*LISP AND SYMBOLIC COMPUTATION: An International Journal, 7, 11–38, 1994*
*© 1994 Kluwer Academic Publishers – Manufactured in The Netherlands*

# Recursion from Iteration*

ANDRZEJ FILINSKI† (*andrzej+@cs.cmu.edu*)
*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891*

**Keywords:** continuations, recursion, iteration, definability, uniformity.

**Abstract.** In a simply-typed, call-by-value (CBV) language with first-class continuations, the usual CBV fixpoint operator can be defined in terms of a simple, infinitely-looping iteration primitive. We first consider a natural but flawed definition, based on exceptions and "iterative deepening" of finite unfoldings, and point out some of its shortcomings. Then we present the proper construction using full first-class continuations, with both an informal derivation and a proof that the behavior of the defined operator faithfully mimics a "built-in" recursion primitive. In fact, given an additional uniformity assumption, the construction is a two-sided inverse of the usual definition of iteration from recursion. Continuing, we show that the CBV looping primitive is in fact the direct-style equivalent of a continuation-passing-style fixpoint, and that this correspondence extends all the way to traditional definitions of these operators in terms of reflexive types.

## 1. Introduction

### 1.1. Background and motivation

Recursive definitions form a cornerstone of functional programming. It is commonly accepted that many algorithms can be expressed much more clearly using recursion rather than iteration, and that iteration itself is easily definable as (tail) recursion. But does this mean that recursion is somehow the more fundamental or general language construct? In this paper we will show that the answer is "not necessarily": in Scheme-like languages full recursion can also be characterized as a particular pattern of iteration!

The results presented here were inspired by a category-theoretic charac-

*An earlier version of this work appeared in *Proceedings of the 1992 ACM SIGPLAN Workshop on Continuations.*

†Supported in part by NSF Grant CCR-8922109 and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

terization of languages with first-class continuations [7] in terms of what could now be called "classically-typed" [9] categories. Specifically, one can interpret the difference between data-driven (or, call-by-value) and demand-driven (call-by-name) evaluation in such a language as an instance of a categorical symmetry principle called *duality*. In this view, the categorical "mirror image" of the CBN recursion operator turns out to be an *iteration* primitive in CBV; very informally, the relation is like between "inside-out" evaluation of $\cdots f(f(x))\cdots$ in CBV and "outside-in" evaluation of of $f(f(\cdots x\cdots))$ in CBN. Moreover, the usual construction used to define iteration from recursion in CBN can be systematically turned "upside down", to express CBV recursion as a sugared form of simple iteration.

We will not pursue this view here, however, but adopt a self-contained presentation not directly tied to category theory or symmetry considerations. This will allow us to draw directly upon the substantial body of existing results about continuations, e.g., [4, 5, 9, 17] for reasoning about the construction. An outline of the categorical approach is sketched in section 5.

In many functional programming languages, it is actually possible to write recursive functions without any "explicit" recursion. For example, the Scheme definition [1] expresses **letrec** in terms of **set!**. More fundamentally, the well-known Y-combinator provides a uniform way of introducing self-reference. However, a closer analysis shows that all such definitions rely in one form or another on the very powerful and general concept of reflexive domains; once we admit those, we cannot make finer distinctions. It is worth investigating, therefore, the essence of recursion in a simply-typed setting; we will briefly return to "recursion from reflexivity" in sections 4.2 and 4.3.

The rest of this section sets up a common framework and introduces some specific notation. Rather than tie ourselves to the idiosyncrasies of any particular language, we will use a neutral, hopefully universally understandable $\lambda$-calculus notation. While we will often omit explicit types, they can easily be reconstructed. In particular, all the definitions can be directly translated into an ML-like language or (a typed variant of) Scheme.

### 1.2. Recursion and iteration

Let us first clarify what we mean by recursion and iteration in a CBV setting. As is well known, a higher-order functional language allows us to express recursive function definitions without any special syntax. Specifically, we can replace syntactic forms like **letrec** with a functional $\mathsf{fix}$ that achieves the same effect:

$$\textbf{letrec } f = E_1 \textbf{ in } E_2 \stackrel{\text{def}}{=} \textbf{let } f = \mathsf{fix}\,(\lambda f.E_1) \textbf{ in } E_2$$

The **let** can now be simply expanded away or replaced by a $\beta$-redex. (The latter choice would interfere with ML-style polymorphic typing, though). $\mathsf{fix}$ owes its name to the fact that for any function $F : \alpha \to \alpha$,

$$\mathsf{fix}\, F \;=\; F\,(\mathsf{fix}\, F) \;:\; \alpha$$

In a CBV language, however, this equation must be taken with a grain of salt. Because all functions are strict, it is easy to see that taking $\mathsf{fix}\, F = \Omega_\alpha$ (a non-terminating term of type $\alpha$) trivially satisfies the equation. In fact, to be useful for CBV evaluation, $\mathsf{fix}\, F$ should reduce to a value in a finite number of steps. More specifically, for every value $F : (\alpha \to \beta) \to \alpha \to \beta$, the following should hold:

$$\mathsf{fix}\, F \;=\; \lambda a.\, F\,(\mathsf{fix}\, F)\, a \;:\; \alpha \to \beta$$

This is usually known as the "CBV fixpoint" equation.

Let us now consider iteration. In a functional setting, it is generally characterized as a special case of recursion, in which the recursive call happens to be a tail call. To capture this pattern as a functional, let us assume that we have a "disjoint union" type with left and right injections and a **case**-construct. Then we can express loops with the following *tail-recursive* definition, which repeatedly applies a function $f : \alpha \to \beta + \alpha$ to a value, until the result is tagged as a left inject:

$$\mathsf{repeat}\, f \;=\; \lambda a.\, \mathbf{case}\ f\, a\ \mathbf{of}\ \mathsf{inl}\,(b) \to b \;[\!]\; \mathsf{inr}\,(a') \to \mathsf{repeat}\, f\, a' \;:\; \alpha \to \beta$$

A degenerate case of this is a loop with *no* exit: for $f : \alpha \to \alpha$,

$$\mathsf{loop}\, f \;=\; \mathsf{repeat}\,(\mathsf{inr} \circ f) \;=\; \lambda a.\, \mathsf{loop}\, f\,(f\, a) \;:\; \alpha \to \beta$$

Since the function $\mathsf{loop}\, f$ never actually returns a result, we can assign it any codomain type $\beta$ whatsoever (but w.l.o.g. we can pick $\beta$ as the empty type, cf. section 3.1). Obviously, $\mathsf{loop}$ by itself is not very useful in a purely functional language, but if we add an escaping construct like exceptions or first-class continuations, it is easy to recover $\mathsf{repeat}$ from it – analogously to the way that a **loop-exit-endloop** construct can simulate **repeat-until** in an imperative language.

### 1.3. Exceptions and continuations

Likewise, let us quickly introduce the notation we will be using for exceptions and continuations. In its most primitive form, an exception facility consists of two special forms:

$$\mathbf{fail} \qquad \text{and} \qquad \mathbf{try}\ M_1\ \mathbf{else}\ M_2$$

When evaluated, **fail** *raises* an exception, which, if not *handled*, terminates program execution. The **try-else** construct first attempts to evaluate $M_1$. If this evaluation terminates normally with some value $V$, the result of the entire expression is $V$; $M_2$ is ignored. But if evaluation of $M_1$ raises an exception (not itself handled by an inner **try**), evaluation of $M_1$ is abandoned, and the result of $M_2$ is returned as the result of the whole expression. Any exception raised in $M_2$ propagates outward as usual and in particular does not reactivate $M_2$.

A simple generalization consists of data-carrying exceptions, where a value can be passed from a **fail** to the handler. This in turn enables named exceptions, but we will not need such generality here. In particular, ML's exception facility uses **raise** $X$ for **fail** and $M_1$ **handle** $X \Rightarrow M_2$ for **try**, where $X$ is an exception name. Scheme has no direct counterpart, though an exception facility can be simulated using call/cc and set!.

There are a number of essentially equivalent ways of introducing first-class continuations in a functional language, all tracing back to Reynolds's **escape**-operator [19] (or, less directly, to Landin's J-operator [14]). In general, we need an operator $\mathcal{C}$ such that an expression $\mathcal{C}M$ invokes the procedure $M$ with a representation $K$ of the *evaluation context* [6] surrounding $\mathcal{C}M$. If $M$ ever invokes $K$ with a value $V$, the then current context of evaluation is abandoned, and control returns to the context represented by $K$, as if $\mathcal{C}M$ had just returned $V$. For example,

$$2 + \mathcal{C}\,(\lambda k.\, 3 + k\,4) \longrightarrow 6$$

The difference from exceptions is that the entire captured context will be reactivated even if the continuation $K$ is actually returned out of the $\mathcal{C}$-expression (e.g., embedded in a closure or other data structure). While potentially more complex to implement, such a facility exhibits a pleasant uniformity of behavior, which makes it superior in many ways to exceptions – both for both theoretical and practical purposes.

For concreteness in the following, we will adopt Griffin's simply-typed formulation of first-class continuations [9], which uses essentially a typed variant of Felleisen's $\mathcal{C}$-operator [6]; the actual choice is not critical, however. We will generally emphasize applications of continuations as $k^\bullet v$. Similarly, we will write $\lambda^\bullet x.\, M$ for the syntactic representation of a continuation. And finally, we will use the notation $\neg\alpha$ for the type of $\alpha$-accepting continuations.

To a first approximation, readers familiar with the continuation facility in Standard ML of New Jersey [4] can simply read $\mathcal{C}$ as callcc, ignore $^\bullet$'s in $\lambda$-abstractions, and read $k^\bullet v$ as throw $k\ v$; Scheme programmers can read $\mathcal{C}$

as call/cc and ignore annotations of both abstractions and applications. A more precise characterization of $\mathcal{C}$ will be given in section 3.1.

### 1.4. The problem

A crucial property underlying the entire development presented here is that reduction in a simply-typed $\lambda$-calculus is strongly normalizing [12], and in particular a CBV strategy is sufficient to reduce every closed term to a value. It can be proved (by CPS conversion back to the original case) that the latter property holds even if we extend the language with first-class continuations [9]. A similar argument works for exceptions not carrying values, or values of *base type* only (if we allow functional values, the domain of exceptions becomes self-referential).

Now, if we have a language that allows recursive definitions, we can directly define fix by its characteristic equation; conversely, we can express all recursive definitions in terms of fix. And either of these can easily express the looping constructs. But we can also pose the question: does fix really give us greater expressive power than repeat or loop? Or could we in fact explicitly *define* a simply-typed function that behaves like fix but uses only iteration and control operators? Perhaps surprisingly, the answer is yes. In fact, we will give two such simulations: a simple, but somewhat problematic, version based on exceptions, and a much better one using general first-class continuations. We will show that the latter is essentially equivalent to the usual CBV fixpoint operator, and consider some implications of this equivalence.

## 2. Informal derivation

In this section, we give an intuitive, stepwise development of the solution. A more formal treatment can be found in section 3.

### 2.1. A first attempt: recursion from loops and exceptions

Let us look at the fixpoint equation again (ignoring for now the value requirements), and unfold it a few times:

$$\mathsf{fix}\, F\, a \;=\; F\,(\mathsf{fix}\, F)\, a \;=\; F\,(F\,(\mathsf{fix}\, F))\, a \;=\; \cdots \;=\; F^n\,(\mathsf{fix}\, F)\, a$$

While we could clearly keep expanding the definition of fix ad infinitum, it can be shown (e.g., [10, 4.4]) that any finite computation needs only a fixed number of $F$'s. In other words, for every *terminating* program (closed term of base type) there exists an $n$ such that if we replace $\mathsf{fix}\, F$ by $F^n\,(\lambda x.\,\mathbf{fail})$, the result is unchanged. Unfortunately, we cannot tell in advance how

many levels of unfolding will be needed, or in fact, whether there even exists a bound – if the program never terminates, any predetermined $n$ will eventually lead to failure.

What we can do, however, is to start the computation with some number of $F$s, and if it runs out (signals an error), restart it with more; eventually, we will either find a large enough $n$ or keep trying forever. We need to strike a balance between wasting work by underestimating the number of levels, and "overshooting" by setting up more $F$s than necessary. A good choice is to double the bound $n$ each time; this ensures that we will only make a constant factor more calls to $F$ than we actually need. We thus want a definition like the following:

$$\begin{aligned}\mathsf{fix}\, F \;=\; \lambda a.\, &\mathbf{try}\ F\,(\lambda x.\,\mathbf{fail})\,a\\ &\mathbf{else\ try}\ (F\circ F)\,(\lambda x.\,\mathbf{fail})\,a\\ &\quad\mathbf{else\ try}\ (F\circ F\circ F\circ F)\,(\lambda x.\,\mathbf{fail})\,a\\ &\qquad\mathbf{else}\ \ldots\end{aligned}$$

which can easily be turned into a finite, iterative procedure:

$$\begin{aligned}\mathsf{fix}\, F \;=\; \lambda a.\, \mathsf{repeat}\,[\lambda F'.\,&\mathbf{try}\ \mathsf{inl}\,(F'\,(\lambda x.\,\mathbf{fail})\,a)\\ &\mathbf{else}\ \mathsf{inr}\,(\lambda f.\,F'\,(F'\,f))]\,F\end{aligned}$$

(using the instance $\phi \to \beta + \phi$ of $\mathsf{repeat}$, where $\phi = (\alpha \to \beta) \to \alpha \to \beta$).

### 2.2. Analysis

So do we really have a working fixpoint combinator? Our initial intuition and some quick tests would say yes. For instance, the canonical example,

$$\mathsf{fix}\,[\lambda f.\,\lambda n.\,\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times f(n-1)]\,5$$

does indeed evaluate to 120. Even non-linear recursive definitions like the naive Fibonacci function work. On closer inspection, however, some serious problems become apparent:

1. The "iterative deepening" paradigm wastes work: we typically need a total of twice as many recursive calls to compute the same result, since the only information we recover from failed attempts is "$n$ was not big enough". This in itself is perhaps not too bad, but is closely related to:

2. The construction is not robust under language extensions: it interferes with almost all computational effects we might want to add. For example, any state manipulations done by the recursively-defined

function will be spuriously duplicated an unpredictable number of times:

$$(\textbf{let } i = \mathsf{ref}\, 0 \textbf{ in}\\ \quad \mathsf{fix}\,[\lambda f.\,\lambda n.\,(i := !i + 1;\ \textbf{if } n = 0 \textbf{ then } !i \textbf{ else } f(n-1))]\,5) \longrightarrow 13$$

A similar problem occurs with effects for communication, in particular I/O operations. Finally, the fixpoint definition monopolizes the exception facility for its own internal purposes, seriously limiting its general usability. In fact, our fix even interferes with itself:

*3.* Nested fixes may not work. For example, in

$$\mathsf{fix}\,(\lambda f.\,\lambda n.\,\textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } \mathsf{fix}\,(\lambda f'.\,\lambda n.\,f\,n)\,(n-1))\,5$$

the definition above will uselessly keep increasing the bound for $f'$ instead of for $f$. To deal with this, we not only need a *named* exception facility, but the ability to *dynamically* generate new exception names. And in fact, even such a scheme is not general enough:

*4.* The definition only works for first-order recursive functions! In particular, we cannot use it to define recursive data structures with embedded functional components, or even curried functions. Consider the following:

$$\mathsf{fix}\,(\lambda a.\,\lambda m.\,\lambda n.\,\textbf{if } m = 0 \textbf{ then } n \textbf{ else } a\,(m-1)\,(n+1))\,3\,4$$

It is easy to see that this definition returns the body $\lambda n, \ldots$ out of the scope of the handler for $a$; after the first recursive call, execution terminates with an unhandled exception.

The first two of these shortcomings are inherent to the basic approach, and there is little we can do about them. But the last two, and perhaps more serious ones are only due to our specific choice of exceptions as the aborting mechanism. In other words, while an exception facility may be a natural feature in a first-order language, it can easily lead to undesirable results when used with higher-order functions – very much like dynamically-scoped variables in Lisp 1.5.

In fact, we recognize problem 3 as essentially the "downward funarg" problem: shadowing of lexical variables (even the "fix" is similar: use unique names). Problem 4 corresponds to an "upward funarg": a function returned from within a **try**-expression retains no record of its associated exception handler. Both can be summarized in the standard observation that the meaning of an exception (i.e., the handler associated to it) is being inappropriately determined by the context of *use* rather than of *definition.* And a partial solution is indeed to use a statically-scoped construct, as shown next.

### 2.3. From exceptions to continuations

To make explicit the context in which to restart a computation that has exceeded its current unfolding bound, we simply replace exceptions with continuations:

$$\mathsf{fix}\, F \;=\; \lambda a.\, \mathsf{repeat}\, [\lambda F'.\mathcal{C}(\lambda^{\bullet} k.\, k^{\bullet}(\mathsf{inl}\,(F'\,[\lambda x.\mathcal{C}(\lambda^{\bullet} d.\, k^{\bullet}(\mathsf{inr}\,(F' \circ F')))]a)))]\, F$$

(The unused continuation $d$ explicitly shows that a context is being discarded.) This continuation-based approach solves the dynamic-scoping problem of exceptions, but does not address our concerns about efficiency and re-execution of computational effects. In fact, such problems are taken to their logical conclusion: if the recursively-defined function returns a closure (containing an embedded continuation), even effects outside of the function body may be duplicated.[1] Note that we are no worse off than before, however: the original definition did not work at all in such cases.

So, to a certain extent, we can simulate recursive definitions with continuations. However, the definition is wasteful, interacts poorly with non-functional extensions of the language, and looks ill-suited for formal reasoning about recursion (notably, relating our defined fix to the underlying domain-theoretical fixpoint). Fortunately, we can do much better.

### 2.4. A proper solution: recursion from loops and continuations

While the above approach has some serious flaws, it contains a core of truth. The problems can be traced back to the fact that the current application context is simply discarded upon reiteration, forcing us to repeatedly recompute the same information. However, the power of full first-class continuations allows us instead to add new levels of recursion "retroactively" as they are needed, rather than committing to a fixed number at the outset.

Consider a recursive function definition in continuation-passing style:

$$\mathsf{fac_c} \;=\; \lambda n.\, \lambda c.\, \mathbf{if}\; n = 0\; \mathbf{then}\; c\, 1\; \mathbf{else}\; \mathsf{fac_c}\,(n-1)\,(\lambda r.\, c\,(n \times r))$$

By general properties of CPS, the recursion has been turned into tail-recursion. The observation we need to focus on is that the entire information about a recursive call is encoded by the pair $(n,c)$ of argument value and return continuation; we will call such a pair an *application context*. Our solution will center around making these contexts explicit without actually converting the program to continuation-passing style.

[1] Particularly problematic are effects on the top-level environment in an interactive setting. This is in fact a general problem with continuations in a typed language, and is currently handled by a run-time test for "stale" continuations in the top-level loop of SML/NJ.

In the examples below, we will use the following abbreviation:

$$\mathsf{FAC} \;=\; \lambda f.\,\lambda n.\,\textbf{if}\; n = 0\; \textbf{then}\; 1\; \textbf{else}\; n \times f(n-1)$$

We need to capture the context of a recursive call, in order to be able to reinstate it later. Our first step is therefore to define the following:

$$\mathsf{switch} \;=\; \lambda l.\,\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\, l^{\bullet}(x, q)) \;:\; \neg(\alpha \times \neg\beta) \to \alpha \to \beta$$

$\mathsf{switch}\, l$ has the type of an ordinary procedure, but when applied to an $\alpha$-typed value in a $\beta$-expecting context, it will capture and pass these to $l$. For example,

$$k_0\,(3 \times \mathsf{switch}\, k_1\,(3-1)) \;\longrightarrow\; k_1\,(2, \lambda a.\, k_0\,(3 \times a))$$

Using $\mathsf{switch}$ as our recursion base (instead of $\lambda x.\,\textbf{fail}$), we can define $\mathsf{step}$, which proceeds until the next recursive call:

$$\begin{aligned} \mathsf{step} \;&=\; \lambda F.\,\lambda(v, c).\,\mathcal{C}(\lambda^{\bullet} l.\, c^{\bullet}(F\,(\mathsf{switch}\, l)\, v)) \\ \mathsf{step} \;&:\; ((\alpha \to \beta) \to \gamma \to \delta) \to \gamma \times \neg\delta \to \alpha \times \neg\beta \end{aligned}$$

(In our definition of $\mathsf{fix}$ we will only use the instance $\gamma = \alpha$, $\delta = \beta$.) $\mathsf{step}$ expresses $F$ as an application-context transformer, mapping an application context for $F\,f$ to one for the first call of $f$. If $F$ never applies $f$, $\mathsf{step}\, F$ does not return. Continuing our example, we have:

$$\begin{aligned} k_1\,(\mathsf{step}\; \mathsf{FAC}\; (3, k_0)) \;&\longrightarrow\; k_1\,(2, \lambda a.\, k_0\,(3 \times a)) \\ k_1\,(\mathsf{step}\; \mathsf{FAC}\; (0, k_0)) \;&\longrightarrow\; k_0\, 1 \end{aligned}$$

Finally, we can define $\mathsf{fix}$, which sets up the initial continuation, and then repeatedly steps through the recursive calls:

$$\begin{aligned} \mathsf{fix} \;&=\; \lambda F.\,\lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\, \mathsf{loop}\;(\mathsf{step}\; F)\,(a, r)) \\ \mathsf{fix} \;&:\; ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \end{aligned}$$

so that

$$\begin{aligned} k_0\,(\mathsf{fix}\; \mathsf{FAC}\; 3) \;&\longrightarrow\; \mathsf{loop}\;(\mathsf{step}\; \mathsf{FAC})\,(3, k_0) \\ &\longrightarrow\; \mathsf{loop}\;(\mathsf{step}\; \mathsf{FAC})\,(\mathsf{step}\; \mathsf{FAC}\; (3, k_0)) \\ &\longrightarrow\; \mathsf{loop}\;(\mathsf{step}\; \mathsf{FAC})\,(2, \lambda a.\, k_0\,(3 \times a)) \\ &\longrightarrow^{*}\; \mathsf{loop}\;(\mathsf{step}\; \mathsf{FAC})\,(0, \lambda a.\, k_0\,(3 \times (2 \times (1 \times a)))) \\ &\longrightarrow\; (\lambda a.\, k_0\,(3 \times (2 \times (1 \times a))))\, 1 \\ &\longrightarrow\; k_0\,(3 \times (2 \times (1 \times 1))) \\ &\longrightarrow\; k_0\, 6 \end{aligned}$$

And in fact, as we will see next, this construction also works for non-linear recursion, higher-order types, and does not duplicate computational effects.

## 3. Formalization

In this section, we will analyze the solution derived above. First, however, we must pin down an exact formulation of first-class continuations.

### 3.1. First-class continuations in a typed setting

There are two major approaches to extending a typed functional language with first-class continuations. The one taken by SML/NJ is based on introducing a new type constructor $\neg\alpha$ to represent continuations accepting $\alpha$-typed values. Continuations are captured with the operator $\mathsf{callcc} : (\neg\alpha \to \alpha) \to \alpha$ and invoked with $\mathsf{throw} : \neg\alpha \to \alpha \to \beta$. The example in the introduction would thus be written as:

$$2 + \mathsf{callcc}(\lambda k.\, 3 + \mathsf{throw}\, k\, 4)$$

The details can be found in [4]; in the context of full ML, one must also worry about potential interactions with **let**-polymorphism [11].

The main pragmatic problem with this approach is that it is awkward to "prepend" an ordinary procedure $f : \alpha \to \beta$ to a continuation $k : \neg\beta$ and get a new continuation $(k \circ f) : \neg\alpha$. More generally, there is no convenient *syntactic* representation of first-class continuations, making an equational theory of program behavior painful at best. On the other hand, having a distinct type of first-class continuations simplifies reasoning (both manual and automated) about CPS versions of programs: unlike an ordinary procedure, a first-class continuation does not itself take an extra, effectively useless, continuation parameter in CPS.

The other main alternative is to represent continuations directly as procedures that do not return. We can make this restriction explicit by using the existing function space $\alpha \to \beta$ with an empty (i.e., containing no closed *values*) codomain type $\beta = 0$, and simply define $\neg\alpha$ as an abbreviation of $\alpha \to 0$. In this setting, the details work out more smoothly if the control operator not only captures (a copy of) the surrounding evaluation context, but also removes it, so that it can only be resumed by an explicit application of the reified continuation [6]. Then the first-class continuation facility can be represented by a single operator $\mathcal{C} : ((\alpha \to 0) \to 0) \to \alpha$. To apply a continuation in (i.e., to escape from) a context expecting a $\beta$-typed result, we can use a second $\mathcal{C}$ to explicitly discard the inner context. Our example would thus be written as:

$$2 + \mathcal{C}(\lambda^{\bullet} k.\, k^{\bullet}(3 + \mathcal{C}(\lambda^{\bullet} d.\, k^{\bullet} 4))) \longrightarrow 6$$

(It is customary to abbreviate the idiom $\lambda z.\mathcal{C}(\lambda^{\bullet} d.\, z) : 0 \to \alpha$ as $\mathcal{A}$, usually pronounced "abort".) This approach to typed first-class continuations is

taken in Griffin's variant of Idealized Scheme [9] (which is actually much closer to ML than to Scheme; its only Scheme-inspired characteristic is the control operator), and it is the one we will be using in the following.[2]

However, the two styles are essentially equivalent [4, 9]. In particular, we can define the SML/NJ operators as

$$\begin{aligned} \mathsf{callcc} &= \lambda f.\mathcal{C}(\lambda^{\bullet} k.k^{\bullet}(f\,k)) \\ \mathsf{throw} &= \lambda k.\lambda x.\mathcal{A}(k^{\bullet}x) \end{aligned}$$

Conversely, given the SML/NJ primitives, we can define a $\mathcal{C}$-operator. A slight problem is that ML does not have a predefined empty type 0. Its natural definition would be as a **datatype** with no summands, but the syntax does not allow this. Instead we can define

$$\textbf{datatype}\ \mathsf{void} = \mathsf{VOID}\ \textbf{of}\ \mathsf{void}$$

$$\textbf{fun}\ \mathsf{i}(\mathsf{VOID}\,v) = \mathsf{i}\,v : \alpha$$

corresponding to the (inductive) type $\mu t.\,t$. The function i is the inclusion from the empty type into another, defined by a degenerate form of primitive recursion. It does not terminate because it does not even begin: since there are no values of type void, i can never actually be invoked in a CBV setting, so its definition does not matter; we could equally well have made the body a **fail** or $\mathcal{A}\,v$. In category-theoretical terms, 0 is an initial object for CBV types and terms, and i is the associated unique morphism from 0 to any type $\alpha$.

Given 0 and i, we can now define $\mathcal{C}$ as:

$$\mathcal{C} = \lambda f.\,\mathsf{callcc}\,(\lambda k.\,\mathsf{i}\,(f\,(\lambda x.\,(\mathsf{throw}\,k\,x) : \mathsf{void})))$$

It is worth emphasizing that in our typed setting, $\mathcal{A}$ actually has absolutely no "aborting" operational behavior – it only serves to keep the types matching up. In fact, expanding its definition in terms of SML/NJ primitives gives simply:

$$\mathcal{A} = \lambda z.\mathcal{C}(\lambda^{\bullet} d.\,z) = \lambda z.\,\mathsf{callcc}\,(\lambda k.\,\mathsf{i}\,([\lambda^{\bullet} d.\,z]\,(\lambda x.\,\mathsf{throw}\,k\,x)) = \lambda z.\mathsf{i}\,z = \mathsf{i}$$

In particular, the instance $\mathcal{A} : 0 \to 0$ is just the identity function on the empty type.

[2] In fact, we can get a reasonable third alternative by taking only negation and products as primitives and *define* $\alpha \to \beta$ as $\neg(\alpha \times \neg\beta)$. While such a minimalist approach has merits (in particular, it seems well suited for reasoning about both direct-style and continuation-passing style versions of a term), we prefer for now a presentation more directly related to existing languages and formalisms.

### 3.2. Equational reasoning about continuations

Before we commence with actual proofs, let us briefly outline the general methodology. The goal is, as usual, to show that certain terms can be substituted for others without changing the meaning of a program. However, rather than attempting to enumerate all such possible replacements valid for a particular language, we take the *axiomatic* (aka. *logical*) approach and concentrate on proving equivalences that hold in *all* languages whose equivalence theories include a set of axioms.

This gives us a more conservative notion of equivalence, but the resulting theory is usually much more robust under language extensions: if we derive a result from a set of axioms, it also holds in any "well-behaved" extension of the language with additional constructs. For example, Moggi's computational $\lambda$-calculus (a strict superset of Plotkin's $\lambda_v$-calculus) is valid for CBV functional languages with a large variety of computational effects (state, nondeterminism, exceptions, continuations, etc.) [16]. The inherent modularity of the axiomatic approach makes feasible program-behavior theories of considerable scope and generality, e.g., [22].

Of particular interest to us are equational theories $Th$ (usually axiomatized as "core" of generic equations together with a set of $\delta$-like rules for the primitive operations) that are "evaluation-complete", in the sense that for any program $M$ (closed term of base type) and value $V$,

$$M \Downarrow V \quad \text{iff} \quad Th \vdash M = V$$

where $M \Downarrow V$ means that $M$ evaluates to $V$. (The "only if" direction is valid for all types, but two functional terms may be provably equal without one evaluating to the other.) Such a property tells us that when a program evaluates to a value, we can "tell why", i.e., what equivalences the evaluation depended on. In particular we only need to verify that an implementation – within the language or externally to it – of any feature (like continuations, state, or fixpoints) satisfies its axiomatic description, to ensure that any program using the implementation will get the correct result.

An axiomatization of CBV $\lambda$-calculus equivalence with this property was developed by Plotkin [17]; while the theory itself is quite simple, the actual proof of the biimplication is non-trivial. (An alternative is to take the above as the declarative *definition* of evaluation; the challenge is then to develop an effective procedure for proving programs equal to values). Both Plotkin's proof technique and others (e.g., taking advantage of typing to use logical relations) can be generalized to larger languages and different evaluation orders (see, e.g., [10] for examples and further references). In our proofs, we will be using Felleisen's extensions for modeling control operators.

Since the language we are using is a proper subset of Felleisen's untyped one, we can directly use the equational reasoning principles developed for the latter. In particular, we have the following rules taken from [5, 20], with the addition of $(\mathcal{A}_{uniq})$, which captures the property noted above about 0. Further, this rule was used to replace two instances of $\mathcal{A}$ in Felleisen's original rules with the identity functions, which were in turn eliminated using $(\beta_\Omega)$:

$$\begin{array}{rcll}
(\lambda x.\,M)\,V & = & [V/x]M & & (\beta_v)\\
(\lambda x.\,E[x])\,M & = & E[M] & (x \notin FV(M)) & (\beta_\Omega)\\
E[(\lambda x.\,M)\,M'] & = & (\lambda x.\,E[M])\,M' & (x \notin FV(E)) & (\beta_{lift})\\
(\lambda x.\,V\,x) & = & V & (x \notin FV(V)) & (\eta_v)\\
\mathcal{C}(\lambda^\bullet k.\,k^\bullet M) & = & M & (k \notin FV(M)) & (\mathcal{C}_{elim})\\
V\,(\mathcal{C}M) & = & \mathcal{C}(\lambda^\bullet k.\,M^\bullet(\lambda^\bullet x.\,k^\bullet(V\,x))) & (k,x \notin FV(V,M)) & (\mathcal{C}_{lift})\\
\mathcal{C}(\lambda^\bullet k.\mathcal{C}M) & = & \mathcal{C}(\lambda^\bullet k.\,M^\bullet(\lambda^\bullet x.\,x)) & & (\mathcal{C}_{idem})\\
V & = & \mathcal{A} & (V : 0 \to \alpha) & (\mathcal{A}_{uniq})
\end{array}$$

Here, $V$'s are values and $M$'s are arbitrary expressions, $E[\cdot]$ is an evaluation context (i.e., a context such that in $E[M]$, $M$ will be evaluated next; in particular, $E[\cdot]$ may be empty, giving the rule $(\lambda x.\,x)\,M = M$ $(\beta_{id})$). Remember also that $^\bullet$-annotated abstractions and applications are special cases of the general forms, and so are also covered by the above laws.

### 3.3. Correctness proof

The failures of our original, exception-based solution should have alerted us to the fact that several things can go wrong. How do we know that the "proper" definition of fix in section 2.4 contains no such surprises? We need to prove that fix actually has the expected behavior in all cases. We will do this in two different ways, since each provides useful insight. In this section, we use direct-style reasoning to show that fix satisfies its defining equation; in section 4.1 we consider the definition of fix in continuation-passing style.

Let us first formalize the meaning of iteration and recursion operators.

**Definition 1** *A (CBV)* iteration operator *is a type-indexed family of functions* $\mathsf{loop}_\alpha : (\alpha \to \alpha) \to \alpha \to 0$ *such that for any value* $f : \alpha \to \alpha$,

$$\mathsf{loop}_\alpha\, f \;=\; \lambda x.\,\mathsf{loop}_\alpha\, f\,(f\,x) \qquad\qquad (loop)$$

*A (CBV)* recursion operator *is a type-indexed family of functions* $\mathsf{fix}_{\alpha,\beta} : (\phi \to \phi) \to \phi$, *where* $\phi = \alpha \to \beta$, *such that for any value* $F : \phi \to \phi$,

$$\mathsf{fix}_{\alpha,\beta}\, F \;=\; \lambda a.\,F\,(\mathsf{fix}_{\alpha,\beta}\, F)\,a \qquad\qquad (fix)$$

(We will usually omit the type subscripts where they are clear form the context.) Coalescing the definitions of section 2.4, we can now state:

**Proposition 1** *If* $\mathsf{loop}$ *is an iteration operator, the* $\mathsf{fix}$ *defined by*

$$\mathsf{fix}_{\alpha,\beta} = \lambda F.\,\lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}_{\alpha\times\neg\beta}\,[\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,c^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,v))]\,(a,r))$$

*is a recursion operator (denoted by* $\mathsf{FIX}(\mathsf{loop})$ *in the following).*

**Proof:** Let $F$ be given. Then

$$\begin{array}{ll}
\mathsf{fix}\,F & \\
\overset{\mathrm{def}}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}\,\overbrace{[\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,c^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,v))]}^{B_F}\,(a,r)) \\
\overset{loop}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,[\lambda x.\,\mathsf{loop}\,B_F\,(B_F\,x)]\,(a,r)) \\
\overset{2\times\beta v}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}\,B_F\,(\mathcal{C}(\lambda^{\bullet} l.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,a)))) \\
\overset{\mathcal{C}_{lift}}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathcal{C}(\lambda k.\,[\lambda^{\bullet} l.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,a)]^{\bullet} \\
 & \qquad\qquad (\lambda^{\bullet} t.\,k^{\bullet}(\mathsf{loop}\,B_F\,t)))) \\
\overset{2\times\beta v}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathcal{C}(\lambda^{\bullet} k.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,k^{\bullet}(\mathsf{loop}\,B_F\,(x,q)))]\,a))) \\
\overset{\mathcal{C}_{idem}}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,[\lambda^{\bullet} k.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,k^{\bullet}(\mathsf{loop}\,B_F\,(x,q)))]\,a)]^{\bullet}(\lambda^{\bullet} x.\,x)) \\
\overset{\beta v}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,(\lambda^{\bullet} x.\,x)^{\bullet}(\mathsf{loop}\,B_F\,(x,q)))]\,a)) \\
\overset{\beta_{id}}{=} & \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,r^{\bullet}(F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,\mathsf{loop}\,B_F\,(x,q))]\,a)) \\
\overset{\mathcal{C}_{elim}}{=} & \lambda a.\,F\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,\mathsf{loop}\,B_F\,(x,q))]\,a \\
\overset{2\times\alpha}{=} & \lambda a.\,F\,[\lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}\,B_F\,(a,r))]\,a \\
\overset{\mathrm{def}}{=} & \lambda a.\,F\,(\mathsf{fix}\,F)\,a
\end{array}$$

(In the step using $(\mathcal{C}_{lift})$ we have used the fact that $\mathsf{loop}\,B_F$, while not syntactically a value, is equal to one by $(loop)$.) ∎

This proof is completely generic in $F$, and hence also applicable to $F$'s which themselves make use of first-class continuations. And since the equational rules are valid even for languages with stores or other effects, we know that the construction will work correctly in any such extension.

Although it almost goes without saying, let us verify at this point that iteration is indeed definable from recursion:

**Proposition 2** *If* $\mathsf{fix}$ *is a recursion operator, the* $\mathsf{loop}$ *defined by*

$$\mathsf{loop}_{\alpha} = \lambda f.\,\mathsf{fix}_{\alpha,0}\,[\lambda l.\,\lambda x.\,l\,(f\,x)]$$

*is an iteration operator (denoted by* $\mathsf{LOOP}(\mathsf{fix})$ *in the following).*

**Proof:** For arbitrary $f$,

$$\begin{array}{rcl} \mathsf{loop}\, f & \stackrel{\mathrm{def}}{=} & \mathsf{fix}\,[\lambda l.\,\lambda x.\,l\,(f\,x)] \\ & \stackrel{fix}{=} & \lambda a.\,[\lambda l.\,\lambda x.\,l\,(f\,x)]\,(\mathsf{fix}\,[\lambda l.\,\lambda x.\,l\,(f\,x)])\,a \\ & \stackrel{2\times\beta_v}{=} & \lambda a.\,(\mathsf{fix}\,[\lambda l.\,\lambda x.\,l\,(f\,x)])\,(f\,a) \\ & \stackrel{\mathrm{def}}{=} & \lambda a.\,\mathsf{loop}\,f\,(f\,a) \end{array}$$

∎

This tells us how to obtain recursion from iteration as well as vice versa. But we can get an even stronger result, as we will see next.

### 3.4. Uniformity

Suppose we start with a "real" language like SML/NJ or Scheme that already includes full recursive definitions of functions. We can directly define loop in such a language, check that it satisfies the iteration equation, and then proceed with reasoning about recursive programs using the "recursion from iteration" approach. We already know that everything we could prove using the CBV fixpoint equation, we can obtain from only (*loop*) and the definition of fix. But might it happen that we could also prove something that would *not* be true of the original fixpoints? In particular, could a program using the new fix terminate normally (i.e., be equal to a value) while the same program written with the original fix looped? Fortunately, given an additional, natural constraint on the recursion operator, the answer is no. This extra condition is *uniformity*.

In domain theory, a *fixpoint operator* is a CPO-indexed family of functions $\mathsf{fix}_\alpha : (\alpha \to \alpha) \to \alpha$ such that for any $F : \alpha \to \alpha$, $\mathsf{fix}_\alpha(F) = F(\mathsf{fix}_\alpha(F))$. Such a fix is said to be *uniform* if for any $F : \alpha \to \alpha$, $G : \alpha' \to \alpha'$, and *strict* $H : \alpha \to \alpha'$,

$$H \circ F = G \circ H \;\Rightarrow\; H(\mathsf{fix}_\alpha(F)) = \mathsf{fix}_{\alpha'}(G)$$

The intuition behind this condition is that we expect to have $H(\mathsf{fix}(F)) = H(F(F(F(\cdots)))) = G(H(F(F(\cdots)))) = \cdots = G(G(G(\cdots))) = \mathsf{fix}(G)$.[3]

In an arbitrary CBV language with first-class continuations and possibly other effects, it is not completely clear what the exact axiomatic analog of uniformity should be. However, the following definitions will suffice for our purposes:

[3] Uniformity does not follow from the fixpoint equation alone. For example, consider a fixpoint operator with $\mathsf{fix}_{N_\perp}(F) = (F(0) = 0) \to 0 \,[]\, F(\perp)$. It is easy to check that $\mathsf{fix}(F)$ is always a fixpoint of $F : N_\perp \to N_\perp$, but taking $F = G = id$, $H = succ$, we have $H \circ F = G \circ H$, yet $H(\mathsf{fix}(F)) = succ(0) \neq 0 = \mathsf{fix}(G)$. And in fact, one can show that in the particular framework of CPO's and *continuous* functions, the uniformity condition is equivalent to finding *least* fixpoints [10, Thm. 4.18].

**Definition 2** *We say that a CBV function $h : \alpha \to \beta$ is* total *if for every value $a : \alpha$, there exists a value $b : \beta$ such that $h\,a = b$. A functional $H : (\alpha_1 \to \alpha_2) \to \beta_1 \to \beta_2$ is called* rigid *if there exist total $H_1$ and $H_2$ such that*

$$H = \lambda t.\, \lambda z.\, (H_1\, z)\, (t\, (H_2\, z))$$

*(This is a stronger condition than domain-theoretic strictness, and ensures that $H$ will actually apply its argument).*

**Definition 3** *A CBV recursion operator will be called* uniform *if for any $F : \phi \to \phi$, $G : \phi' \to \phi'$, and rigid $H : \phi \to \phi'$ ($\phi = \alpha \to \beta$, $\phi' = \alpha' \to \beta'$),*

$$H \circ (\lambda t.\, \lambda z.\, F\, t\, z) = G \circ H \;\Rightarrow\; H(\mathsf{fix}_{\alpha,\beta}\, F) = \mathsf{fix}_{\alpha',\beta'}\, G$$

*(where $f_1 \circ f_2$ is CBV composition $\lambda x.\, f_1\, (f_2\, x)$).*

The double $\eta$-wrapping of $F$ is necessary for the same reason that we need it in the CBV fixpoint equation: $F\, t$ may not be a value even if $t$ is.

First, let us check that uniformity is preserved by our fix-construction, i.e., that we get "uniform recursion from uniform iteration". Again, the uniformity condition is somewhat simpler for iteration:

**Definition 4** *We say that an iteration operator* loop *is uniform if for any $f : \alpha \to \alpha$, $g : \alpha' \to \alpha'$ and* total *$h : \alpha' \to \alpha$,*

$$f \circ h = h \circ g \;\Rightarrow\; (\mathsf{loop}_\alpha\, f) \circ h = \mathsf{loop}_{\alpha'}\, g$$

**Proposition 3** *If* loop *is uniform, so is* FIX(loop).

> **Proof:** Given any $F$, $G$, and $H$ satisfying the premise of the fix-uniformity condition above, let $B_-$ be as in the proof of Prop. 1, and take
>
> $$h = \lambda(u,k).\, (H_2\, u, \lambda^\bullet t.\, k^\bullet (H_1\, u\, t))$$
>
> By our assumptions on $H_1$ and $H_2$, $h$ is a total function. Using the equational theory of CBV with continuations, it is straightforward (though slightly tedious) to verify that $B_F \circ h = h \circ B_G$, meaning that we can use the uniformity of loop to get (taking somewhat larger steps in the equational proofs):
>
> $$\begin{aligned} H(\mathsf{fix}\, F) &= \lambda u.\, (H_1\, u)\, (\mathsf{fix}\, F\, (H_2\, u)) \\ &= \lambda u.\, (H_1\, u)\, (\mathcal{C}(\lambda^\bullet r.\, \mathsf{loop}\, B_F\, (H_2\, u, r))) \\ &= \lambda u.\, \mathcal{C}(\lambda^\bullet k.\, \mathsf{loop}\, B_F\, (H_2\, u, \lambda^\bullet t.\, k^\bullet (H_1\, u\, t))) \\ &= \lambda u.\, \mathcal{C}(\lambda^\bullet k.\, \mathsf{loop}\, B_F\, (h\, (u,k))) \\ &= \lambda u.\, \mathcal{C}(\lambda^\bullet k.\, \mathsf{loop}\, B_G\, (u,k)) \\ &= \mathsf{fix}\, G \end{aligned}$$
>
> I.e., the defined fix is uniform. ∎

Conversely,

**Proposition 4** *If* fix *is uniform, so is* LOOP(fix).

**Proof:** Let $f$, $g$, and total $h$ be given, with $f \circ h = h \circ g$. Take

$$F = \lambda s.\, \lambda x.\, s\,(f\,x), \quad G = \lambda s.\, \lambda x.\, s\,(g\,x), \quad H = \lambda u.\, \lambda y.\, u\,(h\,y)$$

This $H$ is rigid (with $H_1 = \lambda y.\, \lambda z.\, z$, $H_2 = h$), so

$$\mathsf{loop}\, f \circ h \;=\; (\mathsf{fix}\, F) \circ h \;=\; H\,(\mathsf{fix}\, F) \;=\; \mathsf{fix}\, G \;=\; \mathsf{loop}\, g$$

because

$$H \circ [\lambda t.\, \lambda z.\, F\,t\,z] \;=\; \lambda t.\, \lambda y.\, t\,(f\,(h\,y)) \;=\; \lambda u.\, \lambda x.\, u\,(h\,(g\,x)) \;=\; G \circ H$$

∎

Returning to the problem mentioned at the beginning of this section, we can now state:

**Proposition 5** *Let* fix' *be a* uniform *recursion operator. Define an iteration operator* loop = LOOP(fix'), *and let* fix = FIX(loop). *Then* fix = fix'.

**Proof:** For an arbitrary $F$ let

$$G \;=\; \lambda s.\, \lambda(v,c).\, s\,(B_F(v,c)) \quad \text{and} \quad H \;=\; \lambda t.\, \lambda(a,r).\, r^{\bullet}(t\,a)$$

$H$ is rigid (with $H_1 = \lambda(a,r).\, \lambda x.\, r^{\bullet} x$ and $H_2 = \lambda(a,r).\, a$), and the fix-uniformity precondition is satisfied:

$$\begin{aligned}
G \circ H &= \lambda u.\, [\lambda s.\, \lambda(v,c).\, s\,(B_F\,(v,c))]([\lambda t.\, \lambda(a,r).\, r^{\bullet}(t\,a)]\,u) \\
&= \lambda u.\, \lambda(v,c).\, [\lambda(a,r).\, r^{\bullet}(u\,a)]\,(B_F\,(v,c)) \\
&= \lambda u.\, \lambda(v,c).\, [\lambda(a,r).\, r^{\bullet}(u\,a)] \\
&\qquad\qquad (\mathcal{C}(\lambda^{\bullet} l.\, c^{\bullet}(F\,[\lambda x.\, \mathcal{C}(\lambda^{\bullet} q.\, l^{\bullet}(x,q))]\,v))) \\
&= \lambda u.\, \lambda(v,c).\, \mathcal{C}(\lambda^{\bullet} k.\, c^{\bullet}(F\,[\lambda x.\, \mathcal{C}(\lambda^{\bullet} q.\, k^{\bullet}(q^{\bullet}(u\,x)))]\,v)) \\
&= \lambda u.\, \lambda(v,c).\, c^{\bullet}(F\,[\lambda x.\, \mathcal{C}(\lambda^{\bullet} q.\, q^{\bullet}(u\,x))]\,v) \\
&= \lambda u.\, \lambda(v,c).\, c^{\bullet}(F\,[\lambda x.\, u\,x]\,v) \\
&= \lambda u.\, \lambda(v,c).\, c^{\bullet}(F\,u\,v) \\
&= \lambda u.\, [\lambda t.\, \lambda(v,c).\, c^{\bullet}(t\,v)]\,(\lambda z.\, F\,u\,z) \\
&= \lambda u.\, H\,(\lambda z.\, F\,u\,z) \\
&= H \circ (\lambda t.\, \lambda z.\, F\,t\,z)
\end{aligned}$$

so we can use uniformity of $\mathsf{fix}'$ to get (remember that by the fixpoint equation $\mathsf{fix}'\,F$ is a value):

$$\begin{aligned}
\mathsf{fix}\,F &= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}\,B_F\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,[\mathsf{fix}'\,(\lambda l.\,\lambda p.\,l\,(B_F\,p))]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{fix}'\,G\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,H\,(\mathsf{fix}'\,F)\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,[\lambda t.\,\lambda(a,r).\,r^{\bullet}(t\,a)]\,(\mathsf{fix}'\,F)\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,r^{\bullet}(\mathsf{fix}'\,F\,a)) \\
&= \lambda a.\,\mathsf{fix}'\,F\,a \\
&= \mathsf{fix}'\,F
\end{aligned}$$

Thus, since $F$ was arbitrary, we have $\mathsf{fix} = \mathsf{fix}'$. ∎

In particular, there is no way to distinguish the defined $\mathsf{fix}$ from the "native" $\mathsf{fix}'$ in any program context. For completeness, let us also note the converse result:

**Proposition 6** *Let* $\mathsf{loop}'$ *be a uniform iteration operator,* $\mathsf{fix} = \mathsf{FIX}(\mathsf{loop}')$, *and* $\mathsf{loop} = \mathsf{LOOP}(\mathsf{fix})$. *Then* $\mathsf{loop} = \mathsf{loop}'$.

**Proof:** Let $f$ be arbitrary. Then

$$\begin{aligned}
&\mathsf{loop}\,f \\
&= \mathsf{fix}\,(\lambda s.\,\lambda y.\,s\,(f\,y)) \\
&= [\lambda F.\,\lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,B_F\,(a,r))]\,(\lambda s.\,\lambda y.\,s\,(f\,y)) \\
&= \lambda a.\,\mathcal{C}\,(\lambda^{\bullet} r.\,\mathsf{loop}' \\
&\qquad [\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,c^{\bullet}([\lambda s.\,\lambda y.\,s\,(f\,y)]\,[\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,v))]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,[\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,c^{\bullet}([\lambda x.\,\mathcal{C}(\lambda^{\bullet} q.\,l^{\bullet}(x,q))]\,(f\,v)))]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,[\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,[\lambda x.\,l^{\bullet}(x,c)]\,(f\,v))]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,[\lambda(v,c).\,\mathcal{C}(\lambda^{\bullet} l.\,l^{\bullet}(f\,v,c))]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,[\lambda(v,c).\,(f\,v,c)]\,(a,r)) \\
&= \lambda a.\,\mathcal{C}(\lambda^{\bullet} r.\,\mathsf{loop}'\,f\,([\lambda(a,r).\,a]\,(a,r))) \\
&= \lambda a.\,\mathcal{A}\,(\mathsf{loop}'\,f\,a) \\
&= \lambda a.\,\mathsf{loop}'\,f\,a \\
&= \mathsf{loop}'\,f
\end{aligned}$$

using uniformity of $\mathsf{loop}'$ and the fact that

$$f \circ [\lambda(a,r).\,a] \;=\; [\lambda(a,r).\,a] \circ [\lambda(v,c).\,(f\,v,c)]$$

(expressing the observation that the continuation is loop-invariant, as expectable from a tail-recursive definition). ∎

Summarizing the results, iteration and recursion are related as follows:

**Theorem** *In any simply-typed CBV language with first-class continuations, there exist parameterized values* $\mathsf{FIX}(\cdot)$ *and* $\mathsf{LOOP}(\cdot)$ *with the following properties:*

1. *If* $\mathsf{loop}$ *is an iteration operator,* $\mathsf{FIX}(\mathsf{loop})$ *is a recursion operator; if* $\mathsf{loop}$ *is uniform, so is* $\mathsf{FIX}(\mathsf{loop})$*, and then* $\mathsf{LOOP}(\mathsf{FIX}(\mathsf{loop})) = \mathsf{loop}$.
2. *If* $\mathsf{fix}$ *is a recursion operator,* $\mathsf{LOOP}(\mathsf{fix})$ *is an iteration operator; if* $\mathsf{fix}$ *is uniform, so is* $\mathsf{LOOP}(\mathsf{fix})$*, and then* $\mathsf{FIX}(\mathsf{LOOP}(\mathsf{fix})) = \mathsf{fix}$.

Given this equivalence, one may wonder why the construction of $\mathsf{fix}$ from $\mathsf{loop}$ appears so much more complicated than the converse. Part of the reason is that the $\lambda$-syntax used for expressing them is biased towards reasoning directly about values, while continuations must first be reified to be denotable. In a more abstract notation (cf. section 5) the definition of $\mathsf{fix}$ from $\mathsf{loop}$ is no larger than the natural definition of $\mathsf{loop}$ from $\mathsf{fix}$.

## 4. Iteration and recursion in perspective

In this section, we will show some additional results about iteration and recursion, and their relationship to other language constructs.

### 4.1. Recursion from iteration in CPS

Let us first note that the type of $\mathsf{loop}_\alpha\, f : \alpha \to 0$ coincides with the type of $\alpha$-accepting continuations. And in fact, annotating the CBV iteration equation accordingly, we see that $\mathsf{loop}$ actually creates a recursive *continuation*: if $f : \alpha \to \alpha$ then

$$\mathsf{loop}\, f \;=\; \lambda^\bullet a.(\mathsf{loop}\, f)^\bullet(f\, a) : \neg\alpha \qquad\qquad (\mathit{loop}^\bullet)$$

We will adopt this characterization of $\mathsf{loop}$ in the following.

Consider now the CPS counterparts of $\mathsf{loop}$ and $\mathsf{fix}$. We use essentially the usual CBV CPS translation [17], extended to typed $\mathcal{C}$ [9], and written with continuations first for technical convenience (as in [20]). Also, when reasoning about CPS versions of terms, it becomes preferable to make $\neg\alpha$ a separate type from the function space $\alpha \to 0$ (and ensure that $\lambda^\bullet$-abstractions are only used with $^\bullet$-applications). Then we can omit the spurious 0-accepting continuation parameters in the translations of first-class continuations and get the translation scheme:

$$
\begin{aligned}
x_c &= \lambda k.\, k\, x \\
(\lambda x.\, M)_c &= \lambda k.\, k\, (\lambda k.\, \lambda x.\, M_c\, k) \\
(M\, N)_c &= \lambda k.\, M_c\, (\lambda f.\, N_c\, (\lambda a.\, f\, k\, a)) \;=\; \lambda k.\, M_c\, (\lambda f.\, N_c\, (f\, k)) \\
(\lambda^{\bullet} x.\, M)_c &= \lambda k.\, k\, (\lambda x.\, M_c\, \mathsf{i}) \\
(M^{\bullet} N)_c &= \lambda i.\, M_c\, (\lambda f.\, N_c\, (\lambda a.\, f\, a)) \;=\; \lambda i.\, M_c\, N_c \\
(\mathcal{C} M)_c &= \lambda k.\, M_c\, (\lambda f.\, f\, k)
\end{aligned}
$$

(where $\mathsf{i} : 0 \to o$ is the nowhere-defined continuation with answer type $o$).

A further simplification is made possible by the observation that there is no need to transform "trivial" (parts of) functions [19] into CPS. In particular, the computational content of a curried function $t = \lambda x.\, \lambda y.\, M$ is fully captured by the CPS variant $t_{c'} = \lambda x.\, (\lambda y.\, M)_c$, with the property that $(t\, x)_c = \lambda k.\, k\, (t_{c'}\, x)$; if $t$ must be passed unapplied to some other function, it can always be wrapped in an $\eta_v$-redex $\lambda x.\, t\, x$ before transformation.

The CPS version of the iteration equation then becomes, after some administrative simplifications (which can actually be built into the translation itself [2, 20]):

$$\mathsf{loop}_{c'}\, f_c \;=\; \lambda a.\, f_c\, (\lambda a'.\, \mathsf{loop}_{c'}\, f_c\, a')\, a \;\overset{2\times\eta}{=}\; f_c\, (\mathsf{loop}_{c'}\, f_c)$$

I.e, loop satisfies the CBV iteration equation precisely when $\mathsf{loop}_{c'}$ is a fixpoint combinator (with type $((\alpha \to o) \to \alpha \to o) \to \alpha \to o$) at the CPS level!

Now, consider the CPS version of fix (based on the continuation-creating loop, and observing that with the translation above, the CPS counterpart of the idiom $\lambda x.\, \mathcal{C}(\lambda^{\bullet} k.\, M)$ reduces to simply $\lambda k.\, \lambda x.\, M_c$):

$$\mathsf{fix}_{c'}\, F_c \;=\; \lambda a.\, \lambda r.\, \mathsf{loop}_{c'}\, [\lambda l.\, \lambda (v, c).\, F_c\, (\lambda f.\, f\, c\, v)\, [\lambda q.\, \lambda x.\, l\, (x, q)]]\, (a, r)$$

The CPS variant of the CBV recursion equation is:

$$\mathsf{fix}_{c'}\, F_c = \lambda k.\, \lambda a.\, F_c\, (\lambda f.\, f\, k\, a)\, (\mathsf{fix}_{c'}\, F_c)$$

And it is easy to verify that our $\mathsf{fix}_{c'}$ satisfies this equation:

$$
\begin{aligned}
\mathsf{fix}_{c'}\, F_c &= \lambda a.\, \lambda r.\, \mathsf{loop}_{c'}\, \overbrace{[\lambda l.\, \lambda (v, c).\, F_c\, (\lambda f.\, f\, c\, v)\, [\lambda q.\, \lambda x.\, l\, (x, q)]]}^{B_{F_c}}\, (a, r) \\
&= \lambda a.\, \lambda r.\, [\lambda l.\, \lambda (v, c).\, F_c\, (\lambda f.\, f\, c\, v)\, [\lambda q.\, \lambda x.\, l\, (x, q)]]\, (\mathsf{loop}_{c'}\, B_{F_c})\, (a, r) \\
&= \lambda a.\, \lambda r.\, F_c (\lambda f.\, f\, r\, a)\, [\lambda q.\, \lambda x.\, \mathsf{loop}_{c'}\, B_{F_c}\, (x, q)]] \\
&= \lambda a.\, \lambda r.\, F_c (\lambda f.\, f\, r\, a)\, (\mathsf{fix}_{c'}\, F_c)
\end{aligned}
$$

(The proof is somewhat shorter than in the direct-style case because we were able to simplify the CPS term first).

Thus, it is the continuation semantics of CBV *iteration* that corresponds to a domain-theoretical fixpoint, while an explicit CBV fixpoint combinator $\mathsf{fix}$ just adds some administrative argument-shuffling with no apparent domain-theoretical significance. But why did we then need control operators to recover $\mathsf{fix}$ from $\mathsf{loop}$? The reason is that in $\mathsf{fix}_{\mathcal{C}'}$, the continuations are permuted in an "illegal" way, so that the term does not have a $\mathcal{C}$-free direct-style counterpart [3]. Another way of seeing this is that the type of the "loop-to-fixpoint transformer",

$$\begin{aligned}&\mathsf{loop2fix} = \lambda \mathit{loop}.\,\mathsf{FIX}(\mathit{loop})\\ &\quad : [(\alpha \times \neg\beta \to \alpha \times \neg\beta) \to \neg(\alpha \times \neg\beta)] \to [((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta]\end{aligned}$$

is "classical" in the sense of [9], while all pure lambda-terms have types that correspond to intuitionistically valid propositions. In particular, taking $\alpha = \neg\beta$ in the above, the term $\mathsf{loop2fix}\,(\lambda f.\,\lambda^{\bullet}(t,c).\,t^{\bullet}c)\,(\lambda g.\,g)$ has type $\neg\neg\beta \to \beta$ and is in fact equal to $\mathcal{C}$. (This does not mean, of course, that recursion cannot be implemented with a simple control stack: the particular pattern of continuation passing used here is essentially that of applicative coroutining [22] between the recursive computation and a purely iterative "controller" that needs no additional stack space).

### 4.2. Recursion and iteration from reflexive types

Up to now, we have considered only a simply-typed language. In particular, this meant that well-known "non-recursive" definitions of recursion like the Y-combinator were not expressible. It is perfectly possible, however, to consider such definitions in a simply-typed framework with *explicit* domain isomorphisms. This lets us analyze exactly what properties of our semantic domain we rely on to make the definition work.

Consider the CBV Y-combinator [18]:

$$\mathsf{Y}\,F = [\lambda x.\,\lambda a.\,F\,(x\,x)\,a]\,[\lambda x.\,\lambda a.\,F\,(x\,x)\,a]$$

This $\mathsf{Y}$ is untypable because $x$ needs to have a type $\tau$ which is itself of the form $\tau \to \cdots$. While we can clearly not get such an identity in a simply-typed setting, all we really need is an *isomorphism* between the types, i.e., for any pair of types $\alpha$ and $\beta$, a type

$$\mathsf{sapp}_{\alpha,\beta} \;=\; \mu t.\,t \to (\alpha \to \beta)$$

equipped with a pair of functions $\Psi_{\alpha,\beta} : (\mathsf{sapp}_{\alpha,\beta} \to \alpha \to \beta) \leftrightarrow \mathsf{sapp}_{\alpha,\beta} : \Phi_{\alpha,\beta}$, such that for any $f : \mathsf{sapp}_{\alpha,\beta} \to \alpha \to \beta$ and $s : \mathsf{sapp}_{\alpha,\beta}$

$$\Phi_{\alpha,\beta}\,(\Psi_{\alpha,\beta}\,f) = f \qquad \text{and} \qquad \Psi_{\alpha,\beta}\,(\Phi_{\alpha,\beta}\,s) = s$$

Note that $t$ occurs negatively (i.e., to the left of an odd number of arrows) in the $\mu$-expression. This is the hallmark of a "reflexive" definition, which requires us to go from set-theoretic models to domain-theoretic ones to find a solution [21]. (In ML, we would use a parameterized **datatype** to define the recursive type $\mathsf{sapp}_{\alpha,\beta}$: $\Psi$ is the constructor; we get $\Phi$ by pattern matching.) We can now write the Y-combinator as

$$\mathsf{Y}\, F = [\lambda s.\, \lambda a.\, F(\Phi s\, s)\, a]\, (\Psi\, [\lambda s.\, \lambda a.\, F(\Phi s\, s)\, a])$$

What about loops? It is easy to see that the combinator

$$\mathsf{L}\, f = [\lambda x.\, \lambda a.\, x\, x\, (f\, a)]\, [\lambda x.\, \lambda a.\, x\, x\, (f\, a)]$$

has the right operational property (i.e., $\mathsf{L}\, f\, a = \mathsf{L}\ f\, (f\, a)$). We get a clearer picture, however, if we eliminate unnecessary currying and replace general functions by continuations where possible:

$$\mathsf{L}\, f = \lambda^{\bullet} a.\, [\lambda^{\bullet}(x, a).\, (x^{\bullet}(x, f\, a))]^{\bullet}([\lambda^{\bullet}(x, a).\, x^{\bullet}(x, f\, a)], a)$$

The type of $x$ is now

$$\mathsf{sthr}_{\alpha} = \mu t.\, \neg(t \times \alpha)$$

and we can insert the corresponding isomorphisms to get a well-typed term. Again, $t$ occurs negatively (in fact, directly under a negation) in its defining type expression. Since we already know how to recover $\mathsf{Y}$ from $\mathsf{L}$, we have thus reduced our domain-theoretic requirement from a type for self-applicable functions to one for self-throwable continuations (i.e., functions with codomain 0).

Finally, consider the CPS transformation of $\mathsf{L}$ (still omitting the isomorphisms for readability):

$$\mathsf{L}_{\mathsf{c}}\, f_c = \lambda a.\, [\lambda(x, a).\, f_c\, (\lambda a'.\, x\, (x, a'))\, a]\, ([\lambda(x, a).\, f_c\, (\lambda a'.\, x\, (x, a'))\, a], a)$$

or, in curried form,

$$\mathsf{L}_{\mathsf{c}}\, f_c \;=\; \lambda a.\, [\lambda x.\, \lambda a.\, f_c\, (\lambda a'.\, x\, x\, a')\, a]\, [\lambda x.\, \lambda a.\, f_c\, (\lambda a'.\, x\, x\, a')\, a]\, a \stackrel{\beta\eta}{=} \mathsf{Y}\, f_c$$

Once again, looping in direct style corresponds to a fixpoint in CPS!

### 4.3. Recursion from other constructs

Let us finally mention for completeness that even if a language does not explicitly provide reflexive types, many other language features implicitly provide a comparable facility. In particular, their denotational descriptions inherently involve reflexive domains. Consider for example the way recursion is expressed in the Scheme report [1] (essentially):

$$\mathsf{fix}\, F \;=\; \textbf{let}\ r = \mathsf{ref}\, (\lambda x.\, \textbf{fail})\ \textbf{in}\ (r := F\, (\lambda x.\, !r\, x);\, !r)$$

(The $\eta$-redex around $!r$ serves to delay evaluation until the correct value of $r$ has been plugged in.) In such a language, where storable procedures can themselves access the store, the denotation of the type $\alpha \to \beta$ becomes:

$$[\![\alpha \to \beta]\!] \cong [\![\alpha]\!] \to \sigma \to [\![\beta]\!] \times \sigma$$

where $\sigma$ is the domain representing the store. Let us simplify slightly and stipulate that *first-class* (in particular, storable) functions are not allowed to modify the store, only read it; and let us further consider a store containing only a single cell of type $\sigma$. If we are then allowed to store functions of type $\alpha \to \beta$, we must have

$$\sigma = [\![\alpha \to \beta]\!] \cong \sigma \to ([\![\alpha]\!] \to [\![\beta]\!])$$

which is precisely the recursive domain equation for the Y-combinator above. And in fact, writing out the ref-based definition above with explicit store-passing shows a structure essentially identical to the Y-combinator.

A similar observation lets us construct fixpoints from exceptions carrying functional values. In a language with no computational effects other than named exceptions, there is a simple correspondence between exception names (expressions of type exn in SML [15]) and functions of type $1 \to 0$, because such a function must essentially have the form $\lambda().\,\mathbf{raise}\,X$ for some exception expression $X$. And then we need only to introduce an exception Psi carrying values of type $(1 \to 0) \to \alpha \to \beta$ (in effect, a constructor of type $(\mathsf{exn} \to \alpha \to \beta) \to \mathsf{exn}$, with its inverse given through exception-pattern matching) to get our usual reflexive domain.

## 5. Related work

A fair amount is known about *transforming* recursive programs into iterative form, the so-called "flowchartability" problem. Most such work has been done in an explicitly procedural setting (e.g., [8]), or for first-order recursion equations [23]. However, some extensions to higher-order call-by-name functional programs are reported in [13]. Interestingly, the methods in the latter work rely heavily on a notion of contexts, but the author apparently never draws any connections to continuation-passing style, let alone first-class continuations.

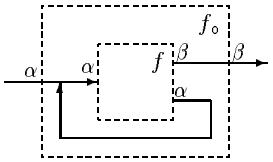
The present paper solves a somewhat different problem: instead of considering general program transformations, we restrict ourselves to *defining* a fixpoint combinator – a purely local construction made possible only by the additional expressive power of a control operator.

Despite the order of presentation above, the definition of fix from iteration and control operators did not evolve from exceptions over the notion

of "single-stepping" a recursive definition. In fact, the exception-based solution was found *last*, during a (failed) attempt to show that exceptions and iteration could *not* express recursion. The operational explanation of fix in section 2.4 was partly suggested by a CW'92 referee. But the actual fix-term (though in a rather different notation) was obtained completely unexpectedly from the category-theoretical analysis of the symmetry between CBV iteration and CBN recursion mentioned in the introduction. Let us briefly sketch it here:

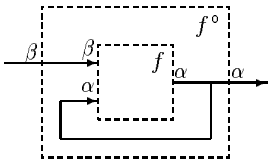
Informally, by taking a language like (typed) Scheme or SML/NJ, but expressed in a notation with syntactic symmetry between values and continuations, the abstract principle of duality can be used to expose a number of otherwise obscured semantic symmetries involving data types, control structures, and evaluation strategies. Specifically, iteration involves "tying a loop" around a function with two (disjunctive) outputs:

$$\frac{f : \alpha \to \beta + \alpha}{f_\circ : \alpha \to \beta}$$




Symmetrically, we get recursion from a function with two (conjunctive) inputs:

$$\frac{f : \beta \times \alpha \to \alpha}{f^\circ : \beta \to \alpha}$$




Both of these use first-order constructs only (i.e., $-^\circ$ is a special form, not a functional). However, if we have a representation of higher-order functions (specifically, for every pair of types $\sigma$ and $\tau$, a type $\sigma \Rightarrow \tau$ representing the function space, together with combinator $\mathsf{apply} : (\sigma \Rightarrow \tau) \times \sigma \to \tau$), we can take $\beta = \alpha \Rightarrow \alpha$, and $f = \mathsf{apply} : (\alpha \Rightarrow \alpha) \times \alpha \to \alpha$ in the above to get $\mathsf{fix} = \mathsf{apply}^\circ : (\alpha \Rightarrow \alpha) \to \alpha$. A dual construction allows us to define a first-class iteration construct similar to loop.

Taking a closer look at the above data-flow diagrams, however, we see that they are essentially equivalent: only the direction of arrows is reversed. Notably, viewing *demands* as *data* in the recursion diagram, we get precisely the iteration diagram. Moreover, there is a natural definition of $f_\circ$ in terms of fix, which gives rise to an abstract "mirror-image" definition of $f^\circ$ from loop. Translating this correspondence back to a traditional syntax (and

completely obscuring the symmetry in the process), we get a representation of recursion from iteration.

## 6. Conclusion and issues

In a simply-typed CBV language with control operators, we can define recursion in terms of iteration. A crude solution using only exceptions is possible, but is fundamentally flawed in several ways. Two major problems of such a definition – its self-interference and first-order nature – are directly related to the dynamic scoping of exceptions, and can be solved by simply switching to a continuation-based escape mechanism. Nevertheless, the definition is still far from perfect. However, using the full power of first-class continuations and a paradigm of *context-switching* as opposed to *aborting* computations, we can get a fixpoint provably equivalent to the "real" one.

In fact, the additional expressive power of first-class continuations lets us decompose the usual CBV fixpoint combinator into an iterative *core*, whose semantics corresponds directly to a fixpoint combinator at the CPS level, and an administrative *wrapping* presenting a more convenient and general interface. In other words, in the presence of a call/cc-like operator, reasoning about CBV recursion can be reduced to reasoning about simple loops – effectively making general recursion a special case of tail recursion!

In perspective, this equivalence of iteration and recursion gives another reason for why a construct for first-class continuations should be considered a natural part of a CBV language, especially one already providing some form of exception mechanism: a sharper tool (continuations) allows us to craft a much better fixpoint than a blunt one (exceptions).

But perhaps the most significant issue raised by the results presented here is that what appears to be a canonical domain-theoretic construct can to a large extent be analyzed in isolation from the actual semantic model. In fact, the line of research sketched in section 5 suggests that closely related concepts like strictness, eager/lazy datatypes, etc., also admit a more abstract, "continuation-theoretic" characterization – applicable even in a language where all evaluations are finite. A further exploration of these ideas and their relation to other current work on continuations would seem a natural direction for further research.

## Acknowledgments

I want to thank John Reynolds for support, and Olivier Danvy, Matthias Felleisen, Dan Friedman, Carolyn Talcott and the referees for their encouragement and helpful comments on various drafts of this paper.

## References


1. Clinger, W. and Rees, J. Revised[4] report on the algorithmic language Scheme. *Lisp Pointers*, 4, 3 (July 1991) 1–55.

2. Danvy, O. and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2, 4 (December 1992) 361–391.

3. Danvy, O. and Lawall, J. L. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California (June 1992) 299–310.

4. Duba, B. F., Harper, R., and MacQueen, D. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida (January 1991) 163–173.

5. Felleisen, M. and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103, 2 (1992) 235–271.

6. Felleisen, M., Friedman, D. P., Kohlbecker, E., and Duba, B. Reasoning with continuations. In *Proceedings of Symposium on Logic in Computer Science*, IEEE, Cambridge, Massachusetts (June 1986) 131–141.

7. Filinski, A. Declarative continuations: An investigation of duality in programming language semantics. In Pitt, D. H. *et al.*, editors, *Category Theory and Computer Science*, Manchester, UK (September 1989) 224–249.

8. Greibach, S. *Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science* 36 (1975).

9. Griffin, T. G. A formulae-as-types notion of control. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California (January 1990) 47–58.

10. Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques.* The MIT Press (1992).

11. Harper, R. and Lillibridge, M. Polymorphic type assignment and CPS conversion. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*, San Francisco, California (June 1992) 13–22. Revised version to appear in *Lisp and Symbolic Computation* (this issue).

12. Hindley, J. R. and Seldin, J. P. *Introduction to Combinators and λ-Calculus*. Volume 1 of *London Mathematical Society Student Texts*, Cambridge University Press (1986).

13. Kfoury, A. J. *The Translation of Functional Programs into Tail-Recursive Form (Part I)*. BUCS Tech Report 87-003, Computer Science Department, Boston University (January 1987).

14. Landin, P. J. A correspondence between ALGOL60 and Church's lambda notation. *Communications of the ACM*, 8 (1965) 89–101 and 158–165.

15. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. The MIT Press (1990).

16. Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE, Pacific Grove, California (June 1989) 14–23.

17. Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1 (1975) 125–159.

18. Reynolds, J. C. Gedanken – a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13, 5 (May 1970) 308–319.

19. Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, Boston (August 1972) 717–740.

20. Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California (June 1992) 288–298. Revised and extended version to appear in *Lisp and Symbolic Computation* (this issue).

21. Scott, D. S. Continuous lattices. In *Proceedings of 1971 Dalhousie Conference*, Springer-Verlag (1972) 97–136.

22. Talcott, C. A theory for program and data type specification. *Theoretical Computer Science*, 104, 1 (1992) 129–159.

23. Walker, S. A. and Strong, H. R. Characterizations of flowchartable recursions. *Journal of Computer and System Sciences*, 7 (1973) 404–447.