Efficient Recursion in the Presence of Effects[*]

Derek Dreyer Robert Harper Karl Crary

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

{dreyer,rwh,crary}@cs.cmu.edu

Abstract

In the interest of designing a recursive module extension to ML that is as simple and general as possible, we study the important subproblem of how to support recursive definitions of effectful expressions. Effects seem to necessitate a Scheme-style backpatching semantics of recursion, which typically results in a performance penalty since all recursive references must check whether or not the recursive variable has been backpatched yet.

We propose a type system that allows such dynamic checks to be safely eliminated in many cases. Our system employs a notion of "names", which track the uses of multiple recursive variables in the presence of nested recursion. So that our type system may eventually be integrated smoothly into ML's, reasoning involving names is confined to recursive expressions and does not need to infect existing ML code.

1 Introduction

One of the most distinguishing features of the programming languages in the ML family, namely Standard ML [15] and Objective Caml [18], 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. Banishing all computational effects, however, would be quite 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 [8]. While such a construct is certainly more flexible than forbidding effects entirely, it imposes a structure on recursive modules that is somewhat arbitrary. Others have suggested abandoning ML-style modules altogether in favor of *mixin modules* [2, 13] or *units* [11], 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 in support of a feature that may only be needed on occasion.

1.1 The Semantics of General Recursion

In the interest of designing a recursive module extension to ML that is as simple and general as possible, it is first necessary to study what is involved in extending the core language of ML with recursive definitions of arbitrary (possibly impure) expressions. First of all, what would it mean to write

$$\texttt{val rec } x \texttt{ = } e$$

if the expression e has effects such as I/O or assignment to state?

The standard interpretation of recursion via a *fixed-point* operator would suggest that the above recursive definition is shorthand for

$$\texttt{val } x \texttt{ = } \mathsf{fix}(x.\,e)$$

where $\mathsf{fix}(x.\,e)$ evaluates to its unrolling $e[\mathsf{fix}(x.\,e)/x]$.[1] However, the fixed-point semantics has undesirable behavior in the presence of effects. In particular, any computational effects in e are re-enacted at every recursive reference to x. Setting aside performance issues, this semantics is senseless if e defines functions operating on locally defined mutable state, as every recursive application of those functions (going through x) will operate on an entirely different state.

[*]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 authors and do not reflect the views of this agency.

[1]We use $e'[e/x]$ to denote the capture-avoiding substitution of e for x in e'.

An alternative semantics for recursion that does make sense in the presence of effects is the *backpatching* semantics employed by Scheme [14], in which the above recursive definition would evaluate as follows: First, x is bound to a fresh location containing an undefined value; then, e is evaluated to a value v; finally, x is backpatched with v. If the evaluation of e attempts to dereference x, a run-time error is reported. This in turn means that every recursive use of x must check that x has been defined before dereferencing it.

In the context of a dynamically-typed language like Scheme, this initialization check is not much of an efficiency penalty because programs already perform ubiquitous dynamic tag-checks. In the context of a statically-typed language like ML, though, it means that functions defined using a general recursive construct will be noticeably slower than ordinary ML functions. For the application to modules, if recursive modules are inherently less efficient than hierarchical modules, their performance penalty may ultimately outweigh their usefulness.

1.2 A Type System for Efficient Recursion

In this paper we propose a way to support general recursion under a backpatching semantics, while avoiding the cost of dynamic initialization checks in many cases. The basic idea is to have the type system track uses of recursive variables, so that a recursive term $\mathsf{rec}(x.e)$ is well-typed only if the type system can ensure that the evaluation of e will never *use* the recursive variable x, although e may *refer* to x in suspended code. Consequently, since recursive uses of x are permitted to occur only *after* x has been backpatched, they do not need to perform an initialization check before dereferencing x.

In the presence of nested recursion, we need to track uses of multiple recursive variables at the same time. To do this we employ a notion of *names*, inspired by the work of Nanevski on a core language for metaprogramming and symbolic computation [17]. Nanevski's language provides the ability to generate code at run time containing free references to undefined symbols called names. Similarly, in our system, we wish to be able to evaluate an expression (such as e in $\mathsf{rec}(x.e)$) containing free references to an undefined variable (x). We use names correspondingly to model recursive variables, and the form of our typing judgment is based on Nanevski's.

There are at least two key questions to ask of our type system:

1. What kinds of programs can we write in it?
2. Is there any hope of incorporating it into ML?

To answer the first question, our type system allows the body of a recursive expression to contain arbitrary λ-abstractions and arbitrary uses of effects. It may also contain function applications so long as the body of the function being applied does not attempt to access the undefined recursive variable. In particular, it may contain applications of ML library functions, as well as any functions defined inside the recursive expression whose bodies do not access the recursive variable.

The key point of difficulty involves the application of higher-order functions. To have any hope of eventually incorporating our type system into ML (the second question), we believe it is important that names not infect the external language, including the types of existing ML library functions. When applying a higher-order library function to a function whose body may dereference a recursive variable, the nameless type of the library function gives us no indication of whether it will attempt to apply its argument. We therefore assume the worst and require such function applications to appear in guarded positions such as λ-abstractions.

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 execute even if it contains free references to undefined names. Through a series of examples, we illustrate how some simpler approaches to tracking evaluability suffer 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 introduce computational effects in the form of mutable state and continuations. These constitute completely orthogonal extensions that are essentially oblivious to the presence of names. Then, in Section 5 we show how to encode a more general, but less efficient, form of recursion in our system by extending the language with memoized computations. In circumstances where the type system is too weak to observe that a recursive term is well-typed, this encoding is useful as a fallback. Finally, in Section 6 we compare our system to related work, and in Section 7 we conclude and discuss future work.

2 Evaluability

Consider a general recursive construct of the form $\mathsf{rec}(x:\tau.e)$, representing an expression of e of type τ that may refer to its ultimate value recursively as x. Under the traditional backpatching semantics of recursion, such a construct would have a typing rule something like this (where Γ represents the typing context):

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

This typing rule does not prevent the evaluation of e from attempting to use x, and so the implementation of this construct must insert initialization checks in e before every use of x.

What would be required of e to allow us to eliminate the dynamic checks on x? The answer is quite simple: the evaluation of e would be required never to trigger the evaluation of x. We call this property *evaluability*: To allow $\mathsf{rec}(x:\tau.e)$ to be implemented efficiently, 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 as the name suggests, evaluability is a prerequisite for ensuring that a term can be safely evaluated in the absence of dynamic checks on 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 we call *names*. We write names with capital letters, as opposed to variables which appear in lowercase. The distinction between them can be seen from their typing rules:

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

Given these extensions, we can now write the following revised typing rule for recursive terms:

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

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, consider the following pseudo-ML example of mutually recursive modules `A` and `B`:

```
rec(X : τ.
  structure A = struct
    fun f(x) = ...X.B.g(x-1)...
    fun g(x) = ...X.B.f(x-1)...
  end
  structure B = struct
    fun f(x) = ...X.A.g(x-1)...
    fun g(x) = ...X.A.f(x-1)...
  end
)
```

The body of this module is a pair of submodules `A` and `B`, each of which contains a pair of function values itself. To define `A` and `B` recursively, we need recursion to be defined over a tuple of tuples of functions, not supported directly by ML's notion of recursion. In general, all values and tuples of evaluable expressions should be considered evaluable.

Having computational effects, though, should not preclude a term from being considered evaluable. Suppose we were to add a ref cell to the modules from the previous example:

```
rec(X : τ.
  structure A = struct
    val debug = ref true
    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
)
```

This kind of code is common, for instance to define externally-visible debugging flags in a module. In general, `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 effects in the sense that they are both hidden by λ-abstractions and unleashed by their 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 an unnecessary assumption. While functions defined inside a recursive expression may very well be hiding references to a recursive name, functions defined in ML libraries are not. For example, suppose that we wish to define some local state in module `A` by a call to the array creation function:

```
rec(X : τ.
  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
  structure B = ...
)
```

The call to `Array.create` is perfectly evaluable, while a call to the function `A.g` inside the above module would *not* be evaluable. 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 must 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:

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

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. Functions defined in ML libraries 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 ML library functions even in the presence of undefined names (*i.e.*, inside a recursive expression), as those function applications will be known to be evaluable.

However, there are still some serious problems.

Recursive Functions 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 name `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 the name `F` to be declared with a total arrow `int` $\to$ `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, it may erroneously turn a truly partial function into a total one, as the following code illustrates:

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

The problem 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 "totalize" the type of `fn () => X`.

One might in turn suggest a more restricted version of the revised typing rule, requiring that the body of $\mathsf{rec}(X : \tau.\, e)$ only contain free references to X and no other name. This would disqualify the above counterexample since the body of the inner recursive expression contains a reference to a name other than `Y`. In the presence of effects, though, this version of the rule is unsound as well, as the following example illustrates:

```
rec(X : τ.
  let
     val c = ref (fn () => X)
     val f = rec(Y : unit → τ. !c)
  in
     f()
  end
)
```

The inner recursive expression is once again well-formed since `!c` does not refer to any names at all. Yet evaluation of this code will attempt to use `X` before it is backpatched. The problem is that while `!c` does not refer to any names itself, a reference to `X` is being concealed in the mutable state. We conclude that totalization is only sensible in the absence of nested recursion.

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 has the following type (monomorphic for simplicity):

```
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 even partially apply `map` to a partial function, as in the following:

```
rec (X : τ.
  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 would at least like to be able to replace `map f` with its eta-expansion `fn xs => map f xs`. Even the eta-expansion is ill-typed, however, because the type of `f` still does not match the argument type of `map`. There is something seriously amiss in our semantics if we cannot write arbitrarily non-evaluable code underneath a λ-abstraction. Instead of attempting to solve this problem with yet another partial solution like totalization, we now present our type system with names, which generalizes the partial/total distinction and clarifies the semantic confusions we have exhibited.

Names	X,Y,Z	$\in$	*Names*
Variables	x,y,z	$\in$	*Var*
Support Variables	m,n	$\in$	*SupportVar*
Supports	M,N	$::=$	$\emptyset \mid N,X \mid N,n$
Types	σ,τ	$::=$	$1 \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow{N} \tau_2 \mid \forall n.\, \tau$
Terms	e	$::=$	$x \mid X \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \pi_i(e)$
			$\mid \lambda x{:}\tau.\, e \mid e_1(e_2) \mid \Lambda n.\, e \mid e\{N\}$
			$\mid \mathsf{rec}(X : \tau.\, e)$
Values	v	$::=$	$x \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \lambda x{:}\tau.\, e \mid \Lambda n.\, e$
Typing Contexts	Γ	$::=$	$\emptyset \mid \Gamma, x : \tau \mid \Gamma, X : \tau \mid \Gamma, n$

Figure 1. Core Language Syntax

3 A Core Calculus for General Recursion

The reason that the totalization idea from the previous section is unsound in the presence of nested recursion is that it does not distinguish between uses of different names. In the presence of more than one name, there is no way to tell from the type of a partial function which names it will use when applied. The basic goal of our core calculus is to address this problem by explicitly tracking uses of individual recursive names. In the process, a solution to the higher-order function problem falls out from our novel name-aware type equivalence judgment.

3.1 Syntax

The syntax of our core language is given in Figure 1. We assume the existence of disjoint countably infinite sets of names (*Names*), variables (*Var*), and support variables (*SupportVar*). We use M and N to stand for supports, which are unordered sets that may contain both names and support variables. Supports are used to identify the set of names that an expression may attempt to dereference when evaluated. We explain the role of support variables below.

The type structure of the language consists of unit (1), pair types ($\tau_1 \times \tau_2$), function types ($\tau_1 \xrightarrow{N} \tau_2$), and support-polymorphic types ($\forall n.\, \tau$). The first two need no explanation. A function type has a support on its arrow, which represents the set of names that a function of this type may attempt to use when applied. We will sometimes write $\tau_1 \to \tau_2$ as shorthand for an arrow type with empty support ($\tau_1 \xrightarrow{\emptyset} \tau_2$). Support-polymorphic types classify support abstractions, as described below.

Terms include variables, names, unit, pairs, projections, λ-abstractions, function applications, support abstractions, support applications, and general recursive terms. Values include variables, unit, pairs of values, λ-abstractions, and support abstractions. Typing contexts are ordered sets of variable-type assignments, name-type assignments, and support variables.

Notational Conventions In the term $\lambda x{:}\tau.\, e$, the variable x is bound in e; in the term $\Lambda n.\, e$, the support variable n is bound in e; in the term $\mathsf{rec}(X : \tau.\, e)$, the name X is bound in e; in the type $\forall n.\, \tau$, the support variable n is bound in τ. As usual, we identify terms and

Term well-formedness: $\Gamma \vdash e : \tau\ [N]$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau\ [N]}\ (1) \qquad \frac{X : \tau \in \Gamma \quad X \in N}{\Gamma \vdash X : \tau\ [N]}\ (2)$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1\ [N]}\ (3) \qquad \frac{\Gamma \vdash e_1 : \tau_1\ [N] \quad \Gamma \vdash e_2 : \tau_2\ [N]}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2\ [N]}\ (4)$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2\ [N] \quad i \in \{1,2\}}{\Gamma \vdash \pi_i(e) : \tau_i\ [N]}\ (5)$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau\ [M]}{\Gamma \vdash \lambda x{:}\sigma.\, e : \sigma \xrightarrow{M} \tau\ [N]}\ (6)$$

$$\frac{\Gamma \vdash e_1 : \sigma \xrightarrow{M} \tau\ [N] \quad \Gamma \vdash e_2 : \sigma\ [N] \quad M \subseteq N}{\Gamma \vdash e_1(e_2) : \tau\ [N]}\ (7)$$

$$\frac{\Gamma, n \vdash e : \tau\ [N]}{\Gamma \vdash \Lambda n.\, e : \forall n.\, \tau\ [N]}\ (8) \qquad \frac{\Gamma \vdash e : \forall n.\, \tau\ [N]}{\Gamma \vdash e\{M\} : \tau[M/n]\ [N]}\ (9)$$

$$\frac{\Gamma, X : \tau \vdash e : \sigma\ [N] \quad \Gamma, X : \tau \vdash \sigma \equiv \tau\ [X]}{\Gamma \vdash \mathsf{rec}(X : \tau.\, e) : \tau\ [N]}\ (10)$$

$$\frac{\Gamma \vdash e : \sigma\ [N] \quad \Gamma \vdash \sigma \equiv \tau\ [N]}{\Gamma \vdash e : \tau\ [N]}\ (11)$$

Figure 2. Core Language Typing Judgment

types that are equivalent modulo α-conversion of bound variables.

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/support variable twice, and (2) for any prefix of Γ of the form $\Gamma', x : \tau$ or $\Gamma', X : \tau$, the free names and support variables 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 or support variables appearing on the right-hand-side 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 $\Gamma \vdash e : \tau\ [N]$, with the interpretation "in context Γ, term e has type τ with support N". The support N represents the set of names that we may assume have been defined (backpatched) by the time e is evaluated. The static semantics is carefully designed to validate this assumption.

Rules 1 through 7 may be summarized as follows: Variables need no support (Rule 1), but to evaluate a name X, X must be in the support (2). Unit needs no support (3), but pairs and projections require the support of their constituent expressions (4 and 5). A λ-abstraction needs no support, but the support of its body must be indicated on its arrow type (6). To evaluate a function application $e_1(e_2)$, the support must contain the supports of e_1 and e_2, as well as the support on e_1's arrow type. Note that the rules are designed to make admissible the rule of *support weakening*, which says that if $\Gamma \vdash e : \tau\ [M]$ and $M \subseteq N$, then $\Gamma \vdash e : \tau\ [N]$. That is why, for instance, the conclusion of the variable rule specifies arbitrary support instead of empty support.

Support equivalence: $\Gamma \vdash N_1 \equiv N_2\ [N]$

$$\frac{N_1 \cup N = N_2 \cup N}{\Gamma \vdash N_1 \equiv N_2\ [N]}$$

Type equivalence: $\Gamma \vdash \tau_1 \equiv \tau_2\ [N]$

$$\frac{}{\Gamma \vdash 1 \equiv 1\ [N]}\ (12) \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2\ [N] \quad \Gamma \vdash \tau_1 \equiv \tau_2\ [N]}{\Gamma \vdash \sigma_1 \times \tau_1 \equiv \sigma_2 \times \tau_2\ [N]}\ (13)$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2\ [N] \quad \Gamma \vdash \tau_1 \equiv \tau_2\ [N] \quad \Gamma \vdash N_1 \equiv N_2\ [N]}{\Gamma \vdash \sigma_1 \xrightarrow{N_1} \tau_1 \equiv \sigma_2 \xrightarrow{N_2} \tau_2\ [N]}\ (14)$$

$$\frac{\Gamma, n \vdash \tau_1 \equiv \tau_2\ [N]}{\Gamma \vdash \forall n.\, \tau_1 \equiv \forall n.\, \tau_2\ [N]}\ (15)$$

Figure 3. Core Language Equivalence Judgments

Rules 8 and 9 govern support abstractions and applications. As the name suggests, the support abstraction $\Lambda n.\, e$ allows one to abstract a term e over a support, represented as a support variable n. Support application is used to instantiate a support variable with a specific support. We anticipate support abstractions being used primarily to generalize the types of expressions with empty support, *e.g.,* λ-abstractions like the `map` example below. Thus, in contrast to arrow types, support-polymorphic types do not bear a support. As a consequence, Rule 8 is forced to give the support abstraction $\Lambda n.\, e$ the same support as e. Given the intended usage, e will typically have empty support, so $\Lambda n.\, e$ will have empty support as well.

Equivalence Modulo a Support Rules 10 and 11 are the most interesting rules in the type system, as they both make use of our type equivalence judgment, defined in Figure 3. The judgment $\Gamma \vdash \tau_1 \equiv \tau_2\ [N]$ means that τ_1 and τ_2 are equivalent types *modulo* the names in support N, *i.e.,* that τ_1 are τ_2 are identical types if we ignore all occurrences of the names in N. For example, the types $\tau_1 \xrightarrow{\emptyset} \tau_2$ and $\tau_1 \xrightarrow{X} \tau_2$ are equivalent *modulo* any support containing X. The type equivalence judgment employs an auxiliary judgment of support equivalence, $\Gamma \vdash N_1 \equiv N_2\ [N]$, stating that N_1 and N_2 are equivalent supports *modulo* N.

The intuition behind our type equivalence judgment is that, once a name has been backpatched, that name can be completely ignored for typechecking purposes because we only care about tracking uses of *undefined* names. If the support of e is N, then as far as e is concerned the names in N are defined. Thus, in the context of typing e with support N, the types $\tau_1 \xrightarrow{\emptyset} \tau_2$ and $\tau_1 \xrightarrow{N} \tau_2$ are as good as equivalent since they only differ with respect to the irrelevant defined names in N.

This notion of equivalence is critical to the typing of recursive terms. Recall the factorial example from Section 2.2:

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

The problem 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 $\mathsf{rec}(X : \tau.\, e)$ works as follows: After adding X to the context, it checks that e has some type σ with a support N that does *not* contain X (since X is undefined while evaluating e). Then it checks that σ and τ are equivalent *modulo* X. With respect to our earlier idea of totalization, one can think of Rule 10 as totalizing the type σ with respect to a particular name (X), so that the rule behaves properly in the presence of multiple names (nested recursion).

In contrast, Rule 11 appears rather straightforward, allowing a term e (with type σ and support N) to be assigned a type that is equivalent under the same support. In fact, this rule solves the higher-order function problem described in Section 2.2! Recall that we wanted to apply a higher-order ML library function like `map` to a partial function (in the context of our core language, a "partial function" is one whose arrow type bears a nonempty support):

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

Intuitively, this code ought to typecheck. Why? Because if we are willing to add `X` to the support of `g`'s arrow type, then `X` must be defined before `g` is ever applied, and so `X` should be ignored when typing the body of `g`.

Rule 11 affords us such ignorance. To see why, note that since `g` only needs the type σ `list` $\xrightarrow{\mathtt{X}}$ τ `list`, we can typecheck `map f xs` under support `X`. Having `X` in the support means that $\sigma \xrightarrow{\mathtt{X}} \tau$, the type of `f`, is equivalent to $\sigma \to \tau$, the argument type of `map`. Thus, by Rule 11, `f` can be assigned $\sigma \to \tau$ under support `X`, rendering `map f xs` well-formed.

Support Polymorphism In our original version of the above example, we wanted to define `g` as simply `map f`. Unlike its eta-expansion, however, `map f` should *not* be allowed because there is no way to tell from the type of `map` that the evaluation of `map f` will not apply `f`. In our core calculus, `map f` is only well-typed if `X` is in the support, which is not the case at the point `g` is defined.

Nonetheless, we "know" that `map f` will *not* attempt to apply `f` because we "know" how the Standard Basis `map` function is implemented. With that knowledge, we can give a more precise type to `map` through the use of *support polymorphism*. In particular, by enclosing the implementation of `map` in a support abstraction, we can give it the type:

```
val map : ∀n. (σ -n-> τ) → (σ list -n-> τ list)
```

The support-polymorphic type indicates that `map` will turn a value of any arrow type into a value of the same arrow type, but will not use any names in the process. Given this type for `map`, the following code will indeed be well-typed:

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

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

Figure 4. Syntax of Core Continuations

The support-polymorphism solution results in better code than 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` in the Standard Basis, infecting the existing ML infrastructure with names. It is therefore important that, in the absence of this solution, our type system is strong enough to typecheck at least the eta-expansion of `map f`, without requiring changes to existing ML code.

3.3 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 4. A store ω is a partial mapping from names to *storable things*. (In the context of the dynamic semantics, names are essentially memory locations.) A storable thing S is either a term (e) or nonsense (**?**). By $\omega(X)$ we denote the storable thing stored at name X in ω, which is only valid if something (possibly nonsense) is stored at X. By $\omega[X \mapsto S]$ we denote the result of creating a new name in ω and storing S at it. By $\omega[X := S]$ we denote the result of updating the store ω to store S 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 5. It takes the form of a stepping relation $\Omega \mapsto \Omega'$. Rules 16 through 25 are all fairly standard. Rule 26 says that, in order to evaluate $\mathsf{rec}(X : \tau.\, e)$, we create a new undefined location X in the store, push the recursive frame $\mathsf{rec}(X : \tau.\, \bullet)$ 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 27 performs the backpatching step: it stores v at name X in the store and returns v. Finally, if a backpatched name X is ever referenced, Rule 28 simply looks up the value it is bound to in the store.

3.4 Type Safety

Observe that the machine is stuck if the expression being evaluated is an undefined name (one 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:

DEFINITION 3.1 (RUNTIME CONTEXTS). A context Γ is *runtime* if $\mathrm{dom}(\Gamma) \cap (\mathit{Var} \cup \mathit{SupportVar}) = \emptyset$.

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

1. Γ is runtime

2. $\mathrm{dom}(\omega) = \mathrm{dom}(\Gamma)$

$$\frac{\langle e_1, e_2\rangle \text{ not a value}}{(\omega;C;\langle e_1,e_2\rangle) \mapsto (\omega;C\circ\langle\bullet,e_2\rangle;e_1)} \quad (16)$$

$$\frac{}{(\omega;C\circ\langle\bullet,e\rangle;v) \mapsto (\omega;C\circ\langle v,\bullet\rangle;e)} \quad (17)$$

$$\frac{}{(\omega;C\circ\langle v_1,\bullet\rangle;v_2) \mapsto (\omega;C;\langle v_1,v_2\rangle)} \quad (18)$$

$$\frac{}{(\omega;C;\pi_i(e)) \mapsto (\omega;C\circ\pi_i(\bullet);e)} \quad (19)$$

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

$$\frac{}{(\omega;C;e_1(e_2)) \mapsto (\omega;C\circ\bullet(e_2);e_1)} \quad (21)$$

$$\frac{}{(\omega;C\circ\bullet(e);v) \mapsto (\omega;C\circ v(\bullet);e)} \quad (22)$$

$$\frac{}{(\omega;C\circ(\lambda x{:}\tau.\,e)(\bullet);v) \mapsto (\omega;C;e[v/x])} \quad (23)$$

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

$$\frac{}{(\omega;C\circ\bullet\,\{M\};\Lambda n.\,e) \mapsto (\omega;C;e[M/n])} \quad (25)$$

$$\frac{X \notin \text{dom}(\omega)}{(\omega;C;\mathsf{rec}(X:\tau.\,e)) \mapsto (\omega[X\mapsto ?];C\circ\mathsf{rec}(X:\tau.\,\bullet);e)} \quad (26)$$

$$\frac{}{(\omega;C\circ\mathsf{rec}(X:\tau.\,\bullet);v) \mapsto (\omega[X{:=}v];C;v)} \quad (27)$$

$$\frac{\omega(X)=v}{(\omega;C;X) \mapsto (\omega;C;v)} \quad (28)$$

Figure 5. Core Language Dynamic Semantics

3. $\forall X \in N.\ \exists v.\ \omega(X) = v$ and $\Gamma \vdash v : \Gamma(X)\ [N]$

Essentially, the judgment $\Gamma \vdash \omega\ [N]$ says that Γ assigns types to all names in the domain of ω, that the names in support N are all defined names, and that they all map to appropriately-typed values.

We define well-formedness of continuations and continuation frames via the judgments $\Gamma \vdash C : \tau\ \mathsf{cont}\ [N]$ and $\Gamma \vdash F : \tau_1 \Rightarrow \tau_2\ [N]$, defined in Figure 6. 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 38 for recursive frames $\mathsf{rec}(X:\tau.\,\bullet)$. Since this frame is not a binder for X, Rule 38 requires that X already be in the typing context. This is a safe assumption since $\mathsf{rec}(X:\tau.\,\bullet)$ 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 machine states:

DEFINITION 3.3 (MACHINE STATE WELL-FORMEDNESS). A

Continuation well-formedness: $\boxed{\Gamma \vdash C : \tau\ \mathsf{cont}\ [N]}$

$$\frac{}{\Gamma \vdash \bullet : \tau\ \mathsf{cont}\ [N]} \quad (29)$$

$$\frac{\Gamma \vdash F : \tau \Rightarrow \sigma\ [N] \quad \Gamma \vdash C : \sigma\ \mathsf{cont}\ [N]}{\Gamma \vdash C\circ F : \tau\ \mathsf{cont}\ [N]} \quad (30)$$

$$\frac{\Gamma \vdash C : \sigma\ \mathsf{cont}\ [N] \quad \Gamma \vdash \sigma \equiv \tau\ [N]}{\Gamma \vdash C : \tau\ \mathsf{cont}\ [N]} \quad (31)$$

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

$$\frac{\Gamma \vdash e : \tau_2\ [N]}{\Gamma \vdash \langle\bullet,e\rangle : \tau_1 \Rightarrow \tau_1\times\tau_2\ [N]} \quad (32)$$

$$\frac{\Gamma \vdash v : \tau_1\ [N]}{\Gamma \vdash \langle v,\bullet\rangle : \tau_2 \Rightarrow \tau_1\times\tau_2\ [N]} \quad (33)$$

$$\frac{i \in \{1,2\}}{\Gamma \vdash \pi_i(\bullet) : \tau_1\times\tau_2 \Rightarrow \tau_i\ [N]} \quad (34)$$

$$\frac{\Gamma \vdash e : \sigma\ [N]}{\Gamma \vdash \bullet(e) : \sigma\to\tau \Rightarrow \tau\ [N]} \quad (35) \qquad \frac{\Gamma \vdash v : \sigma\to\tau\ [N]}{\Gamma \vdash v(\bullet) : \sigma \Rightarrow \tau\ [N]} \quad (36)$$

$$\frac{}{\Gamma \vdash \bullet\,\{M\} : \forall n.\,\tau \Rightarrow \tau[M/n]\ [N]} \quad (37)$$

$$\frac{X:\tau \in \Gamma \quad \Gamma \vdash \sigma \equiv \tau\ [X]}{\Gamma \vdash \mathsf{rec}(X:\tau.\,\bullet) : \sigma \Rightarrow \tau\ [N]} \quad (38)$$

Figure 6. Well-formedness of Core Continuations

machine state Ω is *well-formed*, denoted $\Gamma \vdash \Omega\ [N]$, if $\Omega = (\omega;C;e)$, where:

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

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

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

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 $\Gamma \vdash \Omega\ [N]$, then Ω is not stuck.

Note that when $\Omega = (\omega;C;X)$, the well-formedness of Ω requires that $\Gamma \vdash X : \tau\ [N]$ for some τ, which in turn means that the support N must contain X. 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 28.

COROLLARY 3.8 (TYPE SAFETY). Suppose $\emptyset \vdash e : \tau\ [\emptyset]$. Then

Mutable Locations	$\ell \in \mathit{MutLoc}$
Cont'n Locations	$k \in \mathit{ContLoc}$
Types	$\tau ::= \cdots \mid \mathsf{ref}(\tau) \mid \mathsf{cont}(\tau)$
Terms	$e ::= \cdots \mid \ell \mid k \mid \mathsf{ref}(e) \mid \mathsf{get}(e) \mid \mathsf{set}(e_1, e_2) \mid \mathsf{callcc}_\tau(k.\,e) \mid \mathsf{throw}(e_1, e_2)$
Values	$v ::= \cdots \mid \ell \mid k$
Typing Contexts	$\Gamma ::= \cdots \mid \Gamma, \ell : \mathsf{ref}(\tau) \mid \Gamma, k : \mathsf{cont}(\tau)$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [N]}{\Gamma \vdash \mathsf{ref}(\tau_1) \equiv \mathsf{ref}(\tau_2)\ [N]}\ (39) \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [N]}{\Gamma \vdash \mathsf{cont}(\tau_1) \equiv \mathsf{cont}(\tau_2)\ [N]}\ (40)$$

$$\frac{\ell : \mathsf{ref}(\tau) \in \Gamma}{\Gamma \vdash \ell : \mathsf{ref}(\tau)\ [N]}\ (41) \qquad \frac{k : \mathsf{cont}(\tau) \in \Gamma}{\Gamma \vdash k : \mathsf{cont}(\tau)\ [N]}\ (42)$$

$$\frac{\Gamma \vdash e : \tau\ [N]}{\Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}(\tau)\ [N]}\ (43) \qquad \frac{\Gamma \vdash e : \mathsf{ref}(\tau)\ [N]}{\Gamma \vdash \mathsf{get}(e) : \tau\ [N]}\ (44)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ref}(\tau)\ [N] \quad \Gamma \vdash e_2 : \tau\ [N]}{\Gamma \vdash \mathsf{set}(e_1, e_2) : 1\ [N]}\ (45)$$

$$\frac{\Gamma, k : \mathsf{cont}(\tau) \vdash e : \tau\ [N]}{\Gamma \vdash \mathsf{callcc}_\tau(k.\,e) : \tau\ [N]}\ (46)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{cont}(\sigma)\ [N] \quad \Gamma \vdash e_2 : \sigma\ [N]}{\Gamma \vdash \mathsf{throw}(e_1, e_2) : \tau\ [N]}\ (47)$$

Figure 7. Static Semantics for Effects

Continuation Frames $F ::= \cdots \mid \mathsf{ref}(\bullet) \mid \mathsf{get}(\bullet) \mid \mathsf{set}(\bullet, e) \mid \mathsf{set}(v, \bullet) \mid \mathsf{throw}(\bullet, e) \mid \mathsf{throw}(v, \bullet)$

$$\frac{}{(\omega; C; \mathsf{ref}(e)) \mapsto (\omega; C \circ \mathsf{ref}(\bullet); e)}\ (48)$$

$$\frac{\ell \notin \mathrm{dom}(\omega)}{(\omega; C \circ \mathsf{ref}(\bullet); v) \mapsto (\omega[\ell \mapsto v]; C; \ell)}\ (49)$$

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

$$\frac{\omega(\ell) = v}{(\omega; C \circ \mathsf{get}(\bullet); \ell) \mapsto (\omega; C; v)}\ (51)$$

$$\frac{}{(\omega; C; \mathsf{set}(e_1, e_2)) \mapsto (\omega; C \circ \mathsf{set}(\bullet, e_2); e_1)}\ (52)$$

$$\frac{}{(\omega; C \circ \mathsf{set}(\bullet, e); v) \mapsto (\omega; C \circ \mathsf{set}(v, \bullet); e)}\ (53)$$

$$\frac{}{(\omega; C \circ \mathsf{set}(\ell, \bullet); v) \mapsto (\omega[\ell := v]; C; \langle\rangle)}\ (54)$$

$$\frac{k \notin \mathrm{dom}(\omega)}{(\omega; C; \mathsf{callcc}_\tau(k.\,e)) \mapsto (\omega[k \mapsto C]; C; e)}\ (55)$$

$$\frac{}{(\omega; C; \mathsf{throw}(e_1, e_2)) \mapsto (\omega; C \circ \mathsf{throw}(\bullet, e_2); e_1)}\ (56)$$

$$\frac{}{(\omega; C \circ \mathsf{throw}(\bullet, e); v) \mapsto (\omega; C \circ \mathsf{throw}(v, \bullet); e)}\ (57)$$

$$\frac{\omega(k) = C_k}{(\omega; C \circ \mathsf{throw}(k, \bullet); v) \mapsto (\omega; C_k; v)}\ (58)$$

Figure 8. Dynamic Semantics for Effects

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) will appear in a forthcoming technical report.

4 Adding Computational Effects

Since we have modeled the backpatching semantics of recursion quite operationally in terms of a mutable store, it is easy to incorporate some actual computational effects into our core framework.

Figure 7 gives extensions to the syntax and static semantics for adding mutable state and continuations to the language. The primitives for the former are ref, get and set, and the primitives for the latter are callcc and throw, all with the standard typing rules. Ref cells and continuations are allocated in the store, so the values of types $\mathsf{ref}(\tau)$ and $\mathsf{cont}(\tau)$ are locations in the store. We use ℓ to stand for a mutable location (the canonical form of type $\mathsf{ref}(\tau)$) and k to stand for a continuation location (the canonical form of type $\mathsf{cont}(\tau)$). (In $\mathsf{callcc}_\tau(k.\,e)$, k is bound in e.)

If we think of a continuation as a kind of function with no return type, it may seem surprising that the typing rules for continuations are oblivious to names. Moreover, while the arrow type $\tau_1 \xrightarrow{N} \tau_2$ specifies the support N required to call a function of that type, the continuation type $\mathsf{cont}(\tau)$ does not specify a support, and no support is required in order to throw to a continuation. (One can view $\mathsf{cont}(\tau)$ as always specifying empty support.)

To see why a support is unnecessary, consider what the machine state looks like when we are about to evaluate a callcc: it has the form $\Omega = (\omega; C; \mathsf{callcc}_\tau(k.\,e))$. Assuming that Ω is well-formed ($\Gamma \vdash \Omega\ [N]$), we know that $\Gamma \vdash C : \tau\ \mathsf{cont}\ [N]$ and $\Gamma \vdash \mathsf{callcc}_\tau(k.\,e) : \tau\ [N]$. The current continuation is C and that is what callcc will bind to k before evaluating e.

Then what is the "type" of C? Explicit continuations are not part of our language, but we can nonetheless think of C as a function with argument type τ that, when applied, may use any of the names in the support N. Thus, the most appropriate arrow-like type for C would be $\tau \xrightarrow{N} \sigma$ for some return type σ. Since the names in N are all defined, though, the types $\tau \xrightarrow{N} \sigma$ and $\tau \rightarrow \sigma$ are equivalent. It is therefore correct to assign k a type bearing empty support.

Figure 8 gives the extensions to the dynamic semantics for mutable state and continuations. We extend stores ω to contain mappings from mutable locations ℓ to values v and from continuation locations k to continuations C. The rules for mutable state are completely straightforward. The rules for continuations are also fairly straightforward, since the machine state already makes the current continuation explicit.

$$\frac{}{\Gamma \vdash \mathsf{ref}(\bullet) : \tau \Rightarrow \mathsf{ref}(\tau)\ [N]}\ (59) \qquad \frac{}{\Gamma \vdash \mathsf{get}(\bullet) : \mathsf{ref}(\tau) \Rightarrow \tau\ [N]}\ (60)$$

$$\frac{\Gamma \vdash e : \tau\ [N]}{\Gamma \vdash \mathsf{set}(\bullet, e) : \mathsf{ref}(\tau) \Rightarrow 1\ [N]}\ (61) \qquad \frac{\Gamma \vdash v : \mathsf{ref}(\tau)\ [N]}{\Gamma \vdash \mathsf{set}(v, \bullet) : \tau \Rightarrow 1\ [N]}\ (62)$$

$$\frac{\Gamma \vdash e : \sigma\ [N]}{\Gamma \vdash \mathsf{throw}(\bullet, e) : \mathsf{cont}(\sigma) \Rightarrow \tau\ [N]}\ (63)$$

$$\frac{\Gamma \vdash v : \mathsf{cont}(\sigma)\ [N]}{\Gamma \vdash \mathsf{throw}(v, \bullet) : \sigma \Rightarrow \tau\ [N]}\ (64)$$

Figure 9. Well-formedness of Effects Continuations

Proving type safety for these extensions requires only a simple, orthogonal extension of the proof framework from Section 3.4. The judgment on well-formedness of continuations is extended in the obvious way, as shown in Figure 9. The judgment on store well-formedness is also extended in a straightforward way to include well-formedness conditions on the things stored at mutable and continuation locations:

DEFINITION 4.1 (STORE WELL-FORMEDNESS). A store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [N]$, if $\Gamma \vdash \omega\ [N]$ according to Definition 3.2 and also:

1. $\forall \ell : \mathsf{ref}(\tau) \in \Gamma.\ \exists v.\ \omega(\ell) = v$ and $\Gamma \vdash v : \tau\ [N]$
2. $\forall k : \mathsf{cont}(\tau) \in \Gamma.\ \exists C.\ \omega(k) = C$ and $\Gamma \vdash C : \tau\ \mathsf{cont}\ [N]$

5 Encoding Unrestricted Recursion

Despite all the efforts of our type system, there will always be recursive terms $\mathsf{rec}(X : \tau.\, e)$ 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 $\mathsf{rec}(X : \tau.\, e)$ with the understanding that every dereference of X will incur an initialization check.

One option is to add a second recursive term construct employing implicit dynamic checks, with the typing rule:

$$\frac{\Gamma, x : \tau \vdash e : \tau\ [N]}{\Gamma \vdash \mathsf{urec}(x : \tau.\, e) : \tau\ [N]}$$

(Note that we model the recursive variable with an ordinary variable x, not a name X, because there are no restrictions on the use of the variable within e.) However, adding an explicit urec construct 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 10. First, we introduce a type $\mathsf{comp}_N(\tau)$ of memoized computations. A value of this type is essentially a thunk of type $1 \xrightarrow{N} \tau$ whose result is memoized after the first application.

The primitive $\mathsf{delay}(e)$ creates a memoized location m in the store bound to the unevaluated expression e. When m is forced (by

Memoized Locations	$m \in \mathit{MemoLoc}$
Types	$\tau ::= \cdots \mid \mathsf{comp}_N(\tau)$
Terms	$e ::= \cdots \mid m \mid \mathsf{delay}(e) \mid \mathsf{force}(e)$
Values	$v ::= \cdots \mid m$
Typing Contexts	$\Gamma ::= \cdots \mid \Gamma, m : \mathsf{comp}_N(\tau)$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [N] \quad \Gamma \vdash N_1 \equiv N_2\ [N]}{\Gamma \vdash \mathsf{comp}_{N_1}(\tau_1) \equiv \mathsf{comp}_{N_2}(\tau_2)\ [N]}\ (65)$$

$$\frac{m : \mathsf{comp}_M(\tau) \in \Gamma}{\Gamma \vdash m : \mathsf{comp}_M(\tau)\ [N]}\ (66) \qquad \frac{\Gamma \vdash e : \tau\ [M]}{\Gamma \vdash \mathsf{delay}(e) : \mathsf{comp}_M(\tau)\ [N]}\ (67)$$

$$\frac{\Gamma \vdash e : \mathsf{comp}_M(\tau)\ [N] \quad M \subseteq N}{\Gamma \vdash \mathsf{force}(e) : \tau\ [N]}\ (68)$$

Figure 10. Static Semantics for Memoized Computations

$\mathsf{force}(m)$), the expression e stored at m is evaluated to a value v, and then v is written back to m. During the evaluation of e, the location m is bound to nonsense; if m is forced again during this stage, the machine raises an error. Thus, every force of m 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 $\mathsf{comp}_N(\tau)$, $\mathsf{delay}(e)$ and $\mathsf{force}(e)$ were shorthand for $1 \xrightarrow{N} \tau$, $\lambda\langle\rangle : 1.\, e$ and $e\,\langle\rangle$, respectively. We use $\mathsf{comp}(\tau)$ sometimes as shorthand for $\mathsf{comp}_\emptyset(\tau)$.

We can now encode the urec construct in terms of a recursive memoized computation:

$$\begin{aligned} &\mathsf{urec}(x : \tau.\, e) \overset{\text{def}}{=} \\ &\qquad \mathsf{force}(\mathsf{rec}(X : \mathsf{comp}(\tau).\, \mathsf{delay}(e[\mathsf{force}(X)/x]))) \end{aligned}$$

The idea behind this encoding is that x in $\mathsf{urec}(x : \tau.\, e)$ is not really a value of type τ, but a *computation* of type τ that may diverge if x has not been backpatched yet. Hence, the recursive name X is declared with $\mathsf{comp}(\tau)$ instead of τ. Correspondingly, the *initialization* check implicit in every reference to x is modeled by the *memoization* check made explicit by $\mathsf{force}(X)$. Essentially what we have done is to separate recursion and initialization checks into distinct language constructs, whereas urec combines them.

Observe that if we were to give $\mathsf{comp}(\tau)$ a non-memoizing semantics, *i.e.*, to consider it synonymous with $1 \to \tau$, 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 11. We extend stores to contain mappings from memoized locations m to storable things (either e or **?**). To evaluate $\mathsf{delay}(e)$, we create a new memoized location in the store and bind e to it (Rule 69). To evaluate $\mathsf{force}(e)$, we first evaluate e (70). Once e evaluates to a location m, we look m up in the store. If m 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 m (71 and 72). If m is instead bound to nonsense, then we must be in the middle of evaluating another $\mathsf{force}(m)$, so we step to an Error state which halts the program (73).

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

$$\frac{m \notin \mathrm{dom}(\omega)}{(\omega;C;\mathsf{delay}(e)) \mapsto (\omega[m \mapsto e];C;m)} \quad (69)$$

$$\frac{}{(\omega;C;\mathsf{force}(e)) \mapsto (\omega;C \circ \mathsf{force}(\bullet);e)} \quad (70)$$

$$\frac{\omega(m) = e}{(\omega;C \circ \mathsf{force}(\bullet);m) \mapsto (\omega[m := \mathbf{?}];C \circ \mathsf{memo}(m,\bullet);e)} \quad (71)$$

$$\frac{}{(\omega;C \circ \mathsf{memo}(m,\bullet);v) \mapsto (\omega[m := v];C;v)} \quad (72)$$

$$\frac{\omega(m) = \mathbf{?}}{(\omega;C \circ \mathsf{force}(\bullet);m) \mapsto \mathsf{Error}} \quad (73)$$

Figure 11. Dynamic Semantics of Memoization

$$\frac{}{\Gamma \vdash \mathsf{force}(\bullet) : \mathsf{comp}(\tau) \Rightarrow \tau\ [N]} \quad (74)$$

$$\frac{\Gamma \vdash m : \mathsf{comp}(\tau)\ [N]}{\Gamma \vdash \mathsf{memo}(m,\bullet) : \tau \Rightarrow \tau\ [N]} \quad (75)$$

Figure 12. Well-formedness of Memoization Continuations

As with the extensions of the previous section, extending the type safety proof to handle memoized computations is straightforward. Continuation frame well-formedness is extended with the two rules in Figure 12. Note that the rules refer to $\mathsf{comp}(\tau)$ instead of $\mathsf{comp}_M(\tau)$ for an arbitrary $M \subseteq N$. The reason is that, under support N, $\mathsf{comp}(\tau)$ is equivalent to $\mathsf{comp}_M(\tau)$ for any $M \subseteq N$.

We must also update the definition of well-formed and terminal states to include Error, and the definition of store well-formedness to account for memoized locations:

DEFINITION 5.1 (MACHINE STATE WELL-FORMEDNESS). A machine state Ω is *well-formed* if either $\Omega = \mathsf{Error}$ or Ω is well-formed according to Definition 3.3.

DEFINITION 5.2 (TERMINAL STATES). A machine state Ω is *terminal* if either $\Omega = \mathsf{Error}$ or Ω is terminal according to Definition 3.5.

DEFINITION 5.3 (STORE WELL-FORMEDNESS). A store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [N]$, if $\Gamma \vdash \omega\ [N]$ according to Definition 4.1 and also:

- $\forall m : \mathsf{comp}_M(\tau) \in \Gamma$. either $\omega(m) = \mathbf{?}$ or $\Gamma \vdash \omega(m) : \tau\ [M \cup N]$

6 Related Work

The idea of using names in our core calculus is inspired by Nanevski's work on using a modal logic with names to model a "metaprogramming" language for symbolic computation [17]. (His use of names was in turn inspired by Pitts and Gabbay's FreshML [19].) Nanevski uses names to represent undefined symbols appearing inside "boxed" expressions of a modal $\Box$ type. These boxed expressions can be viewed as pieces of uncompiled syntax whose free names must be defined before they can be compiled.

While the form of our main typing judgment is based closely on Nanevski's, the semantic connections to his system are more tenuous. In addition to the fact that we do not employ a $\Box$ type, our calculus differs from his on the meaning of *support*. In our language, the support of e is the set of names that the evaluation of e may dereference. In his language, the support of e is the set of names appearing free in e, outside of boxed expressions. As a consequence, arrow types in his language do not need to indicate any support because λ-abstractions do not suspend the support of their bodies. Lastly, Nanevski's language does not employ any judgment of type equivalence modulo a support, which plays a critical role in our system.

The closest work to ours thematically is that of Boudol [3], who proposes a type system for statically ensuring that recursive definitions of arbitrary expressions are well-formed. While Boudol's language is designed, like ours, to support efficient general recursion in the presence of effects, his approach is quite different from ours. Instead of distinguishing recursive variables and tracking their usage as we do, he tracks the *degrees* to which expressions depend on variables. 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. The equivalent of $\mathsf{rec}(x : \tau.\,e)$ in Boudol's system requires that the degree to which e depends on x is 1.

In our system an arrow type indicates the names that will be used when a function of that type is applied. An arrow type in Boudol's system indicates the degree to which the body of the function depends on its *argument*. Sometimes this can provide useful information, such as in the following type one might give to `map`:

$$(\sigma \xrightarrow{0} \tau) \xrightarrow{1} (\sigma\ \mathtt{list} \xrightarrow{0} \tau\ \mathtt{list})$$

Much like the support-polymorphic type we gave in Section 3.2, this type indicates that `map` does not use its argument immediately.

In other cases, it is not clear that degrees are tracking the right thing. For instance, Boudol's system assigns both $\lambda y.y$ and $\lambda y.y\langle\rangle$ arrow types bearing degree 0, since y appears nakedly in the bodies of both functions. As a result, $(\lambda y.y)(\lambda\langle\rangle.X)$ and $(\lambda y.y\langle\rangle)(\lambda\langle\rangle.X)$ both depend on the variable X with degree 0, despite the fact that only the latter term will *use* X during its evaluation. The former term could thus not be used to recursively define X in Boudol's system, whereas it can in ours. As this example illustrates, degrees seem to be tracking something fundamentally different from our notion of support. We are interested in better understanding the connection, but at the present time it is difficult for us to compare the expressiveness of our language and Boudol's much further.

Hirschowitz and Leroy [13] propose a variant of Boudol's calculus, which they use as the target language for compiling their call-by-value version of Ancona and Zucca's mixin module language [2] (see below). They extend Boudol's notion of degrees to be arbitrary integers: the degree to which e depends on x becomes the *number* of λ-abstractions under which x appears in e. While this generalization allows more programs to typecheck, it still fails to accept the recursive definition from the previous paragraph.

There has been considerable work recently on adding general 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 [9] 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 [10]. Friedman and Sabry [12] 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 the well-formedness of recursive definitions 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 heavily from recent work by Moggi and Sabry [16], who give an operational semantics for the monadic metalanguage extended with the Friedman-Sabry form of mfix. The main difference between our semantics and theirs is that they always perform an initialization check at references to the recursive variable bound by mfix.

There has been a lot of work on recursive modules. However, as we discussed in the introduction, most proposals restrict the form of the recursive module construct so that recursion is not defined over an effectful expression. Crary, Harper and Puri [4] give a foundational account of recursive modules that models recursion via a fixed-point construct at the module level. They employ a judgment of *valuability*, classifying modules/expressions that are pure and terminating, and require that the body of a fixed-point module be valuable in a context where the recursive variable is non-valuable. Due to this restriction, their recursive constructs are able to employ a standard fixed-point semantics. Our notion of *evaluability* from Section 2 can be seen as a generalization of valuability.

Similarly, Flatt and Felleisen's proposal for *units* [11] 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 [7, 8] 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 [2]. In one direction, recent work by Ancona *et al.* [1] extends *CMS* with computational effects encapsulated by monads. They handle recursive monadic computations using a recursive do construct based on Erkök and Launchbury's [10]. In another direction, Hirschowitz and Leroy [13] transfer *CMS* to a call-by-value setting. Their type system does perform a static analysis of mixin modules to ensure well-formedness of recursive definitions, but it requires the dependencies between module components to be written explicitly in the types of modules. It is not clear how types that contain dependency graphs would be incorporated into a practical programming language.

The most liberal of the recursive module proposals is Russo's extension to Moscow ML [20], which permits arbitrary code in the body of a recursive module. He does not make any attempt, though, to statically detect ill-formed recursive definitions, thus forcing an initialization check at every reference to the recursive module variable.

7 Conclusion and Future Work

We have defined a core type system for general recursion over arbitrary expressions in an impure call-by-value language. The presence of effects necessitates a backpatching semantics for recursion based on Scheme's. To avoid the cost of initialization checks at recursive references, we define a judgment of evaluability, ensuring that the body of a recursive expression will evaluate without attempting to access the undefined recursive variable. In order to track evaluability in the presence of nested recursion, we employ a type system that tracks the usage of individual recursive variables, which we call *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 cost of dynamic checks.

Given that we do not plan to add names to the ML source language, an important direction for future work is to determine the degree to which types involving names can be inferred. Ideally, we will be able to infer principal support-polymorphic types for higher-order functions, such as the one we gave for `map` in Section 3.2. These will in turn allow more applications of locally-defined functions to be judged evaluable. A more immediate problem is to determine if our explicitly-typed core language admits a practical typechecking algorithm.

Another key direction for future work is to scale our approach to the module level. Aside from issues involving recursion at the level of types [6], there is the question of how names and recursion interact with higher-order modules [5] and how effectively names can be used in the presence of separate compilation.

References

[1] 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. To appear.

[2] 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.

[3] Gerard Boudol. The recursive record semantics of objects revisited. Research report 4199, INRIA, 2001. Preliminary version presented at ESOP'01, LNCS 2028.

[4] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, 1999. ACM SIGPLAN.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

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