

Research Report

PROGRAMMING A LANGUAGE

Alexander Aiken

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776

John H. Williams
Edward L. Wimmers

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).



Research Division
Yorktown Heights, New York ■ San Jose, California ■ Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

Programming a Language

Alexander Aiken
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
email: aiken@cs.berkeley.edu

John H. Williams
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: williams@almaden.ibm.com

Edward L. Wimmers
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: wimmers@almaden.ibm.com

Abstract

We explore a simple approach to incorporating new language features into a functional language. New features are realized simply by redefining the meanings of primitive functions. That is, the programming language is not modified at all; rather, new features are just programmed within the existing language. Surprisingly, in a combinator-based functional language the class of language features that can be expressed this way is very large; a theorem is presented characterizing this class. Examples are given which illustrate how features are implemented using this approach and which describe practical experience with the technique in a compiler for a functional language.

1 Introduction

Anyone designing a programming language must decide which of a long list of language features to include. State, exceptions, and continuations are but a few of the most popular candidates. Each additional feature provides some convenience in programming but comes with an associated cost—the semantics of the language is complicated by the presence of the new feature and its potential interactions with other features.

The designers of functional languages have tended to opt for simplicity, in the hope that a clean design and the ability to write higher-order functions compensate for the lack of multiple special-purpose features [Hug89]. However, functional programs can be much less convenient to modify than imperative programs [Fel90, Rie91]. For example, suppose one wished to add a counter to a function written in a functional language. If state—or some similar feature—were available in the language, very little programming would be required. However, because no state can be associated with a pure function, it would probably be necessary to modify not just the one function, but also every function that depends, directly or indirectly, on that particular function [Wad90a].

Recently there have been several proposals to incorporate features of conventional languages in functional languages in a way that preserves clean functional semantics as much as possible. The proposed mechanisms are diverse, ranging from adding special primitive functions to a language [WW88, AWW90], to new type systems [Laf88, Wad90b], to mixed imperative/functional languages [Ode91, SRI91]. The common view shared by these proposals is that some change to the basic functional language is required.

There is a simpler approach: using the power of a functional language to implement desired language features as programs written in the functional language itself. A well developed example of this approach is the *monadic* style of programming [Wad90a]. Using monads as a structuring device for programs, a large class of language features can be implemented within a functional language, including state and continuations, among others. This paper investigates a different, and more general, paradigm for structuring software, based on the ability to redefine functions. Our technique is motivated by existing mechanisms for overloading functions. In many languages the function “+” can be extended to programmer-defined complex numbers as well as built-in integers and reals. This allows existing programs that use “+” to take advantage of this new, non-standard behavior of addition. Extending this idea, the programmer might redefine the primitive combining forms of the language—the “glue” with which programs are built—to realize language extensions. This approach, if successful, has two advantages. First, it solves the problem of expressing new language features without complicating the semantics of the language. Second, it avoids the pitfalls of building features into the language in a particular way, which may or may not be suitable for future applications. Since the features are just programs, they can be modified to meet unforeseen requirements.

To make precise this approach of implementing language features as programs, we must say what a “language feature” is. We take the view that language features are embodied by a language’s semantic function; that is, adding a feature to a language is accomplished by changing its semantic function. Thus, we are interested in investigating non-standard semantics for a language that can be implemented within the language itself. More formally, let L be a programming language with standard semantic function μ and let P be a program written in L . Given a non-standard semantic function γ , is there an L -context $E()$ such that $\forall P \mu(E(P)) = \gamma(P)$? Our main result is that for a certain class of functional languages and a large class of semantic functions, the answer to this question is “yes” (Theorem 4.2). Returning to the above example, let γ be a non-standard semantics that threads a state through a computation, and let $E()$ be a context that implements γ . It is easy to add a counter to a program $E(P)$ by exploiting the state.

To be generally applicable, this technique requires that *all* program forming operations be redefinable by the programmer. Most languages do not have this property and therefore are not suited to this approach; for example, in a language based on the lambda calculus there is no way to redefine the meaning of lambda (the primary program-building form) within the language. However, a combinator-based functional language (such as FL [BWW⁺89]) can have this property, since all functions are built using primitive combinators, each of which can be redefined within the language. This observation leads to the thesis that many language features are not required in a

combinator-based functional language, because they can be implemented directly by redefining the primitive functions. Furthermore, a feature defined in this way is just as concise to use as it would be if it were added as part of the language definition.

The idea that all program forming operations should be redefinable by programmers is, to say the least, radical. The idea has significant consequences, not the least of which is that it gives programmers enormous and relatively unstructured programming power. For this reason, we cannot recommend it as a principle of language design. The purpose of this paper is to make a point about language expressiveness: it is a curious and non-obvious fact that any natural functional programming language should have the property that non-standard semantic functions can be implemented within the language itself. In addition, despite the misgivings outlined above, we have found that the idea has applications. In fact, a component of an FL compiler has been designed and implemented as a non-standard FL semantics implemented in FL (see Section 5). Without the paradigm of “programming a language” within FL, this component would have been much more difficult to implement.

The rest of the paper is organized as follows. Section 2 presents MiniFL, the subset of the language FL required for this paper. Section 3 develops a full example of the technique, showing how to implement lambda abstraction in MiniFL. Section 4 generalizes the technique and shows that it can implement a large class of non-standard semantics. Section 5 presents an extended example and discusses practical experience with the technique. All of this is done within the context of a pure functional language. Section 6 concludes with a summary and discussion of related work.

2 A Combinator Language

FL [BWW⁺89] is a combinator-based functional language based on FP [Bac78]. FL is higher-order, dynamically typed, and strict. This section presents the syntax and semantics of MiniFL, a subset of the full FL language.

The following grammar gives the syntax of MiniFL expressions:

$$\begin{aligned} E &::= \text{Atom} \mid \langle E_1, \dots, E_n \rangle \mid E_1 : E_2 \mid E_1 \text{ where } \{ \text{DF Name}_1 \equiv E_2 \dots \text{DF Name}_n \equiv E_n \} \mid E_1 \text{ where PF} \\ \text{Atom} &::= \text{Name} \mid \text{Bool} \mid \text{Int} \mid \text{Char} \\ \text{DF} &::= \text{def} \mid \text{nrdef} \mid \text{type} \end{aligned}$$

Throughout this paper, we use italic font *v* for semantic values and typewriter font **v** for expressions. The MiniFL atoms are booleans (**true**, **false**), integers, and characters (**'a'**, **'b'**, ...). The application of a function **f** to an argument **x** is written **f : x**. Sequences are written $\langle \mathbf{x}_1, \dots, \mathbf{x}_n \rangle$. Strings "abc" are abbreviations for sequences of characters $\langle \mathbf{'a'}, \mathbf{'b'}, \mathbf{'c'} \rangle$. MiniFL provides **where** clauses for function definition and scoping and provides **type** definitions for programmer-defined abstract data types. The type definition **type x $\equiv$ E** defines three functions: a type constructor

mkx, a type destructor **unx**, and a type predicate **isx**. **PF** denotes the environment of primitive functions (see below).

The MiniFL semantic domain consists of $\perp$, atoms, sequences, strict functions, and tagged values. Sequences are written $\langle x_1, \dots, x_n \rangle$ and are also strict, so that $\langle \dots, \perp, \dots \rangle = \perp$. Function application of function f to argument x is denoted $f \cdot x$. The distinction between syntactic application $f : x$ and semantic application $f \cdot x$ will be important; hence, we emphasize the distinction with different notation. A value v with tag tag is denoted $\prec tag, v \succ$. Tagged values are used to denote elements of abstract types.

A MiniFL environment e is a function from names to function values. The MiniFL semantic function maps an expression **exp** and an environment e to a value.

$$\begin{aligned} \mu(f, e) &= e(f) \text{ if } f \text{ is a function name} \\ \mu(a, e) &= a \text{ if } a \text{ is an atom} \\ \mu(\langle x_1, \dots, x_n \rangle, e) &= \langle \mu(x_1, e), \dots, \mu(x_n, e) \rangle \\ \mu(f : x, e) &= e(\text{apply}) \cdot \mu(\langle f, x \rangle, e) \end{aligned}$$

Note that the last clause makes the syntactic “ $\cdot$ ” an infix alias for the meaning of the function name **apply** in the current environment. This makes “ $\cdot$ ” redefinable by the programmer.¹ When the environment is empty or clear from context we write simply $\mu(\text{exp})$.

To give semantics to **where** clauses, we introduce a semantic function ρ that maps a definition list and an environment to a new environment:

$$\rho(\{\dots \text{DF } f_i \equiv E_i \dots\}, e) = \text{least } e' \text{ s.t. } \left\{ \begin{array}{ll} e'(f_i) = \mu(E_i, e') \text{ and } e'(f_i) \geq \lambda x. \perp & \text{for def } f_i \\ e'(f_i) = \mu(E_i, e) \text{ and } e'(f_i) \geq \lambda x. \perp & \text{for nrdef } f_i \\ \left. \begin{array}{l} e'(\text{mk}f_i) = \text{mk}f_i \\ e'(\text{un}f_i) = \text{un}f_i \\ e'(\text{is}f_i) = \text{is}f_i \end{array} \right\} & \text{for type } f_i \\ e'(f) = e(f) & \text{otherwise} \end{array} \right.$$

A **def** is a recursive definition: it binds names in the function body to definitions in the immediate environment, if they exist. An **nrdef** is a non-recursive definition: it binds names in the function body to definitions in the enclosing environment. The condition $e'(f) \geq \lambda x. \perp$ says simply that e' is a solution of the constraints that guarantees f is a function. The meanings of the type functions **mkf**, **unf**, and **isf** are given below. There is a primitive environment PF such that $PF(f) = f$ for all primitive function names f . Figure 1 gives the set of primitive MiniFL functions. Using ρ , we can give meaning to **where** clauses:

$$\mu(E \text{ where } \{\text{Env}\}, e) = \mu(E, \rho(\{\text{Env}\}, e))$$

¹We could take this one step further and make the meaning of sequences redefinable, too. The expression $\langle a, b, c \rangle$ could be an abbreviation for the application **mkseq** : 3 : $a : b : c$, where **mkseq** is a primitive function name. This generalization is not necessary for the examples we present.

$\circ\bar{\cdot}\langle f, g \rangle\bar{\cdot}x = f\bar{\cdot}(g\bar{\cdot}x)$ $K\bar{\cdot}x\bar{\cdot}y = x, \text{ (if } y \neq \perp\text{)}$ $tt\bar{\cdot}y = true, \text{ (if } y \neq \perp\text{)}$ $\Delta\bar{\cdot}f\bar{\cdot}x = \begin{cases} x & \text{if } f\bar{\cdot}x = true \\ \perp & \text{otherwise} \end{cases}$	$cons\bar{\cdot}\langle f_1, \dots, f_n \rangle\bar{\cdot}x = \langle f_1\bar{\cdot}x, \dots, f_n\bar{\cdot}x \rangle$ $id\bar{\cdot}x = x$ $apply\bar{\cdot}\langle f, x \rangle = f\bar{\cdot}x$ $delay\bar{\cdot}f\bar{\cdot}x\bar{\cdot}y = f\bar{\cdot}x\bar{\cdot}y$ $hd\bar{\cdot}\langle y_1, \dots, y_n \rangle = y_1, \text{ (if } y_i \neq \perp\text{)}$
--	--

Figure 1: The MiniFL primitive functions.

$$\mu(E \text{ where } PF, e) = \mu(E, PF)$$

A type definition `type f ≡ E` defines a type constructor *mkf*, a type destructor *unf*, and a type predicate *isf* as follows. Let *e* be the environment of the type definition.

$$\begin{aligned}
mkf\bar{\cdot}y &= \begin{cases} \prec tag_f, y \succ & \text{if } \mu(E, e)\bar{\cdot}y = true \\ \perp & \text{otherwise} \end{cases} \\
unf\bar{\cdot}\prec tag_f, y \succ &= y \\
isf\bar{\cdot}z &= \begin{cases} true & \text{if } z = \prec tag_f, y \succ \\ false & \text{otherwise} \end{cases}
\end{aligned}$$

These functions are the only mechanism available to the programmer for constructing tagged values; the unique tag *tag_f* is chosen by the language implementation and is not accessible by the programmer. (N.B. For succinctness and clarity in this paper, we will often write $\prec f, v \succ$ instead of the more proper *mkf*:*v*.)

To improve readability, we use infix notation for function composition, so that $f \circ g = \circ\bar{\cdot}\langle f, g \rangle$. The meaning of a complete program *P* is $\mu(P, PF)$, so that top-level expressions are implicitly enclosed in a `where PF`; this binds free names to their primitive definitions.

3 Implementing Lambda Abstraction

This section shows how to implement lambda abstraction using only the function definition and environment facilities available in MiniFL. This example has been chosen for two reasons: it provides motivation for characterizing the power of the technique presented in Section 4, and it illustrates the unfamiliar style of programming with combinators by implementing something very familiar to many people.

Consider the program $f \circ g$ (recall that $f \circ g = \circ\bar{\cdot}\langle f, g \rangle$). If *f* is a function name, then in many functional languages lambda abstraction can be used to create a new, parameterized program $\lambda f.(f \circ g)$. To do the same thing in a combinator language, we can perform combinatorial abstraction [HS86]. While this is straightforward, it results in a completely restructured program. For example, the program $\lambda f.(f \circ g)$ becomes $apply \circ cons\bar{\cdot}\langle K\bar{\cdot}\circ, cons\bar{\cdot}\langle id, K\bar{\cdot}g \rangle \rangle$ (assuming *g* does not depend on

f). The essence of the abstraction algorithm is to “lift” the original expression and then modify it in such a way that the argument (in this case f) is passed in to the appropriate positions. Roughly speaking, the redefinition technique simulates combinatorial abstraction, accomplishing the same thing by redefining the primitive functions so that they distribute the argument correctly.

In this section, we examine three equivalent ways to achieve lambda abstraction in a combinator based language. The first method is simply to add λ as a language construct. The second method is to transform expressions (either by hand or automatically) to pass in the argument. The third method is to redefine the primitive functions to pass in the argument. Note that the first method involves only a constant size change to any program using lambda abstraction but requires a language extension. The second method requires no language extension but does require that the entire program (potentially) needs to be rewritten. The third method combines the best of the first two methods in that no language extension is required and only a constant size change is needed for a program to use lambda abstraction.

A *simple* expression is any MiniFL expression without type definitions. For simplicity, we illustrate the three approaches on simple expressions only. The first method, adding lambda abstraction as a new language construct, is straightforward.

Definition 3.1 $\mu(\lambda x.E, e) = \lambda t. \mu(E, e[x \leftarrow t])$

For the second method, we give an abstraction algorithm for transforming one expression into another in a way that accomplishes lambda abstraction. The intuition behind the following construction is that $\text{Abs}(e, (f_1, \dots, f_n))$ is e abstracted with respect to the function names $f_1, \dots, f_n$. More formally, $\text{Abs}(e, (f_1, \dots, f_n))$ is a function such that $\text{Abs}(e, (f_1, \dots, f_n)) : x$ is equal to $e[f_1 \leftarrow f_1 : x, \dots, f_n \leftarrow f_n : x]$.

Definition 3.2 Let e be a simple expression and let fs be a list of function names. Then $\text{Abs}(e, fs)$ is an expression constructed recursively by cases:

$\text{Abs}(x, fs) = x$, if x is a function name in fs

$\text{Abs}(x, fs) = K : y$, if x is a function name not in fs or an atom

$\text{Abs}(e_1 : e_2) = \text{apply} \circ \text{cons} : (\text{Abs}(e_1, fs), \text{Abs}(e_2, fs))$

$\text{Abs}(\langle e_1, \dots, e_n \rangle, fs) = \text{cons} : (\text{Abs}(e_1, fs), \dots, \text{Abs}(e_n, fs))$

$\text{Abs}(e \text{ where } \{ \dots \text{nrdef } f_i \equiv e_i \dots \text{def } f_j \equiv e_j \dots \}, (g_1, \dots, g_k)) =$

$\text{Abs}(e, (g_1, \dots, g_k, \dots, f_i, \dots, f_j, \dots)) \text{ where } \{$

$\dots$

$\text{nrdef } f_i \equiv \text{delay} : \text{Abs}(e_i, (g_1, \dots, g_k))$

$\dots$

$\text{def } f_j \equiv \text{delay} : \text{Abs}(e_j, (g_1, \dots, g_k, \dots, f_i, \dots, f_j, \dots,))$

$\dots$

$\}$

This translation is straightforward except for the introduction of the **delay** function in **where** clauses. To see why the **delay** is required, consider the expression **f where def f ≡ f**. The meaning of this function is $\lambda y. \perp$, the everywhere non-terminating function.² Thus $\lambda x. (\mathbf{f \ where \ def \ f \equiv f}) = \lambda x. \lambda y. \perp$. But without the **delay** in the **where** clause,

$$\begin{aligned} & \text{Abs}(\mathbf{f \ where \ \{def \ f \equiv f\}}, (\mathbf{x})) \\ &= \text{Abs}(\mathbf{f}, (\mathbf{x}, \mathbf{f})) \ \mathbf{where \ \{def \ f \equiv f\}} \\ &= \mathbf{f \ where \ \{def \ f \equiv f\}} \\ &= \lambda x. \perp \end{aligned}$$

The purpose of adding **delay** to the right-hand side of each function definition in the construction of **Abs** is to guarantee that the function “waits” for the extra argument. When the **delay** is included, the example above works correctly:

$$\begin{aligned} & \text{Abs}(\mathbf{f}, (\mathbf{x}, \mathbf{f})) \ \mathbf{where \ \{def \ f \equiv delay:Abs(f, (x, f))\}} \\ &= \mathbf{f \ where \ \{def \ f \equiv delay:f\}} \\ &= \lambda x. \lambda y. \perp \end{aligned}$$

Definition 3.3 The expression $e[x_1 \leftarrow e_1, \dots, x_n \leftarrow e_n]$ is the expression obtained from e by simultaneously replacing each free occurrence of x_i by e_i for $i = 1, \dots, n$. Renaming is performed on e as necessary to avoid name clashes.

Lemma 3.4 If e is an FL expression, then $\text{Abs}(e, (f_1, \dots, f_n)):x = e[f_1 \leftarrow f_1:x, \dots, f_n \leftarrow f_n:x]$

Proof: The proof is by induction on the structure of e . The proof is very similar to Lemma 3.7 and is omitted. $\square$

Corollary 3.5 Let e be an arbitrary FL expression. $\lambda x. e = \text{Abs}(e, (\mathbf{x})) \ \mathbf{where \ \{def \ x \equiv id\}}$

Proof: Both sides of the equation are functions, so it suffices to show they are equal when applied to any argument. This follows from Lemma 3.4. $\square$

Consider the function $\lambda f. f \circ g$ discussed earlier in this section. By Corollary 3.5, this expression is equivalent to $\text{Abs}(f \circ g, (\mathbf{f})) \ \mathbf{where \ \{def \ f \equiv id\}}$. Assuming g is a function name distinct from f , it is easy to check that

$$\begin{aligned} & \text{Abs}(f \circ g, (\mathbf{f})) \ \mathbf{where \ \{def \ f \equiv id\}} \\ &= \text{Abs}(\circ: \langle f, g \rangle, (\mathbf{f})) \ \mathbf{where \ \{def \ f \equiv id\}} \\ &= \text{apply} \circ \text{cons}: \langle K: \circ, \text{cons}: \langle f, K: g \rangle \rangle \ \mathbf{where \ \{def \ f \equiv id\}} \\ &= \text{apply} \circ \text{cons}: \langle K: \circ, \text{cons}: \langle id, K: g \rangle \rangle \end{aligned}$$

which is exactly the combinatorial abstraction of $\lambda f. f \circ g$.

²Recall that the meaning of **def f ≡ f** is the least function satisfying the equation $f \equiv f$ (see Section 2).

To accomplish the third method of implementing lambda abstraction without rewriting the program, we define a non-standard environment of primitive functions. The idea is to redefine the primitive functions to simulate combinatorial abstraction. To do this, function applications inside of a lambda expression must be delayed and the structure of the expression “remembered” so that a lambda-bound argument is correctly substituted when it is applied. There are three components in the implementation presented below: a new abstract data type which is used to distinguish a lambda-bound argument from other values, a function `suspend` that delays function applications, and a function `abstr` that simulates combinatorial abstraction.

Technically, we are defining an environment L that contains definitions for the functions `suspend` and `abstr`, among others. Instead of writing out this environment explicitly each time, the subscript L is used on function names to indicate that they are enclosed by a `where` clause that defines the environment L , so that $f_L = f$ where L .

We introduce a type definition `type lamb ≡ tt`. Since `tt:x = true` for any value x (except $\perp$), the data type `lamb` simply provides a way of tagging an arbitrary value. The data type `lamb` serves to identify lambda-bound arguments.

The function `suspendL` delays a function application until an argument of type `lamb` is supplied. The function `suspauxL` handles the case of a suspended application applied to an additional argument: if the argument is not of type `lamb`, the additional application is suspended as well.

The function `abstr` simulates combinatorial abstraction by propagating the lambda argument through the structure of the expression and performing any suspended applications.

$$\begin{aligned} \text{suspend}_L: x.y &= \begin{cases} \text{abstr}_L: x.y & \text{if } y = \langle \text{lamb}, z \rangle \\ \text{suspaux}_L: x.y & \text{otherwise} \end{cases} \\ \text{suspaux}_L: x.w.y &= \begin{cases} (\text{abstr}_L: x.y):(\text{abstr}_L: w.y) & \text{if } y = \langle \text{lamb}, z \rangle \\ \text{suspaux}_L: (\text{suspaux}_L: x.w):y & \text{otherwise} \end{cases} \\ \text{abstr}_L: x.y &= \begin{cases} x.y & \text{if } x \text{ is a function} \\ x & \text{if } x \text{ is an atom} \\ \langle \text{abstr}_L: x_1.y, \dots, \text{abstr}_L: x_n.y \rangle & \text{if } x = \langle x_1, \dots, x_n \rangle \end{cases} \end{aligned}$$

There is one more device needed to handle recursively defined functions. As with the definition of `Abs` above, it is necessary to ensure that the right-hand sides of recursive function definitions are delayed inside the scope of a lambda abstraction. We introduce a new primitive function `func` and extend PF with $PF(\text{func}) = id$. A definition `(nr)def f ≡ E` is treated as if it had been written `(nr)def f ≡ func.E`. Thus, by redefining the function name `func`, the evaluation of the right-hand of function definitions can be controlled.

Definition 3.6 `Redef(e, (f1, ..., fn))` is the expression

$$(e \text{ where } \{ \dots, \text{nrdef } g \equiv \text{suspend}_L: (K.g), \dots \} \text{ where } \{ \text{def func} \equiv \text{delay}_L \circ \text{abstr}_L \})$$

where g ranges over all free function names in e except for $f_1, \dots, f_n$.

Intuitively, Definition 3.6 is to the implementation of lambda abstraction via function redefinition as Definition 3.2 is to the technique of combinatorial abstraction. The following lemma is the analog of Lemma 3.4.

Lemma 3.7 If $(f_1, \dots, f_n)$ does not include the name `apply`, then

$$\text{abstr}_L: \text{Redef}(e, (f_1, \dots, f_n)): (\text{mklamb}: x) = e[f_1 \leftarrow f_1: (\text{mklamb}: x), \dots, f_n \leftarrow f_n: (\text{mklamb}: x)]$$

Proof: The proof is by induction on the structure of e . Let fs be the list of functions $(f_1, \dots, f_n)$. Let subx be the substitution $[f_1 \leftarrow f_1: (\text{mklamb}: x), \dots, f_n \leftarrow f_n: (\text{mklamb}: x)]$. Only the two most interesting cases, applications and where clauses, are presented; the other cases are left to the reader.

Let e be the application $e_1: e_2$.

$$\begin{aligned} & \text{abstr}_L: \text{Redef}(e, fs): (\text{mklamb}_L: x) \\ &= \text{abstr}_L: \text{Redef}(e_1: e_2, fs): (\text{mklamb}_L: x) \\ &= \text{abstr}_L: (\text{Redef}(\text{apply}, fs): \text{Redef}(\langle e_1, e_2 \rangle, fs)): (\text{mklamb}_L: x) \end{aligned}$$

The previous line is simply the standard semantics of infix application. The next step substitutes the definition of `apply` in the environment provided by `Redef`.

$$\begin{aligned} &= \text{abstr}_L: (\text{suspend}_L: (K: \text{apply}): \langle \text{Redef}(e_1, fs), \text{Redef}(e_2, fs) \rangle): (\text{mklamb}_L: x) \\ &= \text{abstr}_L: (\text{suspau}_L: (K: \text{apply}): \langle \text{Redef}(e_1, fs), \text{Redef}(e_2, fs) \rangle): (\text{mklamb}_L: x) \\ &= \text{suspau}_L: (K: \text{apply}): \langle \text{Redef}(e_1, fs), \text{Redef}(e_2, fs) \rangle: (\text{mklamb}_L: x) \end{aligned}$$

The previous two lines are calculations using first the definition of `suspendL` and then the definition of `abstrL`. In the next two steps `suspauL` first propagates the lambda argument to the subexpressions and then `abstrL` performs the suspended application:

$$\begin{aligned} &= (\text{abstr}_L: (K: \text{apply}): (\text{mklamb}_L: x)): (\text{abstr}_L: \langle \text{Redef}(e_1, fs), \text{Redef}(e_2, fs) \rangle: (\text{mklamb}_L: x)) \\ &= K: \text{apply}: (\text{mklamb}_L: x): \langle \text{abstr}_L: \text{Redef}(e_1, fs): (\text{mklamb}_L: x), \text{abstr}_L: \text{Redef}(e_2, fs): (\text{mklamb}_L: x) \rangle \end{aligned}$$

Computing using the definition of K we get

$$= \text{apply}: \langle \text{abstr}_L: \text{Redef}(e_1, fs): (\text{mklamb}_L: x), \text{abstr}_L: \text{Redef}(e_2, fs): (\text{mklamb}_L: x) \rangle$$

To complete the proof for application, the induction hypothesis is applied to the components of the pair. Because `apply` is not in fs , the substitution subx can be moved so that it applies to the entire expression.

$$\begin{aligned} &= \text{apply}: \langle e_1[\text{subx}], e_2[\text{subx}] \rangle \\ &= \text{apply}: \langle e_1, e_2 \rangle[\text{subx}] \\ &= e[\text{subx}] \\ &= e[f_1 \leftarrow f_1: (\text{mklamb}: x), \dots, f_n \leftarrow f_n: (\text{mklamb}: x)] \end{aligned}$$

This completes the proof for the application case. To simplify notation in the `where` clause case, we assume that fs contains only a single function name and that there is only one definition in the `where` clause.

$$\begin{aligned}
& \text{abstr}_L:\text{Redef}(e \text{ where } \{\text{def } f_0 \equiv e_0\}, (f_1)):(\text{mklamb}_L:x) \\
= & \text{abstr}_L:\text{Redef}(e, (f_0, f_1)):(\text{mklamb}_L:x) \text{ where } \{\text{def } f_0 \equiv \text{Redef}(\text{func}_L:e_0, (f_0, f_1))\} \\
= & \text{abstr}_L:\text{Redef}(e, (f_0, f_1)):(\text{mklamb}_L:x) \text{ where } \{\text{def } f_0 \equiv \text{delay}_L:(\text{abstr}_L:\text{Redef}(e_0, (f_0, f_1)))\}
\end{aligned}$$

The first step distributes `Redef` over the `where` clause; for this step to be correct, f_1 must not be captured by `Redef`, so it is added to list `fs`. The second step substitutes in the (re)definition of `func`.

The next step exploits the fact that given a definition `def f0 = delay:E`, the expression `f0:y` always represents a function (because of the `delay`) and hence the function `def F0 = E:y` is equal to `f0:y`. The function `F0` does not require an explicit `delay`; the delay is built into the semantics of MiniFL.

$$\begin{aligned}
= & \text{abstr}_L:\text{Redef}(e, (f_0, f_1)):(\text{mklamb}_L:x) \text{ where} \\
& \quad \{\text{def } f_0 \equiv \text{delay}_L:(\text{abstr}_L:\text{Redef}(e_0, (f_0, f_1))) \\
& \quad \text{def } F_0 \equiv (\text{abstr}_L:\text{Redef}(e_0, (f_0, f_1)))(\text{mklamb}_L:x)\} \\
= & e[f_0 \leftarrow f_0:(\text{mklamb}_L:x), f_1 \leftarrow f_1:(\text{mklamb}_L:x)] \text{ where} \\
& \quad \{\text{def } f_0 \equiv \text{delay}_L:(\text{abstr}_L:\text{Redef}(e_0, (f_0, f_1))) \\
& \quad \text{def } F_0 \equiv e_0[f_0 \leftarrow f_0:(\text{mklamb}_L:x), f_1 \leftarrow f_1:(\text{mklamb}_L:x)]\}
\end{aligned}$$

The previous step follows by induction. The next step substitutes `F0` for `f0: (mklambL:x)`. These expressions are equal for the reasons discussed above. After this transformation, the definition of `f0` is not used in `e` and can be dropped.

$$\begin{aligned}
= & e[f_0 \leftarrow F_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)] \text{ where} \\
& \quad \{\text{def } f_0 \equiv \text{delay}_L:\text{abstr}_L:\text{Redef}(e_0, (f_0, f_1)) \\
& \quad \text{def } F_0 \equiv e_0[f_0 \leftarrow F_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)]\} \\
= & e[f_0 \leftarrow F_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)] \text{ where} \\
& \quad \{\text{def } F_0 \equiv e_0[f_0 \leftarrow F_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)]\}
\end{aligned}$$

The final steps rename `F0` to `f0` and simplify the substitution:

$$\begin{aligned}
= & e[f_0 \leftarrow f_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)] \text{ where } \{\text{def } f_0 \equiv e_0[f_0 \leftarrow f_0, f_1 \leftarrow f_1:(\text{mklamb}_L:x)]\} \\
= & e[f_1 \leftarrow f_1:(\text{mklamb}_L:x)] \text{ where } \{\text{def } f_0 \equiv e_0[f_1 \leftarrow f_1:(\text{mklamb}_L:x)]\} \\
= & (e \text{ where } \{\text{def } f_0 \equiv e_0\})[f_1 \leftarrow f_1:(\text{mklamb}_L:x)]
\end{aligned}$$

□

To finish the implementation of lambda abstraction, a mechanism for delimiting the scope of a lambda bound variable is needed. The function `lambdaL:f` takes any argument, injects it into type `lamb`, and then applies an abstracted version of `f` to the tagged value:

$$\text{lambda}_L:f = \text{abstr}_L:f \circ \text{mklamb}_L$$

Finally, it is possible faithfully to implement lambda abstraction.

Theorem 3.8 $\lambda x.e \equiv \text{lambda}_L: (\text{Redef}(e, (x)))$ where $\{\text{def } x \equiv \text{unlambda}_L\}$

Proof: Both sides of the equation are functions, so it suffices to show that they are equal when applied to any argument, which follows from Lemma 3.7. $\square$

4 A Characterization

Section 3 shows how to implement a specific “non-standard semantics”, *i.e.*, lambda abstraction, in MiniFL. In this section, we generalize this example and show that a very large class of non-standard semantics can be implemented within MiniFL.³

We begin by showing that it is possible to define a “universal” environment UPF that redefines the primitives to produce a parse tree of an expression. In this context, a parse tree is any expression that can be derived by unfolding function definitions. Because such parse trees may be infinite (due to the unfolding of recursive definitions), it is necessary to work with finite approximations.

We show that there exists a MiniFL function `parse` and an environment UPF such that for any user expression e , the value `parse:⟨e where UPF, i⟩` is a parse tree for e unfolded to depth i . The construction uses functions `suspendU`, `suspauU`, and `parse` and introduces new types `pdepth` (to track the depth of a parse tree), `app` (to distinguish applications in a parse tree), `fn` (to distinguish functions in a parse tree), and a distinguished value `bot` (to indicate where the depth of the full parse tree exceeds the depth of the approximation).⁴

The construction is reminiscent of the construction for lambda abstraction. The functions `suspendU` and `suspauU` delay applications in exactly the same way as `suspendL` and `suspauL`. The function `parse` plays the same role as `abstrL`, only instead of performing delayed applications, `parse` builds a data structure that represents a parse tree of the original expression.

$$\begin{aligned} \text{suspend}_U: x: y &= \begin{cases} \text{parse}: \langle x, i \rangle & \text{if } y = \prec \text{pdepth}, i \succ \\ \text{suspau}_U: x: y & \text{otherwise} \end{cases} \\ \text{suspau}_U: x: w: y &= \begin{cases} \prec \text{app}, \langle \text{parse}: \langle x, i-1 \rangle, \text{parse}: \langle w, i-1 \rangle \rangle \succ & \text{if } y = \prec \text{pdepth}, i \succ \\ \text{suspau}_U: (\text{suspau}_U: x: w): y & \text{otherwise} \end{cases} \\ \text{parse}: \langle x, i \rangle &= \begin{cases} \text{bot} & \text{if } i = 0 \\ x: \prec \text{pdepth}, i \succ & \text{if } x \text{ is a function} \\ \langle \dots, \text{parse}: \langle x_j, i-1 \rangle, \dots \rangle & \text{if } x = \langle \dots, x_j, \dots \rangle \\ x & \text{otherwise} \end{cases} \end{aligned}$$

Let UPF be the environment $\{\dots \text{def } p \equiv \text{suspend}_U: \prec \text{fn}, "p" \succ \dots \text{nrdef } \text{func} \equiv \text{suspend}_U\}$ where p ranges over all the primitive function names.

³The constructions required to characterize this class are fairly technical. The reader may wish first to browse Section 5, which describes practical experience with this technique.

⁴The value `bot` can be defined as a unique element of another abstract type.

The following examples illustrate the construction of parse trees. Consider the expression $\text{hd} \circ \text{id}$. In the worst approximation, `parse` returns no information about the expression:

$$\text{parse:} \langle \text{hd} \circ \text{id} \text{ where UPF}, 0 \rangle = \text{bot}$$

If the second parameter is 1, the approximation reveals that the expression is an application. Let

$$\mathbf{X} = \langle \text{hd}, \text{id} \rangle \text{ where UPF} = \langle \text{suspendy:} \prec \text{fn}, " \text{hd} " \succ, \text{suspendy:} \prec \text{fn}, " \text{id} " \succ \rangle$$

Calculating using the definitions above:

$$\begin{aligned} & \text{parse:} \langle \text{suspendy:} \prec \text{fn}, " \circ " \succ, \mathbf{X}, 1 \rangle \\ &= \text{suspendy:} \prec \text{fn}, " \circ " \succ, \mathbf{X}: \prec \text{pdepth}, 1 \succ \\ &= \prec \text{app}, \langle \text{parse:} \langle \prec \text{fn}, " \circ " \succ, 0 \rangle, \text{parse:} \langle \mathbf{X}, 0 \rangle \rangle \succ \\ &= \prec \text{app}, \langle \text{bot}, \text{bot} \rangle \succ \end{aligned}$$

Higher values for the second parameter yield better approximations.

$$\begin{aligned} \text{parse:} \langle \text{hd} \circ \text{id} \text{ where UPF}, 2 \rangle &= \prec \text{app}, \langle \prec \text{fn}, " \circ " \succ, \langle \text{bot}, \text{bot} \rangle \rangle \succ \\ \text{parse:} \langle \text{hd} \circ \text{id} \text{ where UPF}, 3 \rangle &= \prec \text{app}, \langle \prec \text{fn}, " \circ " \succ, \langle \prec \text{fn}, " \text{hd} " \succ, \prec \text{fn}, " \text{id} " \succ \rangle \rangle \succ \end{aligned}$$

Given the ability to generate a parse tree, it seems intuitively clear that any non-standard semantic function γ can be implemented in MiniFL. However, there are two problems. The first problem is that MiniFL may lack sufficient power to compute γ (e.g., γ might be non-recursive). Hence, we restrict our attention to functions γ that are computable in MiniFL.⁵ The second problem is that γ may not be referentially transparent, or may otherwise use too much information about the form of the expression (e.g., γ could count the number of characters in the program text). To avoid this problem, we further restrict our attention to the class of *reasonable* semantics, which is defined in terms of transformations that mimic the expansion of a parse tree. In Definition 4.1, part 1 enforces referential transparency, part 2 requires that the semantics be invariant under removal of dead code, and part 3 guarantees that a reasonable semantics is a monotonic, least fixed-point semantics.

Definition 4.1 (Reasonable Semantics) Let P_0 and P_1 be MiniFL programs and let γ be any semantic function. Then γ is *reasonable* iff

$$\gamma(P_0) = \bigsqcup \{ \gamma(P_1) \mid P_0 \rightarrow P_1 \text{ and } P_1 \text{ has no user-defined function names} \}$$

where $\rightarrow$ is the least reflexive, transitive relation such that the following all hold:

1. $P_0 \rightarrow P_1$ if P_1 is obtained from P_0 by substituting a function body for its name.

⁵In this paper, the term “computable” means computable by the underlying language (in this case, MiniFL). Since MiniFL is not expressively complete, this notion of “computable” may be less than one might desire but is sufficient for our purpose here.

2. e_0 where $\{(nr)def\ f_1 \equiv e_1 \dots (nr)def\ f_n \equiv e_n\} \rightarrow e_0$ if e_0 contains no occurrences of any of the user function names $f_1, \dots, f_n$.
3. $P_0 \rightarrow P_1$ if P_1 is obtained from P_0 by substituting bot for the occurrence of a function name.

Theorem 4.2 Let γ be any reasonable semantic function computable in MiniFL. Then there exists a MiniFL context $E()$ such that $\gamma(P) = \mu(E(P))$ for all simple MiniFL expressions P .

The basic idea of the proof of Theorem 4.2 is that by using `parse` to generate sufficiently large unfoldings of an expression e , the function γ can be used to compute increasingly accurate approximations to the meaning of $\gamma(e)$. Throughout the argument it is necessary to distinguish carefully between an expression (a piece of syntax) and a value (an element of the semantic domain). Before the crux of the proof can be presented there are three technical problems that must be overcome.

The first problem is that the parsing function `parse` does not produce expressions but rather values that represent expressions. There is a one to one correspondence between the structures produced by `parse` and FL expressions. This correspondence is made explicit by a function σ that maps values produced by `parse` to the FL expressions used in Definition 4.1:

$$\begin{aligned}
\sigma(\prec \text{app}, \langle f, x \rangle \succ) &= \sigma(f) : \sigma(x) \\
\sigma(\langle x_1, \dots, x_n \rangle) &= \langle \sigma(x_1), \dots, \sigma(x_n) \rangle \\
\sigma(bot) &= bot \\
\sigma(\prec fn, "p" \succ) &= p \\
\sigma(x) &= x \text{ otherwise}
\end{aligned}$$

The function σ maps a parse tree value into the parse tree expression it represents. We will also need a semantic counterpart of σ that maps a parse tree value v into the value denoted by $\sigma(v)$. Let σ' be a function that maps values to values; σ' has the same definition as σ except that the objects on the right-hand side of the equations are interpreted as values rather than expressions and $\sigma'(bot) = \perp$.

The second problem is that the definition of `parse` given above does not generate all finite parse trees, but only the subset of the finite parse trees where all subexpressions are expanded to the same depth. The definition of a reasonable semantics (Definition 4.1) is given in terms of all finite parse trees. To prove the theorem, the following equation is needed:

$$\{\sigma(\text{parse} : \langle e \text{ where UPF}, i \rangle) \mid i \geq 0\} = \{e' \mid e \rightarrow e' \text{ and } e' \text{ has no user function names}\} \quad (1)$$

We assume a definition of `parse` that satisfies the above equation. It is sufficient to redefine `parse` to enumerate all possible finite parse trees in some order. This is clearly computable and the details are omitted.

The third problem is that the semantic function γ maps arbitrary expressions to arbitrary values. However, the function `parse` generates values that represent parse trees. To carry out

the plan of using approximations of the full parse tree to simulate γ , it is necessary to have an interpreter for γ that maps parse tree values to parse tree values. Not surprisingly, this interpreter can be defined by a non-standard interpretation of γ . Since γ is computable in MiniFL, there is an expression Γ such that $\mu(\Gamma) = \gamma$. We require a context E_0 such that for all i

$$\sigma'(\mu(E_0(\Gamma, \text{parse:} \langle e \text{ where UPF, } i \rangle))) = \gamma(\sigma(\text{parse:} \langle e \text{ where UPF, } i \rangle))$$

and such that $\sigma(\mu(E_0(\Gamma, \text{parse:} \langle e \text{ where UPF, } i \rangle)))$ is either **bot** or in *normal form*. An expression is in normal form if no applications can be reduced using the equations in Figure 1. For the purposes of this proof it is assumed that E_0 exists. An explicit construction of E_0 is presented in Section 5.

Proof: [of Theorem 4.2] Define $\gamma_i(e) = \mu(E_0(\Gamma, \text{parse:} \langle e \text{ where UPF, } i \rangle))$. The key step of the proof is to define a context $E(e) = \bigsqcup_i \sigma'(\gamma_i(e))$. The intuition behind the construction of $E()$ is that $E()$ repeatedly computes $\gamma_i(e)$ for increasing values of i . If $\gamma(e)$ is a value other than $\perp$, then $\gamma_i(e)$ evaluates to some value other than **bot** for some finite value of i . By construction, the value $\gamma_i(e)$ is a representation of a normal form; by analyzing this normal form it is possible to recursively compute the meaning of $\gamma(e)$. If $\gamma_i(e)$ is an atom, then $E(e)$ is that atom and the procedure terminates. Otherwise, if $\gamma_i(e)$ has the form of a sequence the procedure is applied recursively to compute the meaning of each component. If $\gamma_i(e)$ is a function then the procedure terminates returning a function whose meaning is $\gamma(e)$.

To define $E()$ MiniFL is extended with a family of selector functions $s_1, \dots, s_n$ such that $s_i: \langle x_1, \dots, x_n \rangle = x_i$ if $i \leq n$. Because several other new primitive functions are required to define γ' entirely in MiniFL, the definition is once again by cases.

$$\begin{aligned} E(e) = & \\ & \text{if } \gamma_i(e) = \text{bot for all } i \text{ then } \perp \\ & \text{otherwise choose min } i \text{ s.t. } \gamma_i(e) = x \neq \text{bot} \\ & \quad \text{if } x = a \quad \quad \quad \text{then } a \quad \text{if } a \text{ is an atom} \\ & \quad \text{if } x = \langle x_1, \dots, x_n \rangle \quad \text{then } \langle E(s_1:e), \dots, E(s_n:e) \rangle \\ & \quad \text{if } x = \langle \text{app}, \langle f, y \rangle \rangle \quad \text{then } \lambda z. E(\langle \text{app}, \langle e, z \rangle \rangle) \\ & \quad \text{if } x = \langle \text{fn}, "p" \rangle \quad \text{then } \lambda z. E(\langle \text{app}, \langle e, z \rangle \rangle) \end{aligned}$$

In this definition it is necessary to consider the case $\gamma_i(e) = \langle \text{app}, \langle f, y \rangle \rangle$ because some functions in normal form are represented by applications (e.g., consider the expression `delay:f`). The proof that $E(e) = \bigsqcup_i \sigma'(\gamma_i(e))$ is by induction on the finite elements of the semantic domain. The argument is straightforward but requires introducing much new notation, so the details are omitted.

To complete the proof of the theorem, note that

$$\begin{aligned} & E(e) \\ = & \bigsqcup_i \sigma'(\gamma_i(e)) \\ = & \bigsqcup_i \sigma'(\mu(E_0(\Gamma, \text{parse:} \langle e \text{ where UPF, } i \rangle))) && \text{definition of } \gamma_i \\ = & \bigsqcup_i \gamma(\sigma(\text{parse:} \langle e \text{ where UPF, } i \rangle)) && \text{definition of } E_0 \\ = & \bigsqcup \{ \gamma(e') \mid e \rightarrow e' \text{ and } e' \text{ has no user function names} \} && \text{by (1)} \\ = & \gamma(e) && \text{by Definition 4.1} \end{aligned}$$

□

5 Compile-Time Evaluation

Theorem 4.2 shows that a very large class of non-standard semantics can be implemented entirely within a combinator-based functional language. We have used the technique of defining a non-standard environment of primitive functions to implement several language features, including state, continuations, and exceptions. In practice, nothing nearly as complex and inefficient as the construction in the proof of Theorem 4.2 is required to implement desired features. This section describes in detail how compile-time evaluation of expressions is implemented in an FL compiler. This application is particularly well-suited to the technique of defining a non-standard semantics.

A common optimization performed by many compilers is to evaluate expressions at compile-time whenever possible. For example, if the expression $+(2, 3)$ appears in a program it is a good idea to replace it by the equivalent expression 5. In general, the evaluated result may be a function; for example, $K:(K:f):g$ evaluates to $K:f$.

Evaluating expressions at compile-time means replacing an application $f:x$ by its normal form. (Recall that a normal form is an expression in which no applications can be reduced—see Section 4). While conceptually this is a trivial task, it is anything but trivial to implement. Simply computing $\mu(f:x)$ is not correct, since $\mu(f:x)$ is a value, not an expression. For example, in the case of $K:(K:f):g$ the value $\mu(K:(K:f):g)$ is a function, so it is probably represented in an implementation as a pointer to a piece of assembly code. What is required for compile-time evaluation, however, is a parse tree for the expression $K:f$. Thus, in general it is not possible to exploit an existing compiler or interpreter for a language to implement compile-time evaluation for the language, because the existing language processor has the wrong functionality; i.e., it does not produce values that match the representation of expressions used in the language processor. This makes implementing compile-time evaluation in the obvious way very tedious, because it is necessary to reimplement the language with a slightly non-standard semantics (mapping expressions to expressions) that is isomorphic to the standard semantics. Even worse, it makes the compiler harder to maintain, because whenever there is a change in the language there are two separate pieces of code—the standard implementation and the compile-time evaluator—that must be consistently modified.

Implementing compile-time evaluation turns out to be an ideal application for the idea of defining a non-standard semantics for a language. In fact, we suspect that in practice compile-time evaluation is often implemented using “tricks” that essentially simulate what we describe here. Compile-time evaluation of FL expressions is implemented by writing an environment of primitive functions EPF and a function `deparse` such that `deparse:(e where EPF)` is the parse tree of a normal form expression whose meaning is $\mu(e)$. More precisely, `e where EPF` and `deparse` have the following properties:

- If $\mu(e) \neq \perp$, then `deparse:(e where EPF) = p` where p is a parse tree such that $\sigma'(p) = \mu(e)$ and $\sigma(p)$ is a normal form.
- `deparse:f = f: < deparse, 0 >` for any function f

(Definitions of σ and σ' are given in Section 4.) The details that make this work are similar in spirit

to the construction for lambda abstraction (Section 3). Every primitive function is overloaded so that it can perform either “normal” computation or produce a parse tree of itself. For example, the primitive function `hd` is redefined as follows:

$$\begin{aligned} \text{hd}_{\text{EPF}}:\langle \text{deparse}, x \rangle &= \langle \text{fn}, "hd" \rangle \\ \text{hd}_{\text{EPF}}:x &= \text{hd}_{\text{PF}}:x && \text{otherwise} \end{aligned}$$

Recall that `PF` is the environment of primitive functions, so that `hdPF` means `hd` where `PF`, thus binding `hd` to its primitive definition. For consistency, parse trees are represented using the notation introduced in Section 4. Other first-order primitives are defined analogously. For higher-order functions, the redefinition depends on the order of the function. Let `D` stand for the set of `deparse` values $\langle \text{deparse}, y \rangle$. The primitive `apply` is a second-order function:

$$\begin{aligned} \text{apply}_{\text{EPF}}:d &= \langle \text{fn}, "apply" \rangle && \text{if } d \in D \\ \text{apply}_{\text{EPF}}:f:d &= \langle \text{app}, \langle \text{fn}, "apply" \rangle, f:d \rangle && \text{if } f \notin D \text{ and } d \in D \\ \text{apply}_{\text{EPF}}:f:x &= f:x && \text{otherwise} \end{aligned}$$

In the definition of `applyEPF` given above, note in the second clause that `f` is applied to a `deparse` value to obtain the parse tree for `f`.

FL has a large number of primitive functions, and it would be very tedious to write out the definition of every primitive in the style outlined above. Fortunately, the definitions of `hdEPF` and `applyEPF` have enough similarity of structure that it is worthwhile introducing a higher-order function that abstracts out the common part. In our implementation of compile-time evaluation, all definitions in `EPF` are written using a single higher-order function `wrap` that takes five arguments: the primitive function `f`, a list of arguments to `f`, the name "`f`" of `f`, the arity of `f`, and a new argument for `f` or a `deparse` value.

$$\begin{aligned} \text{wrap}:f:\langle \rangle:"f":i:d &= \langle \text{fn}, "f" \rangle && \text{if } d \in D \\ \text{wrap}:f:\langle x_1, \dots, x_n \rangle:"f":i:d &= \langle \text{app}, \langle \text{wrap}:f:\langle x_1, \dots, x_{n-1} \rangle:"f":i:d, x_n:d \rangle \rangle && \text{if } d \in D \\ \text{wrap}:f:\langle x_1, \dots, x_{n-1} \rangle:"f":i:x_n &= \text{wrap}:f:\langle x_1, \dots, x_n \rangle:"f":i && \text{if } n < i \text{ and } x_n \notin D \\ \text{wrap}:f:\langle x_1, \dots, x_{n-1} \rangle:"f":i:x_n &= f:x_1:\dots:x_n && \text{if } n = i \text{ and } x_n \notin D \end{aligned}$$

Using `wrap`, the functions `hdEPF` and `applyEPF` can be defined very succinctly.

$$\begin{aligned} \text{nrdef } \text{hd} &\equiv \text{wrap}: \text{hd}:\langle \rangle:"hd":1 \\ \text{nrdef } \text{apply} &\equiv \text{wrap}: \text{apply}:\langle \rangle:"apply":2 \end{aligned}$$

In general, the definition of `fEPF` for an order `i` primitive `f` is

$$\text{nrdef } f \equiv \text{wrap}:f:\langle \rangle:"f":i$$

This implementation of compile-time evaluation solves the problems discussed at the beginning of the section. First, the implementation is easy. The interesting function `wrap` can be written in less

than 10 lines of FL code. Redefining each of the primitives using `wrap` is trivial. The only missing piece, the definition of `deparse`, is equally short:

$$\begin{aligned} \text{deparse:} a &= a && \text{if } a \text{ is an atom} \\ \text{deparse:} \langle x_1, \dots, x_n \rangle &= \langle \text{deparse:} x_1, \dots, \text{deparse:} x_n \rangle \\ \text{deparse:} f &= f \prec \text{deparse}, 0 \succ && \text{if } f \text{ is a function} \end{aligned}$$

Perhaps more important than the ease of implementation, this design depends minimally on the precise semantics of the primitives. So long as the arity of primitives is fixed, the meaning of the primitives can be changed and the compile-time evaluator does not have to be modified.

There is one potentially serious objection to implementing compile-time evaluation in this way. The performance of an implementation along these lines may be terrible because the layer of redefinition adds an enormous overhead at run-time. In fact, the first time EPF was used in program optimization the performance was terrible. In that first experiment, the FL compiler was used to apply compile-time evaluation to the environment EPF itself. That is, the meaning of EPF where EPF was computed to reduce all expressions in EPF to normal form and thereby improve the performance of EPF. This optimization is very similar in spirit to *partial evaluation* [BEJ88], but note that there is no program analysis involved. The compiler simply evaluates any expression that is not in normal form.

Compile time evaluation of EPF, combined with other optimizations in the FL compiler, was sufficient to make a dramatic difference in performance. At present, compile-time evaluation has been used for over two years on a daily basis in the FL compiler and performance has never been a problem. In the current compiler it is estimated that only about 1% of total compilation time is spent evaluating expressions.

6 Related Work and Conclusions

Several researchers have considered the problem of adding features to functional languages. Wadler [Wad85] and Spivey [Spi90] are the closest to our approach in spirit. They show how to use lists and abstract data types to realize exceptions in a lazy language. Our work is a natural generalization of these examples.

A number of efforts have focussed on adding state to a functional language in a particular way. These approaches include new type systems [Laf88, Wad90b] and mixed imperative/functional languages [Ode91, SRI91].

More recently, Wadler has proposed another approach based on monads [Wad90a]. Briefly, Wadler augments the functional language Haskell [HWA⁺88] with new syntax that specifies a monad in which the text of the program should be interpreted; monads are powerful enough to express many useful features (e.g., state and exceptions). This approach has been used to good effect in a compiler for Haskell [PJW93], where, in fact, the features are programmed directly within Haskell rather than in an exotic syntax.

Using environments to implement non-standard semantics is not a new way of structuring software in FL. What is new is the realization that this paradigm is general and that it can be used to modify existing code with a minimum of reimplementation. To gain experience with this technique a compile-time expression evaluator for an FL compiler has been designed and implemented using a non-standard semantics written in FL. The results are surprising. The idea of writing a non-standard semantics led to a small, simply structured implementation. Program optimization (including applying the expression evaluator to itself) resulted in performance sufficient to allow the evaluator to be used routinely in program compilation.

As stated in the introduction, our technique for adding features to a language is very general. It is, for example, more general than monads; there is no analog of Theorem 4.2 for monads. This advantage easily can be read as a disadvantage. Using monads again as a point of comparison, monads are highly structured and restricted to “sensible” things, whereas our technique is essentially unstructured and unrestricted. As Theorem 4.2 shows, it is possible to program some very powerful (and potentially bad) features. For this reason, despite the strong theoretical properties of the technique, more work is necessary to determine whether there are useful restrictions of the ideas presented here that maintain something of the expressive power while providing a better structuring mechanism.

7 Acknowledgements

Thanks to Matthias Felleisen, Joe Halpern, Phil Wadler, and Jennifer Widom for their comments on an earlier draft.

References

- [AWW90] A. Aiken, J. H. Williams, and E. L. Wimmers. Program transformation in the presence of errors. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 210–217, January 1990.
- [Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [BEJ88] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
- [BWW⁺89] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, and A. Aiken. The FL language manual parts 1 and 2. Technical Report RJ 7100 (67163), IBM, 1989.
- [Fel90] M. Felleisen. On the expressive power of programming languages. In *Proceedings of the 1990 European Symposium on Programming*, pages 134–151. Springer Verlag Lecture Notes in Computer Science no. 432, 1990.

- [HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ -Calculus*. Cambridge University Press, 1986.
- [Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):99–107, April 1989.
- [HWA⁺88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. P. Jones, R. Nikhil, M. Reeve, D. Wise, and J. Young. Report on the functional programming language Haskell. Technical Report DCS/RR-666, Yale University, December 1988.
- [Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
- [Ode91] M. Odersky. How to make destructive updates less destructive. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–36, January 1991.
- [PJW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, January 1993.
- [Rie91] J. G. Riecke. Fully abstract translations between functional languages. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, January 1991.
- [Spi90] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.
- [SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *Proceedings of the 1991 Symposium on Functional Programming and Computer Architecture*, pages 192–214, August 1991.
- [Wad85] P. Wadler. How to replace failure by a list of successes. In *Proceedings of the Second International Conference on Functional Programming Languages and Computer Architecture*. Springer Verlag Lecture Notes in Computer Science no. 201, 1985.
- [Wad90a] P. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, June 1990.
- [Wad90b] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.
- [WW88] J. H. Williams and E. L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 169–179, January 1988.

