Recursive Type Generativity

Derek Dreyer
Toyota Technological Institute at Chicago
dreyer@tti-c.org

Abstract

Existential types provide a simple and elegant foundation for understanding generative abstract data types, of the kind supported by the Standard ML module system. However, in attempting to extend ML with support for recursive modules, we have found that the traditional existential account of type generativity does not work well in the presence of mutually recursive module definitions. The key problem is that, in recursive modules, one may wish to define an abstract type in a context where a name for the type already exists, but the existential type mechanism does not allow one to do so.

We propose a novel account of recursive type generativity that resolves this problem. The basic idea is to separate the act of generating a name for an abstract type from the act of defining its underlying representation. To define several abstract types recursively, one may first "forward-declare" them by generating their names, and then define each one secretly within its own defining expression. Intuitively, this can be viewed as a kind of backpatching semantics for recursion at the level of *types*. Care must be taken to ensure that a type name is not defined more than once, and that cycles do not arise among "transparent" type definitions.

In contrast to the usual continuation-passing interpretation of existential types in terms of universal types, our account of type generativity suggests a *destination-passing* interpretation. Briefly, instead of viewing a value of existential type as something that creates a new abstract type every time it is unpacked, we view it as a function that takes as input a pre-existing undefined abstract type and defines it. By leaving the creation of the abstract type name up to the client of the existential, our approach makes it significantly easier to link abstract data types together recursively.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Recursion, Abstract data types, Modules; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords Type systems, abstract data types, recursion, generativity, recursive modules, effect systems

1. Introduction

Recursive modules are one of the most frequently requested extensions to the ML languages. After all, the ability to have cyclic dependencies between different files is a feature that is commonplace in mainstream languages like C and Java. To the novice programmer especially, it seems very strange that the ML module system should provide such powerful mechanisms for data abstraction and code reuse as functors and translucent signatures, and yet not allow mutually recursive functions and data types to be broken into separate modules. Certainly, for simple examples of recursive modules, it is difficult to convincingly argue why ML could not be extended in some *ad hoc* way to allow them. However, when one considers the semantics of a *general* recursive module mechanism, one runs into several interesting problems for which the "right" solutions are far from obvious.

One problem involves the interaction of recursion and computational effects. The evaluation of an ML module may involve impure computations such as I/O or the creation of mutable state. Thus, if recursion is introduced at the module level, it appears necessary to adopt a *backpatching* semantics of recursion (in the style of Scheme's `letrec` construct) in order to ensure that the effects within recursive module definitions are only performed once. Under such a semantics, a recursive definition `letrec X = M in ...` is evaluated by (1) binding `X` to an uninitialized ref cell, (2) evaluating `M` to a value `V`, and (3) backpatching the contents of `X`'s cell with `V`, thereby tying the recursive knot. As a matter of both methodology and efficiency, it is desirable to know statically that this recursive definition is *well-founded, i.e.,* that `M` will evaluate to `V` without requiring the value of `X` prematurely. In previous work [4], we studied this problem in detail and proposed a type-based approach to guaranteeing well-founded recursion.

In this paper, we focus on a second, orthogonal problem with recursive modules that we and other researchers have struggled with. This problem is of particular importance and should be of interest to a general audience because it concerns the interaction of two fundamental concepts in programming, *recursion* and *data abstraction*, and it is possible to understand and explore the problem independently of modules. (In fact, that is precisely what we are going to do later in the paper.) To begin, however, we will use some informal examples in terms of ML modules as a way of illustrating the problem.

1.1 Mutually Recursive Abstract Data Types

Suppose we want to write two mutually recursive ML modules `A` and `B`, as shown in Figure 1. Module `A` (resp. `B`) defines a type `t` (resp. `u`) and a function `f` (resp. `g`) among other components. It is sealed with a signature $\texttt{SIG}_\texttt{A}\texttt{(X)}$ (resp. $\texttt{SIG}_\texttt{B}\texttt{(X)}$) that hides the definition of its type component.[1] Note that the type of the function

[1] We make use of parameterized signatures here as a matter of syntactic convenience, although ML does not currently support them.

```
signature SIG_A(X) = sig
                       type t
                       val f : t -> X.B.u * t
                       ...
                    end
signature SIG_B(X) = sig
                       type u
                       val g : X.A.t -> u * X.A.t
                       ...
                    end
signature SIG(X)  = sig
                       structure A : SIG_A(X)
                       structure B : SIG_B(X)
                    end

structure rec X :> SIG(X) = struct
  structure A :> SIG_A(X) = struct
    type t = int
    fun f (x:t) : X.B.u * t =
       let val (y,z) = X.B.g(x+3) (* Error 1 *)
       in (y,z+5) end              (* Error 2 *)
    ...
  end
  structure B :> SIG_B(X) = struct
    type u = bool
    fun g (x:X.A.t) : u * X.A.t = ...X.A.f(...)...
    ...
  end
end
```

Figure 1. Mutually Recursive Abstract Data Types

component in each module refers to the abstract type provided by the other module.

The code here is clearly contrived—*e.g.*, `A.t` and `B.u` are implemented as `int` and `bool`—but it serves to concisely demonstrate the kind of type errors that can arise very easily due to the interaction of recursion and abstract types. The first type error comes in the first line of the body of `A.f`. The function takes as input a variable `x` of type `t` (which is defined to be `int`), and makes a recursive call to the function `X.B.g`, passing it `x+3`. The error arises because the type of `X.B.g` is `X.A.t -> X.B.u * X.A.t`, and `X.A.t` is not equivalent to `int`. To see this, observe that the variable `X` is bound with the signature `SIG(X)`, whose `A.t` component is specified opaquely.[2] The second type error, appearing in the next line of the same function, is similar. The value `z` returned from the call to `X.B.g` has type `X.A.t`, but the function attempts to use `z` as if it were an integer.

Both of these type errors are of course symptoms of the same problem: Alice, the programmer of module `A`, "knows" that `X.A.t` is implemented internally as `int`, because *she* is writing the implementation! Yet this fact is not observable from the signature of `X`. The only simple way that has been proposed to address this problem is to reveal the identity of `A.t` in the signature `SIG`$_A$`(X)` as transparently equal to `int`. This is not really a satisfactory solution, though, since it exposes the identity of `A.t` to the implementor of module `B` and essentially suggests that we give up on trying to impose any data abstraction *within* the recursive module definition.

A more complex suggestion would be that we change the way that recursive modules are typechecked. Intuitively, when we are typechecking the body of module `A`, we ought to know that `X.A.t` is `int`, but we ought not know anything about `X.B.u`. When we are typechecking the body of module `B`, we ought to know that `X.B.u` is `bool`, but we ought not know anything about `X.A.t`. Additionally, when typechecking `B`, we ought to be able to observe that a direct hierarchical reference to `A.t` is interchangeable with a recursive reference to `X.A.t`.

In the case of the module from Figure 1, such a typechecking algorithm seems fairly straightforward, but it becomes much more complicated if the recursive module body contains, for instance, nested uses of opaque sealing. It is certainly possible to define an algorithm that works in the general case—the author's Ph.D. thesis formalizes such an algorithm [5]—but it is not a pretty sight. Furthermore, we would really like to be able to explain what is going on using a *type system*, not just an algorithm.

[2] Incidentally, you may wonder how it can be legal for the signature `X` is bound with to refer to `X`. This is achieved through the use of *recursively dependent signatures*, which were proposed by Crary, Harper and Puri [3] in theory, and implemented by Russo [19] and Leroy [13] in practice. Subject to certain restrictions, they are not semantically problematic, but they are beyond the scope of this paper.

```
signature ORDERED = sig
                        type t
                        val compare : t * t -> order
                    end
signature HEAP = sig
                     type item; type heap;
                     val insert : item * heap -> heap
                     ...
                 end
functor MkHeap : (X : ORDERED)
                   -> HEAP where type item = X.t

structure rec Boot : ORDERED = struct
  datatype t = ...Heap.heap...
  fun compare (x,y) = ...
end
and Heap : (HEAP where type item = Boot.t) = MkHeap(Boot)
```

Figure 2. Bootstrapped Heap Example

1.2 Recursion Involving Generative Functor Application

Figure 2 exhibits another commonly desired form of recursive module, one that is in some ways even more problematic than the one from Figure 1. In this particular example, the goal is to define a recursive data type `Boot.t` of so-called "bootstrapped heaps," a data structure proposed by Okasaki [17]. The important feature of bootstrapped heaps (for our purposes) is that they are defined recursively in terms of heaps of themselves, where "heaps of themselves" are created by applying the library functor `MkHeap` to the `Boot` module.

The problem with this definition, at least in the case of Standard ML semantics [14], is that functors in SML behave *generatively*, so each application of `MkHeap` produces a fresh abstract `heap` type at run time. The way this is typically modeled in type theory is by treating the return signature of `MkHeap` as synonymous with an existential type. Consequently, while `Boot.t` must be defined before `MkHeap(Boot)` can be evaluated, the type `Heap.heap` will not even exist until `MkHeap(Boot)` has been evaluated and "unpacked." It does not make sense in the ML type system to define `Boot.t` in terms of a type (`Heap.heap`) that does not exist yet.

The usual solution to this problem is to assume that `MkHeap` is not generative, but rather *applicative* [12]. Under applicative functor semantics, `MkHeap(Boot)` is guaranteed to produce the same `heap` type every time it is evaluated, and thus the definition of `Boot.t` is allowed to refer to the type `MkHeap(Boot).heap` statically, without having to evaluate `MkHeap(Boot)` first. This solution is certainly sensible if one is designing a recursive module extension to O'Caml [11], for O'Caml only supports applicative functors. There are good reasons, however, for supporting generative functors. Their interpretation in type theory is simpler than that of applicative functors, and they provide stronger abstraction guarantees that are desirable in many cases.[3] It seems unfortunate that MkHeap is *required* to be applicative.

1.3 Overview

Our exposition thus far begs the question: Is recursion fundamentally at odds with type generativity? In this paper, we will argue that the answer is no. We propose a novel account of type abstraction that resolves the problems encountered in the above recursive module examples and provides an elegant foundation for understanding how recursion can coexist peacefully with generativity.

The basic idea is to separate the act of generating a name for an abstract type from the act of defining its underlying representation. To define several abstract types recursively, one may first "forward-declare" them by generating their names, and then define each one secretly within its own defining expression. Intuitively, this can be viewed as a kind of backpatching semantics for recursion at the level of *types*! The upshot is that there is a unique name for each abstract type, which is visible to everyone (within a certain scope), but the identity of each abstract type is only known inside the term that defines it. This is exactly what was desired in both of the recursive module examples discussed above.

While our new approach to type generativity is operationally quite different from existing approaches, it is fundamentally compatible with the traditional interpretation of ADT's in terms of existential types. The catch is that, while existential types are typically understood via the continuation-passing Church encoding in terms of universals[4], we offer an alternative *destination-passing* interpretation. Briefly, instead of viewing a value of existential type $\exists\alpha.\,\mathrm{A}$ as something that creates a new abstract type every time it is unpacked, we view it as a function that takes as input a pre-existing undefined type name β and defines it, returning a value of type A (with β substituted for α). How the function has defined β, however, we do not know. By leaving the creation of the abstract type name β up to the client of the existential, our approach makes it significantly easier to link abstract data types together recursively.

The rest of the paper is structured as follows. In Section 2, we discuss the details of our approach informally, and give examples to illustrate how it works. In Section 3, we define a type system for recursive type generativity as a conservative extension of System F_ω. In order to ensure that abstract types do not get defined more than once, we treat type definitions as a kind of effect and track them in the manner of an effect system [9, 22]. The intention is that this type system may eventually serve as the basis of a recursive module language. In Section 4, we explore the expressive power of our destination-passing interpretation of ADT's. Finally, in Section 5, we discuss related work, and in Section 6, we conclude.

2. The High-Level Picture

We will now try to paint a high-level picture of how our approach to recursive type generativity works. The easiest way to understand is by example, so in Section 2.1, we use the recursive module examples from Section 1 as a way of introducing the key constructs of our language. In particular, we show how those examples would be encoded in our language and why, under this new encoding, they typecheck. Then, in Section 2.2, we also show how our approach makes it possible to support *separate compilation* of recursive abstract data types. Lastly, in Section 2.3, we discuss some of the subtler issues that we encounter in attempting to prevent "bad" cycles in type definitions.

[3] For more details, see the discussion in Dreyer, Crary and Harper [6].

[4] See Section 2.2 for details.

```
SIG_A = $\lambda\alpha:\mathbf{T}.\,\lambda\beta:\mathbf{T}.\,\{$f:$\alpha\rightarrow\beta\times\alpha$, ...$\}$
SIG_B = $\lambda\alpha:\mathbf{T}.\,\lambda\beta:\mathbf{T}.\,\{$g:$\alpha\rightarrow\beta\times\alpha$, ...$\}$
SIG   = $\lambda\alpha:\mathbf{T}.\,\lambda\beta:\mathbf{T}.\,\{$A:SIG_A$(\alpha)(\beta)$, B:SIG_B$(\alpha)(\beta)\}$

new $\alpha\uparrow\mathbf{T},\beta\uparrow\mathbf{T}$ in
  letrec X : SIG$(\alpha)(\beta)$ =
    {A = (let () = $\alpha$ := int in
              {f = ...})
            : SIG_A$(\alpha)(\beta)$ defines $\alpha$,
     B = (let () = $\beta$ := bool in
              {g = ...})
            : SIG_B$(\alpha)(\beta)$ defines $\beta$}
  in ...
```

Figure 3. New Encoding of Example from Figure 1

2.1 Reworking the Examples

Figure 3 shows our new encoding of the recursive module example from Figure 1. The first thing to notice here is that we have dispensed with modules. $\mathtt{SIG_A}$, $\mathtt{SIG_B}$ and SIG are represented here via the well-known encoding of ML signatures as type operators in System F_ω. The idea is simply to view the types of a signature's value components as being parameterized over the signature's abstract type components. Correspondingly, the ML feature of using where type to add type definitions to signatures is encodable in F_ω by type-level function application.[5] (We employ this encoding here merely so that we can study the interaction of recursion and data abstraction at the foundational level of F_ω, with which we assume the reader is familiar. In the future, we intend to scale the ideas of this paper to a more easily programmable language of recursive modules.)

Starting on the fourth line, however, we see something that is not standard. (The underlined portions of the code indicate new features that are not part of F_ω.) What the "new" declaration does is create two new type variables α and β of kind $\mathbf{T}$, the kind of base types. Throughout this example, you can think of α as standing for A.t, and β as standing for B.u, in the original example of Figure 1.

What does it mean to "create a new type variable"? Intuitively, you can think of it much like creating a reference cell in memory. Imagine that during the execution of the program you maintain a *type store*, mapping locations (represented by type variables) to types. Eventually, each location will get filled in with a type, but when a type memory cell is first created (by the "new" construct), its contents are uninitialized.[6] Formally speaking, what the new declaration does is to insert α and β into the type context with a special binding of the form $\alpha\uparrow\mathbf{T}$, which indicates that they have not yet been defined. We refer to such type variables as *writable*.

Next, we make use of a letrec construct to define A and B. For simplicity, the letrec construct employs an unrestricted (*i.e.*, potentially ill-founded) backpatching semantics for recursion.[7] Specifically, we allocate an uninitialized ref cell X in memory, whose type is rec(SIG$(\alpha)(\beta)$). In order to use X within the body of the recursive definition—*i.e.*, in order to get a value of type SIG$(\alpha)(\beta)$ without the "rec"—one must first dereference the

[5] See Jones [10] for more examples of this encoding. Also, this is essentially how the Definition of SML interprets signatures [14].

[6] In fact, this is exactly how we are going to model "type creation" in the dynamic semantics of our language in Section 3.4.

[7] In principle, we believe it should be straightforward to incorporate static detection of ill-founded recursion [4] into the present calculus, but we have not yet attempted it.

```
ORDERED = $\lambda\alpha:\mathbf{T}.\,\{$compare:$\alpha\times\alpha\rightarrow$ order$\}$
HEAP    = $\lambda\alpha:\mathbf{T}.\,\lambda\beta:\mathbf{T}.\,\{$insert:$\alpha\times\beta\rightarrow\beta$, ...$\}$
HEAPGEN = $\lambda\alpha:\mathbf{T}.\,\underline{\forall\beta\uparrow\mathbf{T}.\,\texttt{unit}\xrightarrow{\beta\downarrow}\texttt{HEAP}(\alpha)(\beta)}$
MkHeap  : $\underline{\forall\alpha\downarrow\mathbf{T}.}\,\texttt{ORDERED}(\alpha)\rightarrow\texttt{HEAPGEN}(\alpha)$

new $\alpha\uparrow\mathbf{T},\beta\uparrow\mathbf{T}$ in
  letrec X : $\{$Boot:ORDERED$(\alpha)$, Heap:HEAP$(\alpha)(\beta)\}$ =
     let Boot = (let () = $\underline{\alpha\ :\approx\ (\ldots\beta\ldots)}$
                 in $\{$compare = ...$\}$) in
     let Heap = MkHeap$[\alpha]$(Boot)$[\beta]()$ in
       $\{$Boot=Boot, Heap=Heap$\}$
  in ...
```

Figure 4. New Encoding of Example from Figure 2

memory location by writing `fetch(X)`. This `fetch` operation must check whether `X`'s contents have been initialized and, if not, raise a run-time error. Finally, when the body of the `letrec` is finished evaluating, the resulting value (of type $\texttt{SIG}(\alpha)(\beta)$) is backpatched into the location `X`. (There are good reasons to require the dereferencing of `X` to be explicit, as we will see in Figure 5.)

Now for the definition of "module" `A`: The first thing we do here is to backpatch the type name α with the definition `int`. The use of "$:=$" notation is appropriate because at run time we can think of this operation as modifying the contents of the α location in the type store. At compile time, it results in a change to the type context so that the typechecking of the remainder of `A` is done with the knowledge that α is equal to `int`. As a result, the type errors from Figure 1 disappear!

Once we have finished typechecking `A`, however, we want to hide the knowledge that α is `int` from the rest of the program. We achieve this in the next line by "sealing" the definition of `A`. Although it is a bit hard to tell from Figure 3, the sealing construct has the form "$e : \tau$ `defines` α", where in this case τ is $\texttt{SIG}_{\texttt{A}}(\alpha)(\beta)$. The sealing construct does two things simultaneously: it exports e at the type τ, and it removes the definition of α from the type context in which the rest of the program is typechecked.[8] While the hiding of α's definition is obviously important, it is critical to understand that the ascription of the type τ is just as important. In the case of module `A`, it is the type ascription $\texttt{SIG}_{\texttt{A}}(\alpha)(\beta)$ that ensures that `A.f` is exported at the type $\alpha\rightarrow\beta\times\alpha$ and not, say, at the type $\texttt{int}\rightarrow\beta\times\texttt{int}$.

Finally, there is the definition of `B`, which works similarly to the definition of `A`.

Voilà! To summarize, by distinguishing the point at which α and β are created from the points at which they are defined, we have made it possible for all parties to refer to these types by the same names, but also for the underlying representation of each type to be specified and made visible only within its own defining expression.

Let us move on to Figure 4, which shows our new encoding of the bootstrapped heap example. As in the previous example, we define two abstract types here, α and β, but now α stands for `Boot.t`, and β for `Heap.heap`. The signatures `ORDERED` and `HEAP` are in parameterized form as expected, with the former parameterized over the type α of items being compared, and the latter parameterized over both the item type α and the heap type β.

The most unusual (and important) part of this encoding is the type that we require for the `MkHeap` functor. Under the standard encoding of generative functors into F_ω, we would expect `MkHeap` to have the type

$$\forall\alpha:\mathbf{T}.\,\texttt{ORDERED}(\alpha)\rightarrow\exists\beta:\mathbf{T}.\,\texttt{HEAP}(\alpha)(\beta)$$

The type shown in Figure 4 differs from our expectations in two ways. First, while α is universally quantified, the quantification is written $\alpha\downarrow\mathbf{T}$. The reason for this has to do with avoiding cycles in transparent type definitions, and we will defer explanation of it until Section 2.3. For the moment, read $\alpha\downarrow\mathbf{T}$ as synonymous with $\alpha:\mathbf{T}$. Second, `MkHeap`'s result type, $\texttt{HEAPGEN}(\alpha)$, is not an existential, but some weird kind of universal!

Indeed, $\texttt{HEAPGEN}(\alpha) = \forall\beta\uparrow\mathbf{T}.\,\texttt{unit}\xrightarrow{\beta\downarrow}\texttt{HEAP}(\alpha)(\beta)$ is a universal type, but a very special one. Specifically, a function of this type requires its type argument to be a type *variable* that has not yet been defined (hence, the notation $\forall\beta\uparrow\mathbf{T}$). Moreover, when the function is applied, it will not only return a value of type $\texttt{HEAP}(\alpha)(\beta)$, but also *define* β in the process. We write $\beta\downarrow$ on the arrow type to indicate that the application of the function engenders the effect of defining β, but how it defines β we cannot tell.

The reason for defining $\texttt{HEAPGEN}(\alpha)$ in this fashion is that it allows us to come up with a name (β) for the `Heap.heap` type ahead of time, *before* the `MkHeap` functor is applied. In this way, it is possible for the definition of α (*i.e.,* `Boot.t`) to refer to β before β has actually been defined. As we explained in Section 1.2, this is something that is not possible under the ordinary interpretation of $\texttt{HEAPGEN}(\alpha)$ as an existential type.

The only other point of interest in this encoding is that α is defined by a new kind of assignment ($:\approx$). One can think of this assignment as analogous with `datatype` definitions in SML, just as $:=$ is analogous with transparent `type` definitions (type synonyms). The definition of α in `Boot` does not change the fact that α is an abstract type, but it introduces `fold` and `unfold` coercions that allow one to coerce back and forth between α and its underlying definition. This form of type definition is necessary in order to break up cycles in the type-variable dependency graph. We discuss this point further in Section 2.3.

2.2 Destination-Passing Style and Separate Compilation

The strange new universal type that we used to define $\texttt{HEAPGEN}(\alpha)$ in the last example can be viewed as a kind of existential type in sheep's clothing. Under the usual Church encoding of existential types in terms of universals, $\exists\alpha:\mathrm{K}.\,\tau$ can be understood as shorthand for $\forall\beta:\mathbf{T}.\,(\forall\alpha:\mathrm{K}.\,\tau\rightarrow\beta)\rightarrow\beta$. This is quite similar to $\forall\alpha\uparrow\mathrm{K}.\,\texttt{unit}\xrightarrow{\alpha\downarrow}\tau$ in the sense that a function of either type has some type constructor α of kind K and some value of type τ hidden inside it, but the function's type won't tell you what α is. The difference is that the Church encoding is a function in continuation-passing style (CPS), whereas our new encoding is a function in *destination-passing style* (DPS) [24]. In Section 4.2, we will make the DPS encoding of existentials precise.

So, one may wonder, if our DPS universal type is really an existential in disguise, why don't we just write, say, $\exists\alpha\uparrow\mathrm{K}.\,\mathrm{A}$ instead of $\forall\alpha\uparrow\mathrm{K}.\,\texttt{unit}\xrightarrow{\alpha\downarrow}\tau$? Why bother with the `unit`? The answer is that in some cases we want to write a function of type $\forall\alpha\uparrow\mathrm{K}.\,\tau_1\xrightarrow{\alpha\downarrow}\tau_2$ where $\alpha\in\mathrm{FV}(\tau_1)$—that is, a function that takes as input a writable type name α, together with a value whose type depends on α. In typical programming this does not come up often, but with recursive modules it arises naturally, especially in the context of separate compilation.

Figure 5 illustrates such a situation. The goal here is to allow the recursive "modules" `A` and `B` from Figure 3 to be compiled separately. We have put the implementations of `A` and `B` inside of two separate "functors" $\texttt{Separate}_{\texttt{A}}$ and $\texttt{Separate}_{\texttt{B}}$, represented as polymorphic functions. $\texttt{Separate}_{\texttt{A}}$ takes β (*i.e.,* `B.u`) as its first

[8] Note that sealing is purely a compile-time notion—at run time, the definition of α is not actually removed from the type store.

$$\begin{array}{lcl}\texttt{Separate}_{\texttt{A}} & : & \forall\beta:\mathbf{T}.\,\forall\alpha\uparrow\mathbf{T}.\,\texttt{rec}(\texttt{SIG}(\alpha)(\beta)) \xrightarrow{\alpha\downarrow} \texttt{SIG}_{\texttt{A}}(\alpha)(\beta) \\ & = & \Lambda\beta:\mathbf{T}.\,\Lambda\alpha\uparrow\mathbf{T}.\,\lambda\texttt{X}:\texttt{rec}(\texttt{SIG}(\alpha)(\beta)).\,\ldots \\ \texttt{Separate}_{\texttt{B}} & : & \forall\alpha:\mathbf{T}.\,\forall\beta\uparrow\mathbf{T}.\,\texttt{rec}(\texttt{SIG}(\alpha)(\beta)) \xrightarrow{\beta\downarrow} \texttt{SIG}_{\texttt{B}}(\alpha)(\beta) \\ & = & \Lambda\alpha:\mathbf{T}.\,\Lambda\beta\uparrow\mathbf{T}.\,\lambda\texttt{X}:\texttt{rec}(\texttt{SIG}(\alpha)(\beta)).\,\ldots\end{array}$$

$$\begin{array}{l}\texttt{new}\ \alpha\uparrow\mathbf{T},\beta\uparrow\mathbf{T}\ \texttt{in} \\ \quad\texttt{letrec X : SIG}(\alpha)(\beta)\ \texttt{=} \\ \qquad\{\texttt{A = Separate}_{\texttt{A}}\,[\beta][\alpha](\texttt{X}),\ \texttt{B = Separate}_{\texttt{B}}\,[\alpha][\beta](\texttt{X})\} \\ \quad\texttt{in ...}\end{array}$$

Figure 5. Separate Compilation of A and B from Figure 3

argument, α (*i.e.,* A.t) as its second argument, and the recursive module variable X as its third argument. The type of $\texttt{Separate}_{\texttt{A}}$ employs a DPS universal type to bind α because $\texttt{Separate}_{\texttt{A}}$ wants to take a writable A.t and define it. Note, however, that β is bound normally as $\beta:\mathbf{T}$. ($\texttt{Separate}_{\texttt{B}}$ of course does the opposite, because it wants to define β, not α.) The important point here is that the type of the argument X refers to both α and β and therefore cannot be moved outside of the DPS universal.[9] If all we had was a DPS universal of the form $\forall\alpha\uparrow\mathrm{K}.\,\texttt{unit}\xrightarrow{\alpha\downarrow}\tau$, we would have no way of typing $\texttt{Separate}_{\texttt{A}}$ and $\texttt{Separate}_{\texttt{B}}$.

If it is so important to be able to write a function that takes a value argument *after* an $\alpha\uparrow\mathrm{K}$ argument, it is natural to ask why we do not just offer two separate type constructs, $\forall\alpha\uparrow\mathrm{K}.\,\tau$ and $\tau_1\xrightarrow{\alpha\downarrow}\tau_2$, of which $\forall\alpha\uparrow\mathrm{K}.\,\tau_1\xrightarrow{\alpha\downarrow}\tau_2$ would be the composition. The former construct would require its argument to be a writable variable, and the latter would be a standard sort of effectful function type, in this case the effect being the definition of some externally-bound type variable α.

The reason we do not divide up the DPS universal type in this way is that such a division would result in serious complications for our type system. The main complication is that, while $\tau_1\xrightarrow{\alpha\downarrow}\tau_2$ looks like a standard sort of effect type, the effect in question is highly unusual. In particular, if f were a function of that type, it could only be *applied* once because, for soundness purposes, we require that a type variable α can only be *defined* once. Another way of saying this is that the *type* $\tau_1\xrightarrow{\alpha\downarrow}\tau_2$ only makes sense while α is writable.

Meta-theoretically speaking, this becomes problematic from the point of view of defining type substitution. If at some point in the program α gets defined as τ, and α's binding in the context changes correspondingly from $\alpha\uparrow\mathbf{T}$ to $\alpha:\mathbf{T}=\tau$, then we should be able to substitute τ for free occurrences of α. But substituting τ for α in $\tau_1\xrightarrow{\alpha\downarrow}\tau_2$ does not make sense. In contrast, our type system has the property that well-formed types stay well-formed, regardless of whether their free type variables go from being writable to being defined.

2.3 Avoiding Cycles in Transparent Type Definitions

We have now presented all the key constructs in our language and shown how they can be used to support recursive definitions of generative abstract data types. In order to make this approach work, there are two points of complexity that our type system has to deal with. One involves making sure that writable type variables get defined once and only once. This is a kind of linearity property and it is not fundamentally difficult to track using a type-and-effect system, as we explain in Section 3.

The other point concerns our desire to avoid cycles in transparent type definitions. While our language is designed to permit recursive definitions of abstract types, we require that every cycle in the type dependency graph must go through a "datatype," *i.e.,* one that was defined by $\alpha:\approx\mathrm{A}$.[10] We make this restriction because we want to keep the definition of type equivalence simple. If we were able to define $\alpha:=\beta\times\beta$ and $\beta:=\alpha\times\alpha$, then we would need to support some form of *equi-recursive types* [1, 3]. In fact, since we allow definitions of type constructors of higher kind, we would need to support equi-recursive type *constructors*, whose equational theory is not fully understood.

The mechanism we employ to guarantee that no transparent type cycles arise is slightly involved, but the reasoning behind it is straightforward to understand. Let us step through it. First of all, if α is defined by $\alpha:\approx\mathrm{A}$, then clearly no restrictions are necessary. If, however, α is defined transparently by $\alpha:=\mathrm{A}$, then we must require at the very least that A does not depend on α. By "depend on," we mean that if all known type synonyms were expanded out, then α would not appear in the free variables of the expanded A.

Unfortunately, in the presence of data abstraction, this restriction alone is not sufficient. Suppose, for instance, that in our example from Figure 3 the type variable α were defined by A and β by B (instead of by int and bool). The definition of α and the definition of β each occur in contexts where the other variable is considered abstract. Consequently, the restriction that A not depend on α and B not depend on β would not prevent A from depending on β and B from depending on α. How can our type system ensure that each definition does not contribute to a transparent cycle without peeking at what the other one is (and hence violating abstraction)?

A simple, albeit conservative, solution to this dilemma is to demand that, if α is defined by $\alpha:=\mathrm{A}$, then A may not depend on *any* abstract type variables except those that are known to be datatypes. We will say that a type A obeying this restriction is *stable*. While this approach does the trick, it is rather limiting. For example, in ML, it is common to define a type transparently in terms of an abstract type imported from another module (which may or may not be known to be a datatype). The stability restriction, however, would prohibit such a type definition inside a recursive module.

Therefore, to make our type system less draconian, we employ a modified form of the above conservative solution, in which the restriction on transparent definitions is relaxed in two ways. First, in order to permit transparent definitions to depend on abstract types that are not datatypes, we expand the notion of stability by allowing type variables to be considered stable if they are bound in the context as such. A stable type variable, bound as $\alpha\downarrow\mathrm{K}$, may only be instantiated with other stable types. We also introduce a new form of universal type, $\forall\alpha\downarrow\mathrm{K}.\,\tau$, describing functions whose type arguments must be stable.

We have already seen an instance where a stable universal is needed, namely in the type of the MkHeap functor from Figure 4. The reason for quantifying the item type α as a stable variable is that it enables the MkHeap functor to define the heap type β transparently in terms of α (*e.g.,* $\beta:=\alpha$ list). If α were only bound as $\alpha:\mathbf{T}$, then β would have to be defined as a datatype in order to ensure stability. Since MkHeap requires its item argument to be stable, it is imperative that the actual type α to which it is applied be stable. In the case of the Boot module, α is defined as a datatype, so all is well.

The second way in which we relax the restriction on transparent type definitions is that, while we require them to be stable, we

[9] Also important to the success of this encoding is the fact that X must be explicitly dereferenced. Otherwise, the references to X in the linking module would result in a run-time error. See Dreyer [4] for more discussion of this issue.

[10] We use A and B here to denote type constructors of arbitrary kind, as opposed to τ, which represents types of kind $\mathbf{T}$.

$$
\begin{array}{lrl}
\text{Type Variables} & \alpha, \beta & \\
\text{Kinds} & \mathrm{K}, \mathrm{L} ::= & \mathbf{T} \mid 1 \mid \mathrm{K}_1 \times \mathrm{K}_2 \mid \mathrm{K}_1 \to \mathrm{K}_2 \\
\text{Constructors} & \mathrm{A}, \mathrm{B} ::= & \alpha \mid b \mid \langle\rangle \mid \langle \mathrm{A}_1, \mathrm{A}_2 \rangle \mid \pi_i \mathrm{A} \mid \\
 & & \lambda\alpha : \mathrm{K}.\, \mathrm{A} \mid \mathrm{A}_1(\mathrm{A}_2) \\
\text{Base Types} & b ::= & \mathtt{unit} \mid \mathrm{A}_1 \times \mathrm{A}_2 \mid \mathrm{A}_1 \to \mathrm{A}_2 \mid \\
 & & \mathtt{rec}(\mathrm{A}) \mid \forall\alpha : \mathrm{K}.\, \mathrm{A} \mid \forall\alpha \downarrow \mathrm{K}.\, \mathrm{A} \mid \\
 & & \forall\alpha \uparrow \mathrm{K}.\, \mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2 \\
\text{Eliminations} & \mathcal{E} ::= & \bullet \mid \pi_i \mathcal{E} \mid \mathcal{E}(\mathrm{A}) \\
\text{Type Contexts} & \Delta ::= & \emptyset \mid \Delta, \alpha : \mathrm{K} \mid \Delta, \alpha \uparrow \mathrm{K} \mid \Delta, \alpha \downarrow \mathrm{K} \mid \\
 & & \Delta, \alpha : \mathrm{K} = \mathrm{A} \mid \Delta, \alpha : \mathrm{K} \approx \mathrm{A} \\
\text{Type Effects} & \varphi ::= & \emptyset \mid \varphi, \alpha := \mathrm{A} \mid \varphi, \alpha :\approx \mathrm{A} \mid \varphi, \alpha\downarrow
\end{array}
$$

Figure 6. Syntax of Types

$$
\begin{array}{lrl}
\text{Value Variables} & x, y & \\
\text{Values} & v ::= & x \mid \mathtt{()} \mid (v_1, v_2) \mid \lambda x : \mathrm{A}.\, e \mid \\
 & & \Lambda\alpha : \mathrm{K}.\, e \mid \Lambda\alpha \downarrow \mathrm{K}.\, e \mid \\
 & & \Lambda\alpha \uparrow \mathrm{K}.\, \lambda x : \mathrm{A}.\, e \mid \\
 & & \mathtt{fold}_{\mathrm{A}} \mid \mathtt{unfold}_{\mathrm{A}} \mid \mathtt{fold}_{\mathrm{A}}(v) \\
\text{Terms} & e, f ::= & v \mid \pi_i v \mid v_1(v_2) \mid v[\mathrm{A}] \mid v_1[\alpha](v_2) \mid \\
 & & \mathtt{rec}_{\mathrm{A}}(x.\, e) \mid \mathtt{fetch}(v) \mid \\
 & & \mathtt{let}\ \alpha = \mathrm{A}\ \mathtt{in}\ e \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \\
 & & \mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ e : \mathrm{A} \mid \\
 & & \alpha := \mathrm{A} \mid \alpha :\approx \mathrm{A} \mid e : \mathrm{A}\ \mathtt{defines}\ \alpha \\
\text{Value Contexts} & \Gamma ::= & \emptyset \mid \Gamma, x : \mathrm{A}
\end{array}
$$

Figure 7. Syntax of Terms

do not need them to be *immediately* stable. For example, say we have two writable type variables α and β. It is clearly ok to define $\beta := \mathtt{int}$, followed by $\alpha := \beta$, but what about processing the definitions in the reverse order? If $\alpha := \beta$ comes first, then α's definition is momentarily unstable. Ultimately, though, the definitions are still perfectly acyclic because α's definition is *eventually* stable. Moreover, there are situations where it is useful to have the flexibility of defining α and β in either order (in particular, see Section 4.1).

To afford this flexibility, when typechecking $\alpha := \mathrm{A}$, we allow A to depend on some set of writable variables $\{\beta_i\}$ not including α, so long as the β_i are all backpatched with stable definitions by the time α is sealed (*i.e.,* by the time e, in "$e : \tau$ $\mathtt{defines}$ α," has finished evaluating). While this requirement is not strictly necessary, it allows us to treat all sealed abstract types as stable, which in turn means that subsequent code may depend on them freely, without any restrictions.

3. The Type System

3.1 Type Structure

The syntax of our type structure is shown in Figure 6. The base type constructors b include all the usual F_ω base types, plus the new type constructs introduced in the examples of Section 2. The language of higher type constructors and kinds is standard F_ω, extended with products. Type eliminations $\mathcal{E}$ are used in the typing rules for `fold`'s and `unfold`'s (see the discussion of Rules 13 and 14 in Section 3.2).

Type contexts Δ include bindings for ordinary types ($\alpha : \mathrm{K}$), writable types ($\alpha \uparrow \mathrm{K}$), stable types ($\alpha \downarrow \mathrm{K}$), transparent type synonyms ($\alpha : \mathrm{K} = \mathrm{A}$) and datatypes ($\alpha : \mathrm{K} \approx \mathrm{A}$). We treat type contexts as unordered sets and assume implicitly that all bound variables are distinct. We write $\mathsf{writable}(\Delta)$ to denote the set of writable type variables bound in Δ. We write $\Delta(\alpha)$ to denote the kind to which α is bound in Δ. Type contexts are permitted to contain cycles as long as those cycles are broken by a datatype binding. To be precise:

Definition 3.1 (Acyclic Type Contexts)
We say that a type context Δ is *acyclic* if there is an ordering of its domain—$\alpha_1, \cdots, \alpha_n$—such that, for all $i \in 1..n$, if $\alpha_i : \mathrm{K}_i = \mathrm{A}_i \in \Delta$, then $\mathrm{FV}(\mathrm{A}_i) \subseteq \{\alpha_1, \cdots, \alpha_{i-1}\}$. In this case, we call $\alpha_1, \cdots, \alpha_n$ an *acyclic ordering* of Δ.

Definition 3.2 (Well-Formed Type Contexts)
We say that a type context Δ is well-formed, written $\vdash \Delta\ \mathsf{ok}$, if:

1. Δ is acyclic
2. $(\alpha : \mathrm{K} = \mathrm{A} \in \Delta \ \vee \ \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta) \Rightarrow \Delta \vdash \mathrm{A} : \mathrm{K}$

The type well-formedness judgment ($\Delta \vdash \mathrm{A} : \mathrm{K}$) referred to in part 2 of Definition 3.2 is defined in the obvious way (some representative rules appear in Appendix A). The only thing that the type well-formedness judgment needs to know from Δ is what the kinds of its bound variables are. It does not care whether type variables are writable or transparent, or even whether Δ is acyclic.

Type equivalence is slightly more complicated, due to the presence of type synonyms. To account for these, we use the equivalence judgment ($\Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$) defined by Stone in Section 9.1 of Pierce's ATTAPL book [20].[11] Our new base types, like the DPS universal type, do not at all complicate type equivalence, which Stone shows is relatively easy to prove decidable (assuming Δ is well-formed). Note that the type equivalence judgment treats datatype bindings ($\alpha : \mathrm{K} \approx \mathrm{A}$) no different from ordinary abstract type bindings ($\alpha : \mathrm{K}$). See Appendix A for some representative equivalence rules.

In order to define what it means to be a stable type constructor, we first define a useful auxiliary notion, which we call the *basis* of a type constructor. Intuitively, the basis of a type constructor A is the set of unstable abstract type variables on which A depends. This is determined by inductively crawling through the type context. Stable types are precisely those types whose bases are empty. Formally:

Definition 3.3 (Basis of a Type Constructor)
Given a type constructor A and an acyclic context Δ, where $\mathrm{FV}(\mathrm{A}) \subseteq \mathsf{dom}(\Delta)$, let $\mathsf{basis}_\Delta(\mathrm{A})$ be defined as $\bigcup\{\mathsf{basis}_\Delta(\alpha) \mid \alpha \in \mathrm{FV}(\mathrm{A})\}$, where $\mathsf{basis}_\Delta(\alpha)$ is defined inductively as follows:

$$
\mathsf{basis}_\Delta(\alpha) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta \text{ or } \alpha \downarrow \mathrm{K} \in \Delta \\ \{\alpha\} & \text{if } \alpha \uparrow \mathrm{K} \in \Delta \text{ or } \alpha : \mathrm{K} \in \Delta \\ \mathsf{basis}_\Delta(\mathrm{A}) & \text{if } \alpha : \mathrm{K} = \mathrm{A} \in \Delta \end{cases}
$$

Note: assuming $\alpha_1, \cdots, \alpha_n$ is an acyclic ordering of Δ, the above definition is sensible because $\mathsf{basis}_\Delta(\alpha_i)$ is defined only in terms of $\mathsf{basis}_\Delta(\alpha_j)$ for $j < i$.

Definition 3.4 (Stable Type Constructor)
We say that a type constructor A (with kind K in acyclic context Δ) is *stable*, written $\Delta \vdash \mathrm{A} \downarrow \mathrm{K}$, if $\Delta \vdash \mathrm{A} : \mathrm{K}$ and $\mathsf{basis}_\Delta(\mathrm{A}) = \emptyset$.

The final and most interesting element of Figure 6 is the definition of *type effects* φ. A type effect is an unordered set of type variable definitions (backpatchings). The domain of φ is the set of type variables being defined in φ (each entry in φ must define a distinct variable). A type variable α can either be defined transparently

[11] N.B. The language we are referring to is *not* the Stone-Harper singleton kind language [21]; it is just F_ω with β-η equivalence, extended with support for type defi nitions in the context.

$\boxed{\textbf{Well-formed terms: } \Delta;\, \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \varphi}$

We write $\Delta;\, \Gamma \vdash e : \mathrm{A}$ as shorthand for $\Delta;\, \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \emptyset$.

$$\frac{x:\mathrm{A} \in \Gamma}{\Delta;\, \Gamma \vdash x : \mathrm{A}}\ (1) \qquad \frac{}{\Delta;\, \Gamma \vdash \texttt{()} : \texttt{unit}}\ (2) \qquad \frac{\Delta;\, \Gamma \vdash v_1 : \mathrm{A}_1 \quad \Delta;\, \Gamma \vdash v_2 : \mathrm{A}_2}{\Delta;\, \Gamma \vdash (v_1, v_2) : \mathrm{A}_1 \times \mathrm{A}_2}\ (3) \qquad \frac{\Delta;\, \Gamma \vdash v : \mathrm{A}_1 \times \mathrm{A}_2 \quad i \in \{1,2\}}{\Delta;\, \Gamma \vdash \pi_i v : \mathrm{A}_i}\ (4)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta;\, \Gamma, x:\mathrm{A} \vdash e : \mathrm{B}}{\Delta;\, \Gamma \vdash \lambda x:\mathrm{A}.\, e : \mathrm{A} \to \mathrm{B}}\ (5) \qquad \frac{\Delta;\, \Gamma \vdash v_1 : \mathrm{A} \to \mathrm{B} \quad \Delta;\, \Gamma \vdash v_2 : \mathrm{A}}{\Delta;\, \Gamma \vdash v_1(v_2) : \mathrm{B}}\ (6)$$

$$\frac{\Delta, \alpha:\mathrm{K};\, \Gamma \vdash e : \mathrm{A}}{\Delta;\, \Gamma \vdash \Lambda\alpha:\mathrm{K}.\, e : \forall\alpha:\mathrm{K}.\, \mathrm{A}}\ (7) \qquad \frac{\Delta;\, \Gamma \vdash v : \forall\alpha:\mathrm{K}.\, \mathrm{B} \quad \Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta;\, \Gamma \vdash v[\mathrm{A}] : \{\alpha \mapsto \mathrm{A}\}\mathrm{B}}\ (8)$$

$$\frac{\Delta, \alpha\downarrow\mathrm{K};\, \Gamma \vdash e : \mathrm{A}}{\Delta;\, \Gamma \vdash \Lambda\alpha\downarrow\mathrm{K}.\, e : \forall\alpha\downarrow\mathrm{K}.\, \mathrm{A}}\ (9) \qquad \frac{\Delta;\, \Gamma \vdash v : \forall\alpha\downarrow\mathrm{K}.\, \mathrm{B} \quad \Delta \vdash \mathrm{A} \downarrow \mathrm{K}}{\Delta;\, \Gamma \vdash v[\mathrm{A}] : \{\alpha \mapsto \mathrm{A}\}\mathrm{B}}\ (10)$$

$$\frac{\Delta, \alpha:\mathrm{K} \vdash \mathrm{A} : \mathbf{T} \quad \Delta, \alpha\uparrow\mathrm{K};\, \Gamma, x:\mathrm{A} \vdash e : \mathrm{B}\ \mathsf{with}\ \alpha\downarrow}{\Delta;\, \Gamma \vdash \Lambda\alpha\uparrow\mathrm{K}.\, \lambda x:\mathrm{A}.\, e : \forall\alpha\uparrow\mathrm{K}.\, \mathrm{A} \xrightarrow{\alpha\downarrow} \mathrm{B}}\ (11)$$

$$\frac{\Delta;\, \Gamma \vdash v_1 : \forall\alpha\uparrow\mathrm{K}.\, \mathrm{A} \xrightarrow{\alpha\downarrow} \mathrm{B} \quad \Delta;\, \Gamma \vdash v_2 : \{\alpha \mapsto \beta\}\mathrm{A} \quad \beta\uparrow\mathrm{K} \in \Delta}{\Delta;\, \Gamma \vdash v_1[\beta](v_2) : \{\alpha \mapsto \beta\}\mathrm{B}\ \mathsf{with}\ \beta\downarrow}\ (12)$$

$$\frac{\Delta \vdash \mathrm{A} \equiv \mathcal{E}\{\alpha\} : \mathbf{T} \quad \alpha:\mathrm{K} \approx \mathrm{B} \in \Delta}{\Delta;\, \Gamma \vdash \texttt{fold}_{\mathrm{A}} : \mathcal{E}\{\mathrm{B}\} \to \mathrm{A}}\ (13) \qquad \frac{\Delta \vdash \mathrm{A} \equiv \mathcal{E}\{\alpha\} : \mathbf{T} \quad \alpha:\mathrm{K} \approx \mathrm{B} \in \Delta}{\Delta;\, \Gamma \vdash \texttt{unfold}_{\mathrm{A}} : \mathrm{A} \to \mathcal{E}\{\mathrm{B}\}}\ (14)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta;\, \Gamma, x:\texttt{rec}(\mathrm{A}) \vdash e : \mathrm{A}\ \mathsf{with}\ \varphi}{\Delta;\, \Gamma \vdash \texttt{rec}_{\mathrm{A}}(x.\, e) : \mathrm{A}\ \mathsf{with}\ \varphi}\ (15) \qquad \frac{\Delta;\, \Gamma \vdash v : \texttt{rec}(\mathrm{A})}{\Delta;\, \Gamma \vdash \texttt{fetch}(v) : \mathrm{A}}\ (16)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathrm{K} \quad \Delta, \alpha:\mathrm{K}=\mathrm{A};\, \Gamma \vdash e : \mathrm{B}\ \mathsf{with}\ \varphi}{\Delta;\, \Gamma \vdash \texttt{let}\ \alpha = \mathrm{A}\ \texttt{in}\ e : \{\alpha \mapsto \mathrm{A}\}\mathrm{B}\ \mathsf{with}\ \{\alpha \mapsto \mathrm{A}\}\varphi}\ (17) \qquad \frac{\Delta;\, \Gamma \vdash e_1 : \mathrm{A}_1\ \mathsf{with}\ \varphi_1 \quad \Delta\,@\,\varphi_1;\, \Gamma, x:\mathrm{A}_1 \vdash e_2 : \mathrm{A}_2\ \mathsf{with}\ \varphi_2}{\Delta;\, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathrm{A}_2\ \mathsf{with}\ \varphi_1, \varphi_2}\ (18)$$

$$\frac{\Delta, \alpha\uparrow\mathrm{K};\, \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \varphi, \alpha\downarrow \quad \alpha \notin \mathrm{FV}(\mathrm{A}) \cup \mathrm{FV}(\varphi)}{\Delta;\, \Gamma \vdash (\texttt{new}\ \alpha\uparrow\mathrm{K}\ \texttt{in}\ e : \mathrm{A}) : \mathrm{A}\ \mathsf{with}\ \varphi}\ (19)$$

$$\frac{\alpha\uparrow\mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K} \quad \mathsf{basis}_{\Delta}(\mathrm{A}) \subseteq \mathsf{writable}(\Delta) \setminus \{\alpha\}}{\Delta;\, \Gamma \vdash \alpha := \mathrm{A} : \texttt{unit}\ \mathsf{with}\ \alpha := \mathrm{A}}\ (20) \qquad \frac{\alpha\uparrow\mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta;\, \Gamma \vdash \alpha :\approx \mathrm{A} : \texttt{unit}\ \mathsf{with}\ \alpha :\approx \mathrm{A}}\ (21)$$

$$\frac{\alpha\uparrow\mathrm{K} \in \Delta \quad \Delta;\, \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \varphi \quad \Delta\,@\,\varphi \vdash \alpha \downarrow \mathrm{K}}{\Delta;\, \Gamma \vdash (e:\mathrm{A}\ \texttt{defines}\ \alpha) : \mathrm{A}\ \mathsf{with}\ (\varphi\ \mathsf{sealing}\ \alpha)}\ (22) \qquad \frac{\Delta;\, \Gamma \vdash e : \mathrm{B}\ \mathsf{with}\ \varphi \quad \Delta\,@\,\varphi \vdash \mathrm{A} \equiv \mathrm{B} : \mathbf{T}}{\Delta;\, \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \varphi}\ (23)$$

Figure 8. Static Semantics

($\alpha := \mathrm{A}$), by a datatype definition ($\alpha :\approx \mathrm{A}$), or abstractly ($\alpha\downarrow$). The last kind of definition only arises as the result of the sealing operation ($e:\mathrm{A}$ `defines` α), as discussed in the next section.

In our type system, it is useful to have a shorthand $\Delta\,@\,\varphi$ for "applying" the type effect φ to Δ. The application results in a type context that reflects the definitions in φ. Assuming that $\mathsf{dom}(\varphi) \subseteq \mathsf{writable}(\Delta)$, we define $\Delta\,@\,\varphi$ as follows:

$$\begin{array}{rl} \Delta\,@\,\varphi \overset{\text{def}}{=} & (\Delta \setminus \{\alpha\uparrow\mathrm{K} \mid \alpha\uparrow\mathrm{K} \in \Delta\ \wedge\ \alpha \in \mathsf{dom}(\varphi)\}) \\ & \cup\{\alpha:\mathrm{K}=\mathrm{A} \mid \alpha\uparrow\mathrm{K} \in \Delta\ \wedge\ \alpha := \mathrm{A} \in \varphi\} \\ & \cup\{\alpha:\mathrm{K}\approx\mathrm{A} \mid \alpha\uparrow\mathrm{K} \in \Delta\ \wedge\ \alpha :\approx \mathrm{A} \in \varphi\} \\ & \cup\{\alpha\downarrow\mathrm{K} \mid \alpha\uparrow\mathrm{K} \in \Delta\ \wedge\ \alpha\downarrow \in \varphi\} \end{array}$$

Note that variables that have been defined abstractly ($\alpha\downarrow$) are classified as stable in the new context. This is sound because $\alpha\downarrow$ arises from uses of sealing, and sealed types are always stable (see the end of Section 2.3).

Definition 3.5 (Well-Formed Type Effects)
We say that a type effect φ is well-formed in type context Δ, written $\Delta \vdash \varphi$ ok, if:

1. $\mathsf{dom}(\varphi) \subseteq \mathsf{writable}(\Delta)$
2. $\vdash \Delta\,@\,\varphi$ ok
3. $\forall\alpha := \mathrm{A} \in \varphi.\ \mathsf{basis}_{\Delta}(\mathrm{A}) \subseteq \mathsf{writable}(\Delta)$

The first two conditions are straightforward. The third condition checks that for all transparent definitions $\alpha := \mathrm{A}$ in φ, the right-hand side A does not depend on any variables β bound as $\beta:\mathrm{K}$. The reason for this is simple: if A depends on an unstable, *non-writable* β, there is no way that A can eventually become stable via the backpatching of β. Thus, since A is irrevocably unstable, there is no point in allowing the definition $\alpha := \mathrm{A}$.

3.2 Term Structure

The syntax of our term structure is shown in Figure 7. After the exposition of Section 2, the new term constructs in our language should all look familiar. A few minor exceptions: `let` $\alpha = \mathrm{A}$ `in` e enables local transparent type definitions inside expressions. One can think of this as shorthand for $\{\alpha \mapsto \mathrm{A}\}e$, that is, e with A substituted for free occurrences of α. Also, instead of a `letrec`, we employ a self-contained $\texttt{rec}_{\mathrm{A}}(x.\, e)$ expression. One can think of this as shorthand for `letrec` $x:\mathrm{A} = e$ `in` x.

For simplicity, we require that all sequencing of operations be done explicitly with the use of a `let` expression (`let` $x = e_1$ `in` e_2).

It is straightforward to code up standard left-to-right (or right-to-left) call-by-value semantics for function application, etc., using a `let`.

We say that a value context Γ is well-formed under type context Δ, written $\Delta \vdash \Gamma\ \mathsf{ok}$, if $\vdash \Delta\ \mathsf{ok}$ and $\forall x : \mathrm{A} \in \Gamma.\ \Delta \vdash \mathrm{A} : \mathbf{T}$.

Figure 8 defines the typing rules for terms. Our typing judgment ($\Delta;\ \Gamma \vdash e : \mathrm{A}\ \mathtt{with}\ \varphi$) is read: "Under type context Δ and value context Γ, the term e has type A and type effects φ." We leave off the "with φ" if $\varphi = \emptyset$.

Rules 1 through 8 are completely standard. Note that function bodies are not permitted to have type effects, *i.e.*, to define externally-bound type variables. If they were, we would need to support effect types like $\mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2$, which we argued in Section 2.2 is a problematic feature.

Rules 9 and 10 for stable universals are completely analogous to the normal universal rules (7 and 8).

Rules 11 and 12 for DPS universals are straightforward as well. The body of a DPS universal is required to define its type argument, but that is the only type effect it is allowed to have since that is the only effect written on its arrow. What if we want to write a function that takes multiple writable type arguments and defines all of them? It turns out that such a function is already encodable within the language by packaging all the writable types together as a single writable type constructor of product kind. See Section 4.1 for details.

Rules 13 and 14 for $\mathtt{fold}_\mathrm{A}$ and $\mathtt{unfold}_\mathrm{A}$ require that the type A that is being folded into or out of is some type path $\mathcal{E}\{\alpha\}$ rooted at a datatype variable α, whose underlying definition is B. These coercions witness the isomorphism between $\mathcal{E}\{\alpha\}$ and $\mathcal{E}\{\mathrm{B}\}$.[12]

Rules 15 and 16 for `rec` and `fetch` are completely straightforward. Notice that the body of a `rec` may have arbitrary type effects. Also, the canonical forms of type $\mathtt{rec}(\mathrm{A})$ are variables. In the dynamic semantics (Section 3.4), we use variables to model backpatchable memory locations.

Rule 17 processes the `let` binding of $\alpha = \mathrm{A}$ by adding that type definition to the context when typechecking the `let` body. It substitutes A for α, however, in the result type and type effect. Note that there is no need to restrict A to be stable because α's definition as A is never hidden.

Rule 18 for $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ is slightly interesting in that the type effect φ_1 engendered by e_1 must be applied to the type context Δ before typechecking e_2.

Rule 19 for $\mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ e : \mathrm{A}$, as a matter of simplicity, requires e to define and seal the new writable variable α. However, α may not escape its scope by appearing in the free variables of the result type A or in any other type effects φ that e might have. We ask for `new` to be annotated with its result type so that the typechecking algorithm does not have to guess one (via normalization) that does not refer to α.

Rule 20 for transparent type definitions, $\alpha := \mathrm{A}$, implements what we described at the end of Section 2.3. In particular, the third premise, $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \mathsf{writable}(\Delta) \setminus \{\alpha\}$, allows A to depend on any writable type variables besides α. The rule does *not* check that A is immediately stable. Stability will be checked later on, at the point when α is sealed (Rule 22).

Rule 21 for datatype definitions simply checks that the variable being defined is in fact writable. There is nothing else interesting to check because datatype definitions cannot introduce any cycles.

Rule 22 concerns the sealing construct ($e : \mathrm{A}\ \mathtt{defines}\ \alpha$). The first premise ensures that α is a writable variable (otherwise, e should certainly not be allowed to define it!). The second premise checks that e has the export type A along with some effects φ. The third premise then checks that, once φ has been applied to the type context Δ, α is stable. It is now safe to seal the definition of α. To denote the sealing of α's definition in the type effect of the conclusion, we write "φ $\mathsf{sealing}$ α," defined as follows:

Definition 3.6 (Sealing a Type Variable in a Type Effect)
Suppose φ is a type effect whose domain includes the type variable α. Let $\varphi = \varphi_1, \varphi_2$, where $\mathsf{dom}(\varphi_2) = \{\alpha\}$ (so $\alpha \notin \mathsf{dom}(\varphi_1)$). Then, we will write "$\varphi$ $\mathsf{sealing}$ α" as shorthand for $\varphi_1, \alpha\downarrow$.

Finally, Rule 23 allows for type conversion. The rule is slightly interesting in that type conversion is done in the context $\Delta @ \varphi$ instead of Δ. This is because the type of e describes the value that e evaluates to. That value exists in the "post-φ" world of $\Delta @ \varphi$, where more type definitions may be available, so it is useful to allow type conversion to occur there.

3.3 Some Interesting Properties of the Static Semantics

Here we discuss some useful properties of our type system. For any type judgment $\mathcal{J}$, we will use the notation "$\Delta \Vdash \mathcal{J}$" to signify that $\vdash \Delta\ \mathsf{ok}$ and $\Delta \vdash \mathcal{J}$. For any term judgment $\mathcal{J}$, we will use the notation "$\Delta;\ \Gamma \Vdash \mathcal{J}$" to signify that $\Delta \vdash \Gamma\ \mathsf{ok}$ and $\Delta;\ \Gamma \vdash \mathcal{J}$.

First, we have a validity property, stating that well-formed terms have well-formed types and well-formed type effects.

Proposition 3.7 (Validity)
If $\Delta;\ \Gamma \Vdash e : \mathrm{A}\ \mathtt{with}\ \varphi$, then $\Delta \vdash \mathrm{A} : \mathbf{T}$ and $\Delta \vdash \varphi\ \mathsf{ok}$.

Next, we have some substitution properties. Let a type substitution δ be a total mapping from type variables to type constructors that behaves like the identity on all but a finite set of variables, called its domain, written $\mathsf{dom}(\delta)$. Let id stand for the identity substitution. We will write $\delta\mathrm{A}$ (resp. δe, $\delta\varphi$, or $\delta\Gamma$) to signify the result of performing the substitution δ on the free variables of A (resp. e, φ, or Γ) in the usual capture-avoiding manner. Note that the free type variables of φ include $\mathsf{dom}(\varphi)$.

We must actually define the notion of *well-formed substitution* quite carefully in order to make the theorems go through:

Definition 3.8 (Well-Formed Type Substitutions)
We say that a type substitution δ maps Δ to Δ', written $\Delta' \vdash \delta : \Delta$, if:

1. $\mathsf{dom}(\delta) \subseteq \mathsf{dom}(\Delta)$
2. $\vdash \Delta\ \mathsf{ok}$ and $\vdash \Delta'\ \mathsf{ok}$
3. $\forall \alpha \uparrow \mathrm{K} \in \Delta.\ \exists \alpha' \uparrow \mathrm{K} \in \Delta'.\ \alpha' = \delta\alpha$
4. $\forall \alpha_1 \uparrow \mathrm{K}_1 \in \Delta.\ \forall \alpha_2 \uparrow \mathrm{K}_2 \in \Delta.\ (\alpha_1 \neq \alpha_2) \Rightarrow (\delta\alpha_1 \neq \delta\alpha_2)$
5. $\forall \alpha : \mathrm{K} = \mathrm{A} \in \Delta.$
   $(\Delta' \vdash \delta\alpha \equiv \delta\mathrm{A} : \mathrm{K})\ \wedge\ (\mathsf{basis}_{\Delta'}(\delta\alpha) \subseteq \mathsf{basis}_{\Delta'}(\delta\mathrm{A}))$
6. $\forall \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta.\ \exists \alpha' : \mathrm{K} \approx \mathrm{A}' \in \Delta'.$
   $(\alpha' = \delta\alpha)\ \wedge\ (\Delta' \vdash \mathrm{A}' \equiv \delta\mathrm{A} : \mathrm{K})$
7. $\forall \alpha \downarrow \mathrm{K} \in \Delta.\ \Delta' \vdash \delta\alpha \downarrow \mathrm{K}$
8. $\forall \alpha : \mathrm{K} \in \Delta.\ \Delta' \vdash \delta\alpha : \mathrm{K}$

Conditions 1, 2 and 8 are completely straightforward. Conditions 3 and 6 require that δ maps writable and datatype variables to type variables of the same classes. Condition 7 guarantees that δ maps stable type variables to stable types (not necessarily variables). Condition 4 ensures that δ does not alias two writable variables that were distinct in the original Δ. This is critical, since writable variables may only be backpatched once.

Condition 5 checks that δ maps transparent type variables to types that match their definitions. The second conjunct of this

[12] For simplicity, we have made $\mathtt{fold}_\mathrm{A}$ and $\mathtt{unfold}_\mathrm{A}$ into new canonical forms of the ordinary arrow type. In practice, one may wish to classify these values using a separate *coercion type*, so as to indicate to the compiler that they behave like the identity function at run time [23].

condition may seem redundant, but it is not. For example, suppose $\alpha : \mathbf{T} = \mathtt{unit} \in \Delta$, $\beta \uparrow \mathbf{T} \in \Delta'$, and $\delta(\alpha) = (\lambda\alpha' : \mathbf{T}. \mathtt{unit})(\beta)$. While it is certainly true that $\delta(\alpha)$ is equivalent to $\delta(\mathtt{unit})$, it is not the case that $\mathsf{basis}_{\Delta'}(\delta(\alpha)) \subseteq \mathsf{basis}_{\Delta'}(\delta(\mathtt{unit}))$ because $\delta(\alpha)$ refers to β (albeit in a useless way). This situation could potentially be avoided by defining $\mathsf{basis}_{\Delta}(\mathrm{A})$ in a more sophisticated way (*e.g.*, by first normalizing A, and then computing the set of type variables that its normal form depends on).

In any case, the reason we care about the second conjunct of Condition 5 is in order to obtain the following monotonicity property, which is useful in proving various theorems. It says essentially that, if a type A only depends on some set of writable variables, then the basis of A cannot grow unexpectedly to include other variables when the type undergoes a well-formed substitution. One obvious instance where this is important is in proving substitution (Proposition 3.12 below) for the construct "$\alpha := \mathrm{A}$" (Rule 20). When we apply a substitution δ to this construct, we want to make sure that $\delta\mathrm{A}$ does not suddenly depend on $\delta\alpha$.

If S is a set of type variables, let δS denote $\{\delta\alpha \mid \alpha \in S\}$.

Proposition 3.9 (Monotonicity)
If $\Delta' \vdash \delta : \Delta$ and $\mathsf{basis}_{\Delta}(\mathrm{A}) \subseteq \mathsf{writable}(\Delta)$,
then $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \delta(\mathsf{basis}_{\Delta}(\mathrm{A}))$.

We can now state several type substitution properties:

Proposition 3.10 (Substitution on Types)
Suppose $\Delta' \vdash \delta : \Delta$. Then:

1. If $\Delta \vdash \mathrm{A} : \mathrm{K}$, then $\Delta' \vdash \delta\mathrm{A} : \mathrm{K}$.
2. If $\Delta \vdash \mathrm{A} \downarrow \mathrm{K}$, then $\Delta' \vdash \delta\mathrm{A} \downarrow \mathrm{K}$.
3. If $\Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$, then $\Delta' \vdash \delta\mathrm{A}_1 \equiv \delta\mathrm{A}_2 : \mathrm{K}$.
4. If $\Delta \vdash \Gamma\ \mathsf{ok}$, then $\Delta' \vdash \delta\Gamma\ \mathsf{ok}$.

Proposition 3.11 (Substitution on Type Effects)
If $\Delta' \vdash \delta : \Delta$ and $\Delta \vdash \varphi\ \mathsf{ok}$,
then $\Delta' \vdash \delta\varphi\ \mathsf{ok}$ and $\Delta' @ \delta\varphi \vdash \delta : \Delta @ \varphi$.

Proposition 3.12 (Type Substitution on Terms)
If $\Delta' \vdash \delta : \Delta$ and $\Delta;\, \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \varphi$,
then $\Delta';\, \delta\Gamma \vdash \delta e : \delta\mathrm{A}\ \mathsf{with}\ \delta\varphi$.

A similar property holds for value substitutions, but the definition of a well-formed value substitution is much more obvious. See Appendix B for details and statements of other useful properties.

In proving type soundness, we have found the following lemma to be very handy. We call it the "use it or lose it" lemma because it states this simple strengthening property: if a term e is well-typed in a context where a type variable α is bound as writable, but e does not use the fact that α is writable, then e will also be well-typed in a context where α is not writable. Like the validity and substitution properties, this lemma is provable by straightforward induction on derivations.

Lemma 3.13 (Use It Or Lose It)
If $\Delta, \alpha \uparrow \mathrm{K};\, \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \varphi$ and $\Delta, \alpha : \mathrm{K} \vdash \varphi\ \mathsf{ok}$,
then $\Delta, \alpha : \mathrm{K};\, \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \varphi$.

One corollary of this lemma says that terms that have no type effects (most notably, *values*) remain well-typed even if type effects are applied to their context. This fact is important in showing that the mutable value store maintained by our dynamic semantics remains well-formed throughout execution.

Corollary 3.14 ("Pure" Terms Stay Well-Typed Under Effects)
If $\Delta;\, \Gamma \Vdash e : \mathrm{A}$ and $\Delta \vdash \varphi\ \mathsf{ok}$, then $\Delta @ \varphi;\, \Gamma \Vdash e : \mathrm{A}$.

Proof:
Let $\Delta' = (\Delta \setminus \{\alpha \uparrow \mathrm{K} \mid \alpha \uparrow \mathrm{K} \in \Delta\}) \cup \{\alpha : \mathrm{K} \mid \alpha \uparrow \mathrm{K} \in \Delta\}$. By Lemma 3.13, $\Delta';\, \Gamma \Vdash e : \mathrm{A}$. It is easy to see that $\Delta @ \varphi \vdash \mathsf{id} : \Delta'$. Thus, the desired result follows by Proposition 3.12. ■

Another corollary says that if we have an expression e referring to two writable type variables of the same kind, and e only depends on one of them being writable, then we can merge them into one writable type variable. As stated here, this is exactly what we need in order to prove type preservation in the case of β-reduction for DPS universal types (cf. Rule 11 in Figure 8, and Rule 28 in Figure 9).

Corollary 3.15 (Merging Together Two Writable Types)
If $\Delta \vdash \Gamma\ \mathsf{ok}$ and $\beta \uparrow \mathrm{K} \in \Delta$
and $\Delta, \alpha \uparrow \mathrm{K};\, \Gamma, x : \mathrm{A} \Vdash e : \mathrm{B}\ \mathsf{with}\ \alpha\downarrow$,
then $\Delta;\, \Gamma, x : \{\alpha \mapsto \beta\}\mathrm{A} \Vdash \{\alpha \mapsto \beta\}e : \{\alpha \mapsto \beta\}\mathrm{B}\ \mathsf{with}\ \beta\downarrow$.

Proof: Let $\Delta = \Delta', \beta \uparrow \mathrm{K}$.
By Lemma 3.13, $\Delta', \beta : \mathrm{K}, \alpha \uparrow \mathrm{K};\, \Gamma, x : \mathrm{A} \Vdash e : \mathrm{B}\ \mathsf{with}\ \alpha\downarrow$. It is easy to see that $\Delta \vdash \{\alpha \mapsto \beta\} : \Delta', \beta : \mathrm{K}, \alpha \uparrow \mathrm{K}$. Thus, the desired result follows by Proposition 3.12. ■

Finally, although we do not provide a typechecking algorithm here, it is completely straightforward to write one that, given well-formed contexts Δ and Γ and a well-formed term e, synthesizes a unique type effect φ for e, as well as a type A that is unique up to type conversion under $\Delta @ \varphi$.

3.4 Dynamic Semantics and Type Soundness

We define the dynamic semantics of our language in Figure 9 using an abstract machine semantics. A machine state Ω is either of the form $\mathsf{BlackHole}$ or $(\Delta;\, \omega;\, \mathcal{C};\, e)$. The former arises when an attempt is made to `fetch` a recursive location whose contents have not yet been initialized. In the normal state, Δ is the current type context (*i.e.*, the type *store*), ω is the current value store, $\mathcal{C}$ is the current continuation, and e is the expression currently being evaluated.

In the language defined here, the only purpose of the value store is to support a backpatching semantics for recursion. It could naturally be extended to support other things, such as mutable references. A value store ω binds variables to either values (v) or junk (**?**). Assuming $x \in \mathsf{dom}(\omega)$, we write $\omega(x)$ to denote the contents of location x in ω. Mirroring the syntax of type effect application, we write $\omega @ x := v$ to signify the store ω' with the property that $\mathsf{dom}(\omega') = \mathsf{dom}(\omega)$, $\omega'(x) = v$, and $\omega'(y) = \omega(y)$ for all $y \in \mathsf{dom}(\omega)$, $y \neq x$.

We define well-formedness of value stores as follows:

Definition 3.16 (Run-Time Value Contexts)
We say that a value context Γ is *run-time* if it only contains bindings of the form $x : \mathtt{rec}(\mathrm{A})$.

Definition 3.17 (Well-Formed Value Stores)
We say that a value store ω is well-formed in Δ and has type Γ, written $\Delta \vdash \omega : \Gamma$, if:

1. $\Delta \vdash \Gamma\ \mathsf{ok}$ and Γ is run-time
2. $\mathsf{dom}(\omega) = \mathsf{dom}(\Gamma)$
3. $\forall x : \mathtt{rec}(\mathrm{A}) \in \mathsf{dom}(\Gamma)$.
   either $\omega(x) = \textbf{?}$ or $\Delta;\, \Gamma \vdash \omega(x) : \mathrm{A}$

Continuations $\mathcal{C}$ are represented as stacks of continuation frames $\mathcal{F}$. There are only two continuation frames. The first is $\mathtt{let}\ x = \bullet\ \mathtt{in}\ e$, which waits for x's binding to evaluate to a value v and then plugs v in for x in e. The second is $\mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet)$, which

$$\begin{array}{lrl}
\text{Machine States} & \Omega ::= & (\Delta;\, \omega;\, \mathcal{C};\, e) \mid \mathsf{BlackHole} \\
\text{Value Stores} & \omega ::= & \emptyset \mid \omega, x \mapsto v \mid \omega, x \mapsto \mathbf{?} \\
\text{Continuations} & \mathcal{C} ::= & \bullet \mid \mathcal{C} \circ \mathcal{F} \\
\text{Continuation Frames} & \mathcal{F} ::= & \mathtt{let}\ x = \bullet\ \mathtt{in}\ e \mid \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet)
\end{array}$$

Reductions: $e \rightsquigarrow e'$

$$\frac{}{\pi_i(v_1, v_2) \rightsquigarrow v_i}\ (24) \qquad \frac{}{(\lambda x{:}\mathrm{A}.\, e)(v) \rightsquigarrow \{x \mapsto v\}e}\ (25)$$

$$\frac{}{(\Lambda\alpha{:}\mathrm{K}.\, e)[\mathrm{A}] \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (26) \qquad \frac{}{(\Lambda\alpha\downarrow\mathrm{K}.\, e)[\mathrm{A}] \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (27)$$

$$\frac{}{(\Lambda\alpha\uparrow\mathrm{K}.\, \lambda x{:}\mathrm{A}.\, e)[\beta](v) \rightsquigarrow \{\alpha \mapsto \beta\}\{x \mapsto v\}e}\ (28) \qquad \frac{}{\mathtt{unfold}_{\mathrm{A}}(\mathtt{fold}_{\mathrm{B}}(v)) \rightsquigarrow v}\ (29)$$

$$\frac{}{\mathtt{let}\ \alpha = \mathrm{A}\ \mathtt{in}\ e \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (30) \qquad \frac{}{e : \mathrm{A}\ \mathtt{defines}\ \alpha \rightsquigarrow e}\ (31)$$

Machine state transitions: $\Omega \rightsquigarrow \Omega'$

$$\frac{e \rightsquigarrow e'}{(\Delta;\, \omega;\, \mathcal{C};\, e) \rightsquigarrow (\Delta;\, \omega;\, \mathcal{C};\, e')}\ (32)$$

$$\frac{}{(\Delta;\, \omega;\, \mathcal{C};\, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\Delta;\, \omega;\, \mathcal{C} \circ \mathtt{let}\ x = \bullet\ \mathtt{in}\ e_2;\, e_1)}\ (33) \qquad \frac{}{(\Delta;\, \omega;\, \mathcal{C} \circ \mathtt{let}\ x = \bullet\ \mathtt{in}\ e;\, v) \rightsquigarrow (\Delta;\, \omega;\, \mathcal{C};\, \{x \mapsto v\}e)}\ (34)$$

$$\frac{x \notin \mathsf{dom}(\omega)}{(\Delta;\, \omega;\, \mathcal{C};\, \mathtt{rec}_{\mathrm{A}}(x.\, e)) \rightsquigarrow (\Delta;\, \omega, x \mapsto \mathbf{?};\, \mathcal{C} \circ \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet);\, e)}\ (35) \qquad \frac{x \in \mathsf{dom}(\omega)}{(\Delta;\, \omega;\, \mathcal{C} \circ \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet);\, v) \rightsquigarrow (\Delta;\, \omega\,@\,x := v;\, \mathcal{C};\, v)}\ (36)$$

$$\frac{x \in \mathsf{dom}(\omega) \quad \omega(x) = v}{(\Delta;\, \omega;\, \mathcal{C};\, \mathtt{fetch}(x)) \rightsquigarrow (\Delta;\, \omega;\, \mathcal{C};\, v)}\ (37) \qquad \frac{x \in \mathsf{dom}(\omega) \quad \omega(x) = \mathbf{?}}{(\Delta;\, \omega;\, \mathcal{C};\, \mathtt{fetch}(x)) \rightsquigarrow \mathsf{BlackHole}}\ (38)$$

$$\frac{\alpha \notin \mathsf{dom}(\Delta)}{(\Delta;\, \omega;\, \mathcal{C};\, \mathtt{new}\ \alpha\uparrow\mathrm{K}\ \mathtt{in}\ e : \mathrm{A}) \rightsquigarrow (\Delta, \alpha\uparrow\mathrm{K};\, \omega;\, \mathcal{C};\, e)}\ (39)$$

$$\frac{\alpha\uparrow\mathrm{K} \in \Delta}{(\Delta;\, \omega;\, \mathcal{C};\, \alpha := \mathrm{A}) \rightsquigarrow (\Delta\,@\,\alpha := \mathrm{A};\, \omega;\, \mathcal{C};\, \mathtt{()})}\ (40) \qquad \frac{\alpha\uparrow\mathrm{K} \in \Delta}{(\Delta;\, \omega;\, \mathcal{C};\, \alpha :\approx \mathrm{A}) \rightsquigarrow (\Delta\,@\,\alpha :\approx \mathrm{A};\, \omega;\, \mathcal{C};\, \mathtt{()})}\ (41)$$

Well-formed continuations: $\Delta;\, \Gamma \vdash \mathcal{C} : \mathrm{A}\ \mathsf{cont}$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T}}{\Delta;\, \Gamma \vdash \bullet : \mathrm{A}\ \mathsf{cont}}\ (42) \qquad \frac{\Delta;\, \Gamma \vdash \mathcal{F} : \mathrm{A} \leadsto \mathrm{B}\ \mathsf{with}\ \varphi \quad \Delta\,@\,\varphi;\, \Gamma \vdash \mathcal{C} : \mathrm{B}\ \mathsf{cont}}{\Delta;\, \Gamma \vdash \mathcal{C} \circ \mathcal{F} : \mathrm{A}\ \mathsf{cont}}\ (43) \qquad \frac{\Delta;\, \Gamma \vdash \mathcal{C} : \mathrm{B}\ \mathsf{cont} \quad \Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathbf{T}}{\Delta;\, \Gamma \vdash \mathcal{C} : \mathrm{A}\ \mathsf{cont}}\ (44)$$

Well-formed continuation frames: $\Delta;\, \Gamma \vdash \mathcal{F} : \mathrm{A} \leadsto \mathrm{B}\ \mathsf{with}\ \varphi$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta;\, \Gamma, x{:}\mathrm{A} \vdash e : \mathrm{B}\ \mathsf{with}\ \varphi}{\Delta;\, \Gamma \vdash \mathtt{let}\ x = \bullet\ \mathtt{in}\ e : \mathrm{A} \leadsto \mathrm{B}\ \mathsf{with}\ \varphi}\ (45) \qquad \frac{x : \mathtt{rec}(\mathrm{A}) \in \Gamma}{\Delta;\, \Gamma \vdash \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet) : \mathrm{A} \leadsto \mathrm{A}\ \mathsf{with}\ \emptyset}\ (46)$$

Figure 9. Dynamic Semantics

waits for the body of a recursive term to evaluate to a value v and then backpatches the recursive memory location x with v.

The typing judgments for continuations and continuation frames are shown in Figure 9. The latter is slightly interesting in that a frame may have type effects. One can read the judgment ($\Delta;\, \Gamma \vdash \mathcal{F} : \mathrm{A} \leadsto \mathrm{B}\ \mathsf{with}\ \varphi$) as: "starting in type context Δ, the frame $\mathcal{F}$ takes a value of type A and returns a value of type B while engendering the effects in φ." Continuations $\mathcal{C}$ may of course have type effects as well, but they are irrelevant because we never return from a continuation.

The dynamic semantics itself is entirely what one would expect given our discussion from Section 2. Like sealing in ML, sealing in our language is a static abstraction mechanism with no run-time significance. The `new` construct, on the other hand, has the effect of creating a new entry in the type store at run time. Backpatching a writable type is modeled by actually updating its entry in the type store. In short, the semantics is faithful to our intuition.

That said, it is worth noting that, while the type store Δ is useful in defining the dynamic semantics and proving type soundness, it does not have any real influence on run-time computation. In other words, the dynamic semantics of Figure 9 never consults the type store in order to determine the identity of a type variable and make a transition based on that information. Consequently, there is no need in an actual implementation to construct and maintain the type store, and the operations for creation and definition of abstract type variables may both be compiled as no-ops.

We can now define what it means to be a well-formed machine state and state the standard preservation and progress theorems leading to type soundness. The interesting part of the definition is that the expression e currently being evaluated may have type

$$\llbracket \forall \alpha_1 \uparrow \mathrm{K}_1, \alpha_2 \uparrow \mathrm{K}_2.\, \mathrm{A}(\alpha_1)(\alpha_2) \xrightarrow{\alpha_1\downarrow,\,\alpha_2\downarrow} \mathrm{B}(\alpha_1)(\alpha_2) \rrbracket$$

$$\overset{\mathrm{def}}{=} \quad \forall \alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2.\, \mathrm{A}(\pi_1\alpha)(\pi_2\alpha) \xrightarrow{\alpha\downarrow} \mathrm{B}(\pi_1\alpha)(\pi_2\alpha)$$

$$\llbracket \Lambda \alpha_1 \uparrow \mathrm{K}_1, \alpha_2 \uparrow \mathrm{K}_2.\, \lambda x : \mathrm{A}(\alpha_1)(\alpha_2).\, (e : \mathrm{B}(\alpha_1)(\alpha_2)) \rrbracket$$

$$\begin{array}{ll} \overset{\mathrm{def}}{=} & \Lambda \alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2.\, \lambda x : \mathrm{A}(\pi_1\alpha)(\pi_2\alpha). \\ & \quad \texttt{new}\ \alpha_1 \uparrow \mathrm{K}_1,\ \alpha_2 \uparrow \mathrm{K}_2\ \texttt{in} \\ & \qquad (\texttt{let ()} = \alpha := \langle \alpha_1, \alpha_2 \rangle\ \texttt{in}\ e) \\ & \qquad :\ \mathrm{B}(\pi_1\alpha)(\pi_2\alpha)\ \texttt{defines}\ \alpha \end{array}$$

$$\llbracket v_1[\alpha_1][\alpha_2](v_2)\ :\ \mathrm{B}(\alpha_1)(\alpha_2) \rrbracket$$

$$\begin{array}{ll} \overset{\mathrm{def}}{=} & \texttt{new}\ \alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2\ \texttt{in} \\ & \quad (\texttt{let ()} = \alpha_1 := \pi_1\alpha\ \texttt{in} \\ & \quad\ \ \texttt{let ()} = \alpha_2 := \pi_2\alpha\ \texttt{in} \\ & \qquad v_1[\alpha](v_2)) \\ & \quad :\ \mathrm{B}(\alpha_1)(\alpha_2)\ \texttt{defines}\ \alpha_1,\ \alpha_2 \end{array}$$

Figure 10. Encoding of Multiple-Argument DPS Universals

effects φ, so these effects must be incorporated into the "starting" context of the continuation $\mathcal{C}$.

Definition 3.18 (Well-Formed Machine States)
We say that a machine state Ω is well-formed, written $\vdash \Omega$ ok, if either $\Omega = \mathsf{BlackHole}$, or $\Omega = (\Delta;\ \omega;\ \mathcal{C};\ e)$ and there exist Γ, A and φ such that:

1. $\Delta \vdash \omega : \Gamma$
2. $\Delta; \Gamma \vdash e : \mathrm{A}$ with φ and $\Delta\,@\,\varphi; \Gamma \vdash \mathcal{C} : \mathrm{A}$ cont

Theorem 3.19 (Preservation)
If $\vdash \Omega$ ok and $\Omega \leadsto \Omega'$, then $\vdash \Omega'$ ok.

Definition 3.20 (Terminal States)
A machine state Ω is *terminal* if it is of the form $\mathsf{BlackHole}$ or $(\Delta;\ \omega;\ \bullet;\ v)$.

Definition 3.21 (Stuck States)
A machine state Ω is *stuck* if it is not terminal and there is no state Ω' such that $\Omega \leadsto \Omega'$.

Theorem 3.22 (Progress)
If $\vdash \Omega$ ok, then Ω is not stuck.

(The progress theorem depends on a standard canonical forms lemma, which is given in Appendix B.)

Corollary 3.23 (Type Soundness)
If $\emptyset;\ \emptyset \vdash e : \mathrm{A}$, then the execution of $(\emptyset;\ \emptyset;\ \bullet;\ e)$ never enters a stuck state.

4. Encodings in Destination-Passing Style

4.1 Multiple-Argument DPS Universal Types

It is likely that in practice one may wish to define a function of DPS universal type that takes multiple writable type arguments and defines all of them. However, our language as presented in Section 3 appears to allow DPS universals to take only a single writable type argument. Figure 10 illustrates that in fact multiple-argument DPS universals can be encoded in terms of single-argument ones. For simplicity, we take "multiple-argument" to mean "two-argument," but the technique can easily be generalized to n arguments.

The idea is to encode a function taking two writable type arguments α_1 and α_2 (of kinds K_1 and K_2) as a function taking one writable type argument α (of kind $\mathrm{K}_1 \times \mathrm{K}_2$). In Figure 10, we assume the value argument and result types have the form $\mathrm{A}(\alpha_1)(\alpha_2)$ and $\mathrm{B}(\alpha_1)(\alpha_2)$, respectively, where $\alpha_1, \alpha_2 \notin \mathrm{FV}(\mathrm{A}) \cup \mathrm{FV}(\mathrm{B})$.

In the introduction form, we divide the single α into two writable variables α_1 and α_2 by creating those variables with a `new` and then defining the original α in terms of them. For the elimination form, it is the reverse. We start with two writable variables, and in order to apply the DPS universal we must package them up as one. This is achieved by simply creating a new α of the pair kind, and then defining the original writable variables as projections from it. For the elimination form to be well-typed, it is important of course that α_1 and α_2 be distinct. Also note that we make use of `new` and sealing constructs that create (or seal) multiple variables. These are simply shorthand for several nested `new`'s or sealings.

In the encoding of both the introduction and elimination forms, we rely heavily on the ability to define a writable variable transparently in terms of another writable variable, which is then subsequently defined in some stable way. This provides good motivation for our policy that definitions of writable variables need not be stable immediately, but only by the time they are sealed (as discussed at the end of Section 2.3).

$$\begin{array}{l} \llbracket \exists \alpha \downarrow \mathrm{K}.\, \mathrm{A} \rrbracket_{\mathrm{DPS}} \\ \quad \overset{\mathrm{def}}{=} \quad \forall \alpha \uparrow \mathrm{K}.\, \texttt{unit} \xrightarrow{\alpha\downarrow} \mathrm{A} \end{array}$$

$$\begin{array}{l} \llbracket \texttt{pack}\ [\mathrm{A}, v]\ \texttt{as}\ \exists \alpha \downarrow \mathrm{K}.\, \mathrm{B} \rrbracket_{\mathrm{DPS}} \\ \quad \overset{\mathrm{def}}{=} \quad \Lambda \alpha \uparrow \mathrm{K}.\, \lambda\texttt{()}. \\ \qquad\qquad (\texttt{let ()} = \alpha := \mathrm{A}\ \texttt{in}\ v) \\ \qquad\qquad :\ \mathrm{B}\ \texttt{defines}\ \alpha \end{array}$$

$$\begin{array}{l} \llbracket \texttt{let}\ [\alpha, x] = \texttt{unpack}\ v\ \texttt{in}\ (e : \mathrm{A}) \rrbracket_{\mathrm{DPS}} \\ \quad \overset{\mathrm{def}}{=} \quad \texttt{new}\ \alpha \uparrow \mathrm{K}\ \texttt{in} \\ \qquad\qquad (\texttt{let}\ x = v[\alpha]\texttt{()}\ \texttt{in}\ e) : \mathrm{A} \end{array}$$

Figure 11. DPS Universal Encoding of Existentials

4.2 Existential Types

In Section 2.2, we argued that the special case of the DPS universal in which the value argument has `unit` type can be viewed as a kind of existential type. We now make that argument precise. Figure 11 shows how existential types and their introduction and elimination forms may be encoded using that special case of the DPS universal type. The caveat is that DPS universals are not capable of encoding arbitrary existentials $\exists\alpha : \mathrm{K}.\, \mathrm{A}$, but only what we call *stable existentials*, which we write $\exists \alpha \downarrow \mathrm{K}.\, \mathrm{A}$. As the name suggests, a value of stable existential type is a package whose type component is stable, and the standard CPS encoding of existentials can be trivially modified to define $\exists \alpha \downarrow \mathrm{K}.\, \mathrm{A}$ as shorthand for $\forall \beta : \mathbf{T}.\, (\forall \alpha \downarrow \mathrm{K}.\, \mathrm{A} \to \beta) \to \beta$.

To package type constructor A with value v, we write a DPS function that asks for a writable abstract type name α, and then returns v after defining α to be A. The data abstraction one normally associates with existential introduction is achieved here by our sealing construct. Note that A must be stable in order for the encoding of `pack` to be well-typed, since A is used to define the writable variable α.

To unpack an existential value v, we (the client) must first create a new writable type name α and then pass it to v to be defined. A potential benefit of the DPS encoding over the CPS encoding is that it allows the body e of the `unpack` to have arbitrary type effects, so long as they do not refer to α. In the CPS encoding of `unpack`, e must be encapsulated in a function, so it is not allowed to define any externally-bound variables.

The DPS encoding is encouraging because it means that our approach to recursive type generativity is fundamentally compatible with the traditional understanding of generativity in terms of existential types. For instance, returning to the bootstrapped heap example from Figure 4, we can now rewrite the type of `MkHeap` as

$$\forall\alpha\downarrow\mathbf{T}.\,\mathtt{ORDERED}(\alpha)\rightarrow\exists\beta\downarrow\mathbf{T}.\,\mathtt{HEAP}(\alpha)(\beta)$$

This looks just like the standard F_ω interpretation of a generative functor signature, except that we have replaced the normal type variable bindings by stable ones. It is not even necessary for the existential in the result type of `MkHeap` to be encoded in DPS—a value of stable existential type (under any encoding) can always be coerced to $[\![\,\exists\alpha\downarrow\mathrm{K}.\,\mathrm{A}\,]\!]_{\mathrm{DPS}}$ by first unpacking its components and then repacking them using the DPS encoding of `pack`.

5. Related Work

As discussed in Section 1, there has been much work on extending ML with recursive modules, but a clear account of recursive type generativity has until now remained elusive. Crary, Harper and Puri [3] have given a foundational type-theoretic account of recursive modules, but it does not consider the interaction of recursion with ML's sealing mechanism (opaque signature ascription). Russo has formalized and implemented recursive modules as an extension to the Moscow ML compiler [15]. Under his extension, any type components of a recursive module that are referred to recursively must have their definitions made public to the whole module. Leroy has implemented recursive modules in O'Caml [11], but has not provided any formal account of its semantics. With none of these approaches is it possible to implement the bootstrapped heap example using a generative `MkHeap` functor.

In reaction to the difficulties of incorporating recursive linking into the ML module system, others have investigated ways of replacing ML's notion of module with some alternative mechanism for which recursive linking is the norm and hierarchical linking a special case. Ancona and Zucca's CMS calculus, in particular, has been highly influential and led to a considerable body of work on "mixin modules" [2]. However, it basically ignores all issues involving type components (and hence, data abstraction) in modules.

More recently, Duggan has developed a language of "recursive DLLs" [7]. His calculus is not intended as the basis of a source-level language, but rather as an "interconnection" language for dynamic linking and loading of shared libraries. Based on his informal discussion, Duggan appears to address some of the problems of recursive ADT's in a manner similar to the typechecking algorithm we suggested in Section 1.1. It is difficult, though, to determine precisely how his approach relates to ours because he is working in a relatively low-level setting. In addition, Duggan simplifies the problem to some extent by not supporting full ML-style transparent `type` definitions, but only a limited form of sharing constraint that is restricted to atomic types.

Interestingly, the work that seems most closely related to our approach comes from the Scheme community. Flatt and Felleisen developed a recursive-module-like construct called "units" for use in MzScheme [16]. While MzScheme is dynamically-typed, their paper formalizes an extension of units to the statically-typed setting as well [8]. A unit has some set of imports and exports, which may include abstract types. Two units may be "compounded" into one, with each unit's exports being used to satisfy the other's imports.

While our approach differs from units in many details, there are considerable similarities in terms of expressive power. For instance, one can think of the DPS universal type $\forall\alpha\uparrow\mathrm{K}.\,\mathrm{A}\xrightarrow{\alpha\downarrow}\mathrm{B}$ as the type of a unit with a value import of type A, a value export of type B, and a *type export* α. (We model type *imports* separately, via standard universal quantification.) The avoidance of transparent type cycles, which we handle by distinguishing between stable and unstable forms of universal quantification, is dealt with in the unit language by means of unit "signatures," which explicitly specify which export types of a unit depend on which import types.

Ultimately, the main distinction between our approach and units is that, while units do many things at once, we have tried instead to isolate orthogonal concerns as much as possible. As a result, our language constructs are more lightweight, and our semantics is easier to follow. In contrast, the unit typing rules are large and complex. Given that units were intended as a realistic, programmable language construct, this complexity is understandable, but there are some other problems with units as well. In particular, they lack support for ML-style type sharing, and their emphasis on "external linking" forces one to program in a recursive analogue of "fully functorized" style. Nonetheless, we hope that our present account of recursive type generativity will help draw attention to some of the interesting and novel features of units that the existing work on recursive ML-style modules has heretofore ignored.

Finally, unrelated to recursive modules, Rossberg [18] gives an account of type generativity that, like ours, provides a `new` construct for creating fresh abstract types at run time. Rossberg's focus, however, is not on recursion but on the interaction of data abstraction and run-time type analysis. Thus, his system requires one to define an abstract type at the same point where it is created.

6. Conclusion

In previous work with Karl Crary and Bob Harper [6], we gave an interpretation of ML-style modularity in which type generativity was treated as a computational effect. We view the present work as a continuation and refinement of that interpretation. Specifically, while we still model the definition of new abstract types as a kind of effect, we allow abstract types to be created and used *before* they are defined, thus making it possible to link such types recursively.

One complaint that can be leveled against our approach is that the interpretation of type-level recursion in terms of backpatching is highly operational and may make it difficult to reason about abstraction guarantees. Admittedly, while it is possible to state a weak syntactic abstraction property—*e.g.,* that if a program contains a subterm e in which a writable abstract type is defined and sealed, then e may be replaced by another subterm e' that defines the abstract type in a different way—it is not clear what a stronger abstraction theorem would look like. This remains an important consideration for future work.

Another interesting question is whether the full power of our language is useful, or only a fragment of it is really needed for practical purposes. For example, our type system allows the programmer to define types at run time based on information that is only dynamically available. If one is only interested in supporting "second-class" recursive modules, then the language we have presented here is more powerful than necessary. In that case, it is worth considering whether there is a weaker subset of the language that suffices and is easier to implement in practice.

This question is of course tied in with the more general problem of scaling the ideas of this paper to the level of a module language. The path from this paper to a full-blown module system is not immediate, primarily because the approach to data abstraction we have taken here is at least superficially quite different from the way that type systems for modules have traditionally been formalized. That said, we believe it should not be fundamentally difficult.

Acknowledgments

We would like to thank Aleks Nanevski for suggesting that we pursue an idea along these lines years ago, as well as the anonymous referees for giving such helpful and detailed comments.

References

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

[2] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.

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

[4] Derek Dreyer. A type system for well-founded recursion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 293–305, Venice, Italy, January 2004.

[5] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, January 2003.

[7] Dominic Duggan. Type-safe linking with recursive DLL's and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, November 2002.

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

[9] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, MA, August 1986.

[10] Mark P. Jones. Using parameterized signatures to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 68–78, St. Petersburg Beach, FL, 1996.

[11] Xavier Leroy. The Objective Caml system: Documentation and user's manual. `http://www.ocaml.org/`.

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

[13] Xavier Leroy. A proposal for recursive modules in Objective Caml, May 2003. Available from the author's website.

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

[15] Moscow ML. `www.dina.kvl.dk/~sestoft/mosml.html`.

[16] MzScheme. `www.plt-scheme.org/software/mzscheme/`.

[17] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[18] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*, Uppsala, Sweden, 2003.

[19] Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming (ICFP)*, pages 50–61, Florence, Italy, September 2001.

[20] Christopher A. Stone. Type defi nitions. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 9. MIT Press, 2005.

[21] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 2005. To appear.

[22] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[23] Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed compilation of recursive datatypes. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, New Orleans, January 2003.

[24] Philip Wadler. *Listlessness is Better than Laziness*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1985.

A. Representative Rules for Kinding and Equivalence of Type Constructors

The definition of well-formedness and equivalence for type constructors in our language is entirely straightforward. Here we give some representative rules.

$\boxed{\textbf{Well-formed type constructors: } \Delta \vdash \mathrm{A} : \mathrm{K}}$

$$\frac{\Delta(\alpha) = \mathrm{K}}{\Delta \vdash \alpha : \mathrm{K}} \qquad \frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}_1 : \mathbf{T} \quad \Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}_2 : \mathbf{T}}{\Delta \vdash \forall\alpha\uparrow\mathrm{K}.\, \mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2 : \mathbf{T}}$$

$\boxed{\textbf{Type constructor equivalence: } \Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}}$

$$\frac{\alpha{:}\mathrm{K}{=}\mathrm{A} \in \Delta}{\Delta \vdash \alpha \equiv \mathrm{A} : \mathrm{K}}$$

$$\frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}_1 \equiv \mathrm{B}_1 : \mathbf{T} \quad \Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}_2 \equiv \mathrm{B}_2 : \mathbf{T}}{\Delta \vdash \forall\alpha\uparrow\mathrm{K}.\, \mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2 \equiv \forall\alpha\uparrow\mathrm{K}.\, \mathrm{B}_1 \xrightarrow{\alpha\downarrow} \mathrm{B}_2 : \mathbf{T}}$$

$$\frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}_1 \equiv \mathrm{B}_1 : \mathrm{K}' \quad \Delta \vdash \mathrm{A}_2 \equiv \mathrm{B}_2 : \mathrm{K}}{\Delta \vdash (\lambda\alpha{:}\mathrm{K}.\, \mathrm{A}_1)(\mathrm{A}_2) \equiv \{\alpha \mapsto \mathrm{B}_2\}\mathrm{B}_1 : \mathrm{K}'}$$

$$\frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{A}(\alpha) \equiv \mathrm{B}(\alpha) : \mathrm{K}' \quad \alpha \notin \mathrm{FV}(\mathrm{A}) \cup \mathrm{FV}(\mathrm{B})}{\Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathrm{K} \to \mathrm{K}'}$$

B. Additional Meta-Theoretic Properties

Proposition B.1 (Properties of Type Effect Application)
Suppose $\Delta \vdash \varphi$ ok. Then:

1. $\Delta \vdash \mathrm{A} : \mathrm{K}$ if and only if $\Delta \mathbin{@} \varphi \vdash \mathrm{A} : \mathrm{K}$.
2. If $\Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$, then $\Delta \mathbin{@} \varphi \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$.
3. If $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \mathsf{basis}_\Delta(\mathrm{B})$, then $\mathsf{basis}_{\Delta \mathbin{@} \varphi}(\mathrm{A}) \subseteq \mathsf{basis}_{\Delta \mathbin{@} \varphi}(\mathrm{B})$.
4. If $\Delta \mathbin{@} \varphi \vdash \varphi'$ ok, then $\Delta \vdash \varphi, \varphi'$ ok and $\Delta \mathbin{@} (\varphi, \varphi') = (\Delta \mathbin{@} \varphi) \mathbin{@} \varphi'$.

Definition B.2 (Well-Formed Value Substitutions)
We say that a value substitution γ maps Γ to Γ' under Δ, written $\Delta;\ \Gamma' \vdash \gamma : \Gamma$, if:

1. $\mathsf{dom}(\gamma) \subseteq \mathsf{dom}(\Gamma)$
2. $\Delta \vdash \Gamma$ ok and $\Delta \vdash \Gamma'$ ok
3. $\forall x{:}\mathrm{A} \in \Gamma.\ \Delta;\ \Gamma' \vdash \gamma x : \mathrm{A}$

Proposition B.3 (Value Substitution on Terms)
If $\Delta;\ \Gamma' \vdash \gamma : \Gamma$ and $\Delta;\ \Gamma \vdash e : \mathrm{A}$ with φ, then $\Delta;\ \Gamma' \vdash \gamma e : \mathrm{A}$ with φ.

Lemma B.4 (Canonical Forms)
Suppose $\Delta;\ \Gamma \Vdash v : \mathrm{A}$ and Γ is run-time. Then:

1. If $\mathrm{A} = \texttt{unit}$, then v is of the form `()`.
2. If $\mathrm{A} = \mathrm{A}_1 \times \mathrm{A}_2$, then v is of the form (v_1, v_2).
3. If $\mathrm{A} = \mathrm{A}_1 \to \mathrm{A}_2$, then v is of the form $\lambda x{:}\mathrm{B}.\, e$ or $\texttt{fold}_\mathrm{B}$ or $\texttt{unfold}_\mathrm{B}$.
4. If $\mathrm{A} = \forall\alpha{:}\mathrm{K}.\, \mathrm{A}$, then v is of the form $\Lambda\alpha{:}\mathrm{K}.\, e$.
5. If $\mathrm{A} = \forall\alpha\downarrow\mathrm{K}.\, \mathrm{A}$, then v is of the form $\Lambda\alpha\downarrow\mathrm{K}.\, e$.
6. If $\mathrm{A} = \forall\alpha\uparrow\mathrm{K}.\, \mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2$, then v is of the form $\Lambda\alpha\uparrow\mathrm{K}.\, \lambda x{:}\mathrm{B}.\, e$.
7. If $\mathrm{A} = \texttt{rec}(\mathrm{A}')$, then v is of the form x.
8. If $\mathrm{A} = \mathcal{E}\{\alpha\}$, where $\alpha{:}\mathrm{K} \approx \mathrm{B} \in \Delta$, then v is of the form $\texttt{fold}_{\mathrm{A}'}(v')$.