathsf{r}}{:}\sigma[s^{\mathsf{c}}/\alpha].e)] : [\alpha{:}\kappa.\sigma] \end{array}}$$

Essentially, phase-splitting a fixed-point module wraps an equi-recursive constructor around the static part $c$ and a term-level fixed-point around the dynamic part $e$. It should be noted that the use of an equi-recursive constructor here is necessary for the term-level fixed-point to typecheck. Specifically, $e$ has type $\sigma[c/\alpha]$ but it needs to match the type of the fixed-point, which is $\sigma[s^{\mathsf{c}}/\alpha]$. With $s^{\mathsf{c}}$ as an abbreviation for $\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c$, it is clear that $c$ (with free references to $s^{\mathsf{c}}$) is the unrolling of $s^{\mathsf{c}}$. Thus, the equality of $\sigma[c/\alpha]$ and $\sigma[s^{\mathsf{c}}/\alpha]$ is predicated on the equality of $s^{\mathsf{c}}$ and its unrolling, i.e. on the use of the equi-recursive $\mu_{\equiv}$.

## 2.3 Opaque Recursive Modules

What CHP call *opaque recursive modules* are simply the fixed-point modules presented above, with no other extensions to the type system. These modules *per se* are termed opaque because the only information afforded the body $M$ about the recursive module variable $s$ in fixed-point module $\mathit{fix}(s{:}S.M)$ is the signature $S$. It should be understood that making $S$ opaque is *not* an abstraction mechanism. It does not hide the implementation of types defined inside the fixed-point from code outside the module, because the fixed-point module can always be phase-split and its static part revealed. (In practice, such abstraction can be achieved by opaquely ascribing $S$ to the entire fixed-point, e.g. $\mathit{fix}(\cdots) :> S$.) Rather, it merely hides some of the static part of $s$ from the code in the body $M$ of the module.

What purpose, then, does this opacity serve? In cases where the fixed-point is used solely to write mutually recursive function definitions that cross module boundaries, there is no reason not to make $S$ transparent. Trouble arises, however, when one wishes to define mutually recursive type definitions that cross module boundaries. In order to encode such definitions, at least one of them must be defined using a *module-recursive type constructor*[1], i.e. a type constructor in the body of the fixed-point that refers to the recursive module variable. The issue is that a module-recursive constructor definition like `type t =` $c$, where $c$ refers to $s$, cannot be realized in the signature $S$, since $S$ must be valid in the ambient context that does not contain $s$. Thus, making $t$ opaque in $S$ is a seemingly necessary evil attendant upon module-recursive constructor definitions.

A simple example of how the opacity forced upon module-recursive constructor definitions causes a problem is the following failed attempt to use an opaque recursive module to implement a (recursive) integer list type through the recursive module. (Assume that the core language has been extended with binary sum types and the corresponding standard term-level constructors, `inl` and `inr`, and destructor, `case`. Note also that we will use a sort of pidgin-ML syntax to convey the examples, but this is intended to represent code in our *internal* language and make it more readable, not to imply extensions to ML itself.)

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

structure List = fix (List : LIST.
  type t = 1 + int * List.t
  val nil : t = inl ()
  fun cons (x : int, L : List.t) : t =
    inr (x,L)
)
```

In this example, the body of the fixed-point is well-typed, but the `cons` function has type `int * List.t -> t`, not the required `int * t -> t`, so the fixed-point is not valid. Alternatively, one could try assigning `L` the type `t` in the definition of `cons`, but this clearly fails as well because the `inr` requires `int * List.t`, not `int * t`.

Another good example given by CHP is the canonical `ExpDec` example. The idea is that one would like to separate two syntactic classes in a mutually recursive abstract syntax tree, such as expressions (`Exp`) and declarations (`Dec`), into two mutually recursive modules. A similar problem arises here as in the `List` example. In addition, the `ExpDec` example illustrates problems in programming with mutually recursive modules that do not affect the single recursive module of `List`. However, the distinction between `List` and `ExpDec` is not the focus of the core analysis of CHP, so we will not be concerned with the details of `ExpDec` until Section 3.2.

### 2.4 Transparent Recursive Modules and Recursively Dependent Signatures

The problem with opaque recursive modules boils down to the inability to make any correlation between the implementation of a type in the body (e.g. `t`) and the corresponding type component of the recursive module variable (e.g. `List.t`), when the type is left abstract in the signature of the recursive module variable. CHP suggest that the way to remedy this is to somehow make the signature transparent, so that the module-recursive type definitions in the body are reflected

[1] *Module-recursive* is our term, not CHP's.

in the signature. As discussed above, naïvely realizing the signature with the module-recursive type definition would result in an ill-formed signature with free references to the recursive module variable `List`. Luckily, the effect of such a realization can be achieved through a more complex signature mechanism introduced by CHP: a *recursively dependent signature*, or *rds*.

An rds takes the form $\rho s^{\mathsf{c}}.S$, where the constructor variable $s^{\mathsf{c}}$ is bound in signature $S$. (Actually, CHP's rds construct takes the form $\rho s.S$, where $s$ is a *module* variable, but since a signature can only contain references to the static part of $s$, the two constructs amount to the same thing.) Intuitively, a type definition in $S$ that refers to $s^{\mathsf{c}}$ is expressing a module-recursive type not expressible in normal signatures; $s^{\mathsf{c}}$ can essentially be viewed as a placeholder for the static part of the module to which the rds is ascribed. The introduction and elimination rules for rds's are simple, but somewhat unusual. Applications of these rules can be thought of as roll and unroll operations, respectively, at the module level:

$$\frac{\Gamma \vdash M : S[\mathit{Fst}\, M/s^{\mathsf{c}}] \qquad \Gamma \vdash \rho s^{\mathsf{c}}.S \;\; \mathit{sig}}{\Gamma \vdash M : \rho s^{\mathsf{c}}.S} \qquad \frac{\Gamma \vdash M : \rho s^{\mathsf{c}}.S}{\Gamma \vdash M : S[\mathit{Fst}\, M/s^{\mathsf{c}}]}$$

Now consider our `List` example again. We can close the free module-recursive references in the fully transparent version of the `LIST` signature by binding `List` in an rds:

```
ρ List.
sig
  type t = 1 + int * List.t
  ...
end
```

When the recursive module variable `List` is assigned the above rds, the rds elimination rule allows `List` to be given the unrolled form of this rds, with free references to itself. Projecting from the unrolled signature, it is then apparent that `List.t` $\equiv$ `1 + int * List.t`, which equals the definition of `t` in the body of the fixed-point.

What this implies about rds's is that their interpretation into the core language requires the use of equi-recursive constructors to "solve" the module-recursive type equations introduced by the roll/unroll rules. The phase-splitting rule for rds's achieves this by wrapping the fully transparent static part of an rds with a $\mu_{\equiv}$ by means of singleton kinds:

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \;\; \mathit{kind} \qquad \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma \;\; \mathit{type}}{\Gamma \vdash \rho s^{\mathsf{c}}.[\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \equiv [s^{\mathsf{c}}{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[s^{\mathsf{c}}/\alpha]] \;\; \mathit{sig}}$$

The way rds phase-splitting works requires the body of the rds to be fully transparent. Hence, the combination of a fixed-point module and an rds is termed a *transparent recursive module*. The typechecking rule for rds's reflects the full transparency requirement:

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \;\; \mathit{kind} \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \;\; \mathit{sig} \qquad \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \rho s^{\mathsf{c}}.S \;\; \mathit{sig}}$$

# 3 Recursive Modules in Theory

The recursive module dichotomy as we have presented it is somewhat unsatisfying because the distinction between opaque and transparent recursive modules is not entirely clear. Consider the following: Both idioms use the same fixed-point module construct, but transparent modules use fully transparent rds's while opaque modules use arbitrary non-rds signatures. Opaque recursive modules appear not to be able to encode module-recursive type constructors, rendering them inexpressive in comparison to transparent recursive modules. However, since rds's can be interpreted through phase-splitting as primitive signatures, it would seem that transparent recursive modules are really a subset of opaque ones!

In fact, this paradox is easily resolved by a clarification of what is meant by a "non-rds signature". When we assert that the signature in an opaque recursive module is not an rds, it is not enough to merely prohibit the signature from having the syntactic form $\rho s^{\mathsf{c}}.S$; the point is that the signature should not require the expressive power necessary to encode transparent definitions of module-recursive type constructors. In our system, module-recursive constructors are implemented with equi-recursive constructors, so the precise definition of a non-rds signature is one whose phase-split form is expressible without the use of equi-recursive constructors. Now there is no question that transparent modules have the expressive edge.

That this confusion in terms was glossed over by CHP indicates an implicit assumption in their presentation of recursive modules: namely, that the notion of a module-recursive constructor in fixed-points and rds's encapsulates the ways that equi-recursiveness is used, so the programmer who is equipped with these recursive module constructs should not need to make explicit use of equi-recursive constructors. It is reasonable to want to isolate equi-recursiveness in the phase-splitting rules for fixed-points and rds's. The core language of ML allows only a very restricted form of recursive type definition with the `datatype` mechanism, whose elaboration only requires the use of *iso-recursive* constructors. Ultimately, we will not want to rely on constructs that are implemented with equi-recursive constructors, so it is unfortunate that the phase-splitting interpretations of both fixed-point modules and rds's seem to require them.

In this section, we will reassess the idioms of opaque and transparent recursive module programming by defining them in terms of opaque and transparent variants, respectively, of the fixed-point module and rds constructs. Splitting fixed-points and rds's this way will help to isolate and clarify the uses of equi-recursiveness in the opaque and transparent idioms. In addition, we will state a "fundamental" property of recursive modules unifying the two idioms and prove that they both adhere to it. We end the section with some remarks on the relevance of our theoretical analysis.

### 3.1 Opaque Fixed-Point Modules

Recall the `List` example from Section 2.3. The problem there was not that the body of the fixed-point was ill-typed, but that its signature contained references to the recursive module variable `List` and thus did not match the required signature `LIST`. Specifically, the type of the `cons` function in the body was `int * List.t -> t`, instead of `int * t -> t` as in `LIST`. Solving this problem by making the `LIST` signature transparent means that not only are the actual and required types for `cons` identified, but `t` and `List.t` are identified *during typechecking* of the body. This solution is unnecessarily generous because the opacity of `List.t` did not cause a problem during typechecking, only when matching the body's signature against the required signature.

This suggests a generalization of the fixed-point module rule. If the recursive module variable $s$ is given signature $[\alpha{:}\kappa.\sigma_1]$, then allow the body to have signature $[\alpha{:}\kappa.\sigma_2]$, where $\sigma_1 \equiv \sigma_2[\alpha/\mathit{Fst}\, s]$. (We view the signatures here in their phase-split form in order to have access to the $\alpha$ that gets substituted for $\mathit{Fst}\, s$; in Section 3.4 we will see how this can be avoided.) For instance, the `List` example would now typecheck, because if we substitute `t` for occurrences of `List.t` in the actual type of `cons` (`int * List.t -> t`), we obtain the required type (`int * t -> t`). Note, however, that the signature of $s$ is still opaque (i.e. $\kappa$ can be anything) during typechecking.

Let's call this generalization an *opaque fixed-point module* and denote it with $\mathit{fix}_O$. The type-checking rule is as one would expect from the description above:

$$\frac{\alpha, s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash S \equiv [\alpha{:}\kappa.\sigma_1]\ \ \mathit{sig} \qquad \Gamma[s \uparrow S] \vdash M \downarrow [\alpha{:}\kappa.\sigma_2] \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/\mathit{Fst}\ s]\ \ \mathit{type}}{\Gamma \vdash \mathit{fix}_O(s{:}S.M) : S}$$

This generalization only makes sense, of course, if it does not adversely affect phase-splitting. Fortunately, it does not affect phase-splitting at all. In fact, the idea for the generalization came from looking at what premises of the phase-splitting rule for fixed-points could be safely relaxed. The intuitive reason the opaque fixed-point works is that the equi-recursive constructor used in phase-splitting turns the static part of the recursive module variable and the static part of the body into the same thing. E.g., after phase-splitting, `List.t` becomes $\mu_\equiv\alpha.\,1 + \mathit{int} \times \alpha$ and `t` becomes $1 + \mathit{int} \times$ `List.t`, which are equal since $\mu_\equiv$ is equi-recursive. For completeness, here is the phase-splitting rule for opaque fixed-points, which differs from the original phase-splitting rule only in the more flexible premises:

$$\frac{\begin{array}{c}\alpha, s, s^{\mathsf{c}}, s^{\mathsf{r}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa\ \ \mathit{kind} \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1\ \ \mathit{type} \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma_2\ \ \mathit{type} \\ \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[s^{\mathsf{c}} : \kappa][s^{\mathsf{r}} \uparrow \sigma_1[s^{\mathsf{c}}/\alpha]] \vdash e \downarrow \sigma_2[c/\alpha] \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/s^{\mathsf{c}}]\ \ \mathit{type}\end{array}}{\Gamma \vdash \mathit{fix}_O(s{:}[\alpha{:}\kappa.\sigma_1].[c[\mathit{Fst}\ s/s^{\mathsf{c}}], e[\mathit{Fst}\ s, \mathit{Snd}\ s/s^{\mathsf{c}}, s^{\mathsf{r}}]]) \equiv [s^{\mathsf{c}} = \mu_\equiv s^{\mathsf{c}}{:}\kappa.c, \mathit{fix}(s^{\mathsf{r}}{:}\sigma_1[s^{\mathsf{c}}/\alpha].e)] : [\alpha{:}\kappa.\sigma_1]}$$

Note that the fixed-point expression on the right-hand side of the equation typechecks because $\sigma_2[c/\alpha] \equiv \sigma_2[s^{\mathsf{c}}/\alpha]$ (where $s^{\mathsf{c}} \equiv \mu_\equiv s^{\mathsf{c}}{:}\kappa.c$), and $\sigma_1[s^{\mathsf{c}}/\alpha] \equiv \sigma_2[\alpha/s^{\mathsf{c}}][s^{\mathsf{c}}/\alpha] \equiv \sigma_2[s^{\mathsf{c}}/\alpha]$.

### 3.2 Opaque Rds's

Although the above generalization of the fixed-point construct solves the problem with opaque recursive modules in the `List` example, it does not entirely solve the similar problem in CHP's `ExpDec` example. We will now examine the details of `ExpDec` because they motivate the idea of an opaque rds.

In `ExpDec`, we essentially want to break up a pair of mutually recursive types `exp` and `dec`, representing abstract syntax for expressions and declarations in some object language, and put them in mutually recursive modules `Exp` and `Dec`, respectively. In the absence of rds's, this is impossible — even if `exp` and `dec` are kept abstract, the types of some of `exp`'s constructors will need to refer to `dec` and vice versa, and there is no way to express this in typical hierarchical signatures (e.g. the boxed types are mutually recursive references, the first of which is illegal):

```
signature EXPDEC = sig
  structure Exp : sig
    type exp
    val make_let : Dec.dec * exp -> exp          (* let DEC in EXP end *)
    ... (* other constructors and destructors *)
  end
  structure Dec : sig
    type dec
    val make_val : identifier * Exp.exp -> dec   (* val ID = EXP *)
    ... (* other constructors and destructors *)
  end
end
```

CHP observe that this problem does not arise when signatures are written in phase-split form because the recursive dependencies are *dynamic-on-static* (i.e. terms on types), not *static-on-static* (i.e. types on types). Once the static and dynamic parts of the signature have been separated, dynamic-on-static dependencies are no longer recursive. Indeed, there was no problem in the `List` example because the `LIST` signature was naturally in phase-split form.

Although writing signatures in phase-split form solves the problem, it is not reasonable to force code to be written in phase-split style in general. CHP illustrate a failed attempt to avoid this by declaring `exp` and `dec` in the signatures of both `Exp` and `Dec`, all opaquely:

```
signature EXPDEC_CHP = sig
  structure Exp : sig
    type exp
    type dec
    val make_let : dec * exp -> exp
  end
  structure Dec : sig
    type dec
    type exp
    val make_val : identifier * exp -> dec
  end
end
```

This rids the signature of recursive references, but now there is no connection between the types `Exp.exp` and `Dec.exp` (and `Dec.dec` and `Exp.dec`). Thus, syntax trees constructed with the datatype constructors in `Exp` are incompatible with those constructed with the datatype constructors in `Dec`, rendering a module with this signature useless.

Our solution is to define an *opaque* variant of the rds construct, denoted $\widehat{\rho}$, that only allows dynamic-on-static recursive dependencies. (The original rds construct, which we will continue to denote with $\rho$, is by its very nature *transparent*.) First, the typechecking rule:

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\kappa.\sigma] \ \mathit{sig}}{\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S \ \mathit{sig}}$$

The restriction on recursive dependencies is enforced by requiring the static part $\kappa$ of signature $S$ to be well-formed in the ambient context (so $S$ cannot contain static-on-static references to $s^{\mathsf{c}}$). The phase-splitting rule makes clear that there is nothing fundamentally recursive about this opaque rds, in contrast to the fully transparent rds defined previously:

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma \ \mathit{type}}{\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.[\alpha{:}\kappa.\sigma] \equiv [s^{\mathsf{c}}{:}\kappa.\sigma[s^{\mathsf{c}}/\alpha]] \ \mathit{sig}}$$

With the opaque rds, we can "close" the dynamic-on-static dependencies desired in the `EXPDEC` signature as follows:

```
ρ̂ ExpDec.
sig
  structure Exp : sig
    type exp
    val make_let : ExpDec.Dec.dec * exp -> exp
```

```
  end
  structure Dec : sig
    type dec
    val make_val : identifier * <ExpDec.>Exp.exp -> dec
  end
end
```

The angle brackets around `ExpDec` in the type of `Dec.make_val` signify that the reference to `Exp.exp` could be chosen to be local or recursive, and the signature would phase-split the same either way.

### 3.3 Transparent Fixed-Point Modules

In the spirit of isolating equi-recursiveness, one may wonder whether the transparent recursive module idiom requires the power of equi-recursive constructors in *both* its fixed-point and rds constructs. Interestingly, the answer is no.

Any fixed-point module with transparent signature is equivalent to a module of the form $\mathit{fix}(s{:}[\alpha{:}\mathfrak{S}(c : \kappa).\sigma].[c', e])$. After phase-splitting, the static part of the module is $\mu_{\equiv}\alpha{:}\mathfrak{S}(c : \kappa).c'[\alpha/\mathit{Fst}\ s]$. Since the kind of this constructor is $\mathfrak{S}(c : \kappa)$, it is provably equal to $c$! So, in fact, equi-recursive constructors are not needed here. This leads us to observe that the notion of a transparent fixed-point module — that is, a fixed-point module with fully transparent signature — is orthogonal to that of a transparent rds and would benefit from being treated separately. This does not imply, of course, that the examples of transparent recursive modules presented in CHP do not require equi-recursive constructors in their interpretation. It simply means that the equi-recursiveness is completely encapsulated in the notion of a transparent rds and those examples require the use of rds's.

We will denote the *transparent fixed-point module* construct with $\mathit{fix}_T$. The typechecking and phase-splitting rules are as one would expect:

$$\frac{s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma[s \uparrow S] \vdash M \downarrow S \qquad \Gamma \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\ \mathit{sig}}{\Gamma \vdash \mathit{fix}_T(s{:}S.M) : S}$$

$$\frac{\begin{array}{c}\alpha, s, s^{\mathsf{c}}, s^{\mathsf{r}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \mathfrak{S}(c : \kappa)\ \mathit{kind} \qquad \Gamma[\alpha : \mathfrak{S}(c : \kappa)] \vdash \sigma\ \mathit{type} \\ \Gamma[s^{\mathsf{c}} \uparrow \mathfrak{S}(c : \kappa)] \vdash c' \downarrow \mathfrak{S}(c : \kappa) \qquad \Gamma[s^{\mathsf{c}} : \mathfrak{S}(c : \kappa)][s^{\mathsf{r}} \uparrow \sigma[s^{\mathsf{c}}/\alpha]] \vdash e \downarrow \sigma[c'/\alpha]\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{fix}_T(s{:}[\alpha{:}\mathfrak{S}(c : \kappa).\sigma].[c'[\mathit{Fst}\ s/s^{\mathsf{c}}], e[\mathit{Fst}\ s, \mathit{Snd}\ s/s^{\mathsf{c}}, s^{\mathsf{r}}]]) \\ \qquad \equiv [s^{\mathsf{c}} = c, \mathit{fix}(s^{\mathsf{r}}{:}\sigma[s^{\mathsf{c}}/\alpha].e)] : [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\end{array}}$$

Note that the generalization from Section 3.1 would not increase the expressiveness of transparent fixed-points. Specifically, any references to $\mathit{Fst}\,s$ in the signature of the body of the fixed-point could be replaced by $\alpha$ anyway, since both $\mathit{Fst}\,s$ and $\alpha$ have kind $\mathfrak{S}(c : \kappa)$ and thus are equivalent.

### 3.4 Fundamental Property of Recursive Modules

Now that we have defined two clearly differentiated recursive module idioms, it is interesting to examine what they have in common.

In their treatment of practical issues regarding recursive modules, CHP suggest an "appealing typechecking strategy" for transparent recursive modules that basically works as follows. Assume the module has the form $\mathit{fix}(s{:}S.M)$, where $S \equiv \rho s^{\mathsf{c}}.S_B$, the notation $S_B$ signifying the body of $S$. Instead of checking that $M$ has signature $S$ as prescribed by the fixed-point typechecking rule, check that $M$ has signature $S_B[\mathit{Fst}\,s/s^{\mathsf{c}}]$, in a context where $s$ has signature $S$. In other words, check that the *body* of the fixed-point can be ascribed the *body* of the rds, under the assumption that the recursive module variable can be ascribed the rds. CHP give a simple proof that this typechecking strategy is sound and complete with respect to the fixed-point typechecking rule.

This approach is appealing because it provides a more intuitive and practical way of defining and implementing fixed-point module typechecking than that supplied by the type system directly. In ordinary fixed-point typechecking, one would have two options for checking that the body $M$ of a fixed-point has rds $\rho s^{\mathsf{c}}.S_B$. One could use the rds roll rule to reduce this to checking that $M$ has $S_B[\mathit{Fst}\,M/s^{\mathsf{c}}]$, but this involves a substitution of $\mathit{Fst}\,M$ when it is not even known that $M$ is well-formed. The substitution of $\mathit{Fst}\,s$ for $s^{\mathsf{c}}$ in the new strategy is much more straightforward to implement. Alternatively, one could compute the principal signature of $M$, phase-split it and check if it is a subtype of the rds. While this is feasible to implement, the new approach is far more elegant because it can be described at a high level above the "implementation" level of phase-split forms.

What we find particularly appealing about this approach is that it turns out to apply to opaque recursive modules as well, albeit in a slightly weaker form. This leads us to believe that, in addition to being a guide to implementation, this typechecking strategy is fundamental to recursive module systems, at least of the kind we have studied. Here we formalize it for arbitrary fixed-point and rds constructs $FIX$ and $R$:

**Definition 1 (Fundamental Property of Recursive Modules)**
*A fixed-point module construct $FIX(s{:}S.M)$ and an rds construct $Rs^{\mathsf{c}}.S$ form a* recursive module system *if they satisfy the following property:*

- $\Gamma \vdash FIX(s{:}S.M) : S$ **if and only if** $\exists S_B$ **such that** $\Gamma \vdash S \equiv Rs^{\mathsf{c}}.S_B\ \ sig$ **and** $\Gamma[s \uparrow S] \vdash M \downarrow S_B[\mathit{Fst}\,s/s^{\mathsf{c}}]$.

The rule described above was stronger than the one given here in that the equivalence of $S$ and $Rs^{\mathsf{c}}.S_B$ was an assumption, whereas here it appears inside the "iff". The reason for this weakening is that, for opaque modules, there *exists* an rds equivalent to $S$ that models the signature of the module body correctly, but not every equivalent rds does so. This makes this strategy arguably less practical than the first one, but more useful in theory as a shared high-level property of recursive modules.

We now proceed to show that both the opaque and transparent idioms obey this fundamental property. The proof requires us to be able to perform inversion on the typechecking rules for $\mathit{fix}_O$ and $\mathit{fix}_T$, which we cannot do with the rds roll and unroll rules from Section 2.4 built into the type system. Luckily, the roll and unroll rules are not essential (for either opaque or transparent rds's) and can be shown to be admissible in the type system without them. The proofs are fairly straightforward. It is worth noting that the admissibility proof for opaque rds's makes use of the "self" rule to reify the static part of the opaque signature; and that the proof for transparent rds's makes use of equi-recursive unrolling for admissibility of unroll and bisimilarity for admissibility of roll.

**Lemma 2 (Admissibility of Roll and Unroll for Opaque Rds's)**
*Suppose* $\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S\ \ sig$. *Then,* $\Gamma \vdash M : \widehat{\rho} s^{\mathsf{c}}.S$ *if and only if* $\Gamma \vdash M : S[\mathit{Fst}\,M/s^{\mathsf{c}}]$.

**Proof:** Suppose $\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S\ \ sig$. Then, by well-formedness of opaque rds's, we know that $\Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\kappa.\sigma]\ \ sig$, and $\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S \equiv [\alpha{:}\kappa.\sigma[\alpha/s^{\mathsf{c}}]]\ \ sig$.

$\Longrightarrow$ : Suppose $\Gamma \vdash M : \widehat{\rho} s^{\mathsf{c}}.S$. Then, $\Gamma \vdash M : [\alpha{:}\kappa.\sigma[\alpha/s^{\mathsf{c}}]]$. By selfification, $\Gamma \vdash M : [\alpha{:}\mathfrak{S}(\mathit{Fst}\,M : \kappa).\sigma[\alpha/s^{\mathsf{c}}]]$. This signature is equivalent to $[\alpha{:}\mathfrak{S}(\mathit{Fst}\,M\ :\ \kappa).\sigma[\mathit{Fst}\,M/s^{\mathsf{c}}]] \leq [\alpha{:}\kappa.\sigma[\mathit{Fst}\,M/s^{\mathsf{c}}]] \equiv$

$S[\mathit{Fst}\, M/s^{\mathsf{c}}]$.

$\Longleftarrow$ : Suppose $\Gamma \vdash M : S[\mathit{Fst}\, M/s^{\mathsf{c}}]$. Then, $\Gamma \vdash M : [\alpha{:}\kappa.\sigma[\mathit{Fst}\, M/s^{\mathsf{c}}]]$. By selfification, $\Gamma \vdash M : [\alpha{:}\mathfrak{S}(\mathit{Fst}\, M : \kappa).\sigma[\mathit{Fst}\, M/s^{\mathsf{c}}]]$. This signature is equivalent to $[\alpha{:}\mathfrak{S}(\mathit{Fst}\, M : \kappa).\sigma[\alpha/s^{\mathsf{c}}]] \leq [\alpha{:}\kappa.\sigma[\alpha/s^{\mathsf{c}}]] \equiv \widehat{\rho} s^{\mathsf{c}}.S$. ■

**Lemma 3 (Admissibility of Roll and Unroll for Transparent Rds's)**
*Suppose $\Gamma \vdash \rho s^{\mathsf{c}}.S\ \ sig$. Then, $\Gamma \vdash M : \rho s^{\mathsf{c}}.S$ if and only if $\Gamma \vdash M : S[\mathit{Fst}\, M/s^{\mathsf{c}}]$.*

**Proof:** Suppose $\Gamma \vdash \rho s^{\mathsf{c}}.S\ \ sig$. Then, by well-formedness of transparent rds's, we know that $\Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\ \ sig$, and $\Gamma \vdash \rho s^{\mathsf{c}}.S \equiv [\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]]\ \ sig$, where $\Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa$.

$\Longrightarrow$ : Suppose $\Gamma \vdash M : \rho s^{\mathsf{c}}.S$. Then, $\Gamma \vdash M : [\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]]$. Since $\Gamma \vdash \mathit{Fst}\, M \equiv \mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa$, $M$'s signature is equivalent to $[\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\mathit{Fst}\, M/s^{\mathsf{c}}]]$, which, by equi-recursive unrolling, is equivalent to $[\alpha{:}\mathfrak{S}(c[\mathit{Fst}\, M/s^{\mathsf{c}}] : \kappa).\sigma[\mathit{Fst}\, M/s^{\mathsf{c}}]] \equiv S[\mathit{Fst}\, M/s^{\mathsf{c}}]$.

$\Longleftarrow$ : Suppose $\Gamma \vdash M : S[\mathit{Fst}\, M/s^{\mathsf{c}}]$. Then, $\Gamma \vdash M : [\alpha{:}\mathfrak{S}(c[\mathit{Fst}\, M/s^{\mathsf{c}}] : \kappa).\sigma[\mathit{Fst}\, M/s^{\mathsf{c}}]]$. Since $\Gamma \vdash \mathit{Fst}\, M \equiv c[\mathit{Fst}\, M/s^{\mathsf{c}}] : \kappa$, bisimilarity implies that $\mathit{Fst}\, M \equiv \mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c$, so $M$'s signature is equivalent to $[\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]] \equiv \rho s^{\mathsf{c}}.S$. ■

**Theorem 4**
*The opaque fixed-point and rds constructs form a recursive module system.*

**Proof:**
$\Longrightarrow$ : Suppose $\Gamma \vdash \mathit{fix}_O(s{:}S.M) : S$. By inversion on typechecking for $\mathit{fix}_O$, $\Gamma \vdash S \equiv [\alpha{:}\kappa.\sigma_1]\ \ sig$, $\Gamma[s \uparrow S] \vdash M \downarrow [\alpha{:}\kappa.\sigma_2[\mathit{Fst}\, s/s^{\mathsf{c}}]]$ and $\Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/s^{\mathsf{c}}]\ \ type$. Choosing $S_B$ to be $[\alpha{:}\kappa.\sigma_2]$, it is clear that $\Gamma[s \uparrow S] \vdash M \downarrow S_B[\mathit{Fst}\, s/s^{\mathsf{c}}]$. By phase-splitting for $\widehat{\rho}$, we also have $\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S_B \equiv [\alpha{:}\kappa.\sigma_2[\alpha/s^{\mathsf{c}}]] \equiv [\alpha{:}\kappa.\sigma_1] \equiv S\ \ sig$.

$\Longleftarrow$ : Suppose $\Gamma \vdash S \equiv \widehat{\rho} s^{\mathsf{c}}.S_B\ \ sig$ and $\Gamma[s \uparrow S] \vdash M \downarrow S_B[\mathit{Fst}\, s/s^{\mathsf{c}}]$. Well-formedness of $\widehat{\rho} s^{\mathsf{c}}.S_B$ requires that $\Gamma[s^{\mathsf{c}} : \kappa] \vdash S_B \equiv [\alpha{:}\kappa.\sigma_2[s^{\mathsf{c}}/\mathit{Fst}\, s]]\ \ sig$, where $\Gamma \vdash \kappa\ \ kind$ (i.e. $S_B$ contains only dynamic-on-static recursive references) and $s^{\mathsf{c}}$ is not free in $\sigma_2$. Choose $\sigma_1$ to be $\sigma_2[\alpha/\mathit{Fst}\, s]$. Then, by phase-splitting for $\widehat{\rho}$, we have that $\Gamma \vdash S \equiv [\alpha{:}\kappa.\sigma_2[\alpha/\mathit{Fst}\, s]] \equiv [\alpha{:}\kappa.\sigma_1]\ \ sig$. Since $S_B[\mathit{Fst}\, s/s^{\mathsf{c}}] \equiv [\alpha{:}\kappa.\sigma_2]$, the initial assumption gives us $\Gamma[s \uparrow S] \vdash M \downarrow [\alpha{:}\kappa.\sigma_2]$. We have thus fulfilled the premises of the typechecking rule for $\mathit{fix}_O$, so $\Gamma \vdash \mathit{fix}_O(s{:}S.M) : S$. ■

**Theorem 5**
*The transparent fixed-point and rds constructs form a recursive module system.*

**Proof:**
$\Longrightarrow$ : Suppose $\Gamma \vdash \mathit{fix}_T(s{:}S.M) : S$. By inversion on typechecking for $\mathit{fix}_T$, $\Gamma \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\ sig$ and $\Gamma[s \uparrow S] \vdash M \downarrow S$. Choose $S_B$ to be $S$. Since $s^{\mathsf{c}} \notin \mathrm{FV}(S)$, $\Gamma \vdash S_B[\mathit{Fst}\, s/s^{\mathsf{c}}] \equiv S\ \ sig$, and so $\Gamma[s \uparrow S] \vdash M \downarrow S_B[\mathit{Fst}\, s/s^{\mathsf{c}}]$. Also, because $s^{\mathsf{c}} \notin \mathrm{FV}(c) \cup \mathrm{FV}(\sigma)$ and by phase-splitting for $\rho$, $\Gamma \vdash \rho s^{\mathsf{c}}.S_B \equiv [\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]] \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \equiv S\ \ sig$.

$\Longleftarrow$ : Suppose $\Gamma \vdash S \equiv \rho s^{\mathsf{c}}.S_B\ \ sig$ and $\Gamma[s \uparrow S] \vdash M \downarrow S_B[\mathit{Fst}\, s/s^{\mathsf{c}}]$. Well-formedness and phase-splitting for $\rho$ tell us that $\Gamma \vdash S_B \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\ \ sig$ and $\Gamma \vdash S \equiv [\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]]\ \ sig$.

(Note of course that $S$ is transparent, a requirement of the $fix_T$ typechecking rule.) By equational reasoning with singleton kinds, $\Gamma[s \uparrow S] \vdash Fst\ s \equiv \mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa$. Therefore, $\Gamma[s \uparrow S] \vdash S_B[Fst\ s/s^{\mathsf{c}}] \equiv [\alpha{:}\mathfrak{S}(c[\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c/s^{\mathsf{c}}] : \kappa).\sigma[\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c/s^{\mathsf{c}}]] \equiv [\alpha{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[\alpha/s^{\mathsf{c}}]] \equiv \rho s^{\mathsf{c}}.S_B \equiv S\ sig$. Combining this with the initial assumption yields $\Gamma[s \uparrow S] \vdash M \downarrow S$. We may now apply the typechecking rule for $fix_T$ to obtain $\Gamma \vdash fix_T(s{:}S.M) : S$. ■

### 3.5 Theoretical Conclusions

What has our theoretical analysis achieved? First of all, we have expanded the scope of the opaque recursive module idiom by generalizing the fixed-point construct and defining an opaque form of rds, so that the examples from CHP do not require the use of transparent modules. Second, we have isolated the uses of equi-recursiveness in the two idioms by demonstrating that opaque rds's and transparent fixed-points do not involve recursion on type constructors. We can therefore freely employ the latter two constructs in a recursive module design extension for the external language.

However, we cannot simply discard the other two equi-recursive constructs as impractical since they are the only constructs that encode module-recursive constructors at all! Enabling the definition of mutually recursive types in separate modules is one of the main motivations behind recursive modules, so a recursive module design that does not handle module-recursive types is highly undesirable. This points to a weakness in our theory, namely that, for simplicity, we have conflated the notions of recursive and equi-recursive.

If the choice between opaque and transparent programming were to be made based on the analysis thus far, the transparent idiom would still be preferable because, while both require equi-recursive constructors, transparent recursive modules are strictly more expressive than opaque recursive modules. That is, every opaque module is, by phase-splitting, equivalent to the transparent module formed by reifying all the abstract type constructors in the opaque rds with their module-recursive definitions in the body of the fixed-point. Intuitively, the transparent idiom is more expressive because it propagates more type equations during typechecking. (It is worth noting that the *reverse* is true for the fixed-point constructs *per se*, in the sense that transparent fixed-points are a special case of opaque fixed-points.)

Why bother, then, with the opaque idiom? Because the power of the opaque idiom, strong enough to handle the examples from CHP, relies less on equi-recursiveness than that of the transparent idiom. In the next section, we will explore if and how iso-recursive constructors can be used to replace equi-recursive constructors. In the process, we will see that the restricted use of equi-recursiveness in opaque programming can be an advantage when the goal is eliminating equi-recursiveness. Thus, understanding why the opaque idiom was able to encode both examples from CHP, despite the fact that it is generally less expressive than the transparent idiom, is a fruitful starting point for our practical analysis.

## 4 Recursive Modules in Practice

We will now take the theoretical constructs studied in the previous section and attempt to develop a recursive module extension for Standard ML based on them. The key criteria for such an extension are how practical it is and how well it generalizes recursive constructs at the ML core level (i.e. datatypes and recursive functions). Not surprisingly, our end result is a proposal that combines aspects of both opaque and transparent programming. Before arriving there, however, we will look at each fixed-point and rds construct and judge its usefulness by the abovementioned criteria.

## 4.1 The Practical Uses of Fixed-Point Modules

We begin the practical comparison of the two recursive module idioms with the question that ended the previous section: If transparent programming is more expressive than opaque programming, why did the examples from CHP work fine in the opaque idiom? The answer is very simple: both the `List` and `ExpDec` examples were essentially implementations of ML-style datatypes.

The `datatype` mechanism, one of ML's strongest successes, provides the only way of defining (mutually) recursive types in the language, specifically recursive (tuples of) labeled sums. The labels corresponding to a datatype serve as constructors of the type, as well as destructors when used in pattern matching against elements of the type. In order to make the relationship between datatypes and opaque programming absolutely clear, it is first necessary to define what we mean by "implementing datatypes". In Section 4.1.1, we provide a brief review of Harper and Stone's formalization of the implementation of datatypes. In Section 4.1.2, we discuss how opaque recursive modules generalize datatypes. Then, in Section 4.1.3, we present an example of non-datatype code in an opaque module that suggests opaque modules are better suited to implementing datatypes than to general programming. Finally, in Section 4.1.4, we look at how transparent fixed-points generalize ML's recursive function definitions.

### 4.1.1 The Harper-Stone Interpretation of Datatypes

Harper and Stone have given an interpretation of Standard ML into an underlying type theory with modules [10], which serves as the basis of the front-end of the TILT compiler for Standard ML in development at Carnegie Mellon University. This interpretation takes the form of a set of elaboration rules that involve typechecking of the ML source program as well as the "desugaring" of ML features such as pattern matching and signature patching. Perhaps the most infamous of these elaboration rules, for its length and complexity, is the one for handling `datatype` declarations.

What datatype elaboration essentially does is to translate a datatype into a module containing a recursive sum type along with a constructor function for each branch of the sum and a single destructor function (labeled `expose`) returning a non-recursive sum. Suppose for the moment that Harper-Stone used equi-recursive types. Then, the constructors would merely be sum injections and the destructor a no-op. For example, an integer list type written in ML as

```
datatype intlist = nil | cons of int * intlist
```

could be elaborated to

```
struct
  type intlist = μ≡α. 1 + int × α
  val nil : intlist = inl ()
  fun cons (x:int, L:intlist) : intlist = inr (x,L)
  fun expose (L:intlist) : 1 + int * intlist = L
end
```

Voilà! This is the phase-split form of the `List` example. One could similarly write `ExpDec` as two mutually recursive datatype declarations (connected with an `and`), and it would elaborate to the phase-split form of the recursive `ExpDec` module. This is what we mean by saying that the CHP examples are implementations of ML datatypes.

In practice, however, equi-recursiveness is not necessary here. An *iso-recursive* constructor (denoted $\mu_{\cong}$) can be used, with rolls and unrolls added to the constructor and destructor definitions, respectively, in order to mediate between the recursive sum type and its unrolling. (More specifics about iso-recursive types and their typing rules will be given in Section 4.2.1.) Thus, Harper-Stone actually elaborates `datatype intlist` to the following:

```
struct
  type intlist = μ≅α. 1 + int × α
  val nil : intlist = roll_intlist(inl ())
  fun cons (x:int, L:intlist) : intlist = roll_intlist(inr (x,L))
  fun expose (L:intlist) : 1 + int * intlist = unroll_intlist(L)
end
```

Note that we are simplifying here by ignoring the details of equality compilation (i.e. the generation of recursive equality functions for datatypes) and polymorphic datatypes, because they are basically orthogonal to the discussion. In addition, there is some controversy over whether or not the module resulting from datatype elaboration should be sealed with an opaque signature hiding the implementation of the datatype, as is prescribed in Harper-Stone. We will return to that issue in Section 4.3.

### 4.1.2 Generalization of ML Datatypes

There are two main ways in which opaque recursive modules generalize ML datatypes. First, the `ExpDec` example exhibits how recursive modules permit mutually recursive type definitions across module boundaries, so long as the type definitions occur under the roof of a single opaque fixed-point and refer to each other through it. We have discussed already that this was one of the primary motivations for a recursive module extension in the first place.

Second, opaque recursive modules generalize and simplify ML's `withtype` mechanism. In ML, `withtype` provides a way of writing ordinary type definitions in a mutually recursive bundle with datatype definitions. In order to avoid the need for equi-recursive types, the ordinary type definitions may only refer to the datatypes, not one another. The Definition of Standard ML [16] specifies that each occurrence of one of the `withtype` identifiers in the `datatype` definitions should be macro-expanded to the type bound to the identifier. This is viewed by some as a rather ugly style of definition, if a necessary one.

In an opaque recursive module, the `withtype` definitions are unnecessary and can be replaced by ordinary type definitions. In order to do this, we simply need to make the `withtype` definitions transparent in the rds, which is feasible because they are not module-recursive. Consider a simple, if admittedly useless, reworking of the `List` example, where the `int * List.t` is extracted into a `withtype` definition. Here is the ML version:

```
datatype list = nil | cons of headtail
withtype headtail = int * list
```

ML macro-expands this to:

```
datatype list = nil | cons of int * list
type headtail = int * list
```

The opaque recursive module can skip the macro-expansion:

```
signature LIST =
sig
  type list
```

```
  type headtail = int * list
  val nil : list
  val cons : headtail -> list
  val expose : t -> 1 + headtail
end

structure List = ofix (List : LIST.
  type list = 1 + List.headtail
  type headtail = int * list
  ...
)
```

Having shown that opaque recursive modules are closely tied to ML datatypes and generalize them in interesting ways, it remains for us to show that they can be implemented without the use of equi-recursive constructors. That issue warrants more extended discussion and is taken up in Section 4.2.

### 4.1.3 The Limitations of Opacity

Suppose we want to extend our trusty `List` example with a simple function called `nthtail` that takes a list `L` and an integer `n` and returns the result of removing the first `n` elements from the list. Writing this function inside the body of the `List` module gives the programmer access to the module-recursive sum implementation of the datatype `t`. Here is a naive attempt to make use of the access to that implementation:

```
fun nthtail (L : t, n : int) : t =
  if n = 0 then L
  else case L of
         inl () => raise Error
       | inr (hd : int, tl : List.t) => nthtail(tl, n-1)
```

This fails to typecheck because the type of `tl` does not match the argument type of `nthtail` in the recursive call. There are two ways to fix this. We could either make the type of argument `L` be `List.t` instead of `t`, or add `nthtail` to the rds `LIST` and replace the recursive call to `nthtail` with a call to `List.nthtail`. Either way, both the argument type and result type of `nthtail` will become `List.t` because the argument `L` is also the result in the case that `n = 0`.

This in turn causes a problem because the `case` expression requires `L` to be of sum type, which `List.t` is not since it is abstract. The simple way to solve this is to make a module-recursive call to `List.expose` before case analysis. The new `nthtail` implementation is as follows:

```
fun nthtail (L : List.t, n : int) : List.t =
  if n = 0 then L
  else case List.expose(L) of
       inl () => raise Error
     | inr (hd : int, tl : List.t) => nthtail(tl, n-1)
```

This example illustrates that "programming *inside* the datatype" is not only of limited utility, but sometimes impossible. Attempts to exploit access to the datatype implementation result in confusing and unintuitive typechecking errors. The solution we have arrived at for `nthtail` is really an instance of "programming *outside* the datatype" because it treats the datatype completely abstractly and could have just as easily been written outside the module. This does not imply that there is anything wrong with writing code like `nthtail` inside the `List` module. In fact, in a recursive module design for ML, we will definitely want to be able to write such functions in the same modules as the datatypes they traverse. The point is simply that there is no expressiveness to be gained from having access to the datatype implementation, and it does not make sense to thrust the "opaque" distinction between `t` and `List.t` on the ML programmer if it is not necessary. We thus turn to transparent fixed-points now to see how they generalize recursive programming in the absence of the opaque distinction.

#### 4.1.4 Transparent Fixed-Points and Polymorphic Recursion

For transparent fixed-point modules, practicality is not an issue since they do not involve any recursion on types at all. The most obvious way that transparent fixed-points generalize ML's recursive function definitions is that the fixed-point allows mutually recursive functions to be defined in separate submodules. In addition, perhaps less immediately clear is that the fixed-point module construct provides a very systematic way of expressing polymorphic recursion.

"Polymorphic recursion" refers to the ability of a polymorphic function to call itself recursively at different instantiations of its polymorphic type. This is particularly useful when writing a function that traverses a *non-uniform* datatype. Analogously, non-uniform datatypes allow a polymorphic datatype definition to refer to itself recursively at different instantiations of its type argument. Here is a simple contrived example to illustrate the two together:

```
datatype 'a t = A | B of 'a * 'a t t

(* val collect : 'a t -> 'a list
   Collects all the data of type 'a in t and forms a list. *)
fun collect (A) = []
  | collect (B(x,FF)) =       (* x : 'a, FF : 'a * 'a t t *)
  let val FL = collect FF     (* collect needs type 'a t t -> 'a t list *)
      val LL = map collect FL (* collect needs type 'a t -> 'a list *)
      val L = foldl (op @) [] LL
  in x::L end
```

The difference between non-uniform and uniform datatypes (and between polymorphic and monomorphic recursion) is simply a matter of where the constructor-level (or term-level) type abstraction occurs — inside or outside the fixed-point. Non-uniform datatypes and polymorphically recursive functions are fixed-points at function kind and function type, respectively, the type lambda occurring underneath the fixed-point construct. Oddly, Standard ML admits non-uniform datatypes, but not polymorphic recursion, the reason being that the latter causes type inference to be undecidable. The issue of type inference in the presence of polymorphic recursion, as well as any debate over the merits of polymorphic recursion, is outside the scope of this paper. (See [17, 11, 12, 18] for more on the subject.)

By forcing the types of the functions in the module body to be specified in the rds, transparent fixed-points enable the coexistence of monomorphic and polymorphic recursion without affecting type inference at all. While ordinary recursive calls remain monomorphic, module-recursive calls are free to be polymorphic. So, for instance, in the example above, the `collect` function could be expressed in a transparent fixed-point as follows:

```
tfix (Mod : sig val collect : 'a t -> 'a list end.
```

```
  fun collect (A) = []
    | collect (B(x,FF)) =
    let val FL = Mod.collect FF  (* polymorphic recursive call *)
        val LL = map collect FL  (* monomorphic recursive call *)
        val L = foldl (op @) [] LL
    in x::L end
)
```

The reason transparent fixed-points have no trouble expressing polymorphic recursion is that they are implemented with the term-level *fix* construct, which is defined to work over an arbitrary type. In a sense, relying on the explicitness of the rds in a transparent fixed-point could be viewed as a non-solution to the type inference problem for polymorphic recursion. On the other hand, it follows the "pay as you go" principle of language design — polymorphic recursion is encapsulated in the transparent fixed-point construct, so ML code that does not need polymorphic recursion may continue to employ type inference.

## 4.2 Implementing Generalized Datatypes

Deeming opaque recursive modules a useful generalization of ML datatypes, we would like to make them practical by replacing the use of equi-recursive constructors with iso-recursive constructors. First, in Section 4.2.1, we will discuss how iso-recursive constructors work. Then, in Sections 4.2.2 and 4.2.3, we will explore a technique for implementing datatype-generalizing opaque fixed-points using iso-recursive constructors.

### 4.2.1 Iso-Recursive Constructors

Despite their practicality, iso-recursive constructors have not been studied as thoroughly as equi-recursive constructors in the type theory literature (perhaps because they are not real fixed-points), and so there are fewer standard formalisms to go by[2]. To ground our presentation in reality, we will begin with the limited form of iso-recursive constructor that appears in the Middle Intermediate Language (MIL) of the TILT compiler [19], and then generalize it to suit our purposes. We believe that our generalized iso-recursive constructors preserve the decidability of the MIL, but a proof will not be attempted here.

In the MIL theory, the iso-recursive constructor is defined only at the kind of a pair of types (i.e. $T{\times}T$, which is easily generalized in practice to $n$-tuple kinds) and takes the form $\mu_{\cong}(\alpha,\beta).(c_1,c_2)$. The typing rule is the same as for equi-recursive constructors, except that there is no contractiveness condition. Modulo $\beta\eta$-equivalence, an iso-recursive constructor can only be equivalent to another iso-recursive constructor, on the condition that their bodies are equivalent.

$$\frac{\Gamma[\alpha:T][\beta:T]\vdash c_1:T \qquad \Gamma[\alpha:T][\beta:T]\vdash c_2:T}{\Gamma\vdash \mu_{\cong}(\alpha,\beta).(c_1,c_2):T{\times}T}$$

$$\frac{\Gamma[\alpha:T][\beta:T]\vdash c_1\equiv c_3:T \qquad \Gamma[\alpha:T][\beta:T]\vdash c_2\equiv c_4:T}{\Gamma\vdash \mu_{\cong}(\alpha,\beta).(c_1,c_2)\equiv \mu_{\cong}(\alpha,\beta).(c_3,c_4):T{\times}T}$$

One may construct an element of type $\mu_{\cong}(\cdots).1$ or $\mu_{\cong}(\cdots).2$ with a `roll` expression and deconstruct one with an `unroll` expression. The typing rules are straightforward:

$$\frac{\Gamma\vdash c:T \qquad \Gamma\vdash c\equiv \mu_{\cong}(\alpha,\beta).(c_1,c_2).i:T \qquad \Gamma\vdash e:T(c_i[c.1,c.2/\alpha,\beta])}{\Gamma\vdash \mathtt{roll}_c(e):T(c)}$$

[2]The authors would appreciate pointers to any references on type theory of iso-recursive constructors.

$$\frac{\Gamma \vdash e : T(c) \qquad \Gamma \vdash c \equiv \mu_{\cong}(\alpha,\beta).(c_1,c_2).i : T}{\Gamma \vdash \mathtt{unroll}_c(e) : T(c_i[\mu_{\cong}(\alpha,\beta).(c_1,c_2).1, \mu_{\cong}(\alpha,\beta).(c_1,c_2).2/\alpha,\beta])}$$

The MIL formalism is sufficient for expressing ML datatypes, but to replace equi-recursive constructors, we need iso-recursive constructors at arbitrary higher kind. The generalized construct can take the form $\mu_{\cong}\alpha{:}\kappa.c$, and the corresponding typing and equivalence rules are easy:

$$\frac{\Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[\alpha : \kappa] \vdash c : \kappa}{\Gamma \vdash \mu_{\cong}\alpha{:}\kappa.c : \kappa}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \ \mathit{kind} \qquad \Gamma[\alpha : \kappa_1] \vdash c_1 \equiv c_2 : \kappa_1}{\Gamma \vdash \mu_{\cong}\alpha{:}\kappa_1.c_1 \equiv \mu_{\cong}\alpha{:}\kappa_2.c_2 : \kappa_1}$$

The interesting question is how to generalize the typing rules for `roll` and `unroll`. The problem is that rolls and unrolls only mediate between constructors of kind type (obviously, since they are term-level constructs). In the case of the MIL, the pair kind limited the $c$ in $\mathtt{roll}_c(e)$ to be either a first or second projection of a $\mu_{\cong}$. In the general case, we propose to allow $c$ to be any series (or "path") of destructions — in our calculus, either projections from a pair or function applications — starting with a $\mu_{\cong}$ and resulting in a constructor of kind $T$. Formally, we define these paths of destruction $P$ as follows:

$$P ::= \bullet \mid P\,c \mid P.1 \mid P.2$$

The $\bullet$ signifies the "head" of the path where the $\mu_{\cong}$ will be substituted, and the notation $P\{c\}$ represents $P$ with $c$ as the head. The syntax for paths does not enforce either that the head is a $\mu_{\cong}$ or that that the path has kind $T$, but the generalized typing rules for `roll` and `unroll` do:

$$\frac{\Gamma \vdash c \equiv P\{\mu_{\cong}\alpha{:}\kappa.c'\} : T \qquad \Gamma \vdash c'' \equiv P\{c'[\mu_{\cong}\alpha{:}\kappa.c'/\alpha]\} : T \qquad \Gamma \vdash e : T(c'')}{\Gamma \vdash \mathtt{roll}_c(e) : T(c)}$$

$$\frac{\Gamma \vdash c \equiv P\{\mu_{\cong}\alpha{:}\kappa.c'\} : T \qquad \Gamma \vdash c'' \equiv P\{c'[\mu_{\cong}\alpha{:}\kappa.c'/\alpha]\} : T \qquad \Gamma \vdash e : T(c)}{\Gamma \vdash \mathtt{unroll}_c(e) : T(c'')}$$

In practice, the unrolled constructors $c''$ can be obtained by $\beta$-normalizing $P\{c'[\mu_{\cong}\alpha{:}\kappa.c'/\alpha]\}$.

### 4.2.2 The Coercion Calculus

Recall the phase-splitting interpretation of opaque fixed-points detailed in Section 3.1. The dynamic part of the module body, $e$, has type $\sigma[(c[\mu_{\equiv}s^{\mathsf{c}}{:}\kappa.c/s^{\mathsf{c}}])/\alpha]$, and the dynamic part of the resulting phase-split structure has type $\sigma[\mu_{\equiv}s^{\mathsf{c}}{:}\kappa.c/\alpha]$. Equi-recursiveness makes the constructors being substituted for $\alpha$ here equal, so $e$'s actual type matches the required type of the phase-split result and the phase-splitting rule was able to just wrap $e$ with a term-level fixed-point. Suppose now that we replace the equi-recursive constructor with an iso-recursive constructor. The types are no longer equal, so we need a way to coerce $e$ from its actual type to the required type, preferably without any added run-time cost for the coercion.

The approach that we have investigated is to use Crary's "coercion calculus" [3]. The broader goal of the coercion calculus is to eliminate inclusive subtyping and bounded quantification in type-directed compilers, without introducing the run-time costs of coercions inherent in the well-known "Penn interpretation" of Breazu-Tannen *et al.* [2]. Crary achieves this by putting the subtyping coercion functions into a separate syntactic class so that they can be easily discarded before run-time.

Crary's system is appealing for our purposes because it can generate static coercions between any two *extensionally equivalent* types. (By extensionally equivalent, we mean that both types are inclusive subtypes of each other.) The coercion calculus is limited, however, in that it is only defined for a type system that lacks higher-order type constructors. In Section 4.2.3, we will specify a normal form in which generalized datatype modules can be written so that the type of the code in the module body and the type of the code in the phase-split form are indeed extensionally equivalent. Fortunately, for this restricted class of normal form modules, the limitations of the calculus can be overcome by replacing its simple roll/unroll coercions with our more general ones that handle $\mu_{\simeq}$'s of higher kind. We now give some idea of how coercions work.

Coercions $q$ are like witnesses to casts from a subtype to a supertype, but they comprise a separate syntactic class from term-level functions in order to simplify their eventual elimination:

$$q ::= \mathtt{id} \mid \mathtt{roll}_c \mid \mathtt{unroll}_c \mid q_1 \to q_2 \mid q_1 \times q_2 \mid q_1 + q_2 \mid \forall\alpha{:}\kappa.q$$

$\mathtt{id}$ is the identity coercion. $\mathtt{roll}_c$ and $\mathtt{unroll}_c$ are as we have defined them. The rest of the coercions correspond to arrow (function), product, sum and universally quantified types, respectively. The typing judgment for coercions takes the form $\Gamma \vdash q : \sigma_1 \Rightarrow \sigma_2$, signifying that $q$ is a coercion from $\sigma_1$ to $\sigma_2$ in context $\Gamma$. The inference rules are shown below:

$$\frac{\Gamma \vdash \sigma \ \ \mathit{type}}{\Gamma \vdash \mathtt{id} : \sigma \Rightarrow \sigma} \qquad \frac{\Gamma \vdash c \equiv P\{\mu_{\simeq}\alpha{:}\kappa.c'\} : T \qquad \Gamma \vdash c'' \equiv P\{c'[\mu_{\simeq}\alpha{:}\kappa.c'/\alpha]\} : T}{\Gamma \vdash \mathtt{roll}_c : T(c'') \Rightarrow T(c)}$$

$$\frac{\Gamma \vdash c \equiv P\{\mu_{\simeq}\alpha{:}\kappa.c'\} : T \qquad \Gamma \vdash c'' \equiv P\{c'[\mu_{\simeq}\alpha{:}\kappa.c'/\alpha]\} : T}{\Gamma \vdash \mathtt{unroll}_c : T(c) \Rightarrow T(c'')}$$

$$\frac{\Gamma \vdash q_1 : \sigma_1' \Rightarrow \sigma_1 \qquad \Gamma \vdash q_2 : \sigma_2 \Rightarrow \sigma_2'}{\Gamma \vdash q_1 \to q_2 : \sigma_1 \to \sigma_2 \Rightarrow \sigma_1' \to \sigma_2'} \qquad \frac{\Gamma[\alpha : \kappa] \vdash q : \sigma_1 \Rightarrow \sigma_2}{\Gamma \vdash \forall\alpha{:}\kappa.q : \forall\alpha{:}\kappa.\sigma_1 \Rightarrow \forall\alpha{:}\kappa.\sigma_2}$$

$$\frac{\Gamma \vdash q_1 : \sigma_1 \Rightarrow \sigma_1' \qquad \Gamma \vdash q_2 : \sigma_2 \Rightarrow \sigma_2'}{\Gamma \vdash q_1 \times q_2 : \sigma_1 \times \sigma_2 \Rightarrow \sigma_1' \times \sigma_2'} \qquad \frac{\Gamma \vdash q_1 : \sigma_1 \Rightarrow \sigma_1' \qquad \Gamma \vdash q_2 : \sigma_2 \Rightarrow \sigma_2'}{\Gamma \vdash q_1 + q_2 : \sigma_1 + \sigma_2 \Rightarrow \sigma_1' + \sigma_2'}$$

Coercions are folded into our core calculus by adding a new term-level form, $q\,e$, that applies coercion $q$ to expression $e$. The typechecking rule for coercion applications is the obvious thing:

$$\frac{\Gamma \vdash q : \sigma_1 \Rightarrow \sigma_2 \qquad \Gamma \vdash e : \sigma_1}{\Gamma \vdash q\,e : \sigma_2}$$

The dynamic semantics of coercion applications are incorporated into the operational semantics of the core calculus through a separate "canonicalization" judgement, which takes applications of a coercion to a value, $q\,v$, and translates them to a canonical form. This organization facilitates a formalization of the idea that coercion applications are static. Specifically, when types, type abstractions and coercions are erased at code generation time, a coercion application $q\,v$ and its canonical form — in essence, the result of evaluating the coercion application — are shown to become the same untyped code. See Crary [3] for more details.

Let's look at what coercion is generated for the `List` example. Let $s = \mu_{\simeq}\alpha{:}T.1 + \mathit{int} \times \alpha$, and let $t = 1 + \mathit{int} \times s$. Here, $s$ corresponds to `List.t` and $t$ corresponds to the definition of type `t` in the module body. The code in the module body (consisting of the definitions of `nil`, `cons` and `expose`) has type

$$t \times (\mathit{int} \times s \to t) \times (t \to 1 + \mathit{int} \times s)$$

while the type we need to coerce to is

$$s \times (int \times s \to s) \times (s \to 1 + int \times s)$$

Since $s$ and $t$ are extensionally equivalent, the coercion judgment will generate the following coercion to be applied to the module body:

$$\texttt{roll}_s \times (\texttt{id} \to \texttt{roll}_s) \times (\texttt{unroll}_s \to \texttt{id})$$

Note here how arrow subtyping — contravariant in the argument type, covariant in the result type — yields a $\texttt{roll}_s$ in the result of `cons` and an $\texttt{unroll}_s$ in the argument of `expose`.

It is interesting to observe that if we use the canonicalization judgment to find the canonical form of the application of the above coercion to the `List` module body, we obtain precisely the Harper-Stone interpretation of the integer list datatype given in Section 4.1.1. For instance, what the arrow coercion $\texttt{id} \to \texttt{roll}_s$ does, when applied to the `cons` function, is to leave the argument alone and apply a $\texttt{roll}_s$ to the result. Likewise, the `nil` value gets `roll`ed, and the argument to the `expose` function gets `unroll`ed while the result is unchanged.

### 4.2.3 Generalized Datatype Normal Form

Define an opaque recursive module to be in *generalized datatype normal form* if:

1. The body (and, recursively, its substructures) consists of only two kinds of things: transparent and abstract type declarations.

2. A transparent type declaration is simply a type constructor definition, which is restricted to appear transparent in the rds and may therefore not contain any references to the recursive module variable.

3. An abstract type declaration consists of a type constructor definition, which is restricted to appear abstract in the rds, of the form

$$t = \lambda\alpha{:}\kappa.\, \Sigma\{l_1 \mapsto c_1, \cdots, l_n \mapsto c_n\}$$

   where the $\Sigma$ notation indicates a labeled sum, with each $l_i$ labeling a branch, and where the $c_i$ (in a dual requirement to the transparent types) may not contain any references to other types defined *in the body*. (Note: Non-polymorphic type definitions can be written by using $\kappa = 1$.) Along with the type definition are $n$ constructor functions for $t$, the $i$-th one having type $\forall\alpha{:}\kappa.\, c_i \to t(\alpha)$ (or just $\forall\alpha{:}\kappa.\, t(\alpha)$ if $c_i = 1$), and a destructor of type $\forall\alpha{:}\kappa.\, t(\alpha) \to \Sigma\{l_1 \mapsto c_1, \cdots, l_n \mapsto c_n\}$. The constructors are implemented as simple polymorphic injections into the sum and the destructor as the polymorphic identity. Abstract type declarations will also be referred to as datatype declarations, for the obvious reason.

We will not formalize a normalization process here, but it is not hard to see how "generalized datatypes" of the kind discussed in Section 4.1.2 can be converted to this normal form. Essentially, (non-uniform) datatypes defined mutually recursively through ML's `datatype` mechanism need to be separated and turned into plain polymorphic sums by making all mutually recursive references to themselves or one another be module-recursive. All the types defined in the module body are accessible in the signature of the recursive module variable, so this is not a problem.

It can also be shown that the coercion judgment discussed in the previous section will generate coercions for normal form modules that are very similar in form to the coercion generated for the

`List` example. It is worthwhile noting that the restriction on the form of datatype declarations — specifically, the full module-recursiveness of the $c_i$'s — is important for ensuring that we can use the identity coercion `id` in precisely the places it occurred in the coercion for `List`, that is, in the argument types of the constructor functions and the result type of the destructor.

## 4.3 Comparison of Rds Constructs

In theory, transparent rds's are more expressive than opaque rds's because they use equi-recursive constructors to encode transparent module-recursive type definitions. In practice, however, the need for equi-recursiveness is an impediment to the success of transparent rds's, whereas the simplicity of opaque rds's makes them readily usable. The question is whether the static-on-static dependencies that transparent rds's support can be limited in a way that does not require equi-recursiveness but still gives transparent rds's the expressive edge. In our estimation the answer is no, but the argument for or against transparent rds's is wholly dependent on whether one follows the opaque or transparent interpretation of datatype specifications in signatures. (This is yet another way in which recursive modules and datatypes are inextricably linked.) We will therefore first compare the two datatype interpretations before arguing for the superiority of opaque rds's.

### 4.3.1 Transparent and Opaque Interpretations of Datatypes

In the Harper-Stone interpretation of Standard ML, datatype specifications appearing in signatures are elaborated in such a way that the (recursive) implementations of the datatypes are held abstract. This is termed the "opaque interpretation" of datatypes. The problem with making datatypes abstract is that uses of the datatype constructors and destructor cannot be safely inlined, and a function call is a heavy cost to pay for a sum injection or a no-op (rolls and unrolls are generally implemented as no-ops).

One solution, incorporated into the FLINT compiler for Standard ML [23] and later into the TILT compiler, is to use a transparent interpretation of datatypes, exposing the iso-recursive sum implementation. While this solves the efficiency problem with the opaque interpretation, Crary *et al.* have pointed out that it introduces a separate problem, namely that certain kinds of valid Standard ML programs fail to typecheck [4]. For example, suppose that the argument signature of a functor `F` includes the specification of a type `t1` and a datatype `t2` that refers to it:

```
functor F (X: sig
                type t1
                datatype t2 = C | D of int * t1
                ...
              end) = ...
```

Then, suppose `F` is applied to a pair of mutually recursive datatypes `t1` and `t2` defined as follows:

```
datatype t1 = A | B of string * t2
     and t2 = C | D of int * t1
```

In the opaque interpretation, the functor application is valid because `t2` is abstract in the argument signature of `F` and the constructors and destructor of `t2` have the correct types. In the transparent interpretation, the argument signature specifies that `t2` must have type $\mu_{\cong}\alpha.1 + \mathit{int} \times \texttt{t1}$. However, the actual argument types `t1` and `t2` are defined as the first and second projections from an iso-recursive constructor $t$ of kind $T \times T$: $t = \mu_{\cong}(\alpha, \beta).(1 + \mathit{string} \times \beta, 1 + \mathit{int} \times \alpha)$. Under the simple equivalence rule for iso-recursive constructors that was given in Section 4.2.1, there is no way to equate the actual and required definitions for `t2`, so the functor application is not valid under the transparent interpretation of datatypes.

One way to remedy this problem, as detailed by Crary *et al.* [4], is to use what they refer to as *Shao's equation* (after its original implementation by Shao in the FLINT compiler). For iso-recursive types, the equation can be written as follows:

$$\mu_{\cong}\alpha.\sigma \equiv \mu_{\cong}\alpha.(\sigma[\mu_{\cong}\alpha.\sigma/\alpha])$$

For iso-recursive constructors of tuple kind (the only kind of iso-recursive constructors generated from ML datatypes), the rule has not to our knowledge been formalized, but can be described in words as: "Projections from iso-recursive constructors are equal if their unrollings are equal." (Note that this does not imply that a type is equal to its unrolling, only that two types are equal if they unroll to equal things.) Shao's equation solves the problem in the above example. The actual definition of `t2` is $t.2$. The required definition of `t2` is $\mu_{\cong}\alpha.1 + int \times t.1$. When unrolled, these both become $1 + int \times t.1$, so Shao's equation dictates that the `t2`'s match.

While Shao's equation is easy to implement and is currently employed in the TILT compiler, it is a troublesome solution because the effect the equation has on the type system is not understood well at all. Petersen *et al.* [20] are developing a different approach that returns to the opaque interpretation of datatypes and involves the coercion calculus. Assume that the constructors of a datatype are all condensed into one function `hide` whose type is the reverse of the `expose` function. In other words, whereas previously each constructor would perform a sum injection followed by a roll, there is now one constructor and it just performs a roll; just as there is only one destructor and it just performs an unroll. In this version of the opaque interpretation, the constructor and destructor are merely static coercions (as in the coercion calculus), so Petersen's idea is to make this apparent in the datatype signature by declaring the constructor and destructor to be coercions instead of ordinary functions. This is actually more complicated than it sounds, mainly because adding coercion variable declarations requires coercions to be threaded through the type system (and the compiler) in non-trivial ways. The result, however, is that applications of datatype constructors and destructors that cannot be inlined due to datatype opacity will eventually be eliminated because they are known to be coercion applications.

### 4.3.2 Datatypes and Rds's

Whether one adopts the opaque or transparent interpretation of datatypes has a major effect on which kind of rds is more appropriate. First, suppose that datatype specifications are transparent and Shao's equation is used to solve the problems discussed above. In this situation, CHP conjecture, based on a small example, that equi-recursiveness is completely eliminable for transparent rds's whose static-on-static recursive dependencies are restricted to occur inside datatypes, i.e. underneath iso-recursive constructors. It seems likely that this "datatype-on-static" restriction, which is similar to our "generalized datatype normal form" for opaque fixed-points, works in general. (We refer the reader to CHP for a detailed description of equi-recursiveness can be eliminated.) However, 1) the success of the CHP restriction relies on Shao's equation, and 2) transparent rds's must be fully transparent. The latter is a particularly odd requirement to justify to an ML programmer.

Suppose now that we adopt the opaque interpretation of datatypes. Since datatypes are abstract, references to the recursive module variable within a datatype specification are purely dynamic-on-static, occurring in the types of the constructors and destructor. It is therefore perfectly acceptable for datatype-on-static references to occur in an opaque rds. Thus, other than the prohibition of static-on-static references in ordinary type definitions, there are no restrictions on opaque rds's.

The only distinction between the opaque and transparent interpretations of datatypes that is visible to the ML programmer is that the opaque interpretation enforces datatype generativity, which is of questionable import either way. We claim therefore that transparent rds's under CHP's datatype-on-static restriction offer no clear benefits over opaque rds's, while they require full transparency. In other words, the very restriction suggested to make transparent rds's practical also make them a special case of opaque rds's. Betting on the success of Petersen's work on efficient compilation of the opaque interpretation of datatypes, we see no convincing arguments in favor of transparent rds's.

### 4.4 Putting It Together

Having analyzed the fixed-point and rds constructs on practical criteria, we now fit them together into a general design for a recursive module extension to Standard ML. Our proposal consists of one simple fixed-point mechanism and one simple rds mechanism. We will describe at a high level their elaboration into the opaque and transparent constructs we have studied. Since recursive modules interact with so many other features in the language, it should be noted that the details of our proposal are not set in stone. Rather, we intend this to serve not as a final solution, but as a starting point for language designers and implementors.

The rds construct is a signature expression of the form

```
sig rec <modname> in <specs> end
```

where `<modname>` is the recursive module variable and `<specs>` is a sequence of type, datatype, value and module specifications comprising the signature body. Datatype and value specifications are permitted to contain references to `<modname>`, but transparent type specifications are not. The `sig rec` construct is implemented directly as an opaque rds — we are assuming the opaque interpretation of datatypes — and obeys the rds roll and unroll rules presented in Section 2.4.

The fixed-point construct is a structure expression of the form

```
struct rec <modname>:<sig> in <decls> end
```

where `<modname>` is the recursive module variable, `<sig>` is the rds, and `<decls>` is a sequence of type, datatype, value and module declarations comprising the module body. The body is subject to the same restriction as rds's: datatype and value declarations are permitted to contain references to `<modname>`, but transparent type declarations are not. Elaboration of the body is quite a bit more complicated because it involves a combination of opaque and transparent fixed-points.

The basic idea is to use transparent fixed-points as the main mechanism for implementing `struct rec` since it is more appropriate for general programming, and to use opaque fixed-points to handle the datatype declarations. This approach gives us the best of both worlds: ease of programming and polymorphic recursion from transparent fixed-points, module-recursive datatypes from opaque fixed-points. The main issue in this setup is that the specified rds `<sig>` is not required to be transparent, so we need to reify it with the implementation of types in the body. To accomplish this, there is a first pass over the module body that extracts all the datatype and type declarations. These are converted into generalized datatype normal form and then "solved" with an opaque fixed-point. The solutions of the module-recursive datatypes given by the opaque fixed-point are linked back into the module body through datatype copying. For instance, suppose that the opaque fixed-point is called `Static` (for the "static" part of the module body) and a datatype `dt` is defined in substructure `A` of the body. Then `dt`'s definition, having been extracted into the definition of `Static`, is replaced in the body by

```
datatype dt = datatype Static.A.dt
```

which copies the type `dt` along with its constructors and destructor.

It is important here to mention that it is not always the case that the type declarations can be extracted and solved separately. Specifically, if the body contains module abstraction operations of the form `Mod :> Sig`, and `Sig` contains an abstract type specification such as `type t`, then writing the transparent definition of `Mod.t` separately would break abstraction. We thus require that the static part of the body, except for datatypes, be fully transparent, which bans most uses of opaque signature ascription. We will return to this restriction and its implications in Section 5 on future work.

Thanks to this restriction, however, replacing the module-recursive datatype definitions with links to their solutions in `Static` removes all module-recursiveness from the static part of the body. It is thus ripe for implementation with a transparent fixed-point, as soon as the types in `Static` are used to fill in the abstract types in the rds. The remainder of elaboration reduces to transparent fixed-point typechecking, that is, the body must satisfy the specifications of the rds and be valuable, in a context where the recursive module variable is assigned the reified rds and is not valuable.

A note on the valuability restriction: In this paper, we have directly adopted CHP's use of the fixed-point expression $\mathit{fix}(x{:}\sigma.e)$ in phase-splitting fixed-point modules, thus taking the compilation of "dynamic-on-dynamic" references in recursive modules for granted. We have done so because the primary focus of this paper has been on compiling module-recursive references to types, not terms. In practice, however, imposing the valuability restriction on the entire recursive module body may prove to be a hard sell to ML programmers. It is simply an unintuitive hindrance, for instance, to be prohibited from defining any top-level non-module-recursive ref cells as one might use to implement "flags" (e.g. `val x = ref true`) in a recursive module. This might suggest that if module-recursive references were restricted to occur under $\lambda$-abstractions, as is implied by the valuability restriction, we could get away without the full restriction. A simple example demonstrates that this is a fallacy (here, `M` is the recursive module variable, and the following code appears in the module body):

```
fun f () = M.x
val x = f()
```

The valuability restriction catches the infinite loop in the definition of `x` because the non-valuability of `M` means that `f` is a partial function and its application is not valuable. Finding an intuitive middle ground between these two levels of restrictions will be an important prerequisite for the acceptance of our recursive module proposal.

Another noteworthy issue involves the ability of the module body to contain declarations that are not specified by the rds. So long as the body provides all the components the rds requires, it seems intuitive that providing more should be allowed, in the same way that a module in ML may be ascribed a signature that specifies a proper subset of the components the module actually provides. However, allowing for the body to declare more than is specified introduces some slight complication because, in a transparent fixed-point, the body must match the rds exactly. This can be solved by first generating a total functor that coerces a module from the full signature of the module body to the rds. Then, the application of that functor coercion to the original module body matches the rds and can be used as the body of the transparent fixed-point. Harper and Stone [10] show how to generate such functor coercions automatically, as they are the same sort of coercions that are generated during the elaboration of signature ascription and functor application.

Finally, for the design to be successful, there must be a straightforward way to explain recursive module typechecking to the ML programmer without referring to opaque and transparent fixed-points. Aside from the valuability restriction, which seems unavoidable, typechecking for our proposed recursive modules is actually very intuitive and can be presented as follows:

1. The purpose of associating an rds with a `struct rec` is to specify what the module body is allowed to know about itself when it makes module-recursive references.

2. Any types that appear abstract in the rds are really equal to their implementations in the module body.

3. Module-recursive references may occur in `datatype` declarations and inside function definitions, but nowhere else.

4. Opaque ascription and functor applications resulting in the definition of abstract types are not permitted in the module body.

5. Assume (as alpha-equivalence allows) that the recursive module variable is the same in both the recursive module and the associated rds. Then the recursive module typechecks if the module body provides the specifications in the rds body and the module body is valuable.

The last part of the intuitive typechecking description is a rewording of the "fundamental property of recursive modules" studied in Section 3.4. Rather than saying that the module body should match the rds, we say it should match the body of the rds, which is an equivalent condition for transparent fixed-points but easier to understand.

# 5 Directions for Future Work

Throughout our entire study of recursive modules, with the exception of disallowing the use of opaque ascription in the bodies of `struct rec`'s, we have made hardly any mention of abstraction. The reason is that the phase-distinction framework of Harper, Mitchell and Moggi, which serves as the very basis of our analysis, does not model abstraction at all. It is not a mere oversight: blithely phase-splitting a module abstraction (such as a module of the form `Mod :> Sig`) would expose the implementation of the static part of the module, breaking abstraction. We have already discussed the importance of ameliorating the harshness of the valuability restriction as one avenue of future work. In this section we will look at two other directions for future research involving the interplay of recursive modules, phase-splitting and abstraction — namely, separate compilation and higher-order modules.

## 5.1 Separate Compilation

A desired feature of recursive modules that our proposal does not provide is the ability to separately compile mutually recursive modules. CHP describe how to separately compile substructures of a transparent recursive module using functors. The idea is that each substructure can be written separately by abstracting over the recursive module variable (assuming that the substructures only refer to each other through the recursive module variable). Then, a transparent recursive module is used to join the implementations, instantiating each functor with the actual recursive module variable. For instance, in the `ExpDec` example, `Exp` and `Dec` could be compiled separately as follows:

```
functor ExpFun (structure ExpDec : EXPDEC) = ...
functor DecFun (structure ExpDec : EXPDEC) = ...
structure ExpDec = tfix (ExpDec : EXPDEC.
  structure Exp = ExpFun(ExpDec)
  structure Dec = DecFun(ExpDec)
end
```

The problem in CHP is that `EXPDEC` is required to be fully transparent. This limits the usefulness of separate compilation by forcing the implementations of all the types to be settled on beforehand.

Our proposed `struct rec` appears on its face to be more amenable to separate compilation than CHP because there is no transparency requirement on the rds. However, `struct rec` elaboration involves reification of the entire opaque rds prior to typechecking of (the dynamic part of) the body, so it would seem that the implementation of all the types still needs to be known in advance in order to compile any one submodule separately. In fact, though, this is an instance where our proposal is overly simplistic. The purpose of the two-pass elaboration is to ensure that all references to a type defined in the recursive module, whether they be local or module-recursive references, are equal (e.g. there is no "opaque" distinction between `t` and `List.t`). Reifying the entire rds before elaborating the body the second time is an overly crude way of achieving this goal because it makes all the submodules fully transparent to one another.

One could imagine a more refined design taking the form of a bundle of mutually recursive module declarations (comprising one big module), each of whose implementations would be transparent to itself but opaque to all the other modules. In order to prevent the inter-module opacity from re-introducing the undesirable "opaque" distinction, we could require that any inter-module references be interpreted as "bundle-recursive" references (i.e. module-recursive references to the recursive bundle variable) rather than local references within the bundle. A simple syntax might be

```
structure rec M₁ :> S₁ = implementation of M₁
and  ...  and Mₙ :> Sₙ = implementation of Mₙ
```

where the associated rds implied by this notation is

```
sig rec M in                     (* Inter-module references replaced by *)
  structure M₁ : S₁[M.Mᵢ/Mᵢ]ᵢ∈1..n   (* bundle-recursive references to M.  *)
  ...
  structure Mₙ : Sₙ[M.Mᵢ/Mᵢ]ᵢ∈1..n
end
```

Although the elaboration of the whole bundle still requires the reification of the entire rds because of our interpretation of recursive modules into transparent fixed-points, the *typechecking* of each particular $M_i$ would only require type information gleaned from $M_i$ alone in order to reify $S_i$.

A design along these lines lends itself more to separate compilation than our original proposal because each submodule need know nothing about the implementation of the types in the other submodules. In addition, by making the submodules opaque to one another, it can be viewed as a way of introducing abstraction into the body of a recursive module, which is not possible in our original proposal. Nevertheless, to actually separate the submodules, we still cannot use functors in the straightforward way that CHP did because functors are incapable of capturing the special two-pass elaboration of `struct rec`. We would need to develop a new separate compilation mechanism, which might only be able to do separate typechecking, not full compilation. As pointed out in recent work by Pierce and Harper [21], functors may not be appropriate as a general separate compilation mechanism anyway, so a special construct for separately compiling recursive modules may be in order.

In any case, it should be understood that separate compilation and separate typechecking are predicated on the specification of the rds *a priori*. Since the rds specifies what each submodule may know about the others, changes to it potentially affect the validity of all submodules and may thus require recompilation of all of them. On the flip side, some may not care so much about separate compilation as about the ability to simply write mutually recursive modules in separate files that are compiled together. Such a "separate-location" mechanism is not fundamentally any different from the `structure rec` bundle declaration sketched above.

### 5.2 Higher-Order Modules

A classic problem that nicely motivates recursive modules and that the usual workarounds described in Section 1 fail to work around is *data-structural bootstrapping* (see Okasaki [18]). For example, suppose we wish to implement a "set of sets", defined to be a datatype `sos` that is either an integer, say, or a set of `sos`'s constructed by means of the predefined total functor `MakeSetFn`. Assume that `MakeSetFn` is defined in the following way:

```
signature KEY = sig type key; val compare : key * key -> order end
signature SET = sig type element; type set; (* set functions *) end
functor MakeSetFn (Key:KEY) : SET where type element = Key.key
 = struct ... end
```

Now, we can define `sos` using `struct rec` rather easily:

```
signature SOS = sig rec Sos in
  datatype sos = Int of int | Set of Sos.SosSet.set
  structure SosSet : SET where type element = sos
end
structure Sos = struct rec Sos : SOS in
  datatype sos = Int of int | Set of Sos.SosSet.set
  structure SosKey = struct
    type key = sos
    fun compare (sos1,sos2) = ...
  end
  structure SosSet = MakeSetFn(SosKey)
end
```

It is important here that the datatype definition come before the functor application so that the transparent type references `type element = sos` in `SOS` and `type key = sos` in `Sos` are *not* module-recursive.

Unfortunately, our success in coding this example was subtly dependent on our use of transparent ascription in defining `MakeSetFn`. If we had used opaque ascription and hidden the implementation of the set type, the application of `MakeSetFn` in the body of `Sos` would have been invalid. The reason is that functors in ML are generative, that is, every application of a functor that has abstract types in its result signature generates a fresh abstract type, and abstract types are not allowed in the body of a `struct rec`.

One might expect this problem to be solved by Leroy's applicative functors [13], which are used in the Objective CAML dialect of ML [14]. Applicative functors do not generate fresh types every time they are applied, so even if `MakeSetFn` were defined opaquely, every application of `MakeSetFn(SosKey)` would result in the same `set` type. To achieve this, Leroy allows functor applications to occur in type paths, so the result of `MakeSetFn(SosKey)` has the fully transparent signature

```
sig type element = sos; type set = MakeSetFn(SosKey).set; ... end
```

If `MakeSetFn` were generative, which is the only possibility in Standard ML, `MakeSetFn(SosKey).set` would not even be well-defined. There is still a problem, however: Leroy restricts functor applications in paths to be in named form, i.e. they can only refer to structure paths, not actual structure definitions. Our elaboration for `struct rec` requires that we be able to extract the definition of `set` into the opaque fixed-point that solves the static part of the module. In that opaque fixed-point, `SosKey` will only consist of the static part of the definition of `SosKey` in the module body, so the functor application below is no longer valid since the static part of `SosKey` does not have signature `KEY`:

```
structure StaticPartOfSos = ofix (...
  datatype sos = ...
  structure SosKey = struct type key = sos end
  structure SosSet = struct
    type element = sos
    type set = MakeSetFn(SosKey).set   (* Error: SosKey has wrong signature *)
  end
)
```

Leroy's applicative functors were motivated by a desire to add type-theoretic support to Standard ML for fully transparent higher-order modules in the presence of abstraction [15]. Shao has also presented a type theory for higher-order modules that extends applicative functors and solves the problems we encountered with applicative functors using "fully syntactic signatures" [24]. In his system, the `set` type above would be expressed as $\overline{\texttt{MakeSetFn}}$({`type key = sos`}).`set`, where $\overline{\texttt{MakeSetFn}}$ denotes the static part of `MakeSetFn` — in essence, *Fst* (`MakeSetFn`). This type could be extracted into the opaque fixed-point with no difficulty.

The reason Shao's system is successful, as he points out, is that his general approach is "to adapt and incorporate the phase-splitting interpretation of higher-order modules into a surface module calculus". However, Shao does not actually make use of a full phase-distinction calculus, and it is unclear why not. One reason may be that he does not use singleton kinds to express type definitions in signatures, and it is difficult to equate an arbitrary translucent signature with a primitive signature of the form $[\alpha{:}\kappa.\sigma]$ without singleton kinds.

During our study of recursive modules, one of our sanity checks has been to attempt to formalize the elaboration of our ML extension in the Harper-Stone framework. The lack of higher-order modules, let alone phase-splitting, in the HIL — the high-level intermediate language that serves as the target of Harper-Stone elaboration — made extension of the HIL with fixed-point and rds constructs rather difficult. The opaque fixed-point typechecking rule, for example, relies on the ability to compare the phase-split forms of the rds and body signatures. The distinction between static-on-static and dynamic-on-static dependencies that defines the rds typechecking rule is hard to express in a calculus without phase-splitting, as is the equivalence between rds's and other signatures that the non-standard equational rules of HMM induce. We therefore believe it would be a useful contribution to the theory and practice of recursive and higher-order modules to devise a way to combine a full phase-splitting calculus along the lines of HMM with a mechanism for abstraction.

## 6 Related Work

Flatt and Felleisen [8] have studied recursive modules in the form of *units*. Units are essentially first-class modules, i.e. modules that can be passed as values, dynamically linked and "invoked" at run-time. Each unit consists of a set of import declarations, a body containing type and function definitions, an initialization expression that is executed when the unit is invoked, and a set of export declarations. The recursive aspect of units comes into play in the linking of two units into a compound unit: the first unit's exports may be used to satisfy the second unit's imports, while the second unit's exports are simultaneously used to satisfy the first unit's imports.

Units were designed to serve as the module system for the MzScheme programming environment for Scheme, which is no surprise considering the emphasis on *dynamic*, *first-class* modules. Only after Flatt and Felleisen define the system over a dynamically-typed (i.e. untyped) core language do they consider the implications of type declarations, so issues involving types are not at the center of the design philosophy. Although the extensions to units for dealing with type declarations are reasonable (e.g. using type dependency declarations to check that there are no cyclic dependencies between transparent type definitions in separate units), the fact that units are first-class precludes them from providing fine-grained control over type propagation. For instance, as Flatt and Felleisen point out, ML-style type sharing is not supported by units, resulting in crude resolutions to situations like the "diamond import problem". On the other hand, units make separate compilation easy.

Duggan and Sourelis [6, 7] have presented a recursive module extension to ML in the context of *mixin modules*. Mixins were originally studied [6] as a mechanism akin to virtual types but in a non-object-oriented (or "data-oriented") setting. They are defined using a special `mixin` construct consisting of two sections: the first containing mutually recursive type and function definitions, the second containing initialization code as in *units*. One can write two separate mixins, each of which defines different branches of the same datatype(s) along with functions that operate over the branches it defines. Then, the two mixins can be combined with the `mixcomp` construct, which "merges" the type and term fixed-points. Typically, in an object-oriented language, one could achieve something similar by defining an abstract or virtual class with each of the mixins as a subclass, but mixins have the advantage that they do not require runtime type checks. Programming with mixins enables a certain kind of recursive module programming. Mutually recursive modules can be written separately as mixins by replacing inter-module references with references to local "stub" definitions of datatypes and functions. When a `mixcomp` is performed, these stubs get merged with the actual definitions of types and functions from the other modules.

In their later work [7], Duggan and Sourelis propose a `mixlink` construct that allows for more natural recursive module programming and has some interesting correspondences with our proposal. `mixlink` is used to declare (or "link") a bundle of mutually recursive mixin modules, which are divided into two groups. The typing rules differ for the two groups in very subtle ways, but it appears that the first group are like a simplified version of the `structure rec` declarations sketched in Section 5.1, while the second group are similar to opaque fixed-points in that they introduce an "opaque" distinction between the types defined in their bodies and bundle-recursive references to those types. `mixlink` is capable of expressing the "set of sets" example discussed in Section 5.2, but requires the functor application to occur in the second section of the `mixlink`. Unfortunately, the construct is not amenable to separate compilation in the way that `mixcomp` is. Moreover, it is simply unclear why the two different styles of recursive module programming afforded by `mixcomp` and `mixlink` are tied to the same concept of a mixin module.

Claudio Russo has designed a recursive module extension for Standard ML and implemented it in the Moscow ML 2.00 compiler [1]. The only documentation seemingly available regarding this extension is on pages 23–24 of the Moscow ML Owner's Manual [22]. What can be discerned from this is that Russo's extension is cosmetically similar to ours, defining two new constructs analogous to fixed-points and rds's. In his extension, however, both module and signature constructs have the same basic syntax:

```
rec (X:<forwardsig>).  <mod | sig>
```

Here, <forwardsig> is used to give forward declarations of any types or values that are referred to module-recursively in the fixed-point body (<mod>) or rds body (<sig>). In our extension, sig rec's have no need for forward declarations because the validity of module-recursive references in the rds body can be determined through phase-splitting it. In addition, the rds associated with a struct rec supplies the signature for the entire module, not just the forward declarations.

An odd feature of Russo's extension is that "any opaque type or datatype specified in the signature [associated with a recursive module] must be implemented in the body [of the module] by *copying* it using a forward reference", demonstrated by the following example taken verbatim from the manual:

```
structure Ok =   (* well-typed *)
  rec (X:sig datatype t = C type u type v = int end)
  struct datatype t = datatype X.t type u = X.u type v = int end
structure Fail = (* ill-typed  *)
  rec (X:sig datatype t = C type u type v = int end)
  struct datatype t = C type u = int type v = int end
```

Perhaps the reason this seems so odd is that in our extension, the analogous encoding of structure Ok would be *ill-typed* and the analogous encoding of structure Fail would be *well-typed*. Still, it is not even clear what the definition of type u in Ok is supposed to be, or what is wrong with giving it a definition in Fail. At any rate, it indicates a marked difference from our proposal, but the extent to which the designs differ is impossible to determine from our limited understanding of the theory and details of Russo's extension.


## Acknowledgements

We would like to give special thanks to Leaf Petersen and Chris Stone for invaluable discussion and insight on all things recursive, modular and TILTed. We would also like to thank Perry Cheng and Dave Walker for helpful comments and Peter Lee for his warm guidance.


## References


[1] The Moscow ML 2.00 compiler. URL: http://www.dina.dk/~sestoft/mosml.html.

[2] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[3] Karl Crary. Typed compilation of inclusive subtyping. In *2000 ACM SIGPLAN International Conference on Functional Programming*, pages 68–81, Montreal, Canada, September 2000.

[4] Karl Crary, Robert Harper, Perry Cheng, Leaf Petersen, and Chris Stone. Transparent and opaque interpretations of datatypes. Technical Report CMU-CS-98-177, Carnegie Mellon University, School of Computer Science, November 1998.

[5] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, Atlanta, GA, May 1999.

[6] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, June 1996.

[7] 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.

[8] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998.

[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*, pages 341–354, San Francisco, California, January 1990.

[10] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, Pittsburgh, PA, June 1997. Also published as Fox Memorandum CMU-CS-FOX-97-01.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[12] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[13] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *1995 ACM Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, CA, January 1995.

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

[15] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1–32, September 1996.

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

[17] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, April 1984.

[18] Chris Okasaki. *Purely Functional Data Structures*, chapter 10. Cambridge University Press, 1998.

[19] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, December 2000.

[20] Leaf Petersen, Karl Crary, and Robert Harper. Coercions for datatypes. To be submitted.

[21] Benjamin Pierce and Robert Harper. Advanced module systems: A guide for the perplexed. Invited talk at the 2000 ACM SIGPLAN International Conference on Functional Programming, September 2000.

[22] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual (Version 2.00)*, June 2000. Available at `ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.ps.gz`.

[23] 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.

[24] Zhong Shao. Transparent modules with fully syntactic signatures. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 220–232, Paris, France, September 1999.

[25] Christopher A. Stone and Robert Harper. Deciding type equivalence for a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 214–227, Boston, MA, January 2000.

[26] Christopher Allan Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, August 2000.

# A Type Theory of HMM + Singleton Kinds + Recursive Modules

## A.1 Core Calculus

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

$$\frac{}{\Gamma \vdash T \ \ kind} \qquad \frac{}{\Gamma \vdash 1 \ \ kind} \qquad \frac{\Gamma \vdash c : T}{\Gamma \vdash \mathfrak{s}(c) \ \ kind}$$

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

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

$\boxed{\Gamma \vdash \kappa_1 \equiv \kappa_2 \ \ kind}$

$$\frac{}{\Gamma \vdash T \equiv T \ \ kind} \qquad \frac{}{\Gamma \vdash 1 \equiv 1 \ \ kind} \qquad \frac{\Gamma \vdash c_1 \equiv c_2 : T}{\Gamma \vdash \mathfrak{s}(c_1) \equiv \mathfrak{s}(c_2) \ \ kind}$$

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

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

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

$$\frac{}{\Gamma \vdash T \leq T \ \ kind} \qquad \frac{}{\Gamma \vdash 1 \leq 1 \ \ kind}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : T}{\Gamma \vdash \mathfrak{S}(c_1) \leq \mathfrak{S}(c_2) \ \ kind} \qquad \frac{\Gamma \vdash c : T}{\Gamma \vdash \mathfrak{S}(c) \leq T \ \ kind}$$

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

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

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

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\alpha \uparrow \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa_1 \ \ kind \qquad \Gamma[\alpha : \kappa_1] \vdash c : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c : \Pi\alpha{:}\kappa_1.\kappa_2} \qquad \frac{\Gamma \vdash c_1 : \Pi\alpha{:}\kappa_1.\kappa_2 \qquad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1 c_2 : \kappa_2[c_2/\alpha]}$$

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

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c.1 : \kappa_1} \qquad \frac{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c.2 : \kappa_2[c.1/\alpha]}$$

$$\frac{}{\Gamma \vdash 1 : T} \qquad \frac{\Gamma \vdash c_1 : T \qquad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \rightharpoonup c_2 : T} \qquad \frac{\Gamma \vdash c_1 : T \qquad \Gamma \vdash c_2 : T}{\Gamma \vdash c_1 \times c_2 : T}$$

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

$$\frac{\Gamma \vdash c : \kappa' \qquad \Gamma \vdash \kappa' \leq \kappa \ \ kind}{\Gamma \vdash c : \kappa} \qquad \frac{\Gamma \vdash c : T}{\Gamma \vdash c : \mathfrak{S}(c)}$$

$$\frac{\alpha \notin \mathrm{FV}(c) \qquad \Gamma \vdash \lambda\alpha{:}\kappa_1.c\,\alpha : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c : \Pi\alpha{:}\kappa_1.\kappa_2} \qquad \frac{\Gamma \vdash \langle c.1, c.2 \rangle : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

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

$$\overline{\Gamma[B]} \stackrel{\text{def}}{=} \begin{cases} \overline{\Gamma}[\alpha : \kappa] & \text{if } [B] \text{ is } [\alpha \uparrow \kappa] \\ \overline{\Gamma}[s : S] & \text{if } [B] \text{ is } [s \uparrow S] \\ \overline{\Gamma}[B] & \text{otherwise} \end{cases}$$

$$\frac{\overline{\Gamma} \vdash c \equiv c' : \kappa \qquad \Gamma \vdash c' \downarrow}{\Gamma \vdash c \downarrow \kappa}$$

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

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha \downarrow} \qquad \frac{}{\Gamma \vdash \star \downarrow}$$

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[\alpha : \kappa] \vdash c \downarrow}{\Gamma \vdash \lambda\alpha{:}\kappa.c \downarrow} \qquad \frac{\Gamma \vdash c_1 \downarrow \qquad \Gamma \vdash c_2 \downarrow}{\Gamma \vdash c_1 c_2 \downarrow}$$

$$\frac{\Gamma \vdash c_1 \downarrow \qquad \Gamma \vdash c_2 \downarrow}{\Gamma \vdash \langle c_1, c_2 \rangle \downarrow} \qquad \frac{\Gamma \vdash c \downarrow}{\Gamma \vdash c.i \downarrow} \ (i = 1, 2)$$

$$\frac{}{\Gamma \vdash 1 \downarrow} \qquad \frac{}{\Gamma \vdash c_1 \rightharpoonup c_2 \downarrow} \qquad \frac{}{\Gamma \vdash c_1 \times c_2 \downarrow}$$

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \mathit{kind} \qquad \Gamma[\alpha \uparrow \kappa] \vdash c \downarrow}{\Gamma \vdash \mu_{\equiv}\alpha{:}\kappa.c \downarrow}$$

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

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c \equiv c : \kappa} \qquad \frac{\Gamma \vdash c_2 \equiv c_1 : \kappa}{\Gamma \vdash c_1 \equiv c_2 : \kappa}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \kappa \qquad \Gamma \vdash c_2 \equiv c_3 : \kappa}{\Gamma \vdash c_1 \equiv c_3 : \kappa}$$

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

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

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

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c.1 \equiv c'.1 : \kappa_1}$$

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash c.2 \equiv c'.2 : \kappa_2[c.1/\alpha]}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : T \qquad \Gamma \vdash c_2 \equiv c_2' : T}{\Gamma \vdash c_1 \rightharpoonup c_2 \equiv c_1' \rightharpoonup c_2' : T}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : T \qquad \Gamma \vdash c_2 \equiv c_2' : T}{\Gamma \vdash c_1 \times c_2 \equiv c_1' \times c_2' : T}$$

$$\frac{\begin{array}{c}\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \equiv \kappa' \ \ kind \qquad \Gamma[\alpha : \kappa] \vdash c \equiv c' : \kappa \\ \Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[\alpha \uparrow \kappa] \vdash c' \downarrow \kappa\end{array}}{\Gamma \vdash \mu_{\equiv}\alpha{:}\kappa.c \equiv \mu_{\equiv}\alpha{:}\kappa'.c' : \kappa}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \kappa' \qquad \Gamma \vdash \kappa' \leq \kappa \ \ kind}{\Gamma \vdash c_1 \equiv c_2 : \kappa}$$

$$\frac{\Gamma \vdash c : \mathfrak{S}(c')}{\Gamma \vdash c \equiv c' : T} \qquad \frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash c \equiv c' : \mathfrak{S}(c)} \qquad \frac{\Gamma \vdash c : 1}{\Gamma \vdash c \equiv \star : 1}$$

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

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash c_1 : \Pi\alpha{:}\kappa_1.\kappa_2' \qquad \Gamma \vdash c_1 : \Pi\alpha{:}\kappa_1.\kappa_2'' \qquad \Gamma[\alpha : \kappa_1] \vdash c_1\alpha \equiv c_2\alpha : \kappa_2}{\Gamma \vdash c_1 \equiv c_2 : \Pi\alpha{:}\kappa_1.\kappa_2}$$

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

$$\frac{\begin{array}{c}\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \ \ kind \\ \Gamma \vdash c_1.1 \equiv c_2.1 : \kappa_1 \qquad \Gamma[\alpha : \kappa_1] \vdash c_1.2 \equiv c_2.2 : \kappa_2[c_1.1/\alpha]\end{array}}{\Gamma \vdash c_1 \equiv c_2 : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[\alpha \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \mu_{\equiv}\alpha{:}\kappa.c \equiv c[(\mu_{\equiv}\alpha{:}\kappa.c)/\alpha] : \kappa}$$

$$\frac{\alpha \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma \vdash c \equiv c'[c/\alpha] : \kappa \qquad \Gamma[\alpha \uparrow \kappa] \vdash c' \downarrow \kappa}{\Gamma \vdash c \equiv \mu_{\equiv}\alpha{:}\kappa.c' : \kappa}$$

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

$$\frac{\Gamma \vdash c : T}{\Gamma \vdash T(c) \ \ type} \qquad \frac{\Gamma \vdash \sigma_1 \ \ type \qquad \Gamma \vdash \sigma_2 \ \ type}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 \ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 \ \ type \qquad \Gamma \vdash \sigma_2 \ \ type}{\Gamma \vdash \sigma_1 \rightharpoonup \sigma_2 \ \ type} \qquad \frac{\Gamma \vdash \sigma_1 \ \ type \qquad \Gamma \vdash \sigma_2 \ \ type}{\Gamma \vdash \sigma_1 \times \sigma_2 \ \ type}$$

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

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

$$\frac{\Gamma \vdash \sigma \ \ type}{\Gamma \vdash \sigma \equiv \sigma \ \ type} \qquad \frac{\Gamma \vdash \sigma_2 \equiv \sigma_1 \ \ type}{\Gamma \vdash \sigma_1 \equiv \sigma_2 \ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 \ \ type \qquad \Gamma \vdash \sigma_2 \equiv \sigma_3 \ \ type}{\Gamma \vdash \sigma_1 \equiv \sigma_3 \ \ type}$$

$$\frac{\Gamma \vdash c \equiv c' : T}{\Gamma \vdash T(c) \equiv T(c') \ \ type}$$

$$\frac{\Gamma \vdash c_1 : T \qquad \Gamma \vdash c_2 : T}{\Gamma \vdash T(c_1 \rightharpoonup c_2) \equiv T(c_1) \rightharpoonup T(c_2) \ \ type} \qquad \frac{\Gamma \vdash c_1 : T \qquad \Gamma \vdash c_2 : T}{\Gamma \vdash T(c_1 \times c_2) \equiv T(c_1) \times T(c_2) \ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \ \ type \qquad \Gamma \vdash \sigma_2 \equiv \sigma_2' \ \ type}{\Gamma \vdash \sigma_1 \rightharpoonup \sigma_2 \equiv \sigma_1' \rightharpoonup \sigma_2' \ \ type} \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \ \ type \qquad \Gamma \vdash \sigma_2 \equiv \sigma_2' \ \ type}{\Gamma \vdash \sigma_1 \times \sigma_2 \equiv \sigma_1' \times \sigma_2' \ \ type}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \ \ type \qquad \Gamma \vdash \sigma_2 \equiv \sigma_2' \ \ type}{\Gamma \vdash \sigma_1 \to \sigma_2 \equiv \sigma_1' \to \sigma_2' \ \ type}$$

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

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{x \uparrow \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash \star : 1}$$

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

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

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

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash e.i : \sigma_i} \ (i = 1, 2)$$

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

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

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

$$\frac{\Gamma \vdash e : \sigma' \qquad \Gamma \vdash \sigma \equiv \sigma'\ \ \mathit{type}}{\Gamma \vdash e : \sigma}$$

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

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e \downarrow}{\Gamma \vdash e \downarrow \sigma}$$

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x \downarrow} \qquad \frac{}{\Gamma \vdash \star \downarrow}$$

$$\frac{}{\Gamma \vdash \lambda x{:}\sigma.e \downarrow} \qquad \frac{\Gamma \vdash e_1 \downarrow \sigma_1 \to \sigma_2 \qquad \Gamma \vdash e_2 \downarrow}{\Gamma \vdash e_1 e_2 \downarrow}$$

$$\frac{\Gamma \vdash e_1 \downarrow \qquad \Gamma \vdash e_2 \downarrow}{\Gamma \vdash \langle e_1, e_2 \rangle \downarrow} \qquad \frac{\Gamma \vdash e \downarrow}{\Gamma \vdash e.i \downarrow}\ (i = 1, 2)$$

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

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

## A.2 Structure Calculus

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

$$\frac{\Gamma \vdash M : [\alpha{:}\kappa.\sigma]}{\Gamma \vdash \mathit{Fst}\, M : \kappa}$$

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

$$\frac{\Gamma \vdash M \downarrow S}{\Gamma \vdash \mathit{Fst}\, M \downarrow}$$

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

$$\frac{\Gamma \vdash M : [\alpha{:}\kappa.\sigma]}{\Gamma \vdash \mathit{Snd}\ M : \sigma[\mathit{Fst}\, M/\alpha]}$$

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

$$\frac{\Gamma \vdash M \downarrow S}{\Gamma \vdash \mathit{Snd}\ M \downarrow}$$

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

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

$\boxed{\Gamma \vdash S_1 \equiv S_2 \ \ sig}$

$$\frac{\Gamma \vdash S \ \ sig}{\Gamma \vdash S \equiv S \ \ sig} \qquad \frac{\Gamma \vdash S_2 \equiv S_1 \ \ sig}{\Gamma \vdash S_1 \equiv S_2 \ \ sig}$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 \ \ sig \qquad \Gamma \vdash S_2 \equiv S_3 \ \ sig}{\Gamma \vdash S_1 \equiv S_3 \ \ sig}$$

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

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

$$\frac{\Gamma \vdash S_1 \equiv S_2 \ \ sig}{\Gamma \vdash S_1 \leq S_2 \ \ sig}$$

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

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

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

$$\frac{s : S \in \Gamma}{\Gamma \vdash s : S} \qquad \frac{s \uparrow S \in \Gamma}{\Gamma \vdash s : S}$$

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

$$\frac{\Gamma \vdash M : S' \qquad \Gamma \vdash S' \leq S \ \ sig}{\Gamma \vdash M : S} \qquad \frac{\Gamma \vdash M : [\alpha{:}\kappa'.\sigma] \qquad \Gamma \vdash \mathit{Fst}\, M : \kappa}{\Gamma \vdash M : [\alpha{:}\kappa.\sigma]}$$

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

$$\frac{s : S \in \Gamma}{\Gamma \vdash s \downarrow S}$$

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

$$\frac{\Gamma \vdash M \downarrow S' \qquad \Gamma \vdash S' \leq S \ \ sig}{\Gamma \vdash M \downarrow S} \qquad \frac{\Gamma \vdash M \downarrow [\alpha{:}\kappa'.\sigma] \qquad \Gamma \vdash \mathit{Fst}\, M : \kappa}{\Gamma \vdash M \downarrow [\alpha{:}\kappa.\sigma]}$$

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

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash M \equiv M : S} \qquad \frac{\Gamma \vdash M_2 \equiv M_1 : S}{\Gamma \vdash M_1 \equiv M_2 : S}$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 : S \qquad \Gamma \vdash M_2 \equiv M_3 : S}{\Gamma \vdash M_1 \equiv M_3 : S}$$

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

$$\frac{\Gamma \vdash M_1 \equiv M_2 : S' \qquad \Gamma \vdash S \leq S' \ \ \mathit{sig}}{\Gamma \vdash M_1 \equiv M_2 : S}$$

## A.3 Recursive Module Calculus

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

$$\frac{\begin{array}{c}\alpha, s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash S \equiv [\alpha{:}\kappa.\sigma_1] \ \ \mathit{sig} \\ \Gamma[s \uparrow S] \vdash M \downarrow [\alpha{:}\kappa.\sigma_2] \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/\mathit{Fst}\ s] \ \ \mathit{type}\end{array}}{\Gamma \vdash \mathit{fix}_O(s{:}S.M) : S}$$

$$\frac{s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma[s \uparrow S] \vdash M \downarrow S \qquad \Gamma \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \ \ \mathit{sig}}{\Gamma \vdash \mathit{fix}_T(s{:}S.M) : S}$$

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

$$\frac{\Gamma \vdash \mathit{fix}_O(s{:}S.M) : S}{\Gamma \vdash \mathit{fix}_O(s{:}S.M) \downarrow S} \qquad \frac{\Gamma \vdash \mathit{fix}_T(s{:}S.M) : S}{\Gamma \vdash \mathit{fix}_T(s{:}S.M) \downarrow S}$$

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

$$\frac{\begin{array}{c}\alpha, s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash S \equiv [\alpha{:}\kappa.\sigma_1] \ \ \mathit{sig} \\ \Gamma[s \uparrow S] \vdash M \downarrow [\alpha{:}\kappa.\sigma_2] \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/\mathit{Fst}\ s] \ \ \mathit{type} \\ \Gamma \vdash S \equiv S' \ \ \mathit{sig} \qquad \Gamma[s \uparrow S] \vdash M' \downarrow [\alpha{:}\kappa.\sigma_2] \qquad \Gamma[s : S] \vdash M \equiv M' : [\alpha{:}\kappa.\sigma_2]\end{array}}{\Gamma \vdash \mathit{fix}_O(s{:}S.M) \equiv \mathit{fix}_O(s{:}S'.M') : S}$$

$$\frac{\begin{array}{c}s \notin \mathrm{Dom}(\Gamma) \qquad \Gamma[s \uparrow S] \vdash M \downarrow S \qquad \Gamma \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \ \ \mathit{sig} \\ \Gamma \vdash S \equiv S' \ \ \mathit{sig} \qquad \Gamma[s \uparrow S] \vdash M' \downarrow S \qquad \Gamma[s : S] \vdash M \equiv M' : S\end{array}}{\Gamma \vdash \mathit{fix}_T(s{:}S.M) \equiv \mathit{fix}_T(s{:}S'.M') : S}$$

$$\frac{\begin{array}{c}\alpha, s, s^{\mathsf{c}}, s^{\mathsf{r}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ \mathit{kind} \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \ \ \mathit{type} \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma_2 \ \ \mathit{type} \\ \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[s^{\mathsf{c}} : \kappa][s^{\mathsf{r}} \uparrow \sigma_1[s^{\mathsf{c}}/\alpha]] \vdash e \downarrow \sigma_2[c/\alpha] \qquad \Gamma[\alpha : \kappa] \vdash \sigma_1 \equiv \sigma_2[\alpha/s^{\mathsf{c}}] \ \ \mathit{type}\end{array}}{\Gamma \vdash \mathit{fix}_O(s{:}[\alpha{:}\kappa.\sigma_1].[c[\mathit{Fst}\ s/s^{\mathsf{c}}], e[\mathit{Fst}\ s, \mathit{Snd}\ s/s^{\mathsf{c}}, s^{\mathsf{r}}]]) \equiv [s^{\mathsf{c}} = \mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c, \mathit{fix}(s^{\mathsf{r}}{:}\sigma_1[s^{\mathsf{c}}/\alpha].e)] : [\alpha{:}\kappa.\sigma_1]}$$

$$\frac{\begin{array}{c}\alpha, s, s^{\mathsf{c}}, s^{\mathsf{r}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \mathfrak{S}(c : \kappa) \ \ \mathit{kind} \qquad \Gamma[\alpha : \mathfrak{S}(c : \kappa)] \vdash \sigma \ \ \mathit{type} \\ \Gamma[s^{\mathsf{c}} \uparrow \mathfrak{S}(c : \kappa)] \vdash c' \downarrow \mathfrak{S}(c : \kappa) \qquad \Gamma[s^{\mathsf{c}} : \mathfrak{S}(c : \kappa)][s^{\mathsf{r}} \uparrow \sigma[s^{\mathsf{c}}/\alpha]] \vdash e \downarrow \sigma[c'/\alpha]\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{fix}_T(s{:}[\alpha{:}\mathfrak{S}(c : \kappa).\sigma].[c'[\mathit{Fst}\ s/s^{\mathsf{c}}], e[\mathit{Fst}\ s, \mathit{Snd}\ s/s^{\mathsf{c}}, s^{\mathsf{r}}]]) \\ \quad \equiv [s^{\mathsf{c}} = c, \mathit{fix}(s^{\mathsf{r}}{:}\sigma[s^{\mathsf{c}}/\alpha].e)] : [\alpha{:}\mathfrak{S}(c : \kappa).\sigma]\end{array}}$$

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

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\kappa.\sigma] \ \ sig}{\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S \ \ sig}$$

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \ \ sig \qquad \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa}{\Gamma \vdash \rho s^{\mathsf{c}}.S \ \ sig}$$

$\boxed{\Gamma \vdash S_1 \equiv S_2 \ \ sig}$

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\kappa.\sigma] \ \ sig \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv S' \ \ sig}{\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.S \equiv \widehat{\rho} s^{\mathsf{c}}.S' \ \ sig}$$

$$\frac{\begin{array}{c}\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \\ \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv [\alpha{:}\mathfrak{S}(c : \kappa).\sigma] \ \ sig \qquad \Gamma[s^{\mathsf{c}} : \kappa] \vdash S \equiv S' \ \ sig\end{array}}{\Gamma \vdash \rho s^{\mathsf{c}}.S \equiv \rho s^{\mathsf{c}}.S' \ \ sig}$$

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma \ \ type}{\Gamma \vdash \widehat{\rho} s^{\mathsf{c}}.[\alpha{:}\kappa.\sigma] \equiv [s^{\mathsf{c}}{:}\kappa.\sigma[s^{\mathsf{c}}/\alpha]] \ \ sig}$$

$$\frac{\alpha, s^{\mathsf{c}} \notin \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash \kappa \ \ kind \qquad \Gamma[s^{\mathsf{c}} \uparrow \kappa] \vdash c \downarrow \kappa \qquad \Gamma[s^{\mathsf{c}} : \kappa][\alpha : \kappa] \vdash \sigma \ \ type}{\Gamma \vdash \rho s^{\mathsf{c}}.[\alpha{:}\mathfrak{S}(c : \kappa).\sigma] = [s^{\mathsf{c}}{:}\mathfrak{S}(\mu_{\equiv} s^{\mathsf{c}}{:}\kappa.c : \kappa).\sigma[s^{\mathsf{c}}/\alpha]] \ \ sig}$$