Call by Effect

Kevin Ellis
Catlin Gabel

Abstract. Evaluation strategies specify rules for evaluation of terms in a programming language. That the choice of evaluation strategy affects the expressiveness of a language is well known: lazy languages facilitate, for example, streams, whereas strict languages typically allow unconstrained side effects. We present a new evaluation strategy, call by effect, for reconciling side effects with lazy evaluation. In the abstract sense, it behaves like lazy evaluation for pure arguments to functions and behaves like strict evaluation for arguments which may have a side effect. At the core of our evaluation strategy are effect types, which infer when to reduce an argument before substitution. We prove that effects occur as they would under call by value evaluation. We discuss our working implementation of an interpreter for a language with call by effect semantics. Our main result is a new evaluation strategy with semantics which blend lazy and strict evaluation.

1 Introduction

1.1 Motivation and Example

Programming in a lazy language is entirely different from programming in a strict one. With a lazy language, we have the advantages of infinite data structures, the ability to define new control primitives as functions, and stronger normalization guarantees.[11]

However, lazy evaluation and state do not often go together.[6] With the introduction of lazy evaluation, the programmer loses much of their control over evaluation order: terms reduce “when needed”, if at all. State requires a before and an after, which in the context of lazy evaluation can be quite difficult.

Some algorithms are most naturally expressed using side effects, and so might be otherwise awkward to implement in a lazy language. However, modern lazy languages such as Haskell and Clean provide means for simulating effects: despite being fundamentally lazy languages, they can still describe algorithms in a manner closely resembling how one might describe them in an impure language like ML.[7, 8] However, these methods for describing effects, monads and uniqueness typings, require explicit programmer control and have generally not caught on outside of purely functional languages.

The goal of call by effect is to create an evaluation strategy which combines the advantages of strict and non-strict evaluation; that is, we wish to allow
streams, non-strict compound structures, etc, while still allowing unencapsu-
lated, unrestricted side effects. Ultimately, this leads to more concise code, with
no explicit unpacking or lifting and no delaying or forcing of expressions.

As an example, consider the following program, written in a hypothetical,
call by effect, functional language with a Haskell-like syntax:

data Tree a = Tree a (Tree a) (Tree a) | Nil
deriving (Show)

oneTree = Tree 1 oneTree oneTree

tmap f Nil = Nil
tmap f (Tree x t t') = Tree (f x) (tmap f t) (tmap f t')

takeDepth 0 t = Nil
takeDepth n Nil = Nil
takeDepth n (Tree a t t') =
  Tree a (takeDepth (n-1) t) (takeDepth (n-1) t')

main =
  let _ = print "Enter depth: 
    pair = (takeDepth (readNumber ()) -- input from stdin
        (tmap (+1) oneTree),
        undefined) -- undefined diverges
  in print (first pair)

The program behaves how it intuitively ought to; it asks the user for a depth
and then shows a tree of that depth, where each node is a 2. Call by effect allows
the programmer to write programs such as these, which both use lazy evaluation,
as in the creation of “pair” and in tmap, and to use side effects identically to how
one would in a strict language, as shown with the IO examples. Note the lack of
any explicit annotations, thunks, or other information to tell the compiler how
to reduce expressions. Under call by effect, the above program “just works.”

1.2 Background

We define lambda terms and values by the grammar below:

\[ t ::= x \mid \lambda x.t \mid t_1 t_2, \] where \( x \) a variable

\[ v ::= \lambda x.t \]

We may annotate terms of the form \( \lambda x.t \) with a type, that is, in the Church
style: \( \lambda x: \tau.t \) for a type \( \tau \).

A large number of evaluation strategies have been invented, including call by
name, call by value, call by push value, etc.[1] These specify rules for reduction of
terms. In this paper, we specify operational semantics when describing evaluation
strategies; that is, relations of the form \( t[q_1] \ldots [q_n] \rightarrow t'[q'_1] \ldots [q'_n] \), where \( t \) and \( t' \)
denote terms and \( q_1 \ldots q_n \) and \( q'_1 \ldots q'_n \) denote some states, eg, a store. We define the relation \( \rightarrow^* \) as the transitive reflexive closure of \( \rightarrow \).

Call by value semantics reduce (by the \( \rightarrow^* \) relation) an argument to a function before substitution within the body of a function. Call by name semantics instead substitute the argument unreduced. Similarly to call by name, call by need does not reduce arguments before substitution, but also will cache the result of reduction of the argument to prevent redundant reductions of the argument.

As an example, we provide the operational semantics of call by value:[2]

\[
\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} (V-App1)
\]

\[
\frac{t_2 \rightarrow t'_2}{(\lambda x.t_1)t_2 \rightarrow (\lambda x.t_1)t'_2} (V-App2)
\]

\[
(\lambda x.t_1)v \rightarrow [x \mapsto v]t_1 (V-\beta)
\]

Effect types describe, using types, the side effects of a term.[3] For now, we define “side effect” abstractly. We will define some set of effects, \( \Sigma \), whose subsets are ranged over by the metavariable \( \sigma \), which will vary in accordance with the effect type system under consideration.

Effects types are generated from the grammar:

\[
\tau ::= \tau \sigma, \tau, \sigma \subseteq \Sigma
\]

along with some number of ground types.

Type judgments are of the form \( \Gamma \vdash t : \tau, \sigma \), for a function \( \Gamma \) of variables to types, \( t \) a term, \( \tau \) a type and \( \sigma \) an effect. Example:

\[
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \emptyset} (T-Var)
\]

\[
\frac{[x \mapsto \tau_1] \Gamma \vdash t : \tau_2, \sigma}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \sigma \rightarrow \tau_2, \emptyset} (T-Abs)
\]

\[
\frac{\Gamma \vdash t_1 : \tau_1 \sigma_1 \rightarrow t_2, \sigma_2}{\Gamma \vdash t_1 t_2 : \tau_2, \sigma_1 \sigma_2 \cup \sigma_3} (T-App)
\]

\[
\frac{\Gamma \vdash t : \tau, \sigma}{\Gamma \vdash t : \tau, \sigma' \supseteq \sigma} (T-Sub)
\]

1.3 Overview

Section 2.1 formally defines call by effect by providing an effect type system and an operational semantics. The meaning of the semantics in the context of a programming language is also discussed. Section 2.2 proves the soundness of effect types in a call by effect context as well as other properties of call by effect, such as its relationship to call by value and call by name.
Section 3 regards the real-world implementation of a call by effect language. In Section 3.1 we discuss an interpreter for call by effect. Section 3.2 concerns much needed optimizations unique to the call by effect evaluation strategy. Section 3.3 discusses programming in a call by effect language, with examples. Section 3.4 addresses adding the ability to cast effects, which makes the language more expressive.

Section 4 concludes the paper: 4.1 compares call by effect to monads and uniqueness types, 4.2 references related work, 4.3 discusses new avenues for research and 4.4 summarizes the paper.

2 Introduction to Call by Effect

2.1 Defining Call by Effect

We first define the calculus we will be discussing. It is simply typed lambda calculus augmented with some number of side-effect primitives. These would vary with the implemented language, and may be, for example, I/O primitives or primitives to access a store of some sort.

Types then are defined as they are in a type and effect system, as specified in (T-Var), (T-Abs), (T-App) and (T-Sub). For the purpose of call by effect, we adopt a quite spartan view upon effects. Either a term exhibits an effect or it doesn’t; thus we have two effects: \( \Sigma = \{ \text{Unit} \} \), and thus an effect \( \sigma \) may be either \( \emptyset \) (no effect) or \( \{ \text{Unit} \} \) (has an effect).

The operational semantics of call by effect, as a function of some state \( q \):

\[
\begin{align*}
\frac{t_1[q] \rightarrow t'_1[q']}{t_1t_2[q] \rightarrow t'_1t'_2[q']} \quad &\text{(E-App)} \\
\vdash t_2 : \tau, \emptyset \\
\frac{\vdash (\lambda x : \tau.t_1)/t_2[q] \rightarrow [x \mapsto t_1]t_1[q]}{(\lambda x : \tau.t_1)/t_2[q] \rightarrow (\lambda x : \tau.t_1)/t_1[q]} \quad &\text{(E-\(\beta\_N\))} \\
\emptyset \vdash t_2 : \tau, \{ \text{Unit} \} \\
\frac{t_2[q] \rightarrow^* v[q']}{(\lambda x : \tau.t_1)/t_2[q] \rightarrow (\lambda x : \tau.t_1)/v[q']} \quad &\text{(E-\(\beta\_V\))}
\end{align*}
\]

In essence, call by effect reduces an argument until it no longer has any effects before substitution into the function. To accomplish this task, we use effect types to, at runtime, type the argument of an application and determine its effect.

2.2 Properties of Call by Effect

We begin with soundness. In an effect type system, the effects are not preserved; rather, they become smaller. We show two mechanical lemmas which will allow us to prove this in the context of call by effect.

Lemma 1. Substitution Let \( [x \mapsto t'] \Gamma \vdash t : \tau, \sigma \) and \( \Gamma \vdash t' : \tau', \emptyset \). Then \( \Gamma' \vdash [x \mapsto t'][t] : \tau, \sigma \).

Proof. Straightforward structural induction on \( t \).
Lemma 2. Weakening Let $\Gamma \vdash t : \tau, \sigma$ and $x \notin \text{dom}(\Gamma)$. Then, for all $\tau'$, $[x \mapsto \tau']\Gamma \vdash t : \tau, \sigma$.

Proof. Straightforward induction on the typing derivations. \qed

Theorem 1. Soundness Let $t : \tau, \sigma$, for $t$ not a value. Then $\vdash q \rightarrow t' | q'$ where $\vdash t' : \tau, \sigma'$ for some $\sigma' \subseteq \sigma$.

Proof. Proof by structural induction on $t$.

Suppose that $t$ is a variable. Then the theorem is vacuously satisfied as it does not hold that $t$ types in $\emptyset$.

Suppose that $t$ is an abstraction. Then the theorem is vacuously satisfied as it does not hold that $t$ is not a value.

Suppose that $t$ is an application of the form $t_1 t_2$. Then, as $\vdash t_1 t_2 : \tau, \sigma$ we have, from inversion of (T-App) that $\vdash t_1 : \tau' \mapsto \tau, \sigma_1$ and $\vdash t_2 : \tau', \sigma_2$ for $\sigma = \sigma_1 \cup \sigma_2 \cup \sigma'$.

- Suppose that $t_1$ is not a value. Then, as it types, we may apply the induction hypothesis to $t_1$ to deduce that $\vdash t_1 | q \rightarrow t_1' | q'$ where $\vdash t_1' : \tau' \mapsto \tau, \sigma_1'$ for $\sigma_1' \subseteq \sigma_1$. Then, by (E-App) we have that $\vdash t_1 t_2 | q \rightarrow t_1' t_2 | q'$. By (T-App) we have that $\vdash t_1' t_2 : \tau, \sigma_1' \cup \sigma_2 \cup \sigma'$. Then $\sigma_1' \cup \sigma_2 \cup \sigma' \subseteq \sigma_1 \cup \sigma_2 \cup \sigma'$.

- Suppose that $t_1$ is a value. As $t_1$ types to $\tau' \mapsto \tau, \sigma_1$ and is a value, it must be of the form $\lambda x : \tau'.B$ and $\sigma_1 = \emptyset$.

If $\sigma_2 = \emptyset$ then $(\lambda x : \tau'.B)t_2 | q \rightarrow [x \mapsto t_2]B | q$ (by (E-$\beta_N$)). As $t_1$ types to $\tau' \mapsto \tau$, we have by inversion on (T-Abs) that $[x \mapsto \tau'] \vdash B : \tau, \sigma'$ and thus by the substitution lemma we have that $\vdash [x \mapsto t_2]B : \tau, \sigma'$. Lastly it holds that $\sigma' \subseteq \sigma_1 \cup \sigma_2 \cup \sigma'$.

If instead $\sigma_2 = \{ \text{Unit} \}$ then, by induction on $t_2$ we have that $t_2 | q \rightarrow v | q'$ for $\vdash v : \tau_2, \sigma_2'$ where $\sigma_2' \subseteq \sigma_2$. Then by (E-$\beta_N$) we have that $(\lambda x : \tau'.B)t_2 \rightarrow [x \mapsto v]B | q'$. As $t_1$ types to $\tau' \mapsto \tau$, we have by inversion on (T-Abs) that $[x \mapsto \tau'] \vdash B : \tau, \sigma'$; and thus by the substitution lemma we have that $\vdash [x \mapsto v]B : \tau, \sigma'$. Lastly it holds that $\sigma' \subseteq \sigma_1 \cup \sigma_2 \cup \sigma'$.

\qed

Then we have proved soundness. Next we establish that pure terms preserve state.

Lemma 3. Let $\vdash t : \tau, \emptyset$ and $\vdash q \rightarrow t' | q'$. Then $q = q'$.

Proof. Structural induction on $t$. First, note that if $t$ is a variable or abstraction then the theorem is vacuously satisfied as it does not hold that $t | q \rightarrow t' | q'$. Let $t$ be an application of the form $t_1 t_2$. As $\vdash t : \tau, \emptyset$ it follows from (T-App) that $\vdash t_1 : \tau' \emptyset \rightarrow \tau, \emptyset$ and $\vdash t_2 : \tau', \emptyset$.

- Suppose that $t_1 | q \rightarrow t_1' | q'$. Then from (E-App) we have that $t_1 t_2 | q \rightarrow t_1' t_2 | q'$. By induction on $t_1$, $q = q'$.
Suppose that \( t_1 \) is a value of the form \( \lambda x : \tau' . t_3 \). Then, as \( \vdash t_2 : \tau', \emptyset \) we have by (E-β) that \( (\lambda x : \tau'.t_3)t_2[q \mapsto [x \mapsto t_2]q]_q \), for \( q = q' \).

The more interesting properties of call by effect relate to other evaluation strategies, as shown below.

For a program written in a purely functional manner, we may establish that call by effect results in the same normal form as call by name would. Firstly, we establish a predicate for identifying such programs.

**Definition 1.** Let \( P \) be the least predicate upon \( t \cup \tau \times \Gamma \) satisfying:

\[
\begin{array}{c}
P(t_1, \Gamma) \quad P(t_2, \Gamma) \\
\hline
P(\tau_1, \emptyset) \quad P(\tau_2, \Gamma) \\
\hline
P(\Gamma(x)) \quad P(t, [x \mapsto \tau]\Gamma) \quad P(\tau, \Gamma) \quad P(t_1, \Gamma) \quad P(t_2, \Gamma)
\end{array}
\]

Clearly \( P \) would not hold for whatever side effects may be present in the particular calculus of interest; for example, \( P \) would not hold for terms in a calculus with stores which update or read from said store.

Next we note the effect of such programs, namely \( \emptyset \):

**Lemma 4.** Let \( \Gamma \vdash t : \tau, \sigma \) where \( P(t, \Gamma) \) holds. Then \( \sigma = \emptyset \) and \( P(\tau, \Gamma) \) holds.

**Proof.** Structural induction on \( t \):

- Let \( t \) be a variable of the form \( x \). As \( P(x, \Gamma) \) holds, so must \( P(\Gamma(x), \Gamma) \), which by (T-Var) is the same as \( P(\tau, \Gamma) \). Immediate from (T-Var) is that \( \sigma = \emptyset \).

- Let \( t \) be an abstraction. Firstly \( \sigma = \emptyset \) from (T-Abs). Now we must show \( P(\tau, \Gamma) \). As \( P(\lambda x : \tau'.t', \Gamma) \) holds, both \( P(\tau', \Gamma) \) and \( P(t', [x \mapsto \tau']\Gamma) \) must also hold. By inversion on (T-Abs) we have that \( t \) is of the form \( \lambda x : \tau'.t' \) where \( [x \mapsto \tau']\Gamma \vdash t' : \tau'', \sigma'' \).

Then, by induction on \( t' \) in \([x \mapsto \tau']\Gamma\) we have that \( \sigma'' = \emptyset \) and that \( P(\tau'') \) holds. Thus \( P(\tau' \xrightarrow{\sigma''} \tau'', \Gamma) \).

- Let \( t \) be an application of the form \( t_1 t_2 \). By inversion of (T-App) we have that \( \Gamma \vdash t_1 : \tau' \xrightarrow{\sigma'} \tau, \sigma_1 \) and \( \Gamma \vdash t_2 : \tau', \sigma_2 \), as \( t \) types in \( \Gamma \). Furthermore as \( P(t_1, \Gamma) \) holds, so does \( P(t_1, \Gamma) \) and \( P(t_2, \Gamma) \). Then, by induction on \( t_1 \) and \( t_2 \) we have that \( \sigma_1 = \sigma_2 = \emptyset \) and \( P(\tau' \xrightarrow{\sigma''} \tau, \Gamma) \). Finally: by the definition of \( P \) it holds that \( \sigma'' = \emptyset \) and that \( P(\tau, \Gamma) \); thus \( \sigma = \sigma' \cup \sigma_1 \cup \sigma_2 = \emptyset \).

We find this mechanical substitution lemma useful in the following proofs:

**Lemma 5.** Let \( P(t, \Gamma) \) and \( P(t', [x \mapsto \tau]\Gamma) \) for \( \Gamma \vdash t : \tau, \emptyset \). Then \( P([x \mapsto t]t', \Gamma) \).
Proof. Straightforward structural induction on $t$.  

Then, we show a semantics preservation result for terms for which $P(t, \emptyset)$ holds:

**Theorem 2.** Equivalence of Pure Programs Let $t|q \longrightarrow t'|q'$ under call by effect for a $t$ where $P(t, \emptyset)$ holds. Then $t|q \longrightarrow t'|q'$ under call by name and $P(t', \emptyset)$ holds.

Proof. Proof by structural induction on $t$. First, note that if $t$ is a variable or abstraction, then the theorem is vacuously satisfied, as it does not hold that $t|q \longrightarrow t'|q'$. Therefore let $t$ be an application of the form $t_1 t_2$. As $P(t_1 t_2, \emptyset)$ holds, so do $P(t_1, \emptyset)$ and $P(t_2, \emptyset)$.

Suppose that $t_1|q \longrightarrow t'_1|q'$, under call by effect, and thus $t_1 t_2|q \longrightarrow t'_1 t_2|q'$. By induction on $t_1$ it also holds that $t_1|q \longrightarrow t'_1|q'$ under call by name where $P(t'_1, \emptyset)$ holds. Therefore $t_1 t_2|q \longrightarrow t'_1 t_2|q'$ under call by name and $P(t'_2 t_2, \emptyset)$.

Suppose instead that $t_1$ is of the form $\lambda x: \tau.t'$. As $P(t_2, \emptyset)$ holds, it follows from lemma (4) that $\vdash t_2: \tau, \emptyset$. Then, from $(E-\beta_N)$ we have that $(\lambda x: \tau.t')t_2|q \longrightarrow [x \mapsto t_2]t'|q$ which holds under both call by name and effect. Now we must show $P([x \mapsto t_2]t', \emptyset)$ given that $P(t_2, \emptyset)$ and $P(t', [x \mapsto \tau])$. But that is exactly the statement of lemma (5).

Lastly, we establish that call by effect maintains the same effects as would occur under call by value. The following relation is quite useful when proving this property.

**Definition 2.** The $\stackrel{\theta}{\rightarrow}$ Relation Let $\stackrel{\theta}{\rightarrow}$ be the least binary relation upon terms for which at least one of the following holds:

$$
\frac{}{t \stackrel{\theta}{\rightarrow} t} \quad (\theta-1)
$$

$$
\frac{\vdash t: \tau_1, \emptyset \quad \vdash v': \tau_2, \emptyset \quad t|q \longrightarrow^* v|q' \quad v \stackrel{\theta}{\rightarrow} v'}{t \stackrel{\theta}{\rightarrow} v'} \quad (\theta-2)
$$

$$
\frac{t_1 \stackrel{\theta}{\rightarrow} t_1' \quad t_2 \stackrel{\theta}{\rightarrow} t_2'}{t_1 t_2 \stackrel{\theta}{\rightarrow} t_1' t_2'} \quad (\theta-3)
$$

$$
\frac{t \stackrel{\theta}{\rightarrow} t'}{\lambda x: \tau.t \stackrel{\theta}{\rightarrow} \lambda x: \tau.t'} \quad (\theta-4)
$$

Intuitively, the $t \stackrel{\theta}{\rightarrow} t'$ relation tells us when two terms differ only by pure subterms for which the normal form of one is related by the $\stackrel{\theta}{\rightarrow}$ relation to the other, with equality of terms as the basis.

We first define a useful substitution lemma for the $\stackrel{\theta}{\rightarrow}$ relation.

**Lemma 6.** Preservation of $\stackrel{\theta}{\rightarrow}$ Substitution Let $t_1 \stackrel{\theta}{\rightarrow} t_1'$ and $t_2 \stackrel{\theta}{\rightarrow} v'$. Then $[x \mapsto t_2]t_1 \stackrel{\theta}{\rightarrow} [x \mapsto v']t_1'$. 

Proof. Proof by structural induction on $t_1$.

Suppose that $t_1 \equiv x$. Then it can only hold by (θ-1) that $t_1 \xrightarrow{θ} t_1'$; that is, $t_1' \equiv x$. Finally: $[x \mapsto t_2]x \xrightarrow{θ} [x \mapsto v']x$ iff $t_2 \xrightarrow{θ} v'$, which is given.

Similarly, if $t_1 \equiv y$, then $t_1' \equiv y$, and $[x \mapsto t_2]y \xrightarrow{θ} [x \mapsto v']y$ iff $y \xrightarrow{θ} y$, which holds by (θ-1).

Suppose that $t_1$ is an abstraction of the form $\lambda x : \tau. t$. Then as $\lambda x : \tau. t \xrightarrow{θ} t_1'$ it must either hold that $t_1 = t_1'$ or $t_1'$ is of the form $\lambda x : \tau'. t'$, for $t \equiv t'$. Either way $t'$ is of the form $\lambda x : \tau. t'$, for $t \xrightarrow{θ} t'$ (if $t_1 = t_1'$ then $t = t'$ and thus $t \equiv t'$). Then $[x \mapsto t_2]t_1 = \lambda x : \tau. [x \mapsto t_2]t$ and $[x \mapsto v']t_1' = \lambda x : \tau. [x \mapsto v']t'$. But by induction on $t$ and $t'$ we have that $[x \mapsto t_2]t \xrightarrow{θ} [x \mapsto v']t'$, and thus by (θ-4) it holds that $\lambda x : \tau. [x \mapsto t_2]t \equiv \lambda x : \tau. [x \mapsto v']t'$.

Suppose that $t_1$ is an application of the form $t_11 \cdot t_12$. Then $[x \mapsto t_2]t_1 = ([x \mapsto t_2]t_11)([x \mapsto t_2]t_12)$. From $t_1 \xrightarrow{θ} t_1'$ at least one of the following holds:

- (θ-1) Suppose that $t_1' \equiv t_11 \cdot t_12$. Then $[x \mapsto v']t_1' = ([x \mapsto v']t_11)([x \mapsto v']t_12)$.

As $t_1 \xrightarrow{θ} t_11$ and $t_1 \xrightarrow{θ} t_12$, we can apply the induction hypothesis to $t_11$ and $t_12$ to obtain that $[x \mapsto t_2]t_11 \xrightarrow{θ} [x \mapsto v']t_11$ and $[x \mapsto t_2]t_12 \xrightarrow{θ} [x \mapsto v']t_12$, respectively. Finally by (θ-4) it holds that $([x \mapsto t_2]t_11)([x \mapsto t_2]t_12) \xrightarrow{θ} ([x \mapsto v']t_11)([x \mapsto v']t_12)$.

- (θ-2) Suppose that both $t_1$ and $t_1'$ type in $\emptyset$. Then they are both closed and thus $[x \mapsto t_2]t_1 = t_1$ and $[x \mapsto v']t_1' = t_1'$; but as given, $t_1 \xrightarrow{θ} t_1'$.

- (θ-3) Proved similarly to the corresponding case for θ-1.

- (θ-4) Suppose that $t_1$ is an abstraction. But this contradicts that $t_1$ is an application.

Thus we have enough to precisely define the relationship between call by effect and call by value. It is:

**Theorem 3.** Preservation of State Let $t$ and $t'$ type and $t|q_1 \rightarrow t_2|q_2$ under call by effect and $t \xrightarrow{θ} t'$. Then $t'|q_1 \rightarrow t_2'|q_2$ under call by value where $t_2 \xrightarrow{θ} t_2'$.

Proof. Proof by structural induction on $t$. First, note that if $t$ is not an application, then the theorem is vacuously satisfied as it does not hold that $t|q_1 \rightarrow t_2|q_2$, so we need only consider the case of application. Therefore let $t$ be of the form $MN$.

Suppose that $M|q_1 \rightarrow M_2|q_2$, and thus by (E-App) $MN|q_1 \rightarrow M_2N|q_2$. As $MN \xrightarrow{θ} M'$, at least one of the following must hold:

- (θ-1) Suppose that $t'$ is of the form $MN$. Then from (θ-1) it follows that $M \xrightarrow{θ} M'$, from which we may apply the induction hypothesis to deduce that $M|q_1 \rightarrow M_2'|q_2$ for $M_2 \xrightarrow{θ} M'_2$. Then from (θ-3) it holds that $M_2N \xrightarrow{θ} M'_2N$.
– \((\theta-2)\) Suppose that \(t'\) is a value and \(MN|q \rightarrow^* v|q\), \(v \overset{\theta}{\rightarrow} t'\); or, equivalently, \(MN|q \rightarrow M_2N|q \rightarrow^* v|q\). By the soundness result, \(M_2N\) still types and is pure, which combined with \(M_2N|q \rightarrow^* v|q\) and \(v \overset{\theta}{\rightarrow} t'\) yields that \(M_2N \overset{\theta}{\rightarrow} t'\); for if \(M_2N = v\) then \(M_2N \overset{\theta}{\rightarrow} t'\) is immediate, otherwise it follows from \((\theta-1)\).

– \((\theta-3)\) Proved similarly to the corresponding case for \((\theta-1)\).

– \((\theta-4)\) Suppose that \(t\) is an abstraction. But this contradicts that \(t\) is an application.

Suppose instead that \(M\) is of the form \(\lambda x: \tau.B\) and that \(\vdash N: \tau, \{\text{Unit}\}\). Then from \((E-\beta)\), it holds that \(N|q_1 \rightarrow^* v|q_2\) and \((\lambda x: \tau.B)|q_1 \rightarrow [x \mapsto v]|B|q_2\). As \((\lambda x: \tau.B)N \overset{\theta}{\rightarrow} t'\), at least one of the following must hold:

– \((\theta-1)\) Proved similarly to the corresponding case for \((\theta-3)\).

– \((\theta-2)\) Suppose that \(\vdash MN: \tau', \emptyset\). But this contradicts that \(\vdash N: \tau, \{\text{Unit}\}\).

– \((\theta-3)\) Let \(t'\) be of the form \(M'N'\) for \(\lambda x: \tau.B \overset{\theta'}{\rightarrow} M'\) and \(N \overset{\theta}{\rightarrow} N'\). Immediate from the definition of \(\overset{\theta'}{\rightarrow}\) it holds that \(M'\) is of the form \(\lambda x: \tau.B'\) for \(B \overset{\theta}{\rightarrow} B'\).

From the operational semantics of call by value, \(N'|q_1 \rightarrow^* v'|q'_2\) and \(M'|q_1 \rightarrow^* [x \mapsto v']|B'|q'_2\). Induction on \(N\) with \(N'\) yields \(q'_2 = q_2\), \(v \overset{\theta}{\rightarrow} v'\). But then by lemma \((6)\) it holds that \([x \mapsto v]|B \overset{\theta}{\rightarrow} [x \mapsto v']|B'\).

– \((\theta-4)\) Suppose that \(t\) is an abstraction. But this contradicts that \(t\) is an application.

Lastly suppose that \(M\) is of the form \(\lambda x: \tau.B\) and that \(\vdash N: \tau, \emptyset\). Then by \((E-\beta_N)\) we have that \((\lambda x: \tau.B)|q_1 \rightarrow [x \mapsto N]|B|q_1\). As \((\lambda x: \tau.B)N \overset{\theta}{\rightarrow} t'\), at least one of the following must hold:

– \((\theta-1)\) Suppose that \(t'\) is of the form \((\lambda x: \tau.B)N\). Then, under call by value, \(N|q_1 \rightarrow^* v'|q_2\) and \((\lambda x: \tau.B)|q_1 \rightarrow^* [x \mapsto v']|B|q_2\). But \(N\) is pure, and thus by lemma \((3)\) \(q_1 = q'_2\).

Induction on \(N\) (as \(N \overset{\theta}{\rightarrow} N\)) yields \(N|q \rightarrow^* v|q\) for \(v \overset{\theta}{\rightarrow} v'\). Then, if \(N\) is not a value, then we have the requirements for \((\theta-2)\) and \(N \overset{\theta}{\rightarrow} v'\). Otherwise \(N = v\) and \(N \overset{\theta}{\rightarrow} v'\). Finally from lemma \((6)\) we have \([x \mapsto N]|B \overset{\theta}{\rightarrow} [x \mapsto v']|B\).

– \((\theta-2)\) Suppose that \(t'\) is a value where \((\lambda x: \tau.B)|q|q \rightarrow^* v|q\) for \(v \overset{\theta}{\rightarrow} t'\). Then \((\lambda x: \tau.B)|q_1 \rightarrow [x \mapsto N]|B|q_1 \rightarrow^* v|q_1\). If \([x \mapsto N]|B\) is not a value, then we have by \((\theta-2)\) that \([x \mapsto N]|B \overset{\theta}{\rightarrow} t'\). If it is a value, then it is \(v\) and \(v \overset{\theta}{\rightarrow} t'\). Finally from lemma \((6)\) we have \([x \mapsto N]|B \overset{\theta}{\rightarrow} t'\).

– \((\theta-3)\) Proved similarly to the corresponding case for \((\theta-1)\).

– \((\theta-4)\) Suppose that \(t\) is an abstraction. But this contradicts that \(t\) is an application. \(\square\)
3 Real-World Call by Effect

3.1 Implementation

We implemented an interpreter for the calculus presented in the past section. However, our implementation includes several features, including effect type inference, a polymorphic let form, conditional branching, Peano arithmetic, and recursion via an explicit fix point operator.

We deviate from the operational semantics presented in one major way. We cache the result of arguments to prevent redundant evaluations of terms. This is done using thunks: lazy evaluation.

3.2 Optimizations

To avoid runtime type checking, we may type a term before interpretation and attempt to statically determine when an argument to an application should reduce before substitution. The interpreter has a preliminary pass which determines the types of arguments to applications, eliminating much runtime type checking. This is then a type-directed source translation into a calculus which also supports explicitly annotated applications as being either strict or lazy.

We define the terms and values of this calculus according to the grammar:

\[
t ::= x \mid \lambda x.t \mid \eta tt \mid \vartheta tt, x \text{ a variable}
\]

and semantics of:

\[
\frac{\eta(\lambda x.t_1)t_2|q \rightarrow [x \mapsto t_2]t_1|q'}{(E-\beta_0)}
\]

\[
\frac{\vartheta(\lambda x.t_1)t_2|q \rightarrow (\lambda x.t_1)t_2'|q'}{(E-\vartheta)}
\]

We define this type-directed translation as a function \( E \), defined as:

**Definition 3. E Translation** Define the function \( E : t \times \Gamma \rightarrow t \) as:

\[
E(x, \Gamma) = x
\]

\[
E(\lambda x : \tau, t, \Gamma) = \lambda x.E(t, [x \mapsto \tau]\Gamma)
\]

\[
E(t_1t_2, \Gamma) = \eta E(t_1, \Gamma)E(t_2, \Gamma), \text{ for } \Gamma \vdash t_2 : \tau, \emptyset
\]

\[
E(t_1t_2, \Gamma) = \vartheta E(t_1, \Gamma)E(t_2, \Gamma), \text{ otherwise}
\]

Intuitively, applications of the form \( \eta tt' \) are non-strict and applications of the form \( \vartheta tt' \) are strict.

We wish to show that \( E \) preserves semantics. First, we show that \( E \) distributes over substitution; that is,
Lemma 7. Distributivity of $E$. Let $t_1$ and $t_2$ be terms where $E([x \mapsto t_2]t_1, \Gamma)$ is defined. Then $E([x \mapsto t_2]t_1, \Gamma) = [x \mapsto E(t_2, \Gamma)]E(t_1, \Gamma)$.

Proof. Straightforward structural induction on $t_1$. □

Enabling us to show the semantics preservation proof of:

Theorem 4. Semantics Preservation of $E$. Let $t|q \longrightarrow t'|q'$. Then $E(t, \emptyset)|q \longrightarrow^* E(t', \emptyset)|q'$.

Proof. Proof by structural induction on $t$. First, as $t|q \longrightarrow t'|q'$ we have that $t$ cannot be a variable or abstraction. Let $t$ be an application of the form $t_1t_2$.

Suppose that $t_1|q \longrightarrow t_1'|q'$, and thus $t_1t_2|q \longrightarrow t_1't_2|q'$. By induction on $t_1$ we have that $E(t_1, \emptyset)|q \longrightarrow^* E(t_1', \emptyset)|q'$, and then from ($E$-App) we have that $E(t_1, \emptyset)E(t_2, \emptyset)|q \longrightarrow^* E(t_1', \emptyset)E(t_2, \emptyset)|q'$.

Suppose that $t_1$ is a value of the form $\lambda x : \tau.t$ and that $\vdash t_2 : \tau, \emptyset$. From the definition of $E$ we have that $E((\lambda x : \tau.t)t_2, \emptyset) = E(\lambda x : \tau.t, \emptyset)E(t_2, \emptyset) = E(\lambda x.E(t,[x \mapsto \tau]))E(t_2, \emptyset)$.

For reductions of the terms, we have from ($E$-βN) that $(\lambda x : \tau.t)t_2|q \longrightarrow [x \mapsto t_2]|t|q$ and from ($E$-β$_{\eta}$) that:

$$
\eta(\lambda x.E(t,[x \mapsto \tau]))E(t_2, \emptyset) \longrightarrow [x \mapsto E(t_2, \emptyset)]E(t,[x \mapsto \tau])
$$

From which, from lemma (7) we conclude with:

$$
E([x \mapsto t_2]t, \emptyset) = [x \mapsto E(t_2, \emptyset)]E(t,[x \mapsto \tau]).
$$

Suppose that $t_1$ is a value of the form $\lambda x : \tau.t$ and instead that $\vdash t_2 : \tau, \{\text{Unit}\}$.

Solving for $E(t_1t_2)$ gives $E((\lambda x : \tau.t)t_2, \emptyset) = \psi(\lambda x.E(t,[x \mapsto \tau]))E(t_2, \emptyset)$. Then $t_2|q \longrightarrow^* v|q'$ and $t_1t_2|q \longrightarrow [x \mapsto v]|t|q'$. By induction on $t_2$ we have that $E(t_2, \emptyset)|q \longrightarrow^* E(v, \emptyset)|q'$ and thus that $E(t_1t_2)|q \longrightarrow^* [x \mapsto E(v, \emptyset)]E(t,[x \mapsto \tau])|q'$. From which, from lemma (7) we conclude with:

$$
E([x \mapsto v]t, \emptyset) = [x \mapsto E(v, \emptyset)]E(t,[x \mapsto \tau]).
$$

Consider the term $\lambda xy.x (y \ 0)$, for which, if let bound, we cannot determine the effect of $(y \ 0)$ (as the function is polymorphic). Thus runtime type checking will be required. Polymorphic higher-order functions thwart the efforts of such a naive source translation.

For polymorphic functions for which not all applications can be resolved through an initial type checking, the function is polyinstantiated; that is, non-polymorphic versions are automatically generated and substituted in place of invocations to the polymorphic version.[5]

In detail, we perform the following transformation: for a polymorphic let-bound identifier with a type of the form $\forall \alpha \beta \ldots. \tau$, where $\alpha \beta \ldots$ range over effects, we instantiate the type with all $2^n$ (for $n$ generalized effect variables) possible instances, creating implementations of the identifier type checked in the context of the instantiation. Then, we substitute within the body of the let form.
the identifiers for the non-polymorphic variants. Thus we eliminate all need for runtime type checking, although at the cost of an increase in code size. We conjecture that, although the number of instantiations of a polymorphic type grows exponentially with the number of type variables ranging over effects, the number of type variables ranging over effects will typically be quite small, as experience using higher-order functions indicates (e.g., map and fold have only one effect variable which affects the evaluation order of an application).

Our use of polyinstantiation is interesting because we can completely enumerate all possible instantiations. For instance, if we were polyinstantiating the identity combinator outside of the context of call by effect, normally we would need to pass over the rest of the program to determine all invocations of the identity combinator to create the monomorphic specialized versions of the function, e.g., \( \mathbb{N} \to \mathbb{N}, (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \), etc, as there are actually an infinite number of monomorphic instantiations of the type. For call by effect, we have a finite number of monomorphic types. Of course, polyinstantiation of all generalized variables would still require whole-program analysis.

### 3.3 Programming with Call by Effect

At times, programming in call by effect resembles programming in a pure, lazy language such as Haskell; at other times, it resembles programming in an impure, strict language such as ML. For example, in some hypothetical, call by effect language with a Haskell-like syntax, one could write:

```haskell
let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

while at the same time writing:

```haskell
if readChar () == 'c' then 0 else 1 + readInt ()
```

for which call by effect more closely resembles a strict language. Of course we may integrate both lazy and strict evaluation in the same term:

```haskell
take (readInteger stdin) (map (**2) [1..])
```

for which the effects analysis and subsequent polyinstantiation automatically decide where to delay evaluation.

These examples show the value of call by effect: the programmer is freed from explicit lifting and unpacking, as may be required to manipulate side effects in a lazy language, while still allowing much of the expressiveness of lazy evaluation.

### 3.4 Casting Effects and Local State

Call by effect assumes that the programmer will never intentionally delay the evaluation of an impure term. This assumption is wrong for two reasons, both of which can be fixed.

Firstly, the term may not actually be impure, but might only use some local state; it is thus pure to an outside caller. Expanding the effects analysis to
perform region analysis allows us to eliminate side effects to local mutable data from the effect type of the term, as done in, for example, [4] for the purpose of eliminating superfluous heap allocations.

Secondly, we may actually wish to delay evaluation of an impure argument, such as when mapping a computation with side effects across a stream where we do wish for the side effect to occur when each element of the stream is demanded. For this purpose, implementations of call by effect languages need to allow the programmer to cast the effect, distinct from verifying the effect. For example, we may, as in ML, use \( : \) to verify types, and in some call by effect language also use \( \Rightarrow : \) to cast effects, verifying the non-effect parts of the type. Suppose that 0 indicated no effect and 1 an effect. Then, if we wished to create a stream of units, for which forcing the \( n \text{th} \) unit will print \( n \), we would write:

\[
\text{map (printInt} :\Rightarrow \text{Int} \rightarrow \text{Unit, 0)} [1..]
\]

while the non-casted version of:

\[
\text{map printInt [1..]}
\]

would never terminate.

4 Conclusion

4.1 Comparison to Monads and Uniqueness Typing

Several strategies for integrating state with lazy evaluation are used by functional programmers, from manually threading a state through computations to uniqueness typing to monads.

Monads can be used for many different purposes and elegantly generalize many structures ranging from lists to exceptions to continuations to, as we will concern ourselves here, computations involving state.[7] Monads allow programmers to explicitly but tersely manage state. The state monad threads some state through a computation and provides a combinator for combining two monadic computations (bind).

Uniqueness typing, as used by, for example, the language Clean, resolves the problem of state by having the programming manually thread states through computations, in a manner similar to how the state monad automatically performs the threading of state.[8]

As an advantage, call by effect need not have both monadic and non-monadic versions of higher order functions such as map (no need for both map and mapM). This facilitates code re-use: only a single version of a higher function needs to be written.

In Haskell, one writes:

\[
\text{map f [ ]} = [ ]
\]
\[
\text{map f (x:xs)} = f x : \text{map f xs}
\]
mapM f [] = return []
mapM f (x:xs) = do
  x' <- f x
  xs' <- mapM f xs
  return (x':xs')

for which in a call by effect setting only the first need be written. Of course, monads have many uses beyond describing impure computations; the mapM function would still be needed for those cases, assuming a monadic style.

In addition, the implicit style will probably be more familiar to programmers with a background in impure strict languages like ML, as state will not have to be managed and unpacked so explicitly. This allows concise expression of impure computations on par with imperative languages. For example, consider the following code taken from [9] which uses uniqueness types:

\[
f \text{world} \text{ file} =
\begin{align*}
&\text{let } (\text{world1}, a) = \text{freadi world file in} \\
&\text{let } (\text{world2}, b) = \text{freadi world1 file in} \\
&(a + b, \text{world2})
\end{align*}
\]

For comparison, here is the equivalent code in a monadic style, written in several different ways:

\[
f \text{file} =
\begin{align*}
&\text{freadi file >>= \ } i1 \to \\
&\text{freadi file >>= \ } i2 \to \\
&\text{return } (i1 + i2)
\end{align*}
\]

\[
\text{-- With Haskell's do notation:}
\begin{align*}
do\ i1 <- \text{freadi file} \\
\text{i2 <- freadi file} \\
\text{return } (i1 + i2)
\end{align*}
\]

\[
\text{-- With higher-order monadic combinators:}
\text{liftM2 (+) (freadi file) (freadi file)}
\]

Or, using call by effect:

\[
\text{freadi file + freadi file}
\]

which, in our opinion, feels more natural and intuitive.

There are definite drawbacks to using call by effect to express state. We are no longer able to very easily conceptually separate a “computation” from the actual running of a computation as we can using monads. For example, the following Haskell code represents a list of “actions” which when run will print their index in the list:

\[
\text{map print [1..10]}
\]
as opposed to the action which actually prints out the indexes:

\[
\text{mapM_ print [1..10]}
\]

Such a subtle and powerful distinction is as awkward to achieve using call by effect as it would be in a strict impure language. To represent the list of actions, one would have to resort to a list of closures which, when applied to, say, Unit, would cause some side effect to occur, which we find quite inelegant and no better than what is already done in strict languages.

Furthermore we cannot express control operators such as if as functions. In Haskell, one may write:

\[
\text{tester True yes no = yes} \\
\text{tester False yes no = no}
\]

In a call by effect setting, any side effects of either yes or no will occur, which is almost surely not the intention of the programmer. Similarly, short-circuiting \( \lor \) and \( \land \) operators cannot be defined as functions for arguments with side effects. The side effects will invariably occur, forbidding common idioms such as

\[
\text{open("file") || die;}
\]

Call by effect requires control primitives just as call by value does.

Currently we handle polymorphism through polyinstantiation. This method necessitates that any type-level features we choose to include in our language must work with polyinstantiation. Thus, at the moment call by effect precludes, for example, polymorphic recursion.

The biggest fundamental difference between call by effect and monads/uniqueness typing is that, rather than making state feel more natural in a lazy language (monads) or guaranteeing the safety of manually threaded state in a lazy language, we change the evaluation strategy used by the language to accommodate something quite close to lazy evaluation while maintaining the same ease of using state as one would find in an impure call by value language such as ML (see the Preservation of State theorem).

Call by effect is significantly less controlled than other systems. The programmer does not explicitly specify what has an effect and what does not, as in Haskell, or have to unpack and pass around state, as in Clean, or when to wrap a computation in a thunk using explicit delays, as in Scheme.

### 4.2 Related Work

Without the polyinstantiation, the optimizations performed (that is, statically determining strict and lazy applications) is a form of type-directed partial evaluation. A good resource for partial evaluation is [10].

For discussions on the importance of lazy evaluation, see [11, 6].

Philip Wadler has authored many papers on monads in the context of functionals programming, including, but not limited to, [12, 7, 13].
Edsko Vries and Rinus Plasmeijer have done much work on uniqueness typing. Examples of their publications include [8, 9].

Effect types are introduced in [14, 3]. Applications to automatic parallelization and concurrency are presented in [4] and [15], respectively.

4.3 Future Work

We see two main areas to investigate in call by effect. First, there is the problem of polyinstantiation. While it solves the problem of run-time typing, it places arbitrary restrictions upon the typing system, disallowing polymorphic recursion, dependent types, etc.

Second, we wish to find more elegant methods of allowing the programmer to explicitly control the reduction semantics other than type and effect casts. The method of casting to control evaluation reveals to the programmer the underlying details of call by effect, which opposes the goal of making everything as automatic and implicit as possible.

4.4 Summary

We have presented the evaluation strategy, call by effect. We have shown how to implement it efficiently and provided small example programs demonstrating its usefulness. We have also compared it both to other evaluation strategies (call by value, call by name) and to other methods of describing state (monads and uniqueness typing).

References