JFP **17** (4 & 5): 433–471, 2007.
doi:10.1017/S0956796807006429 Printed in the United Kingdom

Recursive type generativity

DEREK DREYER

Toyota Technological Institute, Chicago, IL 60637, USA
(*e-mail:* 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 supply each one's identity 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.

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 such as 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*, that is, that `M` will evaluate to `V` without requiring the value of `X` prematurely. In previous work (Dreyer 2004), we studied this problem in detail and proposed a type-based approach to guaranteeing well-founded recursion.

In this article, 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 article.) To begin, however, we 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 component in each module refers to the abstract type provided by the other module.

The code here is clearly contrived—for example, `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 because of the interaction of recursion and abstract types.[2] 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.[3]

[1] We make use here of parameterized signatures (Jones 1996), a mechanism not present in ML. While they are not strictly necessary, they serve to simplify the presentation. See Dreyer (2006) for alternative variants of this example that do not rely on parameterized signatures.

[2] For a more realistic example, see Chapter 5 of Dreyer's thesis (2005b).

[3] Incidentally, one 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 *et al.* (1999) in theory and implemented by Russo (2001) and Leroy (2003) in practice. Subject to certain

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

```
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
```

Fig. 1. Mutually recursive abstract data types.

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 really 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 $\texttt{SIG}_\texttt{A}\texttt{(X)}$ 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

restrictions, they are straightforward to account for semantically, but their semantics is not the focus of this article.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

```
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)
```

Fig. 2. Bootstrapped heap example.

`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`. In addition, 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—Dreyer's thesis (2005b) formalizes such an algorithm—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 *ad hoc* algorithm.

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. The example is adapted and simplified from one given by Russo (2001).

In the present version, the goal is to define a recursive data type `Boot.t` of so-called "bootstrapped heaps," a data structure proposed by Okasaki (1998). The important feature of bootstrapped heaps (for our purposes) is that they are defined recursively in terms of heaps of themselves, where the heap data structure is constructed by applying a library functor (in this case, `MkHeap`) to the `Boot` module.

The problem with this definition, at least in the case of Standard ML semantics (Milner *et al.* 1997), 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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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* (Leroy 1995). 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 `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 (Leroy 2004), for O'Caml supports only 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.[4] It seems unfortunate that `MkHeap` is *required* to be applicative.

1.3 Overview

Given our exposition thus far, one may wonder: Is recursion fundamentally at odds with type generativity? In this article, we 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 supply each one's identity 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 known only 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 ADTs 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,[5] we offer an alternative *destination-passing* interpretation. Briefly, instead of viewing a value of existential type $\exists\alpha.\,A$ 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

[4] For more details, see the discussion in Dreyer *et al.* (2003).
[5] See Section 2.2 for details.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

```
SIG_A = λα:T.λβ:T.{f : α → β × α, ...}
SIG_B = λα:T.λβ:T.{g : α → β × α, ...}
SIG   = λα:T.λβ:T.{A : SIG_A(α)(β), B : SIG_B(α)(β)}

new α↑T, β↑T in
  letrec X : SIG(α)(β) =
    {A = set α := int in {f = ...} : SIG_A(α)(β),
     B = set β := bool in {g = ...} : SIG_B(α)(β)}
  in ...
```

Fig. 3. New encoding of example from Figure 1.

existential, our approach makes it significantly easier to link abstract data types together recursively.

The rest of the article 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. 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 (Gifford & Lucassen 1986; Talpin & Jouvelot 1994). The intention is that this type system may eventually serve as the basis of a recursive module language. In Section 4, we develop the meta-theory of our type system. We state and prove a number of important theorems, leading ultimately to a type soundness result. In Section 5, we explore the expressive power of our destination-passing interpretation of ADTs. In Section 6, we briefly compare the present type system with the one given in the original conference version of this article (Dreyer 2005a) and discuss some of the technical improvements. Finally, in Section 7, we consider related work, and in Section 8, we conclude by discussing future work.

2 The high-level picture

We 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.

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. $\texttt{SIG}_\texttt{A}$, $\texttt{SIG}_\texttt{B}$, and SIG are represented here via the well-known encoding of ML signatures as type

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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. (See Jones (1996) for more examples of this encoding. We merely employ the Jones-style encoding here 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.)

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 **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. 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.[6] Specifically, we allocate an uninitialized ref cell `X` in memory, whose type is $\texttt{rec}(\texttt{SIG}(\alpha)(\beta))$. To use `X` within the body of the recursive definition—that is, in order to get a value of type $\texttt{SIG}(\alpha)(\beta)$ without the "`rec`"—one must first dereference the memory location by writing $\texttt{fetch}(\texttt{X})$. 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 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`. This is accomplished by the command $\texttt{set}\ \alpha := \texttt{int}\ \texttt{in}\ \{\texttt{f = ...}\} : \texttt{SIG}_\texttt{A}(\alpha)(\beta)$. The use of "$:=$" notation is appropriate because at run time we can think of this operation as updating 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` (*i.e.*, $\{\texttt{f = ...}\}$) 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. This behavior is built into the static semantics of the `set` expression. In other words, the typing rule for `set` expressions

[6] In principle, we believe it should be straightforward to incorporate static detection of ill-founded recursion (Dreyer 2004) into the present calculus, but we have not yet attempted it.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

```
ORDERED = λα:T.{compare : α × α → order}
HEAP    = λα:T.λβ:T.{insert : α × β → β, ...}
HEAPGEN = λα:T.∀β↑T.unit -β↓-> HEAP(α)(β)
MkHeap  : ∀α↓T.ORDERED(α) → HEAPGEN(α)

new α↑T, β↑T in
set α :≈ (...β...) in
  letrec X : {Boot:ORDERED(α), Heap:HEAP(α)(β)} =
     {Boot = {compare = ...},
      Heap = MkHeap[α](Boot)[β]()}
  in ...
```

Fig. 4. New encoding of example from Figure 2.

is designed so that all the rest of the program gets to know about the definition of `A` is the following: (1) it defines α in some abstract way, thus rendering α no longer writable, and (2) it has type $\mathtt{SIG_A}(\alpha)(\beta)$. Note that the explicit type annotation $\mathtt{SIG_A}(\alpha)(\beta)$ is critically important here, for it is this annotation that ensures that `A.f` is exported at the type $\alpha \to \beta \times \alpha$ and not, say, at the type $\mathtt{int} \to \beta \times \mathtt{int}$.

It is useful to compare the `set` construct with the well-known `pack` introduction form for existential types, and also with the related "opaque-sealing" construct in the ML module system. All of these mechanisms make the implementation of an abstract type visible only within the scope of a particular term (or module), and all of them require explicit type (or signature) annotations. The key difference between the `set` expression and the traditional mechanisms for abstraction is that the `set` expression is used to define an abstract type that has already been created.

Finally, there is the definition of `B`, which is analogous 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

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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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 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, `HEAPGEN`(α), 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 takes two arguments, a type and a value of type `unit`, and the first argument is required to be a type *variable* that has not yet been defined (*i.e.*, a writable variable—hence, the notation $\forall \beta \uparrow \mathbf{T}$). Although the value argument is clearly superfluous in this instance, we see a use for it in the encoding of separate compilation presented in the following section. When the function is applied, it will not only return a value of type `HEAP`(α)(β) 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 `HEAPGEN`(α) 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 `HEAPGEN`(α) 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 using $:\approx$ 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, as we discuss in Section 2.3. Finally, note that α is defined at the top, outside the definition of `X`, rather than inside the definition of `Boot`. The reason for this is that we do not want the definition of α to be visible only within the implementation of `Boot`; we want it to be visible in the largest scope possible.

2.2 Destination-passing style and separate compilation

The strange new universal type that we used to define `HEAPGEN`(α) 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 $\forall \beta : \mathbf{T}.\, (\forall \alpha : \mathrm{K}.\, \tau \to \beta) \to \beta$. 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 does not 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) (Wadler 1985). In Section 5.2, we make the DPS encoding of existentials precise.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

```
SeparateA : ∀β:T.∀α↑T.rec(SIG(α)(β)) -α↓-> SIGA(α)(β)
          = Λβ:T.Λα↑T.λX:rec(SIG(α)(β)). ...
SeparateB : ∀α:T.∀β↑T.rec(SIG(α)(β)) -β↓-> SIGB(α)(β)
          = Λα:T.Λβ↑T.λX:rec(SIG(α)(β)). ...

new α↑T, β↑T in
  letrec X : SIG(α)(β) =
    {A = SeparateA [β][α](X), B = SeparateB [α][β](X)}
  in ...
```

Fig. 5. Separate compilation of `A` and `B` from Figure 3.

So, one may wonder, if our DPS universal type is really an existential in disguise, why do not we just write, say, $\exists\alpha\uparrow \mathrm{K}.\,\mathtt{A}$ instead of $\forall\alpha\uparrow \mathrm{K}.\,\mathtt{unit} \xrightarrow{\alpha\downarrow} \tau$? 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" $\mathtt{Separate}_\mathtt{A}$ and $\mathtt{Separate}_\mathtt{B}$, represented as polymorphic functions. $\mathtt{Separate}_\mathtt{A}$ takes β (*i.e.*, `B.u`) as its first argument, α (*i.e.*, `A.t`) as its second argument, and the recursive module variable `X` as its third argument. The type of $\mathtt{Separate}_\mathtt{A}$ employs a DPS universal type to bind α because $\mathtt{Separate}_\mathtt{A}$ wants to take a writable `A.t` and define it. Note, however, that β is bound normally as $\beta:\mathbf{T}$. ($\mathtt{Separate}_\mathtt{B}$ 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.[7] If all we had was a DPS universal of the form $\forall\alpha\uparrow \mathrm{K}.\,\mathtt{unit} \xrightarrow{\alpha\downarrow} \tau$, we would have no way of typing $\mathtt{Separate}_\mathtt{A}$ and $\mathtt{Separate}_\mathtt{B}$.

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

[7] 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 (2004) for more discussion of this issue.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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 $\tau_1 \xrightarrow{\alpha\downarrow} \tau_2$ results in a type that does not make sense. In contrast, our type system avoids this problem by not offering any type constructs whose well-formedness depends on a free type variable being writable. Moreover, the joint DPS construct $\forall \alpha \uparrow \mathrm{K}.\, \tau_1 \xrightarrow{\alpha\downarrow} \tau_2$ is sufficient to encode the kinds of recursive module examples that we are interested in.

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. 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," that is, one that was defined by $\alpha :\approx \mathrm{A}$. (We use A and B here to denote type constructors of arbitrary kind, as opposed to τ, which represents types of kind $\mathbf{T}$.) 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* (Amadio & Cardelli 1993; Crary *et al.* 1999). 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 this we mean that the normal form of A, in which all type synonyms have been expanded out, must not refer to α.

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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

and the definition of β each occur in contexts where the other variable is considered abstract. Consequently, the restriction that A does not depend on α and B does 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 polymorphic terms whose type arguments are required to 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.,* `set` $\beta := \alpha$ `list in` ...). If α were only bound as $\alpha : \mathbf{T}$, then β would have to be defined as a datatype 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 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 5.1).

To afford this flexibility, when typechecking `set` $\alpha := \mathrm{A}$ `in` $e : \tau$, we allow A to depend on some set of writable variables σ (not including α), so long as the variables in σ are all backpatched with stable definitions by the time α's definition is hidden (*i.e.*, by the time e has finished evaluating). While this requirement is not strictly necessary, it has the benefit that all the code that is evaluated in the aftermath of α's defining expression may depend on α without any restrictions.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

$$
\begin{array}{lrcl}
\text{Type Variables} & \alpha, \beta & \in & \mathit{TypVars} \\
\text{Type Variable Sets} & \sigma & \in & \mathscr{P}_{\text{fin}}(\mathit{TypVars}) \\
\text{Kinds} & \mathrm{K}, \mathrm{L} & ::= & \mathbf{T} \mid 1 \mid \mathrm{K}_1 \times \mathrm{K}_2 \mid \mathrm{K}_1 \to \mathrm{K}_2 \\
\text{Constructors} & \mathrm{A}, \mathrm{B} & ::= & \alpha \mid b \mid \langle\rangle \mid \langle \mathrm{A}_1, \mathrm{A}_2 \rangle \mid \pi_i \mathrm{A} \mid \lambda \alpha : \mathrm{K}.\, \mathrm{A} \mid \mathrm{A}_1(\mathrm{A}_2) \\
\text{Base Types} & b & ::= & \texttt{unit} \mid \mathrm{A}_1 \times \mathrm{A}_2 \mid \mathrm{A}_1 \to \mathrm{A}_2 \mid \texttt{rec}(\mathrm{A}) \mid \\
 & & & \forall \alpha : \mathrm{K}.\, \mathrm{A} \mid \forall \alpha \downarrow \mathrm{K}.\, \mathrm{A} \mid \forall \alpha \uparrow \mathrm{K}.\, \mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2 \\
\text{Eliminations} & \mathscr{E} & ::= & \bullet \mid \pi_i \mathscr{E} \mid \mathscr{E}(\mathrm{A}) \\
\text{Type Contexts} & \Delta & ::= & \emptyset \mid \Delta, \alpha : \mathrm{K} \mid \Delta, \alpha \uparrow \mathrm{K} \mid \Delta, \alpha \downarrow \mathrm{K} \mid \\
 & & & \Delta, \alpha : \mathrm{K} = \mathrm{A} \mid \Delta, \alpha : \mathrm{K} \approx \mathrm{A} \\
\text{Type Effects} & \varphi & ::= & \alpha := \mathrm{A} \mid \alpha :\approx \mathrm{A} \mid \sigma \downarrow
\end{array}
$$

Fig. 6. Syntax of types.

3 The type system

In this section we present the static and dynamic semantics of a core calculus for recursive type generativity named RTG.

3.1 Type structure

The syntax of RTG's 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 $\mathscr{E}$ are used in the typing rules for `fold`'s and `unfold`'s (see the discussion of Rules 16 and 17 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, assume implicitly that all bound variables are distinct, and take comma (",") to mean disjoint union. We write $\Delta(\alpha)$ to denote the kind to which α is bound in Δ. It is useful to refer to certain subsets of the domain of a context, according to the following definitions:

Definition 3.1 (*common subdomains of a type context*)
Given a type context Δ, the following are subsets of $\mathsf{dom}(\Delta)$:

$$
\begin{array}{rcl}
\mathsf{writable}(\Delta) & \stackrel{\text{def}}{=} & \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta\} \\
\mathsf{unstable}(\Delta) & \stackrel{\text{def}}{=} & \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta \vee \alpha : \mathrm{K} \in \Delta\} \\
\mathsf{abstract}(\Delta) & \stackrel{\text{def}}{=} & \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta \vee \alpha : \mathrm{K} \in \Delta \vee \alpha \downarrow \mathrm{K} \in \Delta \vee \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta\}
\end{array}
$$

Our first task is to define judgments for kinding ($\Delta \vdash \mathrm{A} : \mathrm{K}$) and equivalence ($\Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$). For this purpose, we steal (with only minor extensions) the judgments defined by Stone (2005) in Section 9.1 of Pierce's ATTAPL book. Note that the language we are referring to is *not* the Stone-Harper singleton kind language (2005); it is just F_ω with β-η equivalence, extended with support for type definitions in the context. Our new type constructs require only minimal, straightforward extensions to Stone's language. The kinding and equivalence rules are shown in Figure 7.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Well-formed type constructors: $\Delta \vdash A : K$

$$\frac{\Delta(\alpha) = K}{\Delta \vdash \alpha : K} \qquad \frac{}{\Delta \vdash \mathtt{unit} : \mathbf{T}} \qquad \frac{\Delta \vdash A_1 : \mathbf{T} \quad \Delta \vdash A_2 : \mathbf{T}}{\Delta \vdash A_1 \times A_2 : \mathbf{T}} \qquad \frac{\Delta \vdash A_1 : \mathbf{T} \quad \Delta \vdash A_2 : \mathbf{T}}{\Delta \vdash A_1 \to A_2 : \mathbf{T}}$$

$$\frac{\Delta \vdash A : \mathbf{T}}{\Delta \vdash \mathtt{rec}(A) : \mathbf{T}} \qquad \frac{\Delta, \alpha : K \vdash A : \mathbf{T}}{\Delta \vdash \forall \alpha : K.\, A : \mathbf{T}} \qquad \frac{\Delta, \alpha : K \vdash A : \mathbf{T}}{\Delta \vdash \forall \alpha \downarrow K.\, A : \mathbf{T}}$$

$$\frac{\Delta, \alpha : K \vdash A_1 : \mathbf{T} \quad \Delta, \alpha : K \vdash A_2 : \mathbf{T}}{\Delta \vdash \forall \alpha \uparrow K.\, A_1 \xrightarrow{\alpha\downarrow} A_2 : \mathbf{T}}$$

$$\frac{}{\Delta \vdash \langle\rangle : 1} \qquad \frac{\Delta \vdash A_1 : K_1 \quad \Delta \vdash A_2 : K_2}{\Delta \vdash \langle A_1, A_2 \rangle : K_1 \times K_2} \qquad \frac{\Delta \vdash A : K_1 \times K_2}{\Delta \vdash \pi_i A : K_i}$$

$$\frac{\Delta, \alpha : K_1 \vdash A : K_2}{\Delta \vdash \lambda \alpha : K_1.\, A : K_1 \to K_2} \qquad \frac{\Delta \vdash A_1 : K_2 \to K \quad \Delta \vdash A_2 : K_2}{\Delta \vdash A_1(A_2) : K}$$

Type constructor equivalence: $\Delta \vdash A \equiv B : K$

$$\frac{\Delta \vdash A : K}{\Delta \vdash A \equiv A : K} \qquad \frac{\Delta \vdash B \equiv A : K}{\Delta \vdash A \equiv B : K} \qquad \frac{\Delta \vdash A_1 \equiv B : K \quad \Delta \vdash B \equiv A_2 : K}{\Delta \vdash A_1 \equiv A_2 : K}$$

$$\frac{\Delta \vdash A_1 \equiv B_1 : \mathbf{T} \quad \Delta \vdash A_2 \equiv B_2 : \mathbf{T}}{\Delta \vdash A_1 \times A_2 \equiv B_1 \times B_2 : \mathbf{T}} \qquad \frac{\Delta \vdash A_1 \equiv B_1 : \mathbf{T} \quad \Delta \vdash A_2 \equiv B_2 : \mathbf{T}}{\Delta \vdash A_1 \to A_2 \equiv B_1 \to B_2 : \mathbf{T}}$$

$$\frac{\Delta, \alpha : K \vdash A \equiv B : \mathbf{T}}{\Delta \vdash \forall \alpha : K.\, A \equiv \forall \alpha : K.\, B : \mathbf{T}} \qquad \frac{\Delta, \alpha : K \vdash A \equiv B : \mathbf{T}}{\Delta \vdash \forall \alpha \downarrow K.\, A \equiv \forall \alpha \downarrow K.\, B : \mathbf{T}}$$

$$\frac{\Delta \vdash A \equiv B : \mathbf{T}}{\Delta \vdash \mathtt{rec}(A) \equiv \mathtt{rec}(B) : \mathbf{T}} \qquad \frac{\Delta, \alpha : K \vdash A_1 \equiv B_1 : \mathbf{T} \quad \Delta, \alpha : K \vdash A_2 \equiv B_2 : \mathbf{T}}{\Delta \vdash \forall \alpha \uparrow K.\, A_1 \xrightarrow{\alpha\downarrow} A_2 \equiv \forall \alpha \uparrow K.\, B_1 \xrightarrow{\alpha\downarrow} B_2 : \mathbf{T}}$$

$$\frac{\Delta \vdash A_1 \equiv B_1 : K_1 \quad \Delta \vdash A_2 \equiv B_2 : K_2}{\Delta \vdash \langle A_1, A_2 \rangle \equiv \langle B_1, B_2 \rangle : K_1 \times K_2} \qquad \frac{\Delta \vdash A \equiv B : K_1 \times K_2}{\Delta \vdash \pi_i A \equiv \pi_i B : K_i}$$

$$\frac{\Delta, \alpha : K_1 \vdash A \equiv B : K_2}{\Delta \vdash \lambda \alpha : K_1.\, A \equiv \lambda \alpha : K_1.\, B : K_1 \to K_2} \qquad \frac{\Delta \vdash A_1 \equiv B_1 : K_2 \to K \quad \Delta \vdash A_2 \equiv B_2 : K_2}{\Delta \vdash A_1(A_2) \equiv B_1(B_2) : K}$$

$$\frac{\alpha : K = A \in \Delta}{\Delta \vdash \alpha \equiv A : K} \qquad \frac{\Delta \vdash A_1 : K_1 \quad \Delta \vdash A_2 : K_2}{\Delta \vdash \pi_i \langle A_1, A_2 \rangle \equiv A_i : K_i}$$

$$\frac{\Delta, \alpha : K_1 \vdash A_2 : K_2 \quad \Delta \vdash A_1 : K_1}{\Delta \vdash (\lambda \alpha : K_1.\, A_2)(A_1) \equiv \{\alpha \mapsto A_1\} A_2 : K_2}$$

$$\frac{\Delta \vdash A : 1 \quad \Delta \vdash B : 1}{\Delta \vdash A \equiv B : 1} \qquad \frac{\Delta \vdash \pi_1 A \equiv \pi_1 B : K_1 \quad \Delta \vdash \pi_2 A \equiv \pi_2 B : K_2}{\Delta \vdash A \equiv B : K_1 \times K_2}$$

$$\frac{\Delta, \alpha : K_1 \vdash A(\alpha) \equiv B(\alpha) : K_2 \quad \alpha \notin \mathrm{FV}(A) \cup \mathrm{FV}(B)}{\Delta \vdash A \equiv B : K_1 \to K_2}$$

Fig. 7. Kinding and equivalence rules for type constructors.

The only thing that the kinding judgment needs to know from the context Δ is what the kinds of its bound variables are. The equivalence judgment additionally cares whether a variable is bound as abstract or transparent. Neither judgment, though, cares whether a variable is stable or writable etc., since these notions are only relevant to the term language. Thus, when we make reference to the kinding and equivalence judgments, we make use of the following context erasure:

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Definition 3.2 (*erasure of a context*)
Given a type context Δ, let $\overline{\Delta}$ be its *erasure*, defined as follows:

$$\overline{\Delta} \quad \stackrel{\text{def}}{=} \quad \{\alpha : \Delta(\alpha) \mid \alpha \in \mathsf{abstract}(\Delta)\} \ \cup \ \{\alpha : \mathrm{K} = \mathrm{A} \mid \alpha : \mathrm{K} = \mathrm{A} \in \Delta\}$$

To avoid irritating proliferation of erasure notation, we take $\Delta \vdash \mathrm{A} : \mathrm{K}$ and $\Delta \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$ to be shorthand for $\overline{\Delta} \vdash \mathrm{A} : \mathrm{K}$ and $\overline{\Delta} \vdash \mathrm{A}_1 \equiv \mathrm{A}_2 : \mathrm{K}$, respectively.

Now that we have settled on a kinding judgment, we can define what it means for an RTG type context to be well-formed. The interesting part of this definition is the restriction that a type context may only contain cyclic dependencies if the cycle is broken by a datatype binding. To be precise:

Definition 3.3 (*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 $\forall i \in 1..n$, 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 3.4 (*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. $(\alpha : \mathrm{K} = \mathrm{A} \in \Delta \ \vee \ \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta) \Rightarrow \Delta \vdash \mathrm{A} : \mathrm{K}$

A key concept in our type system is the idea of *stability*. To define what it means for a type constructor to be stable, 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 set is determined by first computing the η-long, β-normal form of A, written $\mathsf{norm}_\Delta(\mathrm{A})$, and then inspecting its free variables. (For details on how to compute $\mathsf{norm}_\Delta(\mathrm{A})$, see Section 4.1.) Stable types are precisely those types whose bases are empty. Formally:

Definition 3.5 (*basis of a type constructor*)
Given a well-formed type context Δ and a type constructor A that is well-formed in Δ, let $\mathsf{basis}_\Delta(\mathrm{A})$ be defined as follows:

$$\mathsf{basis}_\Delta(\mathrm{A}) \quad \stackrel{\text{def}}{=} \quad \mathrm{FV}(\mathsf{norm}_\Delta(\mathrm{A})) \cap \mathsf{unstable}(\Delta)$$

Definition 3.6 (*stable type constructor*)
We say that a type constructor A (with kind K in well-formed type 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 unusual element of our type structure is the notion of *type effects* φ. A type effect is something that changes the state of knowledge about one or more type variables in the context. The effect $\alpha := \mathrm{A}$ changes α from being writable to being a type synonym for A. The effect $\alpha :\approx \mathrm{A}$ changes α from being writable to being a datatype whose underlying definition is A. The effect $\sigma \downarrow$ changes all the variables in σ from writable to stable, but does not reveal how the variables have been defined. We now formalize this description:

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Well-formed type effects: $\Delta \vdash \varphi\ \mathsf{ok}$

$$\frac{\alpha\uparrow \mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K} \quad \mathsf{basis}_\Delta(\mathrm{A}) \subseteq \mathsf{writable}(\Delta) \setminus \{\alpha\}}{\Delta \vdash \alpha := \mathrm{A}\ \mathsf{ok}} \quad (1)$$

$$\frac{\alpha\uparrow \mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta \vdash \alpha :\approx \mathrm{A}\ \mathsf{ok}} \quad (2) \qquad \frac{\sigma \subseteq \mathsf{writable}(\Delta)}{\Delta \vdash \sigma\downarrow\ \mathsf{ok}} \quad (3)$$

Fig. 8. Well-formedness of type effects.

Definition 3.7 (*application of a type effect*)
Let the *application of type effect* φ *to type context* Δ, written $\Delta \mathbin{@} \varphi$, be defined as follows:

$$\begin{aligned}
\Delta \mathbin{@} \alpha := \mathrm{A} &\stackrel{\text{def}}{=} \Delta \setminus \{\alpha\uparrow \Delta(\alpha)\} \cup \{\alpha : \Delta(\alpha) = \mathsf{norm}_\Delta(\mathrm{A})\} \\
\Delta \mathbin{@} \alpha :\approx \mathrm{A} &\stackrel{\text{def}}{=} \Delta \setminus \{\alpha\uparrow \Delta(\alpha)\} \cup \{\alpha : \Delta(\alpha) \approx \mathrm{A}\} \\
\Delta \mathbin{@} \sigma\downarrow &\stackrel{\text{def}}{=} \Delta \setminus \{\alpha\uparrow \Delta(\alpha) \mid \alpha \in \sigma\} \cup \{\alpha\downarrow \Delta(\alpha) \mid \alpha \in \sigma\}
\end{aligned}$$

The reason that $\Delta \mathbin{@} \alpha := \mathrm{A}$ defines α as $\mathsf{norm}_\Delta(\mathrm{A})$ instead of A is to ensure that the resulting context is acyclic. To take a silly example, suppose one tried to set $\alpha := \mathrm{A}$, where $\mathrm{A} = (\lambda\beta : \mathbf{T}.\, \mathtt{int})(\alpha)$. This definition is semantically valid because A is stable (its normal form is `int`). Nevertheless, the binding $\alpha : \mathbf{T} = \mathrm{A}$ is syntactically cyclic. Normalizing A eliminates the potential for a purely syntactic cycle of this sort.

Figure 8 defines a judgment of well-formedness for type effects ($\Delta \vdash \varphi\ \mathsf{ok}$). Rules 2 and 3 are self-explanatory. In Rule 1, the third premise checks that the transparent definition $\alpha := \mathrm{A}$ is semantically acyclic, as well as that A does not depend on any variables β bound as $\beta : \mathrm{K}$. This latter condition is in place to ensure that type substitution holds. Suppose, for instance, that $\mathrm{A} = \beta$, where β is bound as $\beta : \mathrm{K}$. There is nothing preventing β from being substituted with α, and in that case the effect would become $\alpha := \alpha$, which is clearly ill-formed.

Note that the same thing cannot happen if β is bound as $\beta\uparrow \mathrm{K}$ because substitutions are not permitted to alias writable variables. (See Section 4.3 for a full definition of well-formed type substitution.) Moreover, 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 RTG's term structure is shown in Figure 9. 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. It is semantically equivalent to $\{\alpha \mapsto \mathrm{A}\}e$—that is, e with A substituted for free occurrences of α. Also, instead of a `letrec`, we employ a self-contained $\mathtt{rec}_\mathrm{A}(x.\, e)$ 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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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

Fig. 9. Syntax of terms.

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 10 defines the typing rules for terms. Our typing judgment has the form $\Delta; \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \sigma\downarrow$, and is read: "Under type context Δ and value context Γ, the term e has type A and type effect $\sigma\downarrow$." We leave off the "$\mathsf{with}\ \sigma\downarrow$" if $\sigma = \emptyset$.

Rules 4 through 11 are completely standard. Note that function bodies are not permitted to have type effects, that is, to define externally bound type variables. If they were, we would need to support effect types such as $\mathrm{A}_1 \xrightarrow{\alpha\downarrow} \mathrm{A}_2$, which we argued in Section 2.2 is a problematic feature.

Rules 12 and 13 for stable universals are completely analogous to the normal universal rules (10 and 11).

Rules 14 and 15 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 5.1 for details.

Rules 16 and 17 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 $\mathscr{E}\{\alpha\}$ rooted at a datatype variable α, whose underlying definition is B. These coercions witness the isomorphism between $\mathscr{E}\{\alpha\}$ and $\mathscr{E}\{\mathrm{B}\}$. (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 (Vanderwaart *et al.* 2003).)

Note that we have not included any type abstraction mechanisms corresponding to the context bindings $\alpha : \mathrm{K} = \mathrm{A}$ and $\alpha : \mathrm{K} \approx \mathrm{A}$. Intuitively, one might expect such mechanisms to be useful in interpreting ML functors whose arguments contain transparent type or datatype components. Yet, while there is nothing wrong in supporting such mechanisms, we do not believe they are necessary in practice. In particular, a transparent type abstraction construct, $\Lambda\alpha : \mathrm{K} = \mathrm{A}.\, e$, is obviated by the ability to `let`-bind α, that is, $\mathtt{let}\ \alpha = \mathrm{A}\ \mathtt{in}\ e$. A datatype abstraction construct,

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Well-formed terms: $\Delta; \Gamma \vdash e : \mathrm{A}$ with $\sigma\downarrow$

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

$$\frac{x : \mathrm{A} \in \Gamma}{\Delta; \Gamma \vdash x : \mathrm{A}} \ (4) \qquad \frac{}{\Delta; \Gamma \vdash () : \mathtt{unit}} \ (5)$$

$$\frac{\Delta; \Gamma \vdash v_1 : \mathrm{A}_1 \quad \Delta; \Gamma \vdash v_2 : \mathrm{A}_2}{\Delta; \Gamma \vdash (v_1, v_2) : \mathrm{A}_1 \times \mathrm{A}_2} \ (6) \qquad \frac{\Delta; \Gamma \vdash v : \mathrm{A}_1 \times \mathrm{A}_2 \quad i \in \{1,2\}}{\Delta; \Gamma \vdash \pi_i v : \mathrm{A}_i} \ (7)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta; \Gamma, x : \mathrm{A} \vdash e : \mathrm{B}}{\Delta; \Gamma \vdash \lambda x : \mathrm{A}.\, e : \mathrm{A} \to \mathrm{B}} \ (8) \qquad \frac{\Delta; \Gamma \vdash v_1 : \mathrm{A} \to \mathrm{B} \quad \Delta; \Gamma \vdash v_2 : \mathrm{A}}{\Delta; \Gamma \vdash v_1(v_2) : \mathrm{B}} \ (9)$$

$$\frac{\Delta, \alpha : \mathrm{K}; \Gamma \vdash e : \mathrm{A}}{\Delta; \Gamma \vdash \Lambda\alpha : \mathrm{K}.\, e : \forall\alpha : \mathrm{K}.\, \mathrm{A}} \ (10) \qquad \frac{\Delta; \Gamma \vdash v : \forall\alpha : \mathrm{K}.\, \mathrm{B} \quad \Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta; \Gamma \vdash v[\mathrm{A}] : \{\alpha \mapsto \mathrm{A}\}\mathrm{B}} \ (11)$$

$$\frac{\Delta, \alpha\downarrow \mathrm{K}; \Gamma \vdash e : \mathrm{A}}{\Delta; \Gamma \vdash \Lambda\alpha\downarrow \mathrm{K}.\, e : \forall\alpha\downarrow \mathrm{K}.\, \mathrm{A}} \ (12) \qquad \frac{\Delta; \Gamma \vdash v : \forall\alpha\downarrow \mathrm{K}.\, \mathrm{B} \quad \Delta \vdash \mathrm{A}\downarrow \mathrm{K}}{\Delta; \Gamma \vdash v[\mathrm{A}] : \{\alpha \mapsto \mathrm{A}\}\mathrm{B}} \ (13)$$

$$\frac{\Delta, \alpha : \mathrm{K} \vdash \mathrm{A} : \mathbf{T} \quad \Delta, \alpha\uparrow \mathrm{K}; \Gamma, x : \mathrm{A} \vdash e : \mathrm{B} \text{ with } \alpha\downarrow}{\Delta; \Gamma \vdash \Lambda\alpha\uparrow \mathrm{K}.\, \lambda x : \mathrm{A}.\, e : \forall\alpha\uparrow \mathrm{K}.\, \mathrm{A} \xrightarrow{\alpha\downarrow} \mathrm{B}} \ (14)$$

$$\frac{\Delta; \Gamma \vdash v_1 : \forall\alpha\uparrow \mathrm{K}.\, \mathrm{A} \xrightarrow{\alpha\downarrow} \mathrm{B} \quad \Delta; \Gamma \vdash v_2 : \{\alpha \mapsto \beta\}\mathrm{A} \quad \beta\uparrow \mathrm{K} \in \Delta}{\Delta; \Gamma \vdash v_1[\beta](v_2) : \{\alpha \mapsto \beta\}\mathrm{B} \text{ with } \beta\downarrow} \ (15)$$

$$\frac{\Delta \vdash \mathrm{A} \equiv \mathscr{E}\{\alpha\} : \mathbf{T} \quad \alpha : \mathrm{K} \approx \mathrm{B} \in \Delta}{\Delta; \Gamma \vdash \mathtt{fold}_{\mathrm{A}} : \mathscr{E}\{\mathrm{B}\} \to \mathrm{A}} \ (16) \qquad \frac{\Delta \vdash \mathrm{A} \equiv \mathscr{E}\{\alpha\} : \mathbf{T} \quad \alpha : \mathrm{K} \approx \mathrm{B} \in \Delta}{\Delta; \Gamma \vdash \mathtt{unfold}_{\mathrm{A}} : \mathrm{A} \to \mathscr{E}\{\mathrm{B}\}} \ (17)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta; \Gamma, x : \mathtt{rec}(\mathrm{A}) \vdash e : \mathrm{A} \text{ with } \sigma\downarrow}{\Delta; \Gamma \vdash \mathtt{rec}_{\mathrm{A}}(x.e) : \mathrm{A} \text{ with } \sigma\downarrow} \ (18) \qquad \frac{\Delta; \Gamma \vdash v : \mathtt{rec}(\mathrm{A})}{\Delta; \Gamma \vdash \mathtt{fetch}(v) : \mathrm{A}} \ (19)$$

$$\frac{\Delta \vdash \mathrm{A} : \mathrm{K} \quad \Delta, \alpha : \mathrm{K} = \mathrm{A}; \Gamma \vdash e : \mathrm{B} \text{ with } \sigma\downarrow}{\Delta; \Gamma \vdash \mathtt{let}\ \alpha = \mathrm{A}\ \mathtt{in}\ e : \{\alpha \mapsto \mathrm{A}\}\mathrm{B} \text{ with } \sigma\downarrow} \ (20)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathrm{A}_1 \text{ with } \sigma_1\downarrow \quad \Delta @ \sigma_1\downarrow; \Gamma, x : \mathrm{A}_1 \vdash e_2 : \mathrm{A}_2 \text{ with } \sigma_2\downarrow}{\Delta; \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mathrm{A}_2 \text{ with } \sigma_1, \sigma_2\downarrow} \ (21)$$

$$\frac{\Delta, \alpha\uparrow \mathrm{K}; \Gamma \vdash e : \mathrm{A} \text{ with } \alpha, \sigma\downarrow \quad \alpha \notin \mathrm{FV}(\mathrm{A})}{\Delta; \Gamma \vdash \mathtt{new}\ \alpha\uparrow \mathrm{K}\ \mathtt{in}\ e : \mathrm{A} \text{ with } \sigma\downarrow} \ (22)$$

$$\frac{\Delta \vdash \alpha := \mathrm{A}\ \mathsf{ok} \quad \Delta @ \alpha := \mathrm{A}; \Gamma \vdash e : \mathrm{B} \text{ with } \sigma\downarrow \quad \mathsf{basis}_{\Delta}(\mathrm{A}) \subseteq \sigma}{\Delta; \Gamma \vdash (\mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ e : \mathrm{B}) : \mathrm{B} \text{ with } \alpha, \sigma\downarrow} \ (23)$$

$$\frac{\Delta \vdash \alpha :\approx \mathrm{A}\ \mathsf{ok} \quad \Delta @ \alpha :\approx \mathrm{A}; \Gamma \vdash e : \mathrm{B} \text{ with } \sigma\downarrow}{\Delta; \Gamma \vdash (\mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ e : \mathrm{B}) : \mathrm{B} \text{ with } \alpha, \sigma\downarrow} \ (24)$$

$$\frac{\Delta; \Gamma \vdash e : \mathrm{B} \text{ with } \sigma\downarrow \quad \Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathbf{T}}{\Delta; \Gamma \vdash e : \mathrm{A} \text{ with } \sigma\downarrow} \ (25)$$

Fig. 10. Typing rules.

$\Lambda\alpha : \mathrm{K} \approx \mathrm{A}.\, e$, cannot be mimicked as directly in general. However, the kind K of an ML datatype α always has the form $\mathbf{T}$ or $\mathbf{T}^n \to \mathbf{T}$—that is, it represents a single (possibly polymorphic) type component. To parameterize a term over $\alpha : \mathrm{K} \approx \mathrm{A}$ for this special case of K, we follow the interpretation of ML datatypes as ADTs formalized by Harper and Stone (2000): If $\mathrm{K} = \mathbf{T}$, we abstract over a stable

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

$\alpha \downarrow \mathbf{T}$, together with two functions of type $\mathrm{A} \to \alpha$ and $\alpha \to \mathrm{A}$, representing $\texttt{fold}_\alpha$ and $\texttt{unfold}_\alpha$, respectively. If $\mathrm{K} = \mathbf{T}^n \to \mathbf{T}$, we abstract over a stable $\alpha \downarrow \mathrm{K}$, together with two *polymorphic* functions of type $\forall \beta : \mathbf{T}^n.\, \mathrm{A}(\beta) \to \alpha(\beta)$ and $\forall \beta : \mathbf{T}^n.\, \alpha(\beta) \to \mathrm{A}(\beta)$, representing $\Lambda \beta : \mathbf{T}^n.\, \texttt{fold}_{\alpha(\beta)}$ and $\Lambda \beta : \mathbf{T}^n.\, \texttt{unfold}_{\alpha(\beta)}$, respectively. These additional term arguments are enough to support all possible ways that we could fold/unfold α if we had access to the binding $\alpha : \mathrm{K} \approx \mathrm{A}$ directly.

Rules 18 and 19 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 `rec`(A) are variables. In the operational semantics (Section 3.3), we use variables to model backpatchable memory locations.

Rule 20 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. Note that there is no need to restrict A to be stable because α's definition as A is never hidden.

Rule 21 for `let` $x = e_1$ `in` e_2 is slightly interesting in that the type effect $\sigma_1 \downarrow$ engendered by e_1 must be applied to the type context Δ before typechecking e_2.

Rule 22 for `new` $\alpha \uparrow \mathrm{K}$ `in` e introduces α into scope as a writable variable during the typechecking of e and requires e to define it. In addition, α is not permitted to escape its scope by appearing in the free variables of the result type A.

Rule 23 formalizes the semantics for transparent type backpatching that we described at the end of Section 2.3. In particular, the first premise, $\Delta \vdash \alpha := \mathrm{A}$ `ok`, allows α to depend on any writable type variables besides α. It does *not* check that A is immediately stable. Once the body of the `set` has been typechecked, we know what variables will have become stable by the time α is hidden, namely, the ones in σ. The rule then checks that the only variables that A depends on are contained in σ, and it adds α to the list of variables that are defined by the `set` expression.

Rule 24 for datatype definitions is similar to the previous rule, but simpler because datatype definitions are always considered stable. The type annotation B is not actually necessary in this case, since a datatype is an abstract type even when its definition is known, but we include the annotation for symmetry.

Finally, Rule 25 is a standard type conversion rule.

3.3 Operational semantics

We define the operational semantics of our language in Figure 11 using an abstract machine semantics. A machine state Ω is either of the form BlackHole or $(\Delta; \omega; \mathscr{C}; e)$. 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, $\mathscr{C}$ 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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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

Reductions: $e \rightsquigarrow e'$

$$\frac{}{\pi_i(v_1, v_2) \rightsquigarrow v_i}\ (26) \qquad \frac{}{(\lambda x : \mathrm{A}.\, e)(v) \rightsquigarrow \{x \mapsto v\}e}\ (27)$$

$$\frac{}{(\Lambda\alpha : \mathrm{K}.\, e)[\mathrm{A}] \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (28) \qquad \frac{}{(\Lambda\alpha \downarrow \mathrm{K}.\, e)[\mathrm{A}] \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (29)$$

$$\frac{}{(\Lambda\alpha \uparrow \mathrm{K}.\, \lambda x : \mathrm{A}.\, e)[\beta](v) \rightsquigarrow \{\alpha \mapsto \beta\}\{x \mapsto v\}e}\ (30) \qquad \frac{}{\mathtt{unfold}_{\mathrm{A}}(\mathtt{fold}_{\mathrm{B}}(v)) \rightsquigarrow v}\ (31)$$

$$\frac{}{\mathtt{let}\ \alpha = \mathrm{A}\ \mathtt{in}\ e \rightsquigarrow \{\alpha \mapsto \mathrm{A}\}e}\ (32)$$

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

$$\frac{e \rightsquigarrow e'}{(\Delta; \omega; \mathscr{C}; e) \rightsquigarrow (\Delta; \omega; \mathscr{C}; e')}\ (33)$$

$$\frac{}{(\Delta; \omega; \mathscr{C}; \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\Delta; \omega; \mathscr{C} \circ \mathtt{let}\ x = \bullet\ \mathtt{in}\ e_2; e_1)}\ (34)$$

$$\frac{}{(\Delta; \omega; \mathscr{C} \circ \mathtt{let}\ x = \bullet\ \mathtt{in}\ e; v) \rightsquigarrow (\Delta; \omega; \mathscr{C}; \{x \mapsto v\}e)}\ (35)$$

$$\frac{x \notin \mathsf{dom}(\omega)}{(\Delta; \omega; \mathscr{C}; \mathtt{rec}_{\mathrm{A}}(x.e)) \rightsquigarrow (\Delta; \omega, x \mapsto \mathbf{?}; \mathscr{C} \circ \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet); e)}\ (36)$$

$$\frac{x \in \mathsf{dom}(\omega)}{(\Delta; \omega; \mathscr{C} \circ \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet); v) \rightsquigarrow (\Delta; \omega \mathbin{@} x := v; \mathscr{C}; v)}\ (37)$$

$$\frac{x \in \mathsf{dom}(\omega) \quad \omega(x) = v}{(\Delta; \omega; \mathscr{C}; \mathtt{fetch}(x)) \rightsquigarrow (\Delta; \omega; \mathscr{C}; v)}\ (38) \qquad \frac{x \in \mathsf{dom}(\omega) \quad \omega(x) = \mathbf{?}}{(\Delta; \omega; \mathscr{C}; \mathtt{fetch}(x)) \rightsquigarrow \mathsf{BlackHole}}\ (39)$$

$$\frac{\alpha \notin \mathsf{dom}(\Delta)}{(\Delta; \omega; \mathscr{C}; \mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ e) \rightsquigarrow (\Delta, \alpha \uparrow \mathrm{K}; \omega; \mathscr{C}; e)}\ (40)$$

$$\frac{\alpha \uparrow \mathrm{K} \in \Delta}{(\Delta; \omega; \mathscr{C}; \mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ e : \mathrm{B}) \rightsquigarrow (\Delta \mathbin{@} \alpha := \mathrm{A}; \omega; \mathscr{C}; e)}\ (41)$$

$$\frac{\alpha \uparrow \mathrm{K} \in \Delta}{(\Delta; \omega; \mathscr{C}; \mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ e : \mathrm{B}) \rightsquigarrow (\Delta \mathbin{@} \alpha :\approx \mathrm{A}; \omega; \mathscr{C}; e)}\ (42)$$

Fig. 11. Operational semantics.

values (v) or junk (written **?**). 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 $\omega \mathbin{@} x := v$ 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 $y \in \mathsf{dom}(\omega), y \neq x$.

Continuations $\mathscr{C}$ are represented as stacks of continuation frames $\mathscr{F}$. 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 waits for the body of a recursive term to evaluate to a value v and then backpatches the recursive memory location x with v.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

The operational semantics itself is entirely what one would expect given our discussion from Section 2. The `new` construct has the effect of creating a new entry in the type store at run time. The `set` constructs have the effect of updating the type store at run time. 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 operational semantics in such a way that it is easy to prove type soundness, it does not have any real influence on run-time computation. In other words, the operational semantics of Figure 11 never consults the type store in order to determine the identity of a type variable and make a transition on the basis of 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.

4 Meta-theory

In this section, we develop the meta-theory of RTG to the point that we can prove a type soundness theorem. For any type judgment $\mathscr{J}$, we use the notation "$\Delta \Vdash \mathscr{J}$" to signify that $\vdash \Delta\ \mathsf{ok}$ and $\Delta \vdash \mathscr{J}$. For any term judgment $\mathscr{J}$, we use the notation "$\Delta; \Gamma \Vdash \mathscr{J}$" to signify that $\Delta \vdash \Gamma\ \mathsf{ok}$ and $\Delta; \Gamma \vdash \mathscr{J}$.

4.1 Normalization

Figure 12 defines $\mathsf{norm}_\Delta(\mathrm{A})$, the η-long, β-normal form of type constructor A in context Δ. The style of definition follows Stone and Harper (2005). Kind-directed normalization, $\Delta \vdash \mathrm{A} : \mathrm{K} \Longrightarrow \mathrm{B}$, converts A into η-long form and reduces the normalization problem to one for constructors of kind **T**. Weak head normalization, $\Delta \vdash \mathrm{A} \overset{\mathrm{wh}}{\Longrightarrow} \mathrm{A}'$, reduces A to a *path* A', which is either a base type, b, or a sequence of eliminations rooted at an abstract variable, $\mathscr{E}\{\alpha\}$. Finally, structural normalization, $\Delta \vdash \mathrm{A}' \longrightarrow \mathrm{B}$, descends recursively into A', normalizing its subterms.

We begin with a few basic facts about normalization. Theorem 4.1 can be proven using a standard logical relations argument, but the proof is beyond the scope of this article. See Stone (2005) and Stone and Harper (2005) for details.

Theorem 4.1 (*fundamental theorem of normalization*)
Suppose $\vdash \Delta\ \mathsf{ok}$. Then:

1. If $\Delta \vdash \mathrm{A} : \mathrm{K}$, then $\Delta \vdash \mathrm{A} \equiv \mathsf{norm}_\Delta(\mathrm{A}) : \mathrm{K}$.
2. If $\Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathrm{K}$, then $\mathsf{norm}_\Delta(\mathrm{A}) = \mathsf{norm}_\Delta(\mathrm{B})$.

Lemma 4.2 (*useful facts about normalization*)
Suppose $\Delta \Vdash \mathrm{A} : \mathrm{K}$ and $\mathrm{B} = \mathsf{norm}_\Delta(\mathrm{A})$. Then:

1. If $\mathrm{FV}(\mathrm{A}) \subseteq \mathsf{abstract}(\Delta)$, then $\mathrm{FV}(\mathrm{B}) \subseteq \mathrm{FV}(\mathrm{A})$ (and thus, as a corollary, $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \mathrm{FV}(\mathrm{A}) \cap \mathsf{unstable}(\Delta)$).
2. $\mathrm{FV}(\mathrm{B}) \subseteq \mathsf{abstract}(\Delta)$.
3. If $\Delta' \subseteq \Delta$ and $\Delta' \Vdash \mathrm{A} : \mathrm{K}$, then $\mathsf{norm}_{\Delta'}(\mathrm{A}) = \mathrm{B}$ (and thus, as a corollary, $\mathrm{FV}(\mathrm{B}) \subseteq \mathsf{dom}(\Delta')$.)

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Type constructor normalization: $\mathsf{norm}_\Delta(A) \quad \Delta \vdash A \Longrightarrow B$

$\mathsf{norm}_\Delta(A) = B$	if $\Delta \vdash A \Longrightarrow B$
$\Delta \vdash A \Longrightarrow B$	if $\Delta \vdash A : K$ and $\Delta \vdash A : K \Longrightarrow B$

Kind-directed normalization: $\Delta \vdash A : K \Longrightarrow B$

$\Delta \vdash A : 1 \Longrightarrow \langle\rangle$	
$\Delta \vdash A : \mathbf{T} \Longrightarrow B$	if $\Delta \vdash A \overset{\text{wh}}{\Longrightarrow} A'$ and $\Delta \vdash A' \longrightarrow B$
$\Delta \vdash A : K_1 \times K_2 \Longrightarrow \langle B_1, B_2\rangle$	if $\Delta \vdash \pi_1 A \Longrightarrow B_1$ and $\Delta \vdash \pi_2 A \Longrightarrow B_2$
$\Delta \vdash A : K_1 \to K_2 \Longrightarrow \lambda\alpha : K_1.\, B$	if $\Delta, \alpha : K_1 \vdash A(\alpha) \Longrightarrow B$

Weak head normalization: $\Delta \vdash A \overset{\text{wh}}{\Longrightarrow} B$

$\Delta \vdash A \overset{\text{wh}}{\Longrightarrow} B$	if $\Delta \vdash A \overset{\text{wh}}{\longrightarrow} A'$ and $\Delta \vdash A' \overset{\text{wh}}{\Longrightarrow} B$
$\Delta \vdash A \overset{\text{wh}}{\Longrightarrow} A$	if $A = b$, or $A = \mathscr{E}\{\alpha\}$ and $\alpha : K \in \Delta$

Weak head reduction: $\Delta \vdash A \overset{\text{wh}}{\longrightarrow} B$

$\Delta \vdash \mathscr{E}\{\pi_i\langle A_1, A_2\rangle\} \overset{\text{wh}}{\longrightarrow} \mathscr{E}\{A_i\}$	always
$\Delta \vdash \mathscr{E}\{(\lambda\alpha : K.\, A_2)(A_1)\} \overset{\text{wh}}{\longrightarrow} \mathscr{E}\{\{\alpha \mapsto A_1\}A_2\}$	always
$\Delta \vdash \mathscr{E}\{\alpha\} \overset{\text{wh}}{\longrightarrow} \mathscr{E}\{A\}$	if $\alpha : K = A \in \Delta$

Structural normalization: $\Delta \vdash A \longrightarrow B$

$\Delta \vdash \mathtt{unit} \longrightarrow \mathtt{unit}$	always
$\Delta \vdash A_1 \times A_2 \longrightarrow B_1 \times B_2$	if $\Delta \vdash A_1 \Longrightarrow B_1$ and $\Delta \vdash A_2 \Longrightarrow B_2$
$\Delta \vdash A_1 \to A_2 \longrightarrow B_1 \to B_2$	if $\Delta \vdash A_1 \Longrightarrow B_1$ and $\Delta \vdash A_2 \Longrightarrow B_2$
$\Delta \vdash \forall\alpha : K.\, A \longrightarrow \forall\alpha : K.\, B$	if $\Delta, \alpha : K \vdash A \Longrightarrow B$
$\Delta \vdash \forall\alpha \downarrow K.\, A \longrightarrow \forall\alpha \downarrow K.\, B$	if $\Delta, \alpha : K \vdash A \Longrightarrow B$
$\Delta \vdash \forall\alpha \uparrow K.\, A_1 \overset{\alpha\downarrow}{\longrightarrow} A_2 \longrightarrow \forall\alpha \uparrow K.\, B_1 \overset{\alpha\downarrow}{\longrightarrow} B_2$	if $\Delta, \alpha : K \vdash A_1 \Longrightarrow B_1$ and $\Delta, \alpha : K \vdash A_2 \Longrightarrow B_2$
$\Delta \vdash \alpha \longrightarrow \alpha$	always
$\Delta \vdash \pi_i A \longrightarrow \pi_i B$	if $\Delta \vdash A \longrightarrow B$
$\Delta \vdash A_1(A_2) \longrightarrow B_1(B_2)$	if $\Delta \vdash A_1 \longrightarrow B_1$ and $\Delta \vdash A_2 \Longrightarrow B_2$

Fig. 12. Normalization of type constructors.

Proof

By inspection of the normalization algorithm. Specifically:

For Part 1, the only way B can refer to type variables outside of FV(A) is if A refers to type synonym variables that the normalization algorithm expands. This possibility is precluded, however, by the premise that $\mathrm{FV}(A) \subseteq \mathsf{abstract}(\Delta)$. The second conclusion of Part 1 follows directly from the definition of $\mathsf{basis}_\Delta(A)$.

For Part 2, suppose that $\mathrm{FV}(B) \not\subseteq \mathsf{abstract}(\Delta)$. It is straightforward to define a type constructor B' that is equivalent to B but where $\mathrm{FV}(B') \subseteq \mathsf{abstract}(\Delta)$, by simply expanding out all type synonym variables that B refers to. Then, by appealing to Part 1 of the lemma, we see that $\mathrm{FV}(B) \subseteq \mathrm{FV}(B')$, yielding a contradiction.

For Part 3, since $\vdash \Delta'\ \mathsf{ok}$, the bindings in Δ' are self-contained. Since $\mathrm{FV}(A) \subseteq \mathsf{dom}(\Delta')$, there is no way that the normalization algorithm, when given A and Δ,

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

can ever access one of the bindings that is in Δ but not in Δ'. Thus, $B = \mathsf{norm}_{\Delta'}(A)$. The corollary follows from the fact that $\mathsf{norm}_{\Delta'}(A)$ is well-formed in Δ'. □

4.2 Validity

Next, we have a standard validity (aka regularity) property, the proof of which relies on the fact that the "static" judgments of kinding, equivalence, stability, and context well-formedness are unaffected by the application of well-formed type effects to their contexts.

Proposition 4.3 (*context well-formedness preserved under effects*)
If $\Delta \Vdash \varphi\ \mathsf{ok}$, then $\vdash \Delta @ \varphi\ \mathsf{ok}$.

Proof
The cases for $\varphi = \alpha :\approx A$ and $\varphi = \sigma\downarrow$ are straightforward. In the case that $\varphi = \alpha := A$, we have $\mathsf{basis}_\Delta(A) \subseteq \mathsf{writable}(\Delta) \setminus \{\alpha\}$. We therefore know that $\alpha \notin \mathrm{FV}(\mathsf{norm}_\Delta(A))$. By Part 2 of Lemma 4.2, we have that $\mathrm{FV}(\mathsf{norm}_\Delta(A)) \subseteq \mathsf{abstract}(\Delta)$. Thus, replacing the binding $\alpha \uparrow \Delta(\alpha)$ with $\alpha : \Delta(\alpha) = \mathsf{norm}_\Delta(A)$, which is what the definition of $\Delta @ \alpha := A$ does, will not introduce any cycles among the transparent definitions of Δ. □

Proposition 4.4 (*kinding and equivalence preserved under effects*)
Suppose $\Delta \Vdash \varphi\ \mathsf{ok}$. Then:

1. $\Delta \vdash A : K$ if and only if $\Delta @ \varphi \vdash A : K$.
2. If $\Delta \vdash A_1 \equiv A_2 : K$, then $\Delta @ \varphi \vdash A_1 \equiv A_2 : K$.
3. If $\varphi = \sigma\downarrow$ and $\Delta @ \varphi \vdash A_1 \equiv A_2 : K$, then $\Delta \vdash A_1 \equiv A_2 : K$.
4. $\Delta \vdash \Gamma\ \mathsf{ok}$ if and only if $\Delta @ \varphi \vdash \Gamma\ \mathsf{ok}$.

Proof
First of all, recall that both the kinding and equivalence judgments only expect contexts that have undergone erasure, and that the notation $\Delta \vdash A_1 \equiv A_2 : K$ is really shorthand for $\overline{\Delta} \vdash A_1 \equiv A_2 : K$. Furthermore, recall that the kinding judgment cares only about the kinds of the variables in the context. Part 1 is valid because φ does not change the kinds of the variables in Δ. Part 2 is valid because, if anything, $\overline{\Delta @ \varphi}$ is a more informative context than $\overline{\Delta}$. Part 3 is valid because $\overline{\Delta @ \sigma\downarrow} = \overline{\Delta}$. Part 4 follows from Part 1. □

Proposition 4.5 (*stability preserved under effects*)
If $\Delta \vdash A \downarrow K$ and $\Delta \Vdash \varphi\ \mathsf{ok}$, then $\Delta @ \varphi \vdash A \downarrow K$.

Proof
Let $B = \mathsf{norm}_\Delta(A)$ and let $\Delta' = \Delta @ \varphi$. By Proposition 4.4, since $\Delta \vdash A \equiv B : K$, we have $\Delta' \vdash A \equiv B : K$. Thus, $\mathsf{basis}_{\Delta'}(A) = \mathsf{basis}_{\Delta'}(B)$. By Part 2 of Lemma 4.2, together with the fact that A is stable, we have that $\mathrm{FV}(B) \subseteq \mathsf{abstract}(\Delta) \setminus \mathsf{unstable}(\Delta)$. Now, let $B' = \mathsf{norm}_{\Delta'}(B)$. By Part 1 of Lemma 4.2, $\mathrm{FV}(B') \subseteq \mathrm{FV}(B)$. In addition, note (by definition of $\Delta @ \varphi$) that $\mathsf{unstable}(\Delta') \subseteq \mathsf{unstable}(\Delta)$. Consequently, since $\mathsf{basis}_{\Delta'}(B) = \mathrm{FV}(B') \cap \mathsf{unstable}(\Delta')$, we have that $\mathsf{basis}_{\Delta'}(A) \subseteq (\mathsf{abstract}(\Delta) \setminus \mathsf{unstable}(\Delta)) \cap \mathsf{unstable}(\Delta) = \emptyset$. □

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Proposition 4.6 (*validity*)
If $\Delta; \Gamma \Vdash e : \mathrm{A}$ with $\sigma\downarrow$, then $\Delta \vdash \mathrm{A} : \mathbf{T}$ and $\Delta \vdash \sigma\downarrow$ ok.

Proof
By straightforward induction on derivations, with appropriate uses of Propositions 4.3 and 4.4 for Rules 21, 23, and 24. □

4.3 Type substitution

Next, we prove a type substitution property. This is a bit more involved. 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 write $\delta\mathrm{A}$ (resp. δe, $\delta\Gamma$) to signify the result of performing the substitution δ on the free variables of A (resp. e, Γ) in the usual capture-avoiding manner. For sets σ, we take $\delta\sigma$ to mean $\{\delta\alpha \mid \alpha \in \sigma\}$, and for effects $\varphi = \alpha := \mathrm{A}$, $\alpha :\approx \mathrm{A}$, or $\sigma\downarrow$, we take $\delta\varphi$ to mean $\delta\alpha := \delta\mathrm{A}$, $\delta\alpha :\approx \delta\mathrm{A}$, or $\delta\sigma\downarrow$, respectively.

The definition of well-formed type substitution is long (because there are five different kinds of context bindings) but fairly straightforward:

Definition 4.7 (*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. $\vdash \Delta$ ok and $\vdash \Delta'$ ok
3. $\forall \alpha \in \mathsf{dom}(\Delta).\ \Delta' \vdash \delta\alpha : \Delta(\alpha)$
4. $\forall \alpha : \mathrm{K} = \mathrm{A} \in \Delta.\ \Delta' \vdash \delta\alpha \equiv \delta\mathrm{A} : \mathrm{K}$
5. $\forall \alpha : \mathrm{K} \approx \mathrm{A} \in \Delta.\ \exists \alpha' : \mathrm{K} \approx \mathrm{A}' \in \Delta'.\ \Delta' \vdash \alpha' \equiv \delta\alpha : \mathrm{K}$ and $\Delta' \vdash \mathrm{A}' \equiv \delta\mathrm{A} : \mathrm{K}$
6. $\forall \alpha\downarrow \mathrm{K} \in \Delta.\ \Delta' \vdash \delta\alpha\downarrow \mathrm{K}$
7. $\forall \alpha\uparrow \mathrm{K} \in \Delta.\ \exists \alpha'\uparrow \mathrm{K} \in \Delta'.\ \alpha' = \delta\alpha$
8. $\forall \alpha_1\uparrow \mathrm{K}_1 \in \Delta.\ \forall \alpha_2\uparrow \mathrm{K}_2 \in \Delta.\ (\delta\alpha_1 = \delta\alpha_2) \Rightarrow (\alpha_1 = \alpha_2)$

The only conditions that are really unusual are the last two. We require that δ map writable variables to *variables*, not arbitrary type expressions, and furthermore that it not alias any two writable variables that were originally distinct. These conditions are critical, since it is only safe to backpatch a writable variable once.

Let us first restate some substitution properties from Stone (2005):

Proposition 4.8 (*substitution on types and contexts*)
Suppose $\Delta' \vdash \delta : \Delta$. Then:

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

Definition 4.9 (*equivalent type substitutions*)
We say that δ_1 and δ_2 are equivalent substitutions, written $\Delta' \vdash \delta_1 \equiv \delta_2 : \Delta$, if:

1. $\Delta' \vdash \delta_1 : \Delta$ and $\Delta' \vdash \delta_2 : \Delta$
2. $\forall \alpha \in \mathsf{dom}(\Delta).\ \Delta' \vdash \delta_1\alpha \equiv \delta_2\alpha : \Delta(\alpha)$

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Proposition 4.10 (*functionality*)
If $\Delta' \vdash \delta_1 \equiv \delta_2 : \Delta$ and $\Delta \vdash A_1 \equiv A_2 : K$, then $\Delta' \vdash \delta_1 A_1 \equiv \delta_2 A_2 : K$.

The following lemma states a monotonicity property that is useful in proving the subsequent 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 type substitution for the construct "`set` $\alpha := A$ `in` ..." (Rule 23). When we apply a substitution δ to this construct, we want to make sure that δA does not suddenly grow to depend on $\delta\alpha$. While the monotonicity property itself is fairly intuitive, the proof is somewhat fiddly.

Lemma 4.11 (*monotonicity of basis*)
If $\Delta \Vdash A : K$, $\mathsf{basis}_\Delta(A) \subseteq \mathsf{writable}(\Delta)$, and $\Delta' \vdash \delta : \Delta$,
then $\mathsf{basis}_{\Delta'}(\delta A) \subseteq \delta(\mathsf{basis}_\Delta(A))$.

Proof
Let $B = \mathsf{norm}_\Delta(A)$. By Proposition 4.8, since $\Delta \vdash A \equiv B : K$, we also have $\Delta' \vdash \delta A \equiv \delta B : K$. Since equivalent type constructors have the same basis, $\mathsf{basis}_{\Delta'}(\delta A) = \mathsf{basis}_{\Delta'}(\delta B)$ and $\delta(\mathsf{basis}_\Delta(A)) = \delta(\mathsf{basis}_\Delta(B))$. The proof reduces to showing that $\mathsf{basis}_{\Delta'}(\delta B) \subseteq \delta(\mathsf{basis}_\Delta(B))$.

As a technical device, we find it useful now to define a substitution δ_n that is equivalent to δ but whose output is normalized (except for writable variables, where the definition of well-formed substitution requires δ_n to return a *variable*, not the η-long β-normal form of a variable):

$$\delta_n(\alpha) \stackrel{\text{def}}{=} \begin{cases} \delta\alpha & \text{if } \alpha \in \mathsf{writable}(\Delta) \\ \mathsf{norm}_{\Delta'}(\delta\alpha) & \text{otherwise} \end{cases}$$

From Part 1 of Lemma 4.2, for all $\alpha \in \mathsf{dom}(\Delta)$, we have that $\mathrm{FV}(\delta_n\alpha) \subseteq \mathsf{abstract}(\Delta')$. (In the case that α is writable, $\delta_n\alpha$ is not explicitly normalized, but it does not matter: the well-formedness of δ guarantees that $\delta_n\alpha = \delta\alpha$ is also writable, and therefore abstract.)

We can now observe that $\mathrm{FV}(\delta_n B) \subseteq \mathsf{abstract}(\Delta')$ (actually, this is true for any B that is well-formed in Δ, not just $B = \mathsf{norm}_\Delta(A)$). By Part 2 of Lemma 4.2, $\mathsf{basis}_{\Delta'}(\delta_n B) \subseteq \mathrm{FV}(\delta_n B) \cap \mathsf{unstable}(\Delta')$. It is easy to check that $\Delta' \vdash \delta \equiv \delta_n : \Delta$. By functionality, $\Delta' \vdash \delta B \equiv \delta_n B : K$. Consequently, $\mathsf{basis}_{\Delta'}(\delta B) = \mathsf{basis}_{\Delta'}(\delta_n B)$, and so $\mathsf{basis}_{\Delta'}(\delta B) \subseteq \mathrm{FV}(\delta_n B) \cap \mathsf{unstable}(\Delta')$.

Suppose that $\alpha \in \mathsf{basis}_{\Delta'}(\delta B)$. Then, $\alpha \in \mathrm{FV}(\delta_n B) \cap \mathsf{unstable}(\Delta')$. We need to show that $\alpha \in \delta(\mathsf{basis}_\Delta(B))$. Note that, since $\alpha \in \mathrm{FV}(\delta_n B)$, there must exist $\beta \in \mathrm{FV}(B)$ such that $\alpha \in \mathrm{FV}(\delta_n\beta)$. We proceed by cases on the binding of β in Δ:

- Case: $\beta : K \in \Delta$. Since B is in normal form, we know that $\beta \in \mathsf{basis}_\Delta(B)$. However, by assumption, $\mathsf{basis}_\Delta(B) = \mathsf{basis}_\Delta(A) \subseteq \mathsf{writable}(\Delta)$. Thus, we have a contradiction.
- Case: $\beta : K = B' \in \Delta$. By Part 2 of Lemma 4.2, we have a contradiction.
- Case: $\beta SK \in \Delta$. By well-formedness of δ_n, we have $\Delta' \vdash \delta_n\beta \downarrow K$. This means that $\mathsf{basis}_{\Delta'}(\delta_n\beta) = \emptyset$. Since $\delta_n\beta$ is already in normal form, we have

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

that $\mathrm{FV}(\delta_n\beta) \cap \mathsf{unstable}(\Delta') = \emptyset$. However, by assumption, $\alpha \in \mathrm{FV}(\delta_n\beta) \cap \mathsf{unstable}(\Delta')$. Thus, we have a contradiction.

- Case: $\beta : \mathrm{K} \approx \mathrm{B}' \in \Delta$. By well-formedness of δ_n, there exists $\beta'' : \mathrm{K} \approx \mathrm{B}'' \in \Delta'$ such that $\Delta' \vdash \beta'' \equiv \delta_n\beta : \mathrm{K}$. By Part 1 of Lemma 4.2, $\mathsf{basis}_{\Delta'}(\beta'') \subseteq \mathrm{FV}(\beta'') \cap \mathsf{unstable}(\Delta') = \emptyset$. Thus, $\mathsf{basis}_{\Delta'}(\delta_n\beta) = \emptyset$. By the same reasoning as in the previous case, we have a contradiction.
- Case: $\beta \uparrow \mathrm{K} \in \Delta$. By well-formedness of δ, we have that $\delta_n\beta = \delta\beta = \alpha$. Since B is in normal form, we know that $\beta \in \mathsf{basis}_\Delta(\mathrm{B})$. Thus, $\alpha \in \delta(\mathsf{basis}_\Delta(\mathrm{B}))$.

□

Corollary 4.12 (*substitution on stable types*)
If $\Delta \vdash \mathrm{A} \downarrow \mathrm{K}$ and $\Delta' \vdash \delta : \Delta$, then $\Delta' \vdash \delta\mathrm{A} \downarrow \mathrm{K}$.

Proof
By monotonicity, $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \delta(\mathsf{basis}_\Delta(\mathrm{A})) = \emptyset$. □

Proposition 4.13 (*substitution on effects*)
If $\Delta \vdash \varphi\ \mathsf{ok}$ and $\Delta' \vdash \delta : \Delta$, then $\Delta' \vdash \delta\varphi\ \mathsf{ok}$ and $\Delta' \mathbin{@} \delta\varphi \vdash \delta : \Delta \mathbin{@} \varphi$.

Proof
Let us begin with the first conclusion ($\Delta' \vdash \delta\varphi\ \mathsf{ok}$). For $\varphi = \alpha :\approx \mathrm{A}$ and $\varphi = \sigma\downarrow$, the proof is completely straightforward. For $\varphi = \alpha := \mathrm{A}$, we know that $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \mathsf{writable}(\Delta)\backslash\{\alpha\}$, and we need to show that $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \mathsf{writable}(\Delta')\backslash\{\delta\alpha\}$. Thanks to the Monotonicity Lemma, this is easy. Specifically, monotonicity gives us that $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \delta(\mathsf{basis}_\Delta(\mathrm{A}))$. By assumption, the latter is a subset of $\delta(\mathsf{writable}(\Delta) \setminus \{\alpha\})$, which is, in turn, a subset of $\mathsf{writable}(\Delta') \setminus \{\delta\alpha\}$ (by well-formedness of δ).

For the second conclusion, first observe that $\vdash \Delta \mathbin{@} \varphi\ \mathsf{ok}$ and $\vdash \Delta' \mathbin{@} \delta\varphi\ \mathsf{ok}$ (by the first conclusion together with Proposition 4.3). For the bindings in $\Delta \mathbin{@} \varphi$ that are holdovers from Δ, the proof follows easily from Propositions 4.4 and 4.5. For the remaining bindings, we argue the proof by cases on φ:

- Case: $\varphi = \alpha :\approx \mathrm{A}$. Here, $\Delta \mathbin{@} \varphi$ changes $\alpha \uparrow \mathrm{K}$ to $\alpha : \mathrm{K} \approx \mathrm{A}$. There will be a corresponding binding $\delta\alpha : \mathrm{K} \approx \delta\mathrm{A}$ in $\Delta' \mathbin{@} \delta\varphi$, so we are done.
- Case: $\varphi = \sigma\downarrow$. For each variable α in $\sigma\downarrow$, $\Delta \mathbin{@} \varphi$ will bind α as stable, but $\Delta' \mathbin{@} \delta\varphi$ will bind $\delta\alpha$ as stable, too, so we are done.
- Case: $\varphi = \alpha := \mathrm{A}$. As usual, the most interesting case. Here, $\Delta \mathbin{@} \varphi$ changes $\alpha \uparrow \mathrm{K}$ to $\alpha : \mathrm{K} = \mathsf{norm}_\Delta(\mathrm{A})$, but $\Delta' \mathbin{@} \delta\varphi$ changes $\delta\alpha \uparrow \mathrm{K}$ to $\delta\alpha : \mathrm{K} = \mathsf{norm}_{\Delta'}(\delta\mathrm{A})$. By Proposition 4.4, it suffices to show $\Delta' \vdash \delta(\mathsf{norm}_\Delta(\mathrm{A})) \equiv \mathsf{norm}_{\Delta'}(\delta\mathrm{A}) : \mathrm{K}$. Then, since $\Delta \vdash \mathsf{norm}_\Delta(\mathrm{A}) \equiv \mathrm{A} : \mathrm{K}$, we have by Proposition 4.8 that $\Delta' \vdash \delta(\mathsf{norm}_\Delta(\mathrm{A})) \equiv \delta\mathrm{A} : \mathrm{K}$. Finally, since $\Delta' \vdash \delta\mathrm{A} \equiv \mathsf{norm}_{\Delta'}(\delta\mathrm{A}) : \mathrm{K}$, the desired result follows by transitivity.

□

Proposition 4.14 (*type substitution on terms*)
If $\Delta; \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \sigma\downarrow$; and $\Delta' \vdash \delta : \Delta$, then $\Delta'; \delta\Gamma \vdash \delta e : \delta\mathrm{A}\ \mathsf{with}\ \delta\sigma\downarrow$.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Proof
By straightforward induction on derivations, with appropriate uses of the propositions proved above. In particular, note that monotonicity pops up again in the case of Rule 23. The third premise of the rule tells us that $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \sigma$, and from the second premise we know that $\sigma \subseteq \mathsf{writable}(\Delta)$. We need to show that $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \delta\sigma$. By monotonicity, $\mathsf{basis}_{\Delta'}(\delta\mathrm{A}) \subseteq \delta(\mathsf{basis}_\Delta(\mathrm{A})) \subseteq \delta\sigma$. □

4.4 Value substitution and the "use it or lose it" lemma

Now we come to value substitutions. Unlike Definition 4.7, the definition of a well-formed value substitution is very simple:

Definition 4.15 (*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. $\Delta \vdash \Gamma\ \mathsf{ok}$ and $\Delta \vdash \Gamma'\ \mathsf{ok}$
3. $\forall x : \mathrm{A} \in \Gamma.\ \Delta; \Gamma' \vdash \gamma x : \mathrm{A}$

Proving the value substitution property, however, is a bit tricky. In the cases for Rules 21, 23, and 24, the premises of the rules have type contexts of the form $\Delta \mathbin{@} \varphi$, so we need to be able to show that value substitutions that are well-formed in Δ are also well-formed in $\Delta \mathbin{@} \varphi$. This boils down to showing that the typing derivation for a value cannot possibly rely on any variables in the context being writable.

To make the induction go through, we prove instead a more general result, which we call the "use it or lose it" lemma. It says that, if a term e is well-typed in a context where α is writable, then either e must *use* the fact that α is writable (*i.e.*, e defines α) or e can afford to *lose* the fact that α is writable (*i.e.*, e will also be well-typed even if α is not bound as writable).

Lemma 4.16 (*use it or lose it*)
If $\Delta, \alpha \uparrow K; \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \sigma \downarrow$; and $\alpha \notin \sigma$, then $\Delta, \alpha : K; \Gamma \Vdash e : \mathrm{A}\ \mathsf{with}\ \sigma \downarrow$.

Proof
By straightforward induction on derivations. □

Corollary 4.17 (*"pure" term typing preserved under effects*)
If $\Delta; \Gamma \Vdash e : \mathrm{A}$ and $\Delta \vdash \varphi\ \mathsf{ok}$, then $\Delta \mathbin{@} \varphi; \Gamma \Vdash e : \mathrm{A}$.

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 4.16, $\Delta'; \Gamma \Vdash e : \mathrm{A}$. It is easy to see that $\Delta \mathbin{@} \varphi \vdash \mathsf{id} : \Delta'$. Thus, the desired result follows from Proposition 4.14. □

Corollary 4.18 (*value substitution typing preserved under effects*)
If $\Delta; \Gamma' \vdash \gamma : \Gamma$ and $\Delta \vdash \varphi\ \mathsf{ok}$, then $\Delta \mathbin{@} \varphi; \Gamma' \vdash \gamma : \Gamma$.

Proof
By Corollary 4.17, together with Part 4 of Proposition 4.4. □

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Proposition 4.19 (*value substitution on terms*)
If $\Delta; \Gamma \vdash e : \mathrm{A}$ with $\sigma\downarrow$ and $\Delta; \Gamma' \vdash \gamma : \Gamma$, then $\Delta; \Gamma' \vdash \gamma e : \mathrm{A}$ with $\sigma\downarrow$.

Proof
By straightforward induction on derivations. In the cases for Rules 21, 23, and 24, the premises of the rules have type contexts of the form $\Delta @ \varphi$, so we need to obtain $\Delta @ \varphi; \Gamma' \vdash \gamma : \Gamma$ in order to make progress, but this is precisely what we get from Corollary 4.18. □

Corollary 4.17 is similarly useful for guaranteeing that the mutable value store maintained by our dynamic semantics remains well-formed throughout execution (see Corollary 4.24 in Section 4.6).

Another corollary of the "use it or lose it" lemma says that if we have an expression e referring to two writable type variables of the same kind, and e depends only 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 (see the case for Rule 30 in the proof of Theorem 4.28 below).

Corollary 4.20 (*merging together two writable types*)
If $\Delta \vdash \Gamma$ ok and $\beta \uparrow \mathrm{K} \in \Delta$ and $\Delta, \alpha \uparrow \mathrm{K}; \Gamma, x : \mathrm{A} \Vdash e : \mathrm{B}$ with $\alpha\downarrow$,
then $\Delta; \Gamma, x : \{\alpha \mapsto \beta\}\mathrm{A} \Vdash \{\alpha \mapsto \beta\}e : \{\alpha \mapsto \beta\}\mathrm{B}$ with $\beta\downarrow$.

Proof
Let $\Delta = \Delta', \beta \uparrow \mathrm{K}$. By Lemma 4.16, $\Delta', \beta : \mathrm{K}, \alpha \uparrow \mathrm{K}; \Gamma, x : \mathrm{A} \Vdash e : \mathrm{B}$ 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 from Proposition 4.14. □

4.5 Typechecking

As the RTG calculus is explicitly typed, it is completely straightforward to define a syntax-directed type synthesis algorithm, $\Delta; \Gamma \vdash e \Rightarrow \mathrm{A}$ with $\sigma\downarrow$, that takes Δ, Γ, and e as input and returns A and σ as output. The soundness and completeness of this algorithm can be summarized as follows:

Theorem 4.21 (*soundness and completeness of type synthesis*)
Suppose $\Delta \vdash \Gamma$ ok. Then:

1. If $\Delta; \Gamma \vdash e \Rightarrow \mathrm{A}$ with $\sigma\downarrow$, then $\Delta; \Gamma \vdash e : \mathrm{A}$ with $\sigma\downarrow$.
2. If $\Delta; \Gamma \vdash e : \mathrm{A}$ with $\sigma\downarrow$, then there exists B such that $\Delta; \Gamma \vdash e \Rightarrow \mathrm{B}$ with $\sigma\downarrow$ and $\Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathbf{T}$.

We omit the details of this algorithm, as they are entirely standard. Essentially, wherever a premise requires a subterm e to have a type of a particular form, say, for example, $\forall\alpha\downarrow \mathrm{K}.\,\mathrm{A}$, the synthesis algorithm synthesizes the type B of e, and then normalizes B to get it into the shape of $\forall\alpha\downarrow \mathrm{K}.\,\mathrm{A}$. (Full normalization is not actually necessary for this purpose; weak head normalization will suffice.) See Chapter 3 of Dreyer's thesis (2005b) for a fully specified example of such an algorithm.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

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

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T}}{\Delta; \Gamma \vdash \bullet : \mathrm{A}\ \mathsf{cont}}\ (43) \qquad \frac{\Delta; \Gamma \vdash \mathscr{F} : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow \quad \Delta \mathbin{@} \sigma\downarrow; \Gamma \vdash \mathscr{C} : \mathrm{B}\ \mathsf{cont}}{\Delta; \Gamma \vdash \mathscr{C} \circ \mathscr{F} : \mathrm{A}\ \mathsf{cont}}\ (44)$$

$$\frac{\Delta; \Gamma \vdash \mathscr{C} : \mathrm{B}\ \mathsf{cont} \quad \Delta \vdash \mathrm{A} \equiv \mathrm{B} : \mathbf{T}}{\Delta; \Gamma \vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}}\ (45)$$

Well-formed continuation frames: $\Delta; \Gamma \vdash \mathscr{F} : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow$

$$\frac{\Delta \vdash \mathrm{A} : \mathbf{T} \quad \Delta; \Gamma, x : \mathrm{A} \vdash e : \mathrm{B}\ \mathsf{with}\ \sigma\downarrow}{\Delta; \Gamma \vdash \mathtt{let}\ x = \bullet\ \mathtt{in}\ e : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow}\ (46)$$

$$\frac{x : \mathtt{rec}(\mathrm{A}) \in \Gamma}{\Delta; \Gamma \vdash \mathtt{rec}_{\mathrm{A}}(x \leftarrow \bullet) : \mathrm{A} \rightsquigarrow \mathrm{A}\ \mathsf{with}\ \emptyset\downarrow}\ (47)$$

Fig. 13. Well-formedness of continuations.

There is one case worth noting, namely, the `new` construct. When synthesizing the type for $\mathtt{new}\ \alpha\uparrow \mathrm{K}\ \mathtt{in}\ e$, the algorithm synthesizes the type A of e. However, it is possible that the synthesized type A may refer, in a non-essential way, to α. The synthesized type for $\mathtt{new}\ \alpha\uparrow \mathrm{K}\ \mathtt{in}\ e$ is therefore not A, but $\mathsf{norm}_{\Delta'}(\mathrm{A})$ (where $\Delta' = \Delta, \alpha\uparrow \mathrm{K}$). If the `new` is well-formed—that is, if there exists a type B equivalent to A that avoids mention of α—then Part 3 of Lemma 4.2 guarantees that $\mathsf{norm}_{\Delta'}(\mathrm{A})$ will not mention α either.

4.6 Type soundness

We define well-formedness of value stores as follows. The notion of run-time context is useful as a way of describing the contexts that classify stores.

Definition 4.22 (*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 4.23 (*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. $\forall x : \mathtt{rec}(\mathrm{A}) \in \Gamma$. either $\omega(x) = \mathbf{?}$ or $\Delta; \Gamma \vdash \omega(x) : \mathrm{A}$

Corollary 4.24 (*value store typing preserved under effects*)
If $\Delta \vdash \omega : \Gamma$ and $\Delta \vdash \varphi\ \mathsf{ok}$, then $\Delta \mathbin{@} \varphi \vdash \omega : \Gamma$.

Proof
Follows directly from Corollary 4.17. □

The typing judgments for continuations and continuation frames are shown in Figure 13. The latter is slightly interesting in that a frame may have a type effect. One can read the judgment $(\Delta; \Gamma \vdash \mathscr{F} : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow)$ as: "starting in type

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

context Δ, the frame $\mathscr{F}$ takes a value of type A and returns a value of type B while defining the variables in σ." Continuations $\mathscr{C}$ may of course have a type effect as well, but they are irrelevant because we never return from a continuation.

Proposition 4.25 (*validity for continuations*)

1. If $\Delta; \Gamma \Vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}$, then $\Delta \vdash \mathrm{A} : \mathbf{T}$.
2. If $\Delta; \Gamma \Vdash \mathscr{F} : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow$, then $\Delta \vdash \mathrm{A} : \mathbf{T}$, $\Delta \vdash \mathrm{B} : \mathbf{T}$, and $\Delta \vdash \sigma\downarrow\ \mathsf{ok}$.

Proposition 4.26 (*type substitution on continuations*)
Suppose $\Delta' \vdash \delta : \Delta$. Then:

1. If $\Delta; \Gamma \Vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}$, then $\Delta'; \delta\Gamma \Vdash \delta\mathscr{C} : \delta\mathrm{A}\ \mathsf{cont}$.
2. If $\Delta; \Gamma \Vdash \mathscr{F} : \mathrm{A} \rightsquigarrow \mathrm{B}\ \mathsf{with}\ \sigma\downarrow$, then $\Delta'; \delta\Gamma \Vdash \delta\mathscr{F} : \delta\mathrm{A} \rightsquigarrow \delta\mathrm{B}\ \mathsf{with}\ \delta\sigma\downarrow$.

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 effect $\sigma\downarrow$, so these effects must be incorporated into the "starting" context of the continuation $\mathscr{C}$.

Definition 4.27 (*well-formed machine states*)
We say that a machine state Ω is well-formed, written $\vdash \Omega\ \mathsf{ok}$, if either $\Omega = \mathsf{BlackHole}$ or $\Omega = (\Delta; \omega; \mathscr{C}; e)$ and there exist Γ, A, and σ such that:

1. $\Delta \vdash \omega : \Gamma$
2. $\Delta; \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \sigma\downarrow$ and $\Delta @ \sigma\downarrow; \Gamma \vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}$

Theorem 4.28 (*preservation*)
If $\vdash \Omega\ \mathsf{ok}$ and $\Omega \rightsquigarrow \Omega'$, then $\vdash \Omega'\ \mathsf{ok}$.

Proof
Straightforward. We sketch the interesting cases.

- Case: Rule 30. By inversion on typing, we know that $\Delta, \alpha\uparrow \mathrm{K}; \Gamma, x : \mathrm{A} \vdash e : \mathrm{B}\ \mathsf{with}\ \alpha\downarrow$, $\Delta; \Gamma \vdash v : \{\alpha \mapsto \beta\}\mathrm{A}$, and $\beta\uparrow \mathrm{K} \in \Delta$. By Corollary 4.20, $\Delta; \Gamma, x : \{\alpha \mapsto \beta\}\mathrm{A} \vdash \{\alpha \mapsto \beta\}e : \{\alpha \mapsto \beta\}\mathrm{B}\ \mathsf{with}\ \beta\downarrow$. By value substitution, $\Delta; \Gamma \vdash \{\alpha \mapsto \beta\}\{x \mapsto v\}e : \{\alpha \mapsto \beta\}\mathrm{B}\ \mathsf{with}\ \beta\downarrow$, so we are done.
- Case: Rule 40. By assumption, $\Delta; \Gamma \vdash \mathtt{new}\ \alpha\uparrow \mathrm{K}\ \mathtt{in}\ e : \mathrm{A}\ \mathsf{with}\ \sigma\downarrow$ and $\Delta @ \sigma\downarrow; \Gamma \vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}$. By inversion on typing, $\Delta, \alpha\uparrow \mathrm{K}; \Gamma \vdash e : \mathrm{A}\ \mathsf{with}\ \alpha, \sigma\downarrow$. By Proposition 4.26, $(\Delta @ \sigma\downarrow), \alpha\downarrow \mathrm{K}; \Gamma \vdash \mathscr{C} : \mathrm{A}\ \mathsf{cont}$. Since $(\Delta @ \sigma\downarrow), \alpha\downarrow \mathrm{K} = (\Delta, \alpha\uparrow \mathrm{K}) @ (\alpha, \sigma\downarrow)$, we are done.
- Case: Rule 41. By assumption, $\Delta; \Gamma \vdash (\mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ e : \mathrm{B}) : \mathrm{B}\ \mathsf{with}\ \alpha, \sigma\downarrow$ and $\Delta @ \alpha, \sigma\downarrow; \Gamma \vdash \mathscr{C} : \mathrm{B}\ \mathsf{cont}$. By inversion on typing, $\Delta \vdash \alpha := \mathrm{A}\ \mathsf{ok}$, $\Delta @ \alpha := \mathrm{A}; \Gamma \vdash e : \mathrm{B}\ \mathsf{with}\ \sigma\downarrow$, and $\mathsf{basis}_\Delta(\mathrm{A}) \subseteq \sigma$. From the first and third of these, it is easy to see that $(\Delta @ \alpha := \mathrm{A}) @ \sigma\downarrow \vdash \mathsf{id} : \Delta @ \alpha, \sigma\downarrow$. Then, by Proposition 4.26, $(\Delta @ \alpha := \mathrm{A}) @ \sigma\downarrow; \Gamma \vdash \mathscr{C} : \mathrm{B}\ \mathsf{cont}$. In addition, since $\Delta \vdash \omega : \Gamma$, Corollary 4.24 gives us that $\Delta @ \alpha := \mathrm{A} \vdash \omega : \Gamma$, so we are done.
- Case: Rule 42. Analogous to (and simpler than) the previous case.

□

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

Definition 4.29 (*terminal states*)
A machine state Ω is *terminal* if it is of the form BlackHole or $(\Delta; \omega; \bullet; v)$.

Definition 4.30 (*stuck states*)
A machine state Ω is *stuck* if it is not terminal and there is no state Ω' such that $\Omega \rightsquigarrow \Omega'$.

Lemma 4.31 (*canonical forms*)
Suppose $\Delta; \Gamma \Vdash v : \mathrm{A}$ and Γ is run-time. Then:

1. If $\mathrm{A} = \mathtt{unit}$, 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 $\mathtt{fold}_{\mathrm{B}}$ or $\mathtt{unfold}_{\mathrm{B}}$.
4. If $\mathrm{A} = \forall \alpha : \mathrm{K}.\, \mathrm{B}$, then v is of the form $\Lambda \alpha : \mathrm{K}.\, e$.
5. If $\mathrm{A} = \forall \alpha \downarrow \mathrm{K}.\, \mathrm{B}$, 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 $\mathrm{A} = \mathtt{rec}(\mathrm{B})$, then v is of the form x.
8. If $\mathrm{A} = \mathscr{E}\{\alpha\}$, where $\alpha : \mathrm{K} \approx \mathrm{B} \in \Delta$, then v is of the form $\mathtt{fold}_{\mathrm{A}'}(v')$.

Proof
By inversion on typing. □

Theorem 4.32 (*progress*)
If $\vdash \Omega$ ok, then Ω is not stuck.

Proof
Straightforward, using the Canonical Forms Lemma. □

Corollary 4.33 (*type soundness*)
If $\emptyset; \emptyset \vdash e : \mathrm{A}$, then for all Ω, $(\emptyset; \emptyset; \bullet; e) \rightsquigarrow^* \Omega$ implies that Ω is not stuck.

Proof
By Progress and Preservation Theorems. □

5 Encodings in Destination-Passing Style (DPS)

5.1 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 14 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 14, 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 $\alpha_1, \alpha_2 \notin \mathrm{FV}(\mathrm{A}) \cup \mathrm{FV}(\mathrm{B})$.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

$$[\![\forall\alpha_1 \uparrow \mathrm{K}_1, \alpha_2 \uparrow \mathrm{K}_2.\, \mathrm{A}(\alpha_1)(\alpha_2) \xrightarrow{\alpha_1\downarrow,\alpha_2\downarrow} \mathrm{B}(\alpha_1)(\alpha_2)]\!]$$
$$\stackrel{\text{def}}{=} \quad \forall\alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2.\, \mathrm{A}(\pi_1\alpha)(\pi_2\alpha) \xrightarrow{\alpha\downarrow} \mathrm{B}(\pi_1\alpha)(\pi_2\alpha)$$

$$[\![\Lambda\alpha_1 \uparrow \mathrm{K}_1, \alpha_2 \uparrow \mathrm{K}_2.\, \lambda x : \mathrm{A}(\alpha_1)(\alpha_2).\, (e : \mathrm{B}(\alpha_1)(\alpha_2))]\!]$$
$$\stackrel{\text{def}}{=} \quad \Lambda\alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2.\, \lambda x : \mathrm{A}(\pi_1\alpha)(\pi_2\alpha).$$
$$\texttt{new}\ \ \alpha_1 \uparrow \mathrm{K}_1\ \ \texttt{in}$$
$$\texttt{new}\ \ \alpha_2 \uparrow \mathrm{K}_2\ \ \texttt{in}$$
$$\texttt{set}\ \ \alpha := \langle \alpha_1, \alpha_2 \rangle\ \ \texttt{in}$$
$$e\ :\ \mathrm{B}(\pi_1\alpha)(\pi_2\alpha)$$

$$[\![v_1[\alpha_1][\alpha_2](v_2)\ :\ \mathrm{B}(\alpha_1)(\alpha_2)]\!]$$
$$\stackrel{\text{def}}{=} \quad \texttt{new}\ \ \alpha \uparrow \mathrm{K}_1 \times \mathrm{K}_2\ \ \texttt{in}$$
$$\texttt{set}\ \ \alpha_1 := \pi_1\alpha\ \ \texttt{in}$$
$$\texttt{set}\ \ \alpha_2 := \pi_2\alpha\ \ \texttt{in}$$
$$v_1[\alpha](v_2)\ :\ \mathrm{B}(\alpha_1)(\alpha_2)$$

Fig. 14. Encoding of multiple-argument DPS universals.

$$[\![\exists\alpha \downarrow \mathrm{K}.\, \mathrm{A}]\!]_{\mathrm{DPS}} \stackrel{\text{def}}{=} \forall\alpha \uparrow \mathrm{K}.\, \texttt{unit} \xrightarrow{\alpha\downarrow} \mathrm{A}$$

$$[\![\texttt{pack}\ [\mathrm{A}, v]\ \texttt{as}\ \exists\alpha \downarrow \mathrm{K}.\, \mathrm{B}]\!]_{\mathrm{DPS}} \stackrel{\text{def}}{=} \Lambda\alpha \uparrow \mathrm{K}.\, \lambda().$$
$$\texttt{set}\ \ \alpha := \mathrm{A}\ \ \texttt{in}\ \ v\ :\ \mathrm{B}$$

$$[\![\texttt{let}\ [\alpha, x] = \texttt{unpack}\ v\ \texttt{in}\ e]\!]_{\mathrm{DPS}} \stackrel{\text{def}}{=} \texttt{new}\ \ \alpha \uparrow \mathrm{K}\ \ \texttt{in}$$
$$\texttt{let}\ \ x = v[\alpha]()\ \ \texttt{in}\ \ e$$

Fig. 15. DPS universal encoding of existentials.

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.

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 hidden (as discussed at the end of Section 2.3).

5.2 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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

now make that argument precise. Figure 15 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 as $\exists\alpha\downarrow\mathrm{K}.\,\mathrm{A}$. 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 `set` construct. Note that A must be stable in order for the encoding of `pack` to be well-typed, since A serves as the definition for 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 that 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

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

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`.

6 Comparison with conference version

The type system and meta-theoretic development presented in Sections 3 and 4 exhibit some technical improvements over the type system presented in the conference version of this article (Dreyer 2005a). We now briefly summarize these improvements.

The most noticeable change is that in the conference version of the type system the act of backpatching an abstract type and the act of sealing its definition were separated into two distinct constructs, whereas in this system they are merged into one (the `set` construct). This change was suggested to us by Andreas Rossberg, and we feel the present approach marks an improvement for several reasons. First of all, none of the examples we gave in the original paper exploited the separation of constructs. If anything, the examples are all easier to read when phrased in terms of the `set` construct. Second, as we remarked in Section 2.1, the `set` construct is close in form to traditional mechanisms for data abstraction. In some sense, it feels very

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

natural for the construct that defines an abstract type to also specify the scope in which that definition is visible. Third, the `set` construct simplifies the type system. In the conference version, the type effect engendered by a term e could be a set of arbitrary φ's. In this system, it is only possible for a term to engender an effect of the form $\sigma\downarrow$.

The other, less noticeable, but semantically more significant, change is in how we define the basis of a type constructor. In the conference version of the type system, we defined $\mathsf{basis}_\Delta(\mathrm{A})$ as the set of unstable free variables of B, where B is A with all references to type synonym variables fully expanded out. In this system, we define $\mathsf{basis}_\Delta(\mathrm{A})$ as the set of unstable free variables of A's normal form, $\mathsf{norm}_\Delta(\mathrm{A})$. Superficially, these definitions might seem interchangeable, but they are not. For instance, consider the context $\Delta = \alpha\uparrow\mathbf{T}, \beta : \mathbf{T}\times\mathbf{T} = \langle\alpha, \mathtt{int}\rangle$. What is $\mathsf{basis}_\Delta(\pi_2(\beta))$? Under our present definition, it is the empty set, since $\pi_2(\beta)$ is equal to `int`. However, under our conference definition, it is $\{\alpha\}$—if we expand out the type synonym β, we get $\pi_2\langle\alpha, \mathtt{int}\rangle$, which has a free, albeit inessential, reference to α. The present definition of $\mathsf{basis}_\Delta(\mathrm{A})$ is thus clearly superior.

Moreover, the present approach simplifies the definition of well-formed substitution. In the conference version, we were forced to include an ugly extra side condition on Part 4 of Definition 4.7 in order for the Monotonicity Lemma (Lemma 4.11) to go through. No such side condition is necessary under the present definition of $\mathsf{basis}_\Delta(\mathrm{A})$.

7 Related work

7.1 Recursive modules

As discussed in the introduction, 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 *et al.* (1999) 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 (Russo 2001; Romanenko *et al.* 2000). 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 (2004) has implemented recursive modules in O'Caml, but has not provided any formal account of their 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 (1999) CMS calculus, in particular, has been highly influential and led to a considerable body of work on "mixin modules". However, it basically ignores all issues involving type components (and hence, data abstraction) in modules.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

More recently, Duggan (2002) has developed a language of "recursive DLLs". 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. On the basis of his informal discussion, Duggan appears to address some of the problems of recursive ADTs 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 (Flatt 2005). While MzScheme is dynamically typed, their article formalizes an extension of units to the statically typed setting as well (Flatt & Felleisen 1998). 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 specify explicitly how the export types of a unit depend on the import types. The unit approach to cycle avoidance is potentially more accurate, but at the expense of infecting interfaces with complex dependency information.

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.

7.2 Data abstraction, effects, and linearity

Rossberg (2003) 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

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

analysis. Thus, his system requires one to define an abstract type at the same point where it is created.

Dreyer *et al.* (2003) give an interpretation of ML-style modularity in which data abstraction is treated as a computational effect. In fact, they consider two kinds of effects—*static* and *dynamic*—corresponding to the different varieties of data abstraction offered by different dialects of ML, and use an effect system to track what kinds of abstract types (if any) a module defines. In this work, we also use effects (of a somewhat different flavor) to track abstract type definitions. We ignore static effects, though, and restrict attention to dynamic effects, which correspond to the form of type generativity supported by Standard ML. Our interpretation of dynamic effects is more refined than that of Dreyer *et al.* (2003) in that we allow abstract types to be created and referred to *before* they are defined, thus making it possible to link such types recursively.

Our type system treats the definition of abstract types in a strictly *linear* way—that is, once a variable α is introduced as writable, it must be defined *exactly once* before it leaves scope. It is natural, therefore, to ask whether it is possible and/or worthwhile to recast our tracking of abstract type definitions in terms of a *linear type system* (Wadler 1990) instead of an effect system (Gifford & Lucassen 1986). Traditionally, linear type systems use linear types as a way of ensuring that variables are used exactly once. In our language, however, it is never our goal to restrict a type variable α to be used exactly once. Rather, the resource that we wish to restrict to be used exactly once is the *ability* to define a writable variable.

For this reason, we believe that our tracking of type effects has less in common with linear type systems than with type systems that track *capabilities*, most notably that of Walker *et al.* (2000).[8] Developed in the setting of region-based memory management, the type system Walker *et al.* uses capabilities to track the permissibility of certain effectful operations, such as memory access and deallocation. In the context of our type system, the writable binding $\alpha \uparrow \mathrm{K}$ can be viewed as a combination of an ordinary binding $\alpha : \mathrm{K}$ and a linear capability specifying that α may be written to. While we think it would be interesting to split our writable binding in this way and make the writable capability explicit in the style of Walker *et al.*, we fail to see how it would lead to any significant simplification of our type system or its meta-theory.

8 Future work

There exists a rich body of work on semantic methods—both denotational (Mitchell 1996) and operational (Pitts 2005)—for reasoning about programs with existential types. One criticism of our approach to recursive type generativity is that it is not clear how easy it will be to develop analogous semantic methods for reasoning about data abstraction in the presence of type-level recursive backpatching. We concede that this is a valid criticism, and we are currently investigating whether it is feasible to adapt to our setting any of the recently proposed methods for semantic reasoning

[8] At least conceptually. Formally speaking, as Morrisett *et al.* (2005) have recently shown, capability tracking can be expressed as a particular mode of use of linear typing.

https://doi.org/10.1017/S0956796807006429 Published online by Cambridge University Press

in the presence of mutable state (Benton & Leperchey 2005; Bohr & Birkedal 2006; Koutavas & Wand 2006). In any case, 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 on the basis of information that is only available dynamically. 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 tied in with the more general question of whether the ideas of this article are scalable to the level of a module language. When the conference version of this article (Dreyer 2005a) was originally published, the answer to this latter question was unclear. Since then, however, we have adapted the RTG type system to serve as the basis of a preliminary proposal for recursive SML-style modules (Dreyer 2006). We have also found, somewhat surprisingly, that the ability to forward-declare types that RTG offers is useful independently of recursive modules—namely, in establishing a soundness and completeness result for Damas-Milner type inference in the presence of non-recursive modules (Dreyer & Blume 2007). These results suggest that RTG has the potential to serve as a foundation for future evolution of the ML module system.

Acknowledgments

We thank Andreas Rossberg for his suggestion about combining definition and sealing into a single `set` construct, Aleks Nanevski for suggesting that we pursue an idea along these lines years ago, and all of the anonymous *ICFP* and *JFP* referees for giving such helpful and detailed comments.

References

Amadio, R. & Cardelli, L. (1993) Subtyping recursive types. *ACM Trans. Program. Lang. Syst.* **15**(4), 575–631.

Ancona, D. & Zucca, E. (1999) A primitive calculus for module systems. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*. Lecture Notes in Computer Science, vol. 1702. London, UK: Springer-Verlag, pp. 62–79.

Benton, N. & Leperchey, B. (2005) Relational reasoning in a nominal semantics for storage. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, pp. 86–101.

Bohr, N. & Birkedal, L. (2006) Relational reasoning for recursive types and references. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pp. 79–96.

Crary, K., Harper, R. & Puri, S. (1999) What is a recursive module? In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 50–63.

Dreyer, D. (2004) A type system for well-founded recursion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 293–305.

Dreyer, D. (2005a) Recursive type generativity. In *International Conference on Functional Programming (ICFP)*, pp. 41–53.

Dreyer, D. (2005b) *Understanding and Evolving the ML Module System*, Ph.D. thesis. Pittsburgh, PA: Carnegie Mellon University.

Dreyer, D. (2006) *Practical Type Theory for Recursive Modules*, Tech. rept. TR-2006-07. Department of Computer Science, University of Chicago.

Dreyer, D. & Blume, M. (2007) Principal type schemes for modular programs. In *European Symposium on Programming (ESOP)*. pp. 441–457.

Dreyer, D., Crary, K. & Harper, R. (2003) A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL)*. pp. 236–249.

Duggan, D. (2002) Type-safe linking with recursive DLL's and shared libraries. *ACM Trans. Program. Lang. Syst.*, **24**(6), 711–804.

Flatt, M. (2005) *PLT MzScheme: Language Manual*. Available at: `www.plt-scheme.org`.

Flatt, M. & Felleisen, M. (1998) Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 236–248.

Gifford, D. K. & Lucassen, J. M. (1986) Integrating functional and imperative programming. *ACM Conference on LISP and Functional Programming*. pp. 28–38.

Harper, R. & Stone, C. (2000) A type-theoretic interpretation of Standard ML. In Plotkin, G., Stirling, C. & Tofte, M. (eds) *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. Cambridge, MA: MIT Press. pp. 341–387.

Jones, M. P. (1996) Using parameterized signatures to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 68–78.

Koutavas, V. & Wand, M. (2006) Small bisimulations for reasoning about higher-order imperative programs. *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 141–152.

Leroy, X. (1995) Applicative functors and fully transparent higher-order modules. *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* pp. 142–153.

Leroy, X. (2003) *A Proposal for Recursive Modules in Objective Caml*. Available from the author's website.

Leroy, X. (2004) *The Objective Caml System: Documentation and User's Manual*. Available at: `www.ocaml.org`.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. Cambridge, MA: MIT Press.

Mitchell, J. C. (1996) *Foundations for Programming Languages*. Cambridge, MA: MIT Press.

Morrisett, G., Ahmed, A. & Fluet, M. (2005) L^3: A linear language with locations. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, pp. 293–307.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge, UK: Cambridge University Press.

Pitts, A. (2005) Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed). Cambridge, MA: MIT Press, chap. 7.

Romanenko, S., Russo, C. & Sestoft, P. (2000) *Moscow ML Language Overview*. Available at: `www.dina.kvl.dk/~sestoft/mosml.html`.

Rossberg, A. (2003) Generativity and dynamic opacity for abstract types. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*. pp. 241–252.

Russo, C. V. (2001) Recursive structures for Standard ML. In *International Conference on Functional Programming (ICFP)*, pp. 50–61.

Stone, C. A. (2005) Type definitions. In *Advanced Topics in Types and Programming Languages* Pierce, B. C. (ed). Cambridge, MA: MIT Press, chap. 9.

Stone, C. A. & Harper, R. (2006) Extensional equivalence and singleton types. *ACM Trans. Comput. Logic.*, **7**(4), 676–722.

Talpin, J.-P. & Jouvelot, P. (1994) The type and effect discipline. *Inf. Comput.* **111**(2), 245–296.

Vanderwaart, J. C., Dreyer, D., Petersen, L., Crary, K., Harper, R. & Cheng, P. (2003) Typed compilation of recursive datatypes. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*. pp. 98–108.

Wadler, P. (1985) *Listlessness is Better than Laziness*, Ph.D. thesis. Pittsburgh, PA: Carnegie Mellon University.

Wadler, P. (1990) Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*. pp. 347–359.

Walker, D., Crary, K. & Morrisett, G. (2000) Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.* **22**(4), 701–771.