A Type System for Well-Founded Recursion*

Derek Dreyer

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dreyer@cs.cmu.edu

Abstract

In the interest of designing a recursive module extension to ML that is as simple and general as possible, we propose a novel type system for general recursion over effectful expressions. The presence of effects seems to necessitate a backpatching semantics for recursion similar to that of Scheme. Our type system ensures statically that recursion is well-founded—that the body of a recursive expression will evaluate without attempting to access the undefined recursive variable—which avoids some unnecessary run-time costs associated with backpatching. To ensure well-founded recursion in the presence of multiple recursive variables and separate compilation, we track the usage of individual recursive variables, represented statically by "names". So that our type system may eventually be integrated smoothly into ML's, reasoning involving names is only required inside code that uses our recursive construct and need not infect existing ML code, although instrumentation of some existing code can help to improve the precision of our type system.

Categories and Subject Descriptors

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

General Terms

Languages, Theory

Keywords

Type systems, recursion, recursive modules, effect systems

*This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the author(s) and do not reflect the views of this agency.

POPL'04, January 14–16, 2004, Venice, Italy.

1 Introduction

A distinguishing feature of the programming languages in the ML family, namely Standard ML [21] and Objective Caml [25], is their strong support for modular programming. The module systems of both languages, however, are strictly hierarchical, prohibiting cyclic dependencies between program modules. This restriction is unfortunate because it means that mutually recursive functions and types must always be *defined* in the same module, regardless of whether they *belong* conceptually in the same module. As a consequence, recursive modules are one of the most commonly requested extensions to the ML languages.

There has been much work in recent years on recursive module extensions for a variety of functional languages. One of the main stumbling blocks in designing such an extension for an impure language like ML is the interaction of module-level recursion and core-level computational effects. Since the core language of ML only permits recursive definitions of λ-abstractions (functions), recursive linking could arguably be restricted to modules that only contain `fun` bindings. The banishing of all computational effects, however, would place a severe restriction on recursive module programming.

Some recursive module proposals attempt to ameliorate this restriction by splitting modules into a *recursively linkable* section and an *initialization* section, and only subjecting the former to syntactic restrictions [10]. While such a construct is certainly more flexible than one that forbids effects entirely, it imposes a structure on recursive modules that is rather arbitrary. Others have suggested abandoning ML-style modules altogether in favor of *mixin modules* [3, 17] or *units* [13], for which recursive linking is the norm and hierarchical linking a special case. For the purpose of extending ML, though, this would constitute a rather drastic revision of the language.

1.1 Recursion and Effects

In the interest of designing a recursive module extension to ML that is as simple and general as possible, suppose that we were to introduce a new form of structure declaration

```
structure rec X = M
```

in which the structure M may refer to itself recursively as X, and there are no *a priori* limitations on M. How should recursion interact with any computational effects that may occur during evaluation of M?

```
structure rec X = struct
  structure A = struct
    val debug = ref false
    fun f(x) = ...X.B.g(x-1)...
  end
  structure B = struct
    val trace = ref false
    fun g(x) = ...X.A.f(x-1)...
  end
end
```

Figure 1. Example of Recursive Module with Effects

```
functor myA (X : SIG) = ...

functor yourB (X : SIG) = ...

structure rec X = struct
  structure A = myA(X)
  structure B = yourB(X)
end
```

Figure 2. Separate Compilation of Recursive Modules

Under the standard interpretation of recursion via a fixed-point operator, the new recursive structure declaration would be tantamount to `structure X =` $\mathsf{fix}(X.M)$, where $\mathsf{fix}(X.M)$ evaluates to its unrolling $M[\mathsf{fix}(X.M)/X]$.[1] Such a fixed-point semantics has the property that any computational effects in M are re-enacted at every recursive reference to X.

While there is nothing inherently wrong with this behavior, it is undesirable for many intended uses of recursive modules. For example, consider the declaration of two mutually recursive structures `A` and `B` in Figure 1. Here, `debug` and `trace` are externally-accessible debugging flags used by `f` and `g`, respectively. Under the above fixed-point semantics, every recursive reference between `f` and `g` prompts a re-evaluation of the entire module, including the creation of brand new ref cells for `debug` and `trace`. In other words, each recursive call operates in an entirely different mutable state, so setting `debug` to `true` externally would not alter the fact that `!debug` is `false` during all recursive calls to `X.A.f` and `X.B.g`.

An alternative semantics for recursion that exhibits more appropriate behavior with respect to computational effects is the *backpatching* semantics of Scheme [19], in which `structure rec X = M` would evaluate as follows: First, X is bound to a fresh location containing an undefined value; then, M is evaluated to a module value V; finally, X is backpatched with V. If the evaluation of M attempts to dereference X, a run-time error is reported. Unlike the fixed-point semantics, backpatching ensures that the effects in M only happen once.

One might argue that what the backpatching semantics really achieves is the ability to write "excessively recursive" definitions. In the example in Figure 1, the effectful definitions of `debug` and `trace` do not really participate in the recursion. One might therefore imagine a semantics for `structure rec` that models the recursion via a fixed-point, but hoists the effects outside of the fixed-point so that they only occur once. However, while hoisting the effects may result in the same behavior as the backpatching semantics when the effect in question is *state*, it is well-known that the same is not true for *continuations*, as it matters whether a continuation is captured inside or outside of the recursive definition [14].

Moreover, hoisting the effects is impossible in the context of separate compilation. In particular, consider Figure 2, which shows how the structures `A` and `B` from Figure 1 may be developed apart from each other by abstracting each one over the recursive variable `X`. The `structure rec` linking them may also be compiled separately, in which case we do not have access to the implementations of `myA` and `yourB` and there is no way to hoist the effects out of `myA(X)` and `yourB(X)`. The backpatching semantics thus seems to be a simpler, cleaner and more general approach.

1.2 Well-Founded Recursion

Russo employs the backpatching semantics described above in his recursive module extension to Moscow ML [28]. Russo's extension has the advantage of being relatively simple, largely because the type system does not make any attempt to statically ensure that `structure rec X = M` is *well-founded*, *i.e.,* that the evaluation of M will not dereference X.

If possible, though, compile-time error detection is preferable. In addition, statically ensuring well-foundedness would allow recursive modules to be implemented more efficiently. In the absence of static detection, there are two well-known implementation choices: (1) the recursive variable X can be implemented as a pointer to a value of `option` type (initially `NONE`), in which case every dereference of X must also perform a tag check to see if it has been backpatched yet, or (2) X can be implemented as a pointer to a thunk (initially `fn () => raise Error`), in which case every dereference of X must also perform a function call. Either way, mutually recursive functions defined across module boundaries will be noticeably slower than ordinary ML functions. If recursion is statically known to be well-founded, however, the value pointed to by X will be needed only after X has been backpatched, so each access will require just a pointer dereference without any additional tag check or function call.

In this paper we propose a type-theoretic approach to ensuring well-founded recursive definitions under a backpatching semantics of recursion. The basic idea is to model recursive variables statically as *names*, and to use names to track the set of recursive variables that a piece of code may attempt to dereference when evaluated. Our use of names is inspired by the work of Nanevski on a core language for metaprogramming and symbolic computation [23], although it is closer in detail to his work (concurrent with ours) on using names to model control effects [24].

Names are important both for tracking uses of multiple recursive variables in the presence of nested recursion and for supporting separate compilation of recursive modules. An equally important feature of our approach is that recursive modules may invoke functions defined in existing ML code without requiring them to be changed or recompiled to account for name reasoning. Nevertheless, as we discuss in Section 3.3, there are useful recursive module idioms for which instrumentation of existing ML code appears to be unavoidable if one wants to statically ensure that the recursion is well-founded.

As there are a number of difficult issues surrounding static (type) components of recursive modules [5, 8], we restrict our attention

[1] We use $M[N/X]$ to denote the capture-avoiding substitution of N for X in M.

here to the dynamic (code) components of recursive modules. Correspondingly, we develop our type system at the level of recursive (core-level) *expressions*. We do not intend this as an extension to the core language of ML, but as the basis of a future extension to the module language.

1.3 Overview

The remainder of the paper is organized as follows: In Section 2 we introduce the notion of *evaluability*, which ensures that a program is safe to evaluate even if it contains free references to undefined recursive variables. Through a series of examples, we illustrate how a simple approach to tracking evaluability suffers from a number of theoretical and practical problems. In Section 3 we present our core type system for solving these problems, in the context of the (pure) simply-typed λ-calculus. While effects necessitate the backpatching semantics of recursion, all of the subtleties involving names can in fact be explored here in the absence of effects. We give the static and dynamic semantics of our core language, along with meta-theoretic properties including type safety.

In Section 4 we show how to encode an unrestricted form of recursion by extending the language with memoized computations. While this unrestricted construct does not ensure well-founded recursion, it is useful as a fallback in circumstances where our type system is too weak to observe that a recursive term is well-founded. In Section 5 we compare our approach with related work on well-founded recursion and recursive modules. Finally, in Section 6 we conclude and suggest future work.

2 Evaluability

Consider a general recursive construct of the form $\mathsf{rec}(x{:}\tau.\,e)$, representing an expression e of type τ that may refer to its ultimate value recursively as x. What is required of e to ensure that $\mathsf{rec}(x{:}\tau.\,e)$ is well-founded? Crary *et al.* [5] require that e be *valuable* (that is, pure and terminating) in a context where x is not. We generalize their notion of valuability to one permitting effects, which we call *evaluability*: a term may be judged evaluable if its evaluation does not access an undefined recursive variable. Thus, to ensure $\mathsf{rec}(x{:}\tau.\,e)$ is well-founded, the expression e must be *evaluable* in a context where uses of the variable x are *non-evaluable*. An expression can be non-evaluable and still well-formed, but only evaluable expressions are safe to evaluate in the presence of undefined recursive variables.

Formally, we might incorporate evaluability into the type system by dividing the typing judgment into one classifying evaluable terms ($\Gamma \vdash e \downarrow \tau$) and one classifying non-evaluable terms ($\Gamma \vdash e \uparrow \tau$). (There is an implicit inclusion of the former in the latter.) In addition, we need to extend the language with a notion of undefined variables, which are bound in the context as $x \uparrow \tau$, as opposed to ordinary variables which are bound as $x : \tau$. The distinction between them can be seen from their typing rules:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \downarrow \tau} \qquad \frac{x \uparrow \tau \in \Gamma}{\Gamma \vdash x \uparrow \tau}$$

Given these extensions, we can now give the following typing rule for recursive expressions:

$$\frac{\Gamma, x \uparrow \tau \vdash e \downarrow \tau}{\Gamma \vdash \mathsf{rec}(x{:}\tau.\,e) \downarrow \tau}$$

2.1 The Evaluability Judgment

While true evaluability is clearly an undecidable property, there are certain kinds of expressions that we can expect the type system to recognize as evaluable. For instance, recall the example from Figure 1, which recursively defines a pair of submodules, each of which is a pair of a `ref` expression and a λ-abstraction. In general, all values and tuples of evaluable expressions should be considered evaluable. In addition, `ref`(e), `!`e, and e_1`:=`e_2 should all be evaluable as long as their constituent expressions are. Evaluability is thus independent of computational purity.

There is, however, a correspondence between *non-evaluability* and *computational impurity* in the sense that both are hidden by λ-abstractions and unleashed by function applications. In ML we assume (for the purpose of the value restriction) that all function applications are potentially impure. In the current setting we might similarly assume for simplicity that all function applications are potentially non-evaluable.

Unfortunately, this assumption has one major drawback: it implies that we can never evaluate a function application inside a recursive expression! Furthermore, it is usually unnecessary: while functions defined inside a recursive expression may very well be hiding references to an undefined variable, functions defined in existing ML code will not. For example, instead of defining local state with a `ref` expression, suppose that we wish to define a mutable array in submodule `A` (of Figure 1) by a call to the array creation function:

```
...
  structure A = struct
    val a = Array.array(n,0)
    fun f(x) = ...Array.update(a,i,m)...
    fun g(x) = ...X.B.f(x-1)...
  end
...
```

The call to `Array.array` is perfectly evaluable, while a call to the function `A.g` inside the above module might *not* be. Lumping them together and assuming the worst makes the evaluability judgment far too conservative.

2.2 A Partial Solution

At the very least, then, we should distinguish between the types of *total* and *partial* functions. For present purposes, a *total* arrow type $\tau_1 \to \tau_2$ classifies a function whose body is evaluable, and a *partial* arrow type $\tau_1 \rightharpoonup \tau_2$ classifies a function whose body is potentially non-evaluable:[2]

$$\frac{\Gamma, x : \sigma \vdash e \downarrow \tau}{\Gamma \vdash \lambda x.\, e \downarrow \sigma \to \tau} \qquad \frac{\Gamma, x : \sigma \vdash e \uparrow \tau}{\Gamma \vdash \lambda x.\, e \downarrow \sigma \rightharpoonup \tau}$$

Correspondingly, applications of total evaluable functions to evaluable arguments will be deemed evaluable, whereas applications of partial functions will be assumed non-evaluable:

$$\frac{\Gamma \vdash e_1 \downarrow \sigma \to \tau \quad \Gamma \vdash e_2 \downarrow \sigma}{\Gamma \vdash e_1(e_2) \downarrow \tau} \qquad \frac{\Gamma \vdash e_1 \uparrow \sigma \rightharpoonup \tau \quad \Gamma \vdash e_2 \uparrow \sigma}{\Gamma \vdash e_1(e_2) \uparrow \tau}$$

The total/partial distinction addresses the concerns discussed in the previous section, to an extent. Existing ML functions can now be classified as total, and the arrow type τ_1`->`τ_2 in ML proper is synonymous with a total arrow. Thus, we may now evaluate calls to

[2] The "total/partial" nomenclature arises from viewing non-evaluability as a kind of computational effect.

existing ML functions in the presence of undefined recursive variables, as those function applications will be known to be evaluable. However, there are still some serious problems.

2.3 Problems

Nested Recursion First, consider what happens when we use general recursion to define a recursive function, such as factorial:

```
rec(f : int ⇀ int. fn x => ... x * f(x-1) ...)
```

Note that we are forced to give the recursive expression a partial arrow type because the body of the factorial function uses the recursive variable `f`. Nonetheless, exporting factorial as a partial function is bad because it means that no application of factorial can ever be evaluated inside a recursive expression!

To mend this problem, we observe that while the factorial function is indeed partial during the evaluation of the general recursive expression defining it, it becomes total as soon as `f` is backpatched with a definition. One way to incorporate this observation into the type system is to revise the typing rule for recursive terms $\mathsf{rec}(x:\tau.\,e)$ so that we ignore partial/total discrepancies when matching the declared type τ with the actual type of e. For example, in the factorial definition above, we would allow `f` to be declared with a total arrow `int` $\rightarrow$ `int`, since the body of the definition has an equivalent type *modulo* a partial/total mismatch.

Unfortunately, such a revised typing rule is only sound if we prohibit nested recursive expressions. Otherwise, the rule may allow a truly partial function to be erroneously assigned a total type, as the following code illustrates:

```
rec(x : τ.
  let
     val f = rec(y : unit → τ. fn () => x)
  in
     f()
  end
)
```

The trouble here is that the evaluation of the recursive expression defining `f` results only in the backpatching of `y`, not `x`. It is therefore unsound for that expression to make the type of `fn () => x` total. In short, the problem is that the total/partial dichotomy is too coarse because it does not distinguish between uses of different recursive variables. In the type system of Section 3, we will be able to give `f` a more appropriate type specifying that `f` will dereference `x` when applied, but not `y`.

Higher-Order Functions Another problem with the total/partial distinction arises in the use of higher-order functions. Suppose we wish to use the Standard Basis `map` function for lists, which can be given the following type (for any σ and τ):

```
val map : (σ → τ) → (σ list → τ list)
```

Since the type of `map` is a pure ML type, all the arrows are total, which means that we cannot apply `map` to a partial function, as in the following:

```
rec (X : SIG.
  let
     val f : σ ⇀ τ = ...
     val g : σ list ⇀ τ list = map f
  ...
)
```

Given the type of `map`, this is reasonable: unless we know how `map` is implemented, we have no way of knowing that evaluating `map f` will not try to apply `f`, resulting in a potential dereference of `X`.

Nevertheless, we should at least be able to replace `map f` with its eta-expansion `fn xs => map f xs`, which is clearly evaluable since it is a value. Even its eta-expansion is ill-typed, however, because the type of `f` still does not match the argument type of `map`. The way we propose to resolve this problem is to view a partial/total type mismatch not as a sign that the offending expression (in this case, `map f`) is ill-typed, but merely that it is potentially non-evaluable. The type system of Section 3 will reflect this intuition, and will correspondingly consider the function `fn xs => map f xs` to be well-typed with a *partial* arrow, but not a total one.

Separate Compilation Russo points out a problem with separate compilation of recursive modules in Moscow ML [28] that the system we have sketched thus far suffers from as well: there is no way to refer to a recursive variable without dereferencing it. For instance, recall the separate compilation scenario from Figure 2. The code in Figure 2 is ill-typed under our current setup because, under call-by-value semantics, the functor applications `myA(X)` and `yourB(X)` will begin by evaluating the recursive variable `X`, which is undefined.

What we really intended, however, was not for the functor applications to dereference the recursive variable `X` and pass the resulting module value as an argument, but rather to pass the recursive variable `X` as an argument itself without dereferencing it. The way to account for this intended semantics is to treat a recursive variable not as a (potentially divergent) expression, but as a *value* of a new *location* type that must be dereferenced explicitly. This idea will be fleshed out further in the next section.

3 A Type System for Well-Founded Recursion

In this section we present a type system for well-founded recursion that addresses all the problems enumerated in the previous section. To address the nested recursion problem, we generalize the judgment of evaluability to one that tracks uses of individual recursive variables. We achieve this by introducing along with each recursive variable a *name* that is used as a static representative of the variable. The new evaluability judgment has the form $\Gamma \vdash e : \tau\ [S]$, with the interpretation "under context Γ, term e has type τ and is evaluable *modulo* the names in set S". In other words, e will evaluate without dereferencing any recursive variables *except* possibly those whose associated names appear in S. Following Nanevski [23], we call a finite set of names a *support*. Our previous judgment of evaluability ($\Gamma \vdash e \downarrow \tau$) can be understood as evaluability modulo the empty support ($\Gamma \vdash e : \tau\ [\emptyset]$), while non-evaluability ($\Gamma \vdash e \uparrow \tau$) corresponds to evaluability modulo *some* non-empty support.

Similarly, we generalize the types of functions to bear a support indicating which particular recursive variables may be dereferenced in their bodies. Thus, the total arrow type of the previous section becomes an arrow type bearing empty support, while the partial arrow type corresponds to an arrow type bearing *some* non-empty support.

To address the higher-order function problem, we employ a novel judgment of *type equivalence modulo a support*, which allows type mismatches in an expression to be ignored so long as they only involve names that are in the support of the expression. The intu-

Variables	$x, y, z \in$	*Variables*
Names	$\mathrm{X}, \mathrm{Y}, \mathrm{Z} \in$	*Names*
Supports	$\mathrm{S}, \mathrm{T} \in$	$\mathcal{P}_{\mathrm{fin}}(\mathit{Names})$
Types	$\sigma, \tau ::=$	$1 \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow{\mathrm{S}} \tau_2$ $\mid \forall \mathrm{X}.\, \tau \mid \mathsf{box}_{\mathrm{S}}(\tau)$
Terms	$e, f ::=$	$x \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \pi_i(e)$ $\mid \lambda x.\, e \mid f(e) \mid \lambda \mathrm{X}.\, e \mid f(\mathrm{S})$ $\mid \mathsf{box}(e) \mid \mathsf{unbox}(e)$ $\mid \mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e)$
Values	$v ::=$	$x \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \lambda x.\, e \mid \lambda \mathrm{X}.\, e$
Typing Contexts	$\Gamma ::=$	$\emptyset \mid \Gamma, x : \tau \mid \Gamma, \mathrm{X}$

Figure 3. Core Language Syntax

ition behind this judgment is that we are only interested in tracking uses of *undefined* recursive variables; since the names that appear in the support of an expression correspond to recursive variables that must be *defined* before the expression can be evaluated, they can be safely ignored when typing it.

To address the separate compilation problem, our type system treats recursive variables as values of a new box type classifying (potentially uninitialized) memory locations. Recursive expressions in our type system have the form $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e)$[3], introducing (in the scope of e) the name X and the recursive variable x, which is bound in the context with type $\mathsf{box}_{\mathrm{X}}(\tau)$. The type of x indicates that x is a memory location and that any expression attempting to dereference (unbox) it must have X in its support. Consequently, what we previously wrote as $\mathsf{rec}(x{:}\tau.\, e)$ would now be written as $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e')$, where e' replaces occurrences of the *expression* x in e with an explicit dereference of the *value* x (written $\mathsf{unbox}(x)$). In addition, note that we no longer need to distinguish recursive variables from ordinary variables through a separate context binding like $x \uparrow \tau$; a recursive variable is distinguished simply by its box type.

3.1 Syntax

The syntax of our core language is given in Figure 3. We assume the existence of countably infinite sets of names (*Names*) and variables (*Variables*), and use S and T to range over supports. We often write the name X as shorthand for the singleton support $\{\mathrm{X}\}$.

The type structure of the language is as follows. Unit (1) and pair types ($\tau_1 \times \tau_2$) require no explanation. An arrow type ($\tau_1 \xrightarrow{\mathrm{S}} \tau_2$) bears a support on the arrow, which indicates the set of names whose associated recursive variables must be defined before a function of this type may be applied. We will sometimes write $\tau_1 \to \tau_2$ as shorthand for an arrow type with empty support ($\tau_1 \xrightarrow{\emptyset} \tau_2$).

The language also provides the ability to abstract an expression over a name. The type $\forall \mathrm{X}.\, \tau$ classifies name abstractions $\lambda \mathrm{X}.\, e$, which suspend the evaluation of their bodies and are treated as values. Application of a name abstraction, $f(\mathrm{S})$, allows the name parameter of f to be instantiated with a support S, not just a single name. The reasons for allowing names to be instantiated with supports are discussed in Section 3.3.

Lastly, the location type $\mathsf{box}_{\mathrm{S}}(\tau)$ classifies a memory location that will contain a *value* of type τ once the recursive variables associated with the names in S have been defined. Locations are most commonly introduced by recursive expressions, but they may also be introduced by $\mathsf{box}(e)$, which evaluates e and then "boxes" the resulting value, *i.e.,* stores it at a new location. Since each boxing may potentially create a new location, $\mathsf{box}(v)$ is not a value; the only values of location type are variables. The elimination form for location types is $\mathsf{unbox}(e)$, which dereferences the location resulting from the evaluation of e. We will sometimes write $\mathsf{box}(\tau)$ as shorthand for $\mathsf{box}_{\emptyset}(\tau)$.

Notational Conventions In the term $\lambda x.\, e$, the variable x is bound in e; in the term $\lambda \mathrm{X}.\, e$ and type $\forall \mathrm{X}.\, \tau$, the name X is bound in e and τ; in the term $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e)$, the name X and variable x are bound in e. As usual, we identify terms and types that are equivalent modulo α-conversion of bound variables/names.

For notational convenience, we enforce several implicit requirements on the well-formedness of contexts and judgments. A context Γ is well-formed if (1) it does not bind the same variable/name twice, and (2) for any prefix of Γ of the form $\Gamma', x : \tau$, the free names of τ are bound in Γ'. A judgment of the form $\Gamma \vdash \cdots$ is well-formed if (1) Γ is well-formed, and (2) any free names appearing to the right of the turnstile are bound in Γ. We assume and maintain the implicit invariant that all contexts and judgments are well-formed.

3.2 Static Semantics

The main typing judgment has the form $\Gamma \vdash e : \tau\ [\mathrm{S}]$. The support S represents the set of names whose associated recursive variables we may assume have been defined (backpatched) by the time e is evaluated. Put another way, the only recursive variables that e may dereference are those associated with the names in S. The static semantics is carefully designed to validate this assumption.

The rules of the type system (Figure 4) are designed to make admissible the principle of *support weakening*, which says that if $\Gamma \vdash e : \tau\ [\mathrm{S}]$ then $\Gamma \vdash e : \tau\ [\mathrm{T}]$ for any $\mathrm{T} \supseteq \mathrm{S}$. Thus, for instance, since a variable x does not require any support, Rule 1 allows x to be assigned any support $\mathrm{S} \subseteq \mathrm{dom}(\Gamma)$, not just the empty support.

The remainder of the rules may be summarized as follows. Unit needs no support (Rule 2), but pairs and projections require the support of their constituent expressions (Rules 3 and 4). A function $\lambda x.\, e$ has type $\sigma \xrightarrow{\mathrm{T}} \tau$ in any support S, so long as the body e is well-typed under the addition of support T (Rule 5). To evaluate a function application $f(e)$, the support must contain the supports of f and e, as well as the support on f's arrow type (Rule 6).

Although a name abstraction $\lambda \mathrm{X}.\, e$ suspends the evaluation of e, the body is typechecked under the same support as the abstraction itself (Rule 7). In other words, one can view $\forall \mathrm{X}.\, \tau$ as another kind of arrow type that always bears empty support (compare with Rule 5 when $\mathrm{T} = \emptyset$). Note also that our assumptions about the well-formedness of judgments ensures that the support S cannot contain X, since $\mathrm{S} \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{X} \notin \mathrm{dom}(\Gamma)$. Restricting name abstractions in this way is motivated by the fact that, in all our intended uses of name abstractions, the body of the abstraction is a value (with empty support).

[3] Our notation here is inspired by, but not to be confused with, Harper and Lillibridge's notation for *labels* and *variables* in a module calculus [16]. They use labels to distinguish external names of module components from internal α-variable names. In our recursive construct, both X and x are bound inside e.

Term well-formedness: $\Gamma \vdash e : \tau\ [\mathrm{S}]$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau\ [\mathrm{S}]}\ (1) \qquad \frac{}{\Gamma \vdash \langle\rangle : 1\ [\mathrm{S}]}\ (2)$$

$$\frac{\Gamma \vdash e_1 : \tau_1\ [\mathrm{S}] \quad \Gamma \vdash e_2 : \tau_2\ [\mathrm{S}]}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2\ [\mathrm{S}]}\ (3) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2\ [\mathrm{S}]}{\Gamma \vdash \pi_i(e) : \tau_i\ [\mathrm{S}]}\ (4)$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau\ [\mathrm{S} \cup \mathrm{T}]}{\Gamma \vdash \lambda x.\, e : \sigma \xrightarrow{\mathrm{T}} \tau\ [\mathrm{S}]}\ (5)$$

$$\frac{\Gamma \vdash f : \sigma \xrightarrow{\mathrm{T}} \tau\ [\mathrm{S}] \quad \Gamma \vdash e : \sigma\ [\mathrm{S}] \quad \mathrm{T} \subseteq \mathrm{S}}{\Gamma \vdash f(e) : \tau\ [\mathrm{S}]}\ (6)$$

$$\frac{\Gamma, \mathrm{X} \vdash e : \tau\ [\mathrm{S}]}{\Gamma \vdash \lambda \mathrm{X}.\, e : \forall \mathrm{X}.\, \tau\ [\mathrm{S}]}\ (7) \qquad \frac{\Gamma \vdash f : \forall \mathrm{X}.\, \tau\ [\mathrm{S}]}{\Gamma \vdash f(\mathrm{T}) : \tau[\mathrm{T}/\mathrm{X}]\ [\mathrm{S}]}\ (8)$$

$$\frac{\Gamma \vdash e : \tau\ [\mathrm{S}]}{\Gamma \vdash \mathsf{box}(e) : \mathsf{box}_{\mathrm{T}}(\tau)\ [\mathrm{S}]}\ (9) \qquad \frac{\Gamma \vdash e : \mathsf{box}_{\mathrm{T}}(\tau)\ [\mathrm{S}] \quad \mathrm{T} \subseteq \mathrm{S}}{\Gamma \vdash \mathsf{unbox}(e) : \tau\ [\mathrm{S}]}\ (10)$$

$$\frac{\Gamma, \mathrm{X}, x : \mathsf{box}_{\mathrm{X}}(\tau) \vdash e : \sigma\ [\mathrm{S}] \quad \Gamma, \mathrm{X} \vdash \sigma \equiv_{\mathrm{X}} \tau}{\Gamma \vdash \mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e) : \tau\ [\mathrm{S}]}\ (11)$$

$$\frac{\Gamma \vdash e : \sigma\ [\mathrm{S}] \quad \Gamma \vdash \sigma \equiv_{\mathrm{S}} \tau}{\Gamma \vdash e : \tau\ [\mathrm{S}]}\ (12)$$

Type equivalence: $\Gamma \vdash \tau_1 \equiv_{\mathrm{S}} \tau_2$

$$\frac{}{\Gamma \vdash 1 \equiv_{\mathrm{S}} 1}\ (13) \qquad \frac{\Gamma \vdash \sigma_1 \equiv_{\mathrm{S}} \sigma_2 \quad \Gamma \vdash \tau_1 \equiv_{\mathrm{S}} \tau_2}{\Gamma \vdash \sigma_1 \times \tau_1 \equiv_{\mathrm{S}} \sigma_2 \times \tau_2}\ (14)$$

$$\frac{\mathrm{S} \cup \mathrm{S}_1 = \mathrm{T} = \mathrm{S} \cup \mathrm{S}_2 \quad \Gamma \vdash \sigma_1 \equiv_{\mathrm{T}} \sigma_2 \quad \Gamma \vdash \tau_1 \equiv_{\mathrm{T}} \tau_2}{\Gamma \vdash \sigma_1 \xrightarrow{\mathrm{S}_1} \tau_1 \equiv_{\mathrm{S}} \sigma_2 \xrightarrow{\mathrm{S}_2} \tau_2}\ (15)$$

$$\frac{\Gamma, \mathrm{X} \vdash \tau_1 \equiv_{\mathrm{S}} \tau_2}{\Gamma \vdash \forall \mathrm{X}.\, \tau_1 \equiv_{\mathrm{S}} \forall \mathrm{X}.\, \tau_2}\ (16)$$

$$\frac{\mathrm{S} \cup \mathrm{S}_1 = \mathrm{T} = \mathrm{S} \cup \mathrm{S}_2 \quad \Gamma \vdash \tau_1 \equiv_{\mathrm{T}} \tau_2}{\Gamma \vdash \mathsf{box}_{\mathrm{S}_1}(\tau_1) \equiv_{\mathrm{S}} \mathsf{box}_{\mathrm{S}_2}(\tau_2)}\ (17)$$

Figure 4. Core Language Static Semantics

Instantiating a name abstraction f of type $\forall \mathrm{X}.\, \tau$ with a support T has the type resulting from substituting T for X in τ (Rule 8). The substitution $\tau[\mathrm{T}/\mathrm{X}]$ is defined by replacing every support S appearing in τ with $\mathrm{S}[\mathrm{T}/\mathrm{X}]$, which is in turn defined as follows:

$$\mathrm{S}[\mathrm{T}/\mathrm{X}] \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathrm{S} \cup \mathrm{T} - \{\mathrm{X}\} & \text{if } \mathrm{X} \in \mathrm{S} \\ \mathrm{S} & \text{if } \mathrm{X} \notin \mathrm{S} \end{cases}$$

Since boxing an expression first evaluates it, $\mathsf{box}(e)$ has the same support as e (Rule 9). Furthermore, $\mathsf{box}(e)$ may be given a location type $\mathsf{box}_{\mathrm{T}}(\tau)$ with arbitrary T since the resulting location contains a defined value and may be unboxed immediately. Unboxing an expression e of type $\mathsf{box}_{\mathrm{T}}(\tau)$ is only permitted if the recursive variables associated with the names in T have been defined, *i.e.,* if T is contained in the support S (Rule 10).

Rules 11 and 12 are the most interesting rules in the type system since they both make use of our type equivalence judgment, defined in Figure 4. The judgment $\Gamma \vdash \tau_1 \equiv_{\mathrm{S}} \tau_2$ means that τ_1 and τ_2 are equivalent types *modulo* the names in support S, *i.e.,* that τ_1 are τ_2 are identical types if we ignore all occurrences of the names in S. For example, the types $\tau_1 \xrightarrow{\emptyset} \tau_2$ and $\tau_1 \xrightarrow{\mathrm{X}} \tau_2$ are equivalent *modulo* any support containing X.

The intuition behind our type equivalence judgment is that, once a recursive variable has been backpatched, its associated name can be completely ignored for typechecking purposes because we only care about tracking uses of *undefined* recursive variables. If the support of e is S, then by the time e is evaluated, the recursive variables associated with the names in S will have been defined. Thus, in the context of typing e under support S, the types $\tau_1 \xrightarrow{\emptyset} \tau_2$ and $\tau_1 \xrightarrow{\mathrm{S}} \tau_2$ are as good as equivalent since they only differ with respect to the names in S, which are irrelevant. (Note: when checking equivalence of arrow types $\sigma_1 \xrightarrow{\mathrm{S}_1} \tau_1$ and $\sigma_2 \xrightarrow{\mathrm{S}_2} \tau_2$ modulo S, we compare the argument types and result types at the extended modulus $\mathrm{S} \cup \mathrm{S}_1 = \mathrm{S} \cup \mathrm{S}_2$. This makes sense because a function of one of these types may only be applied with $\mathrm{S} \cup \mathrm{S}_1$ in the support. The rule for box can be justified similarly.)

This notion of equivalence modulo a support is critical to the typing of recursive terms (Rule 11). Recall the factorial example from Section 2.2, adapted to our present type system:

```
rec(F ▷ f : int → int.
      fn x => ... x * unbox(f)(x-1) ...)
```

The issue here is that the declared type of `F` does not match the actual type of the body, `int` $\xrightarrow{\mathtt{F}}$ `int`. Once `F` is backpatched, however, the two types do match *modulo* `F`.

Correspondingly, our typing rule for recursive terms $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, e)$ works as follows. First, the context Γ is extended with the name X, as well as the recursive variable x of type $\mathsf{box}_{\mathrm{X}}(\tau)$. This location type binds the name and the variable together because it says that X must be in the support of any expression that attempts to dereference (unbox) x. The rule then checks that e has some type σ in this extended context, under a support S that does *not* include X (since x is undefined while evaluating e). Finally, it checks that σ and τ are equivalent *modulo* X. It is easiest to understand this last step as a generalization of our earlier idea of ignoring discrepancies between partial and total arrows when comparing σ and τ. The difference here is that we ignore discrepancies with respect to a particular name X instead of all names, so that the rule behaves properly in the presence of multiple names (nested recursion).

In contrast, Rule 12 appears rather straightforward, allowing a term with type σ and support S to be assigned a type that is equivalent to σ modulo the names in S. In fact, this rule solves the higher-order function problem described in Section 2.2! Recall that we wanted to apply an existing higher-order ML function like `map` to a partial function, *i.e.,* one whose arrow type bears non-empty support:

```
rec (X ▷ x : SIG.
  let
     val f : σ -X-> τ = ...
     val g : σ list -X-> τ list = fn xs => map f xs
  ...
)
```

The problem here is that the type of `f` does not match the argument type $\sigma \to \tau$ of `map`. Intuitively, though, this code ought to typecheck: if we are willing to add `X` to the support of `g`'s arrow type, then `x` must be backpatched before `g` is ever applied, so `X` should be ignored when typing the body of `g`.

Rule 12 encapsulates this reasoning. Since `g` is specified with type σ `list` $\xrightarrow{X}$ τ `list`, we can assume support `X` when typechecking its body (`map f xs`). Under support `X`, Rule 12 allows us to assign `f` the type $\sigma \to \tau$, as it is equivalent to `f`'s type *modulo* `X`. Thus, `g`'s body is well-typed under support `X`.

3.3 Name Abstractions and Non-strictness

We have so far illustrated how the inclusion of supports in our typing and equivalence judgments addresses the first two problems described in Section 2.3. Our system addresses the problem of separate compilation as well by (1) making the dereferencing of a recursive variable an explicit operation, and (2) providing the ability to abstract an expression over a name.

Recall the separate compilation scenario from Figure 2. Since recursive variables in our core language are no longer dereferenced implicitly, we might attempt to rewrite the linking module as:

```
structure rec X ▷ x : SIG =
  structure A = myA(x)
  structure B = yourB(x)
end
```

The recursive variable `x` is now a value of location type $\mathsf{box}_{\mathtt{X}}(\mathtt{SIG})$, so passing it as an argument to `myA` and `yourB` does not dereference it. Assuming then that `myA` and `yourB` are *non-strict*, *i.e.,* that they do not dereference their argument when applied, the recursion is indeed well-founded.

But what types can we give to `myA` and `yourB` to reflect the property that they are non-strict? Suppose that `myA`'s return type is `SIG_A`. We would like to give it the type $\mathsf{box}_{\mathtt{X}}(\mathtt{SIG}) \to \mathtt{SIG\_A}$, so that (1) its argument type matches the type of `x`, and (2) the absence of `X` on the arrow indicates that `myA` can be applied under empty support. However, this type makes no sense where `myA` is defined, because the name `X` is not in scope outside of the recursive module.

This is where name abstractions come in. To show that `myA` is non-strict, it is irrelevant what particular support is required to unbox its argument, so we can use a name abstraction to allow any name or support to be substituted for `X`. Figure 5 shows the resulting well-typed separate compilation scenario, in which the type of `myA` is $\forall \mathtt{X}.\, \mathsf{box}_{\mathtt{X}}(\mathtt{SIG}) \to \mathtt{SIG\_A}$.

Our recursive construct is still not quite as flexible for separate compilation purposes as one might like. In particular, suppose that we wanted to parameterize `myA` over *just* `yourB` instead of *both* `myA` and `yourB`. There is no way in our system to extract a value of type $\mathsf{box}_{\mathtt{X}}(\mathtt{SIG\_B})$ from `x` without unboxing it. It is easy to remedy this problem, however, by generalizing the recursive construct to an n-ary one, $\mathsf{rec}(\vec{X} \triangleright \vec{x} : \vec{\tau}.\, \vec{e})$, where each of the n recursive variables x_i is boxed separately with type $\mathsf{box}_{X_i}(\tau_i)$.

Name abstractions can also be used to express non-strictness of *general-purpose* ML functors, which in turn allows better static detection of well-founded recursion in certain cases. For instance, the recursive module in Figure 6 provides a type `C.t`, which is defined in terms of *sets* of itself.[4] The definition of module `C` refers recursively to the `CSet` module, which is defined by applying the `MakeSet` functor to the `C` module. The only way we can be sure that the recursion is well-founded is if we know that the application of the `MakeSet` functor will not attempt to apply the partial function `C.compare`, *i.e.,* that the `MakeSet` functor is non-strict. With name abstractions, we can instrument the implementation of `MakeSet` in order to assign it a non-strict type[5] such as $\forall \mathtt{X}.\, \mathsf{box}_{\mathtt{X}}(\mathtt{ORDERED}) \to \mathtt{SET}$.

```
myA = λX. λx : box_X(SIG). ...

yourB = λX. λx : box_X(SIG). ...

structure rec X ▷ x : SIG = struct
  structure A = myA{X}(x)
  structure B = yourB{X}(x)
end
```

Figure 5. Revised Separate Compilation Scenario

```
structure rec C : ORDERED = struct
  datatype t = ...CSet.set...
  fun compare (x,y) = ...CSet.compare(a,b)...
end
and CSet = MakeSet(C)
```

Figure 6. Recursive Data Structure Example

Similarly, name abstractions can be used to give more precise types to core-level ML functions. For instance, suppose we had access to the code for the `map` function. By wrapping the definition of `map` in a name abstraction, we could assign the function the type

$$\forall \mathtt{X}.\ (\sigma \xrightarrow{\mathtt{X}} \tau) \xrightarrow{\emptyset} (\sigma\ \mathtt{list} \xrightarrow{\mathtt{X}} \tau\ \mathtt{list})$$

This type indicates that `map` will turn a value of any arrow type into a value of the same arrow type, but will not apply its argument in the process. Given this type for `map`, we can write our recursive module example involving `map` the way we wanted to write it originally in Section 2.3:

```
rec (X ▷ x : SIG.
  let
     val f : σ -X-> τ = ...
     val g : σ list -X-> τ list = [map {X} f]
  ...
)
```

The more precise non-strict type for `map` allows us to avoid eta-expanding `map f`, but it also requires having access to the implementation of `map`. Furthermore, it requires us to modify the type of `map`, infecting the existing ML infrastructure with names. It is therefore important that, in the absence of this solution, our type system is strong enough (thanks to Rule 12) to typecheck at least the eta-expansion of `map f`, without requiring changes to existing ML code. Unfortunately, there is no corresponding way to eta-expand the functor application `MakeSet(C)` in the example from Figure 6. To statically ensure that the recursion in that example is

[4] See Okasaki [26] for similar, more realistic examples, such as "bootstrapped heaps".

[5] For simplicity, we are ignoring here that the result signature `SET` really depends on the type components of the functor argument.

well-founded, it appears that one *must* have access to the implementation of the `MakeSet` functor in order to instrument it with name abstractions and assign it a more precise non-strict interface.

This example also illustrates why it is useful to be able to instantiate a name abstraction with a *support* instead of a single name. In particular, suppose that `f`'s type were $\sigma \xrightarrow{S} \tau$ for some non-singleton support S. The definition of `g` would become `map S f`, which is only possible given the ability to instantiate `map` with a support.

Finally, note that while our system does not contain any notion of subtyping, it is important to be able to coerce a non-strict function into an ordinary (potentially strict) arrow type. The coercion from $\forall X.\, \mathsf{box}_X(\sigma) \to \tau$ to $\sigma \to \tau[\emptyset/X]$ is easily encodable within our language as $\lambda f.\lambda x. f(\emptyset)(\mathsf{box}(x))$.

3.4 Dynamic Semantics

We formalize the dynamic semantics of our core language in terms of a virtual machine. Machine states $(\omega; C; e)$ consist of a store ω, a continuation C, and an expression e currently being evaluated. We sometimes use Ω to stand for a machine state.

A continuation C consists of a stack of continuation frames F, as shown in Figure 7. A store ω is a partial mapping from variables (of location type) to *storable things*. A storable thing θ is either a term (e) or nonsense (**?**). By $\omega(x)$ we denote the storable thing stored at x in ω, which is only valid if something (possibly nonsense) is stored at x. By $\omega[x \mapsto \theta]$ we denote the result of creating a new location x in ω and storing θ at it. By $\omega[x := \theta]$ we denote the result of updating the store ω to store θ at x, where x is already in $\mathrm{dom}(\omega)$. We denote the empty store by ε.

The dynamic semantics of the language is shown in Figure 7 as well. It takes the form of a stepping relation $\Omega \mapsto \Omega'$. Rules 18 through 31 are all fairly straightforward. Rule 32 says that, in order to evaluate $\mathsf{rec}(X \triangleright x : \tau.\, e)$, we create a new location x in the store bound to nonsense, push the recursive frame $\mathsf{rec}(X \triangleright x : \tau.\, \bullet)$ on the continuation stack, and evaluate e. (We can always ensure that x is not already a location in the store by α-conversion.) Once we have evaluated e to a value v, Rule 33 performs the backpatching step: it stores v at location x in the store and returns v. Finally, if a location is ever dereferenced (unboxed), Rule 31 simply looks up the value it is bound to in the store.

3.5 Type Safety

Observe that the machine is stuck if we attempt to unbox a location that is bound to nonsense. The point of the type safety theorem is to ensure that this will never happen for well-formed programs. We begin by defining a notion of well-formedness for stores, which is dependent on a notion of a *run-time context*. A run-time context is a context that only binds variables representing memory locations, *i.e.,* variables of box type. In addition, we distinguish *backpatchable* locations and connect them to their associated names by introducing a new context binding form $X \triangleright x : \tau$, which behaves semantically the same as the two bindings $X, x : \mathsf{box}_X(\tau)$, but is distinguished syntactically.

DEFINITION 3.1 (RUN-TIME CONTEXTS). A context Γ is *run-time* if the only bindings in Γ take the form $X \triangleright x : \tau$ or $x : \mathsf{box}_T(\tau)$.

DEFINITION 3.2 (STORE WELL-FORMEDNESS). A store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [S]$, if:

1. Γ is run-time and $\mathrm{dom}(\omega) = \mathrm{vardom}(\Gamma)$
2. $\forall X \triangleright x : \tau \in \Gamma.$ if $X \in S$ then $\exists v.\ \omega(x) = v$ and $\Gamma \vdash v : \tau\ [S]$
3. $\forall x : \mathsf{box}_T(\tau) \in \Gamma.\ \exists v.\ \omega(x) = v$ and $\Gamma \vdash v : \tau\ [S]$

Continuations	$C ::=$	$\bullet \mid C \circ F$
Continuation Frames	$F ::=$	$\langle \bullet, e \rangle \mid \langle v, \bullet \rangle \mid \pi_i(\bullet) \mid \bullet(e) \mid v(\bullet)$
		$\mid \bullet(T) \mid \mathsf{box}(\bullet) \mid \mathsf{unbox}(\bullet)$
		$\mid \mathsf{rec}(X \triangleright x : \tau.\, \bullet)$

Small-step semantics: $\Omega \mapsto \Omega'$

$$\frac{\langle e_1, e_2 \rangle \text{ not a value}}{(\omega; C; \langle e_1, e_2 \rangle) \mapsto (\omega; C \circ \langle \bullet, e_2 \rangle; e_1)} \quad (18)$$

$$\frac{}{(\omega; C \circ \langle \bullet, e \rangle; v) \mapsto (\omega; C \circ \langle v, \bullet \rangle; e)} \quad (19)$$

$$\frac{}{(\omega; C \circ \langle v_1, \bullet \rangle; v_2) \mapsto (\omega; C; \langle v_1, v_2 \rangle)} \quad (20)$$

$$\frac{}{(\omega; C; \pi_i(e)) \mapsto (\omega; C \circ \pi_i(\bullet); e)} \quad (21)$$

$$\frac{}{(\omega; C \circ \pi_i(\bullet); \langle v_1, v_2 \rangle) \mapsto (\omega; C; v_i)} \quad (22)$$

$$\frac{}{(\omega; C; e_1(e_2)) \mapsto (\omega; C \circ \bullet(e_2); e_1)} \quad (23)$$

$$\frac{}{(\omega; C \circ \bullet(e); v) \mapsto (\omega; C \circ v(\bullet); e)} \quad (24)$$

$$\frac{}{(\omega; C \circ (\lambda x.\, e)(\bullet); v) \mapsto (\omega; C; e[v/x])} \quad (25)$$

$$\frac{}{(\omega; C; e(T)) \mapsto (\omega; C \circ \bullet(T); e)} \quad (26)$$

$$\frac{}{(\omega; C \circ \bullet(T); \lambda X.\, e) \mapsto (\omega; C; e[T/X])} \quad (27)$$

$$\frac{}{(\omega; C; \mathsf{box}(e)) \mapsto (\omega; C \circ \mathsf{box}(\bullet); e)} \quad (28)$$

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; C \circ \mathsf{box}(\bullet); v) \mapsto (\omega[x \mapsto v]; C; x)} \quad (29)$$

$$\frac{}{(\omega; C; \mathsf{unbox}(e)) \mapsto (\omega; C \circ \mathsf{unbox}(\bullet); e)} \quad (30)$$

$$\frac{\omega(x) = v}{(\omega; C \circ \mathsf{unbox}(\bullet); x) \mapsto (\omega; C; v)} \quad (31)$$

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; C; \mathsf{rec}(X \triangleright x : \tau.\, e)) \mapsto (\omega[x \mapsto \mathbf{?}]; C \circ \mathsf{rec}(X \triangleright x : \tau.\, \bullet); e)} \quad (32)$$

$$\frac{}{(\omega; C \circ \mathsf{rec}(X \triangleright x : \tau.\, \bullet); v) \mapsto (\omega[x := v]; C; v)} \quad (33)$$

Figure 7. Core Language Dynamic Semantics

$$\boxed{\textbf{Continuation well-formedness: } \Gamma \vdash C : \tau \text{ cont } [S]}$$

$$\frac{}{\Gamma \vdash \bullet : \tau \text{ cont } [S]} \quad (34)$$

$$\frac{\Gamma \vdash F : \tau \Rightarrow \sigma \ [S] \quad \Gamma \vdash C : \sigma \text{ cont } [S]}{\Gamma \vdash C \circ F : \tau \text{ cont } [S]} \quad (35)$$

$$\frac{\Gamma \vdash C : \sigma \text{ cont } [S] \quad \Gamma \vdash \sigma \equiv_S \tau}{\Gamma \vdash C : \tau \text{ cont } [S]} \quad (36)$$

$$\boxed{\textbf{Continuation frame well-formedness: } \Gamma \vdash F : \tau_1 \Rightarrow \tau_2 \ [S]}$$

$$\frac{\Gamma \vdash e : \tau_2 \ [S]}{\Gamma \vdash \langle \bullet, e \rangle : \tau_1 \Rightarrow \tau_1 \times \tau_2 \ [S]} \quad (37)$$

$$\frac{\Gamma \vdash v : \tau_1 \ [S]}{\Gamma \vdash \langle v, \bullet \rangle : \tau_2 \Rightarrow \tau_1 \times \tau_2 \ [S]} \quad (38)$$

$$\frac{i \in \{1,2\}}{\Gamma \vdash \pi_i(\bullet) : \tau_1 \times \tau_2 \Rightarrow \tau_i \ [S]} \quad (39)$$

$$\frac{\Gamma \vdash e : \sigma \ [S]}{\Gamma \vdash \bullet(e) : \sigma \rightarrow \tau \Rightarrow \tau \ [S]} \quad (40) \qquad \frac{\Gamma \vdash v : \sigma \rightarrow \tau \ [S]}{\Gamma \vdash v(\bullet) : \sigma \Rightarrow \tau \ [S]} \quad (41)$$

$$\frac{}{\Gamma \vdash \bullet(\mathrm{T}) : \forall \mathrm{X}.\, \tau \Rightarrow \tau[\mathrm{T}/\mathrm{X}] \ [S]} \quad (42)$$

$$\frac{}{\Gamma \vdash \mathsf{box}(\bullet) : \tau \Rightarrow \mathsf{box}_{\mathrm{T}}(\tau) \ [S]} \quad (43)$$

$$\frac{}{\Gamma \vdash \mathsf{unbox}(\bullet) : \mathsf{box}(\tau) \Rightarrow \tau \ [S]} \quad (44)$$

$$\frac{\mathrm{X} \triangleright x : \tau \in \Gamma \quad \Gamma \vdash \sigma \equiv_{\mathrm{X}} \tau}{\Gamma \vdash \mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, \bullet) : \sigma \Rightarrow \tau \ [S]} \quad (45)$$

Figure 8. Well-formedness of Core Continuations

Essentially, the judgment $\Gamma \vdash \omega$ [S] says that Γ assigns types to all locations in the domain of ω, and that all locations map to appropriately-typed values except those backpatchable ones associated with names which are not in the support S.

We define well-formedness of continuations and continuation frames via the judgments $\Gamma \vdash C : \tau \text{ cont } [S]$ and $\Gamma \vdash F : \tau_1 \Rightarrow \tau_2$ [S], defined in Figure 8. The former judgment says that continuation C expects a value of type τ to fill in its $\bullet$; the latter judgment says that F expects a value of type τ_1 to fill in its $\bullet$ and that F produces a value of type τ_2 in return. The only rule that is slightly unusual is Rule 45 for recursive frames $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, \bullet)$. Since this frame is not a binder for X or x, Rule 45 requires that $\mathrm{X} \triangleright x : \tau$ be in the context. This is a safe assumption since $\mathsf{rec}(\mathrm{X} \triangleright x : \tau.\, \bullet)$ only gets pushed on the stack after a binding for x has been added to the store.

We can now define a notion of well-formedness for a machine state, which requires that the type of its expression component matches the type of the hole in the continuation component:

DEFINITION 3.3 (MACHINE STATE WELL-FORMEDNESS). A machine state Ω is *well-formed*, denoted $\Gamma \vdash \Omega$ [S], if $\Omega = (\omega; C; e)$, where:

1. $\Gamma \vdash \omega$ [S]
2. $\exists \tau.\ \Gamma \vdash C : \tau \text{ cont } [S]$ and $\Gamma \vdash e : \tau$ [S]

We can now state the preservation and progress theorems leading to type safety:

THEOREM 3.4 (PRESERVATION). If $\Gamma \vdash \Omega$ [S] and $\Omega \mapsto \Omega'$, then $\exists \Gamma', S'.\ \Gamma' \vdash \Omega'$ [S'].

DEFINITION 3.5 (TERMINAL STATES). A machine state Ω is *terminal* if it has the form $(\omega; \bullet; v)$.

DEFINITION 3.6 (STUCK STATES). A machine state Ω is *stuck* if it is non-terminal and there is no state Ω' such that $\Omega \mapsto \Omega'$.

THEOREM 3.7 (PROGRESS). If $\Gamma \vdash \Omega$ [S], then Ω is not stuck.

Note that when $\Omega = (\omega; C \circ \mathsf{unbox}(\bullet); x)$, the well-formedness of Ω implies that $\Gamma \vdash x : \mathsf{box}(\tau)$ [S] for some type τ. The well-formedness of ω then ensures that there is a value v such that $\omega(x) = v$, so Ω can make progress by Rule 31.

COROLLARY 3.8 (TYPE SAFETY). Suppose $\emptyset \vdash e : \tau$ $[\emptyset]$. Then for any machine state Ω, if $(\varepsilon; \bullet; e) \mapsto^* \Omega$, then Ω is not stuck.

The full meta-theory of our language (along with proofs) appears in the companion technical report [7].

3.6 Practical Issues

Efficient Implementation Our dynamic semantics treats values of type $\mathsf{box}_S(\tau)$ as memory locations that will eventually contain values of type τ. It is quite likely, though, that values of type τ (*i.e.,* the kinds of values one wants to define recursively) have a naturally boxed representation. For instance, in the case of recursive modules, τ will typically be a record type, and a module value of type τ will be represented as a pointer to a record stored on the heap.

Only one level of pointer indirection is needed to implement backpatching. Thus, a direct implementation of our semantics that represents all values of type $\mathsf{box}_S(\tau)$ as pointers to values of type τ will introduce an unnecessary level of indirection when values of type τ are already pointers. Our semantics, however, does not require one to employ such a naïve representation. Indeed, for types τ with naturally boxed representations, a realistic implementation of our semantics should represent values of type $\mathsf{box}_S(\tau)$ the same as values of type τ and should compile the unbox'ing of such values as a no-op. (See Hirschowitz *et al.* [18] for an example of such a compilation strategy.) At the level of our type system, though, there is still an important semantic distinction to be made between τ and $\mathsf{box}_S(\tau)$ that transcends such implementation details.

Effects Since we have modeled the semantics of backpatching operationally in terms of a mutable store, it is easy to incorporate some actual computational effects into our framework as well. In the companion technical report [7] we extend the language and type safety proof with primitives for mutable state and continuations. The extensions are completely straightforward and are essentially oblivious to the presence of supports in typing judgments.

Types $\tau ::= \cdots \mid \mathsf{comp}_{\mathrm{S}}(\tau)$
Terms $e ::= \cdots \mid \mathsf{delay}(e) \mid \mathsf{force}(e)$

$$\frac{\mathrm{S}\cup\mathrm{S}_1 = \mathrm{T} = \mathrm{S}\cup\mathrm{S}_2 \quad \Gamma\vdash\tau_1 \equiv_{\mathrm{T}} \tau_2}{\Gamma\vdash \mathsf{comp}_{\mathrm{S}_1}(\tau_1) \equiv_{\mathrm{S}} \mathsf{comp}_{\mathrm{S}_2}(\tau_2)} \quad (46)$$

$$\frac{\Gamma\vdash e:\tau\ [\mathrm{S}\cup\mathrm{T}]}{\Gamma\vdash \mathsf{delay}(e):\mathsf{comp}_{\mathrm{T}}(\tau)\ [\mathrm{S}]} \quad (47)$$

$$\frac{\Gamma\vdash e:\mathsf{comp}_{\mathrm{T}}(\tau)\ [\mathrm{S}] \quad \mathrm{T}\subseteq\mathrm{S}}{\Gamma\vdash \mathsf{force}(e):\tau\ [\mathrm{S}]} \quad (48)$$

Figure 9. Static Semantics for Memoized Computations

Typechecking It is also important that our language admit a practical typechecking algorithm. In the implicitly-typed form of the language that we have used here, it is not obvious that such an algorithm exists because terms do not have unique types. For example, if $\lambda x.\,e$ has type $\sigma \xrightarrow{\mathrm{S}} \tau$, it can also be given $\sigma \xrightarrow{\mathrm{T}} \tau$ for any $\mathrm{T}\supseteq\mathrm{S}$. It is easy to eliminate this non-determinism, however, by making the language explicitly-typed. In particular, if λ-abstractions are annotated as $\lambda^{\mathrm{T}} x:\sigma.e$ and boxed expressions are annotated as $\mathsf{box}_{\mathrm{T}}(e)$, along with the revised typing rules

$$\frac{\Gamma, x:\sigma\vdash e:\tau\ [\mathrm{S}\cup\mathrm{T}]}{\Gamma\vdash \lambda^{\mathrm{T}} x:\sigma.e:\sigma\xrightarrow{\mathrm{T}}\tau\ [\mathrm{S}]} \qquad \frac{\Gamma\vdash e:\tau\ [\mathrm{S}]}{\Gamma\vdash \mathsf{box}_{\mathrm{T}}(e):\mathsf{box}_{\mathrm{T}}(\tau)\ [\mathrm{S}]}$$

then it is easy to synthesize unique types for explicitly-typed terms up to equivalence modulo a given support S. (See the companion technical report for details [7].) It remains an important question for future work how much of the type and support information in explicitly-typed terms can be inferred.

4 Encoding Unrestricted Recursion

Despite all the efforts of our type system, there will always be recursive terms $\mathsf{rec}(\mathrm{X}\triangleright x:\tau.\,e)$ for which we cannot statically determine that e can be evaluated without dereferencing x. For such cases it is important to have a fallback approach that would allow the programmer to write $\mathsf{rec}(\mathrm{X}\triangleright x:\tau.\,e)$ with the understanding that the recursion may be ill-founded and dereferences of x will be saddled with an additional run-time cost.

One option is to add a second *unrestricted* recursive term construct, with the following typing rule:

$$\frac{\Gamma, x:1\rightarrow\tau\vdash e:\tau\ [\mathrm{S}]}{\Gamma\vdash \mathsf{urec}(x:\tau.\,e):\tau\ [\mathrm{S}]}$$

Note that we do not introduce any name X, so there are no restrictions on when x can be dereferenced. Since the dereferencing of x may diverge and cannot therefore be a mere pointer dereference, we assign x the thunk type $1\rightarrow\tau$ instead of $\mathsf{box}(\tau)$, with dereferencing achieved by applying x to $\langle\rangle$. Adding an explicit urec construct, however, makes for some redundancy in the recursive mechanisms of the language. It would be preferable, at least at the level of the theory, to find a way to encode unrestricted recursion in terms of our existing recursive construct.

We achieve this by extending the language with primitives for memoized computations. The syntax and static semantics of this extension are given in Figure 9. First, we introduce a type $\mathsf{comp}_{\mathrm{S}}(\tau)$ of locations storing memoized computations. A value of this type is essentially a thunk of type $1\xrightarrow{\mathrm{S}}\tau$ whose result is memoized after the first application.

Machine States $\Omega ::= \cdots \mid \mathsf{Error}$
Continuation Frames $F ::= \cdots \mid \mathsf{force}(\bullet) \mid \mathsf{memo}(x,\bullet)$

$$\frac{x\notin\mathrm{dom}(\omega)}{(\omega;C;\mathsf{delay}(e))\mapsto(\omega[x\mapsto e];C;x)} \quad (49)$$

$$\frac{}{(\omega;C;\mathsf{force}(e))\mapsto(\omega;C\circ\mathsf{force}(\bullet);e)} \quad (50)$$

$$\frac{\omega(x)=e}{(\omega;C\circ\mathsf{force}(\bullet);x)\mapsto(\omega[x:=\mathbf{?}];C\circ\mathsf{memo}(x,\bullet);e)} \quad (51)$$

$$\frac{}{(\omega;C\circ\mathsf{memo}(x,\bullet);v)\mapsto(\omega[x:=v];C;v)} \quad (52)$$

$$\frac{\omega(x)=\mathbf{?}}{(\omega;C\circ\mathsf{force}(\bullet);x)\mapsto\mathsf{Error}} \quad (53)$$

Figure 10. Dynamic Semantics for Memoized Computations

The primitive $\mathsf{delay}(e)$ creates a memoized location x in the store bound to the unevaluated expression e. When x is forced (by $\mathsf{force}(x)$), the expression e stored at x is evaluated to a value v, and then v is written back to x. During the evaluation of e, the location x is bound to nonsense; if x is forced again during this stage, the machine raises an error. Thus, every force of x must check to see whether it is bound to an expression or nonsense. Despite the difference in operational behavior, the typing rules for memoized computations appear just as if $\mathsf{comp}_{\mathrm{S}}(\tau)$, $\mathsf{delay}(e)$ and $\mathsf{force}(e)$ were shorthand for $1\xrightarrow{\mathrm{S}}\tau$, $\lambda\langle\rangle.\,e$ and $e\,\langle\rangle$, respectively. We use $\mathsf{comp}(\tau)$ sometimes as shorthand for $\mathsf{comp}_{\emptyset}(\tau)$.

We can now encode urec via a recursive memoized computation:

$$\mathsf{urec}(x:\tau.\,e) \stackrel{\text{def}}{=}$$
$$\mathsf{force}(\mathsf{rec}(\mathrm{X}\triangleright x:\mathsf{comp}(\tau).\,\mathsf{delay}(e[\lambda\langle\rangle.\,\mathsf{force}(\mathsf{unbox}(x))/x])))$$

It is easiest to understand this encoding by stepping through it. First, a new *recursive* location x is created, bound to nonsense. Then, the delay creates a new *memoized* location y bound to the expression $e[\ldots/x]$. Next, the rec backpatches x with the value y and returns y. Finally, y is forced, resulting in the evaluation of $e[\ldots/x]$ to a value v, and y is backpatched with v. If the recursive variable (encoded as $\lambda\langle\rangle.\,\mathsf{force}(\mathsf{unbox}(x))$) is dereferenced (applied to $\langle\rangle$) during the evaluation of $e[\ldots/x]$, it will result in another forcing of y, raising a run-time error.

Essentially, one can view the rec in this encoding as merely tying the recursive knot on the memoized computation, while the memoization resulting from the force is what actually performs the backpatching. Observe that if we were to give $\mathsf{comp}(\tau)$ a non-memoizing semantics, *i.e.,* to consider it synonymous with $1\rightarrow\tau$, the above encoding would have precisely the fixed-point semantics of recursion. Memoization ensures that the effects in e only happen once, at the first force of the recursive computation.

The dynamic semantics for this extension is given in Figure 10. To evaluate $\mathsf{delay}(e)$, we create a new memoized location in the store and bind e to it (Rule 49). To evaluate $\mathsf{force}(e)$, we first evaluate e (Rule 50). Once e evaluates to a location x, we look x up in the store. If x is bound to an expression e, we proceed to evaluate e, but first push on the continuation stack a memoization frame to remind us that the result of evaluating e should be memoized at x (Rules 51 and 52). If x is instead bound to nonsense, then we must be in the middle of evaluating another $\mathsf{force}(x)$, so we step to an Error state which halts the program (Rule 53). Extending the type safety proof of Section 3.5 to handle memoized computations is straightforward; the details appear in the companion technical report [7].

5 Related Work

Well-Founded Recursion Boudol [4] proposes a type system for well-founded recursion that, like ours, employs a backpatching semantics. Boudol's system tracks the *degrees* to which expressions depend on their free variables, where the degree to which e depends on x is 1 if x appears in a guarded position in e (*i.e.,* under an unapplied λ-abstraction), and 0 otherwise. What we call the support of an expression corresponds in Boudol's system to the set of variables on which the expression depends with degree 0. Thus, while there is no distinction between recursive and ordinary variables in Boudol's system, his equivalent of $\mathsf{rec}(x{:}\tau.\,e)$ ensures that the evaluation of e will not dereference x by requiring that e depend on x with degree 1.

In our system an arrow type indicates the recursive variables that may be dereferenced when a function of that type is applied. An arrow type in Boudol's system indicates the degree to which the body of a function depends on its argument. Thus, $\sigma \xrightarrow{0} \tau$ and $\sigma \xrightarrow{1} \tau$ classify functions that are *strict* and *non-strict* in their arguments, respectively. As we discussed in Section 3.3, the ability to identify non-strict functions is especially important for purposes of separate compilation. For example, in order to typecheck our separate compilation scenario from Figure 2, it is necessary to know that the separately-compiled functors `myA` and `yourB` are non-strict.

In contrast to our system, which requires the code from Figure 2 to be rewritten as shown in Figure 5, Boudol's system can typecheck the code in Figure 2 as is. The reason is that function applications of the form $f(x)$ (*i.e.,* where the argument is a variable) are treated as a special case in his semantics: while the expression "x" depends on the variable x with degree 0, the expression "$f(x)$" merely passes x to f without dereferencing it. This implies that ordinary λ-bound variables may be instantiated at run time with recursive variables. Thus, viewed in terms of our semantics, Boudol's system treats *all* variables as implicitly having box type.

The simplicity of Boudol's system is achieved at the expense of being rather conservative. In particular, a function application $f(e)$ is considered to depend on all the free variables of f with degree 0. Suppose that f is a curried function $\lambda y.\lambda z.e'$, where e' dereferences a recursive variable x. In Boudol's system, even a single application of f will be considered to depend on x with degree 0 and thus cannot appear unguarded in the recursive term defining x.

To address the limitations of Boudol's system, Hirschowitz and Leroy [17] propose a generalization of it, which they use as the target language for compiling their call-by-value mixin module calculus (see below). Specifically, they extend Boudol's notion of degrees to be arbitrary integers: the degree to which e depends on x becomes, roughly, the number of λ-abstractions under which x appears in e. Thus, continuing the above example, the function $\lambda y.\lambda z.e'$ would depend on x with degree 2, so instantiating the first argument would only decrement that degree to 1, not 0.

Nevertheless, Hirschowitz and Leroy's system still suffers from a paucity of types. Consider the same curried function example, except where we let-bind $\lambda y.\lambda z.e'$ first instead of applying it directly: $\mathsf{let}\ f = \lambda y.\lambda z.e'\ \mathsf{in}\ f(e)$. The most precise degree-based type one can give to f when typing the body of the let is $\tau_1 \xrightarrow{1} \tau_2 \xrightarrow{0} \tau_3$. This type tells us nothing about the degree to which f depends on the recursive variable x dereferenced by e'. Thus, Hirschowitz and Leroy's system must conservatively assume that $f(e)$ may dereference x. In contrast, our type system can assign f a type such as $\tau_1 \xrightarrow{\emptyset} \tau_2 \xrightarrow{X} \tau_3$, which would allow its first argument (but not its second) to be instantiated under the empty support.

We believe the let expression above is representative of code that one might want to write in the body of a recursive module, which suggests that our name-based approach is a more appropriate foundation for recursive modules. However, the weaknesses of the degree-based approaches are not necessarily problematic in the particular applications for which they were developed. For the purpose of compiling mixin modules, the primary feature required of Hirschowitz and Leroy's target language is the ability to link mutually recursive λ-abstractions that have been compiled separately. As we have illustrated in Section 3.3, our language supports this feature as well.

Weak Polymorphism and Effect Systems There seems to be an analogy between the approaches discussed here for tracking well-founded recursion and the work on combining polymorphism and effects in the early days of ML. Boudol's 0-1 distinction is reminiscent of Tofte's distinction between imperative and applicative type variables [30]. Hirschowitz and Leroy's generalization of Boudol is similar to the idea of *weak polymorphism* [15] (implemented by MacQueen in earlier versions of the SML/NJ compiler), wherein a type variable α carries a numeric "strength" representing, roughly, the number of function applications required before a ref cell is created storing a value of type α. Our system has ties to effect systems in the style of Talpin and Jouvelot [29], in which an arrow type indicates the set of effects that may occur when a function of that type is applied. For us, the effect in question is the dereferencing of an undefined recursive variable.

A common criticism leveled at both effect systems and weak polymorphism is that functional and imperative implementations of a polymorphic function have different types, and it is impossible to know which type to expect when designing a specification for a module separate from its implementation [31]. To a large extent, this criticism does not apply to our type system: names infect types *within* recursive modules, but the *external* interface of a module will be the same regardless of whether or not the module is implemented recursively. To ensure that certain recursive modules (like the one in Figure 6) are well-founded, however, one needs to observe that a general-purpose functor (like the `MakeSet` functor) is non-strict, and it is debatable whether the (non-)strictness of such a functor should be reflected in its specification. Choosing not to expose strictness information in the specification of a functor imposes fundamental limitations on how the functor can be used, not just in our system, but in any type system for well-founded recursion.

Strictness Analysis One can think of static detection of well-founded recursion as a kind of *non-strictness* analysis, in contrast to the well-known problem of *strictness* analysis [1]. Both problems are concerned with identifying whether an expression, such as the body of a function, will access the value of a particular variable when evaluated. Strictness analysis, however, is used as an optimization technique for lazy languages, in which any function may be conservatively classified as non-strict. In call-by-value languages, on the other hand, functions are strict by default—observing that a function is non-strict requires us to explicitly treat its argument as boxed and to show that applying the function will not unbox it. It is thus unclear how techniques from strictness analysis might be applied to the well-founded recursion problem.

Names The idea of using names in our type system is inspired by Nanevski's work on using a modal logic with names to model a "metaprogramming" language for symbolic computation [23]. (His use of names was in turn inspired by Pitts and Gabbay's FreshML [27].) Nanevski uses names to represent undefined symbols appearing inside expressions of a modal $\Box$ type. These expressions can be viewed as pieces of uncompiled syntax whose free names must be defined before they can be compiled.

Our use of names is conceptually closer to Nanevski's more recent work (concurrent with ours) on using names to model control effects for which there is a notion of handling [24]. As mentioned earlier, one can think of the dereferencing of a recursive variable as an effect that is in some sense "handled" by the backpatching of the variable. Formally, though, Nanevski's system is quite different, especially in that it does not employ any judgment of type equivalence modulo a support.

Monadic Recursion There has been considerable work recently on adding effectful recursion to Haskell. Since effects in Haskell are isolated in monadic computations, adding a form of recursion over effectful expressions requires an understanding of how recursion interacts with monads. Erkök and Launchbury [11] propose a monadic fixed-point construct mfix for defining recursive computations in monads that satisfy a certain set of axioms. They later show how to use mfix to define a recursive form of Haskell's do construct [12]. Friedman and Sabry [14] argue that the backpatching semantics of recursion is fundamentally stateful, and thus defining a recursive computation in a given monad requires the monad to be combined with a state monad. This approach allows recursion in monads that do not obey the Erkök-Launchbury axioms, such as the continuation monad.

The primary goal of our type system is to statically ensure well-founded recursion in an impure call-by-value setting, and thus the work on recursive monadic computations for Haskell (which avoids any static analysis) is largely orthogonal to ours. Nevertheless, the dynamic semantics of our language borrows from recent work by Moggi and Sabry [22], who give an operational semantics for the monadic metalanguage extended with the Friedman-Sabry mfix.

Recursive Modules Most recursive module proposals restrict the form of the recursive module construct so that recursion is not defined over effectful expressions. One exception is Russo's extension to Moscow ML [28], which employs an unrestricted form of recursion similar to our urec construct from Section 4. Another is Leroy's experimental extension to O'Caml [20], which permits arbitrary effects in recursive modules but restricts backpatching to modules of pointed type, *i.e.,* modules that export only functions and lazy computations. This restriction enables more efficient implementation, since for pointed types there is an appropriate "bottom" value with which to initialize the recursive variable. One can apply the same optimization to our $\mathsf{urec}(x : \tau. e)$ in the case that τ is pointed. Our system, however, permits examples like the one in Figure 1, which Leroy's extension does not.

Crary, Harper and Puri [5] give a foundational account of recursive modules that models recursion via a fixed-point at the module level. For the fixed-point semantics to make sense, they require that the body of a fixed-point module is *valuable* (*i.e.,* pure and terminating) in a context where the recursive variable is non-valuable. Our judgment of *evaluability* from Section 2 can be seen as a generalization of valuability. Similarly, Flatt and Felleisen's proposal for *units* [13] divides the recursive module construct into a recursive section, restricted to contain only valuable expressions, and an unrestricted initialization section evaluated after the recursive knot is tied. Duggan and Sourelis [9, 10] study a *mixin* module extension to ML, which allows function and datatype definitions to span module boundaries. Like Flatt and Felleisen, they confine such extensible function definitions to the "mixin" section of a mixin module, separate from the effectful initialization section.

There have also been several proposals based on Ancona and Zucca's calculus *CMS* for purely functional call-by-name mixin modules [3]. In one direction, recent work by Ancona *et al.* [2] extends *CMS* with computational effects encapsulated by monads. They handle recursive monadic computations using a recursive do construct based on Erkök and Launchbury's [12]. In another direction, Hirschowitz and Leroy [17] transfer *CMS* to a call-by-value setting. Their type system performs a static analysis of mixin modules to ensure well-founded recursive definitions, but it requires the strictness dependencies between module components to be written explicitly in the interfaces of modules.

6 Conclusion and Future Work

We have proposed a novel type system for general recursion over effectful expressions, to serve as the foundation of a recursive module extension to ML. The presence of effects seems to necessitate a backpatching semantics for recursion similar to that of Scheme. Our type system ensures statically that recursion is well-founded, avoiding some unnecessary run-time costs associated with backpatching. To ensure well-founded recursion in the presence of multiple recursive variables and separate compilation, we track the usage of individual recursive variables, represented statically by *names*. Our core system is easily extended to account for the computational effects of mutable state and continuations. In addition, we extend our language with a form of memoized computation, which allows us to write arbitrary recursive definitions at the expense of an additional run-time cost.

The explicitly-typed version of our type system admits a straightforward typechecking algorithm, and could serve as a target language for compiling a recursive extension to ML. An important direction for future work is to determine the extent to which names should be available to the ML programmer. This will depend heavily on the degree to which types involving names can be inferred when typechecking recursive modules.

Another key direction for future work is to scale our approach to the module level. In addition to the issues involving recursion at the level of types [8], there is the question of how names and recursion interact with other module-level features such as type generativity. We are currently investigating this question by combining the language presented here with our previous work on a type system for higher-order modules [6].

Acknowledgments

The author would like to thank Bob Harper and Karl Crary for invaluable discussions and guidance throughout the development of this work.

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] Davide Ancona, Sonia Fagorzi, Eugenio Moggi, and Elena Zucca. Mixin modules and computational effects. In *2003 International Colloquium on Languages, Automata and Programming*, Eindhoven, The Netherlands, 2003.

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

[4] Gerard Boudol. The recursive record semantics of objects revisited. Research report 4199, INRIA, 2001. To appear in the Journal of Functional Programming.

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

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *2003 ACM Symposium on Principles of Programming Languages*, pages 236–249, 2003.

[7] Derek Dreyer, Robert Harper, and Karl Crary. A type system for well-founded recursion. Technical Report CMU-CS-03-163, Carnegie Mellon University, July 2003.

[8] Derek R. Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, School of Computer Science, Carnegie Mellon University, March 2001.

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

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

[11] Levent Erkök and John Launchbury. Recursive monadic bindings. In *2000 International Conference on Functional Programming*, pages 174–185, Paris, France, 2000.

[12] Levent Erkök and John Launchbury. A recursive do for Haskell. In *2002 Haskell Workshop*, October 2002.

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

[14] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report TR546, Indiana University, December 2000.

[15] John Greiner. Weak polymorphism can be sound. *Journal of Functional Programming*, 6(1):111–141, 1996.

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

[17] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *2002 European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 6–20, 2002.

[18] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *2003 International Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden.

[19] Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

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

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

[22] Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. In *2003 Workshop on Fixed Points in Computer Science*, April 2003.

[23] Aleksandar Nanevski. Meta-programming with names and necessity. In *2002 International Conference on Functional Programming*, pages 206–217, Pittsburgh, PA, 2002. A significant revision is available as a technical report CMU-CS-02-123R, Carnegie Mellon University.

[24] Aleksandar Nanevski. A modal calculus for effect handling. Technical Report CMU-CS-03-149, Carnegie Mellon University, June 2003.

[25] Objective Caml. `http://www.ocaml.org`.

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

[27] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer, 2000.

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

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

[30] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

[31] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.