Yield: Mainstream Delimited Continuations

Roshan P. James
Indiana University
Bloomington, Indiana, U.S.A.
rpjames@indiana.edu

Amr Sabry
Indiana University
Bloomington, Indiana, U.S.A.
sabry@indiana.edu

Abstract

Many mainstream languages have operators named *yield* that share common semantic roots but differ significantly in their details. We present the first known formal study of these mainstream *yield* operators, unify their many semantic differences, adapt them to a functional setting, and distill the operational semantics and type theory for a generalized *yield* operator. The resultant *yield*, with its delimiter *run*, turns out to be a delimited control operator of comparable expressive power to *shift-reset*, with translations from one to the other. The mainstream variants of *yield* turn out to be one-shot or linearly used restrictions of delimited continuations. These connections may serve as a means of transporting ideas from the rich theory on delimited continuations to mainstream languages which have largely shied away from them. Dually, the restrictions of the various existing *yield* operations may be treated as *shift-reset* variants that have found mainstream acceptance and thus worthy of study.

1 Introduction

Many mainstream programming languages such as Ruby, Python, C#, and JavaScript have variations of an operator called *yield*. In all of these languages *yield* provides a means of suspending a computation temporarily with the ability to resume it later — a feature that immediately suggests the presence of continuations. A variant of the operator appeared in CLU and Icon, but its recent popularity may be attributed to its use in Ruby. In the Ruby language, *yield* forms a central means of composition with almost every iteration/enumeration mechanism being defined in terms of it.

For an operator that enjoys such widespread acceptance, *yield* has been largely ignored by the literature on continuations and control operators. The semantics of *yield* varies from language to language, often to the point where algorithms in one language cannot be directly translated into another. We provide the first formal investigation of *yield*. We do this by investigating the semantics of several popular *yield* operators, unifying their behavior into one operator. Further we disentangle *yield* from its imperative roots and adapt it to a functional setting thereby providing a semantics using a typed lambda calculus extended with pairs and sums, $\lambda^{\rightarrow\times+}$.

The resulting *yield* operator turns out to be a new sort of delimited continuation operator equivalent to *shift-reset* and this is our main result. Establishing such a connection between theoretically well understood delimited continuations and widely accepted programming practice carries benefits for both theory and practice:

- The literature of languages that support *yield* operators do not describe *yield* as a delimited continuation operator. In fact, such languages have explicitly distanced themselves from (delimited) continuations which are perceived as complex, inaccessible, and inefficient. By making the connection between *yield* and standard delimited continuations precise we expect the rich literature on delimited continuations to be more accessible and hence to inform more precise cost-benefit decisions in mainstream languages.

- From the perspective of language theory, the generalized *yield* provides a new perspective on delimited continuations, providing the full power of delimited continuations without giving direct

access to the continuation. Moreover the variants of *yield* from mainstream languages provide several new operators that encapsulate stylized uses of such delimited continuations. Stylized uses of continuations have been proposed early on [FH85] and studied extensively as a linear discipline on continuation use [BORT02]. In the case of *yield*, stylized uses of continuations can be defined by simple syntactic restrictions, suggesting that *yield* provides a superior interface for delimited continuations.

2 A Survey of *yield* operators

Languages such as C#, Ruby, Python, Cω, JavaScript, Sather, F#, and CLU all support *yield* operators. Describing the full syntactic and library-level support provided by these languages is beyond the scope of this paper and hence we look at only a few of these languages in detail, describing the relevant properties of *yield* at each step. Unfortunately, many languages only give informal descriptions of their *yield* variants with just a few examples, and hence our presentation informally proceeds from the informal to the formal.

This section helps develop an insight into *yield*'s semantics and establishes relevant terminology. Since the terminology of each language is different, as we explore the design space of *yield* operators we establish our own common vocabulary for concepts common to different languages.

Ultimately, we use this section to provide an informal derivation of our generalized *yield*. We reconcile existing semantic differences by merging functionality whenever possible, often biasing design choices in favor of a functional programming style.

2.1 `yield return` in C#

C# 2.0 [ECM06] introduced a *yield* operator with the addition of the keyword `yield return`. The function `fibonacci` below uses the *yield* operator to generate the infinite Fibonacci series. Its return type, `IEnumerable<int>`, indicates that it yields a (potentially infinite) sequence of `int` values.

```
IEnumerable<int> fibonacci() {
  int a = 0, b = 1;
  while(true) {
    yield return a;
    b = a + b;
    a = b - a;
  }
}
```

```
void useFibonacci() {
  foreach(int n in fibonacci()) {
    if(n > 100) break;
    Console.WriteLine(n);
  }
}
```

We use the term ***iterator*** to refer to computations that *yield* values. The argument to the *yield* operator becomes an *output* of the *iterator*. We refer to these outputs as *yielded values*. The usage of the word *output* here needs some clarification: yielding values is not an output operation in the conventional sense of a process interacting with the operating system. Rather the yielded values are "interactive" outputs from the *iterator* to its calling context. In this sense it is handy to think of *yield* as a delimited IO operator.

The caller, `useFibonacci`, invokes the *iterator* using a `foreach` loop. The loop body executes each time a value is output from the *iterator*. Variable `n` is bound to the yielded value at each loop iteration. Each time `fibonacci` yields, its execution is suspended. When the loop body finishes, execution of the *iterator* resumes from the point of suspension. The state of the *iterator* is preserved across a *yield* suspension. Thus *yield* lets us model streams and lazy lists in a natural way.

When an *iterator* yields, it need not be resumed by its caller even though it may have pending computations to perform. For instance, `useFibonacci` breaks out of the loop eventually leaving `fibonacci`

suspended for good. In the absence of break, one may be led to think of *yield* as merely the invocation of an implicit function which is the loop body. However under that interpretation break would be a non-local escape. In addition to break, the loop body can also execute return which will cause control to go to the caller of useFibonacci.

To summarize, *yield* allows a function to output values to its calling context. State is preserved while yielding and resuming a suspended *iterator* causes it to continue from the point of yielding. An infinite sequence of values may be yielded and the caller only resumes the *iterator* at its discretion.

2.2 Ruby

With these insights from C#, let's look at *yield* in Ruby [TH00]. Ruby is a *"duck typed"* language primarily developed by Yukihiro Matsumoto. Much of the present day popularity of *yield* is attributed to its implementation in Ruby, which was in turn motivated by *yield* in CLU and control blocks in Smalltalk.

The important difference between Ruby and C# is that *yield* in Ruby returns a value, whereas in C# there is no such return value. Further in Ruby the *iterator* can return a final result at termination, which is distinguished from yielded values. We illustrate *yield* using the inject function below which is a simplification of its standard Ruby library implementation. The inject function folds over a Ruby object.

```
def inject(state)
  self.each{|v| state = yield(v, state) }
  state
end
```

```
def useInject()
  total = [1..1000].inject(0) {|v, sum|
    if prime?(v)
      sum+v
    else
      sum
    end
  }
end
```

The components of the object self are enumerated by each. The *iterator* inject yields each component of self and the current state. The variable state is updated with the value returned by *yield*. After the components of the current object have been enumerated, inject returns the accumulated state.

The caller, useInject, invokes the inject method of an array object and builds the sum of primes. The *block* construct, syntactically {| variable | body}, is Ruby's equivalent of the C# foreach. This usage of *yield* simulates higher-order programming in an imperative context. However, the code block following the inject invocation is not just a lambda abstraction; like the body of the foreach loop in C#, the block can execute a break or a return.

Summarizing, *iterators* in Ruby take *input* from their calling context to resume their computation, *i.e.*, unlike C#, each invocation of *yield* in Ruby returns a value. Further, yielding computations can also return a value on completion. The execution of the *iterator* is suspended each time it yields. The *iterator* maintains state and when resumed, behaves exactly as if the operator *yield* had returned. What the *iterator* does next on resumption can depend on its input value. The return value of an *iterator* is distinguished from a yielded value, in particular the input, output and return values of an *iterator* can have different types. In Ruby nomenclature *iterators* are sometimes referred to as *asymmetric coroutines*, ones that are restricted to transfer control only to their calling context.

2.3 JavaScript

JavaScript 1.7 [Moz06, Fla06] introduced a *yield* operator that is yet to be ratified in the ECMA specification [ECM99] of the language. In the previous examples, we used *yield* along with a loop construct that interacted with one *iterator* at a time. This usage is sometimes referred to as *internal iterators*, in the sense that the caller's interaction with the *iterator* is encapsulated by the loop construct. The suspended

iterator is always implicit. To use more than one *iterator* at a time, we have to make this suspension an explicit object.

Let us look at a JavaScript example that decouples the *iterator* usage from the loop construct, thereby enabling us to interleave interaction with multiple *iterators*. These *iterators* are first class values — a usage that is referred to as *external iterators*:

```
function fibonacci() {
  val a = 0, b = 1
  while(true) {
    yield a
    b = b + a
    a = b - a
  }
}

function each(array) {
  for(val i = 0; i < array.length(); i++) {
    yield array[i]
  }
}
```

```
function compare(array) {
  try {
    iter1 = fibonacci()
    iter2 = each(array)
    while(true) {
      if(iter1.next() == iter2.next())
        return true
    }
  }
  catch(err instanceOf StopIteration) {}
  return false
}
```

In the example above, compare invokes two *iterators*, executing them in lock step. Each Fibonacci number is computed only when required until either a match is found or the array is exhausted. In JavaScript, when an *iterator* terminates its .next() method invocation raises a StopIteration exception.

Since one cannot enumerate all the ways in which someone might want to interleave *iterators*, having external iterators is more expressive than having only internal iterators. We quote the classical *Design Patterns* book [GHJV95]:

> External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators [...] But on the other hand internal iterators are easier to use because they define the iteration logic for you.

C#, F# and Python also have the ability to treat *iterators* as first class values, though they differ in how termination is handled. Given first class *iterators* of this form, a programming language may always add a handy loop construct to recover the common foreach paradigm. (See Section 3.2 for a realization of this idea.)

Traditionally Ruby has only supported iteration over one *iterator* at a time. An *ad hoc* solution to provide external iterators is the Generator library which relies on Ruby's callcc operator. Using the Generator library however limits the *iterator*'s input and return capabilities. Ruby version 1.9, that is currently under development, is expected to support *internal* and *external iterators* with an implementation based on OS Fibers.

In the imperative languages (C#, Ruby and JavaScript) discussed here, a *iterator* is inherently a stateful object. Invoking .next() on a *iterator* to resume it, also changes its state. Since we are interested in modeling *yield* in a functional setting, we will modify this stateful behavior to an immutable one in Section 2.5.

2.4 Python, F#, Cω, CLU and other languages

Python [vRD03] versions prior to 2.5 [YvR01] have support for *yield* as an output only operator (like in C#). Python version 2.5, based on PEP 342 [vRE05], supports *yield* as an *expression instead of a statement i.e.*, it has support for *yield* returning values. Both Python and JavaScript iterators can *yield* out values and take input values. They do not return values at the end of evaluation, but instead indicate termination by raising a distinguished exception. Given that these languages lack static types,

in principle, the return value could be encoded as the last yielded value or by some other convention. However, in practice this is inconvenient since one would have to always keep track of the last yielded value.

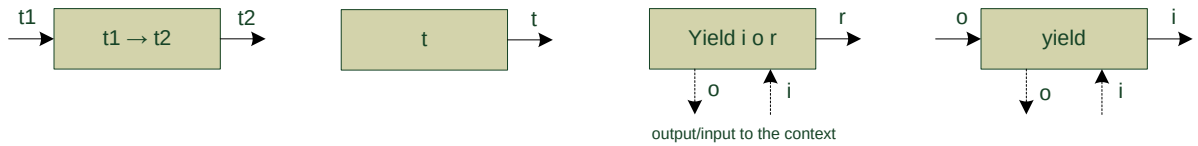
CLU [Lis93] has support for an output only variadic *yield* and a `for` loop construct for consuming such iterators. It also attempts to capture the type of the yielded values in the type signature of iterators. Sather [MOSS96, Omo91] has an output only *yield* construct with weaker restrictions on the usage within loop constructs. Due to the absence of a formal description of the semantics, we are unsure of the precise nature of these restrictions.

Cω [MSB03, BMS05] and F# [Sym03] have `yield return` statements, much like C#'s, that are primarily used to model potentially infinite lazy lists/streams. The Cω type system is enhanced with types such as `int*` and `int+` to indicate if these are streams of zero or more integers. There are other languages, such as Groovy [KGK+07], Icon [GG83], that implement *yield* or *iterator*-like constructs that we have not detailed here.

2.5 Towards a generalized *yield*

In summary, an *iterator* may perform some arbitrarily complex computation. Its consumer, the calling context of the *iterator*, is abstracted from the details of this computation. Dually, the *iterator* does not know anything about the computation performed by its calling context, or even if it will be resumed again. The *yield* operator separates control flow concerns and allows us to encapsulate *iterators* and their callers into reusable software components.

Based on our brief survey, we unify *yield* and *iterator* properties from the various languages, introducing types as appropriate. An *iterator* can be thought of as a *sub-process*, a special computation, that is restricted to communicate with its calling context using *yield*. In the course of its computation it can *yield* out many values of type o and receive inputs of type i. When an *iterator* terminates it returns a final value of type r. Hence we can give *iterators* the abstract type *Yield i o r*. The construct *yield* is a *delimited IO* operation available to such *iterators* that enables them to communicate with their calling context. Each invocation of *yield* suspends the state of its *iterator* and suspended *iterators* are first-class values. Resuming a suspended *iterator* is at the discretion of the calling context and the value used to resume the *iterator* becomes available as the return value of the suspending *yield*. A single *yield* statement may be seen as the simplest possible interesting *iterator* with the abstract type $o \to$ *Yield i o i*, allowing more interesting *iterators* to be built by composition. A pure computation in contrast does not do any such IO, and only returns a result. Diagrammatically, if we visualize pure functions $t_1 \to t_2$ and computations t as shown to the left, then we can visualize an iterator *Yield i o r* and the *yield* operator of type $o \to$ *Yield i o i* as shown to the right:



A first attempt to formalize *yield* might be to simply add *yield* to the call-by-value λ-calculus. However this is quickly seen to be inadequate. The main complication is to decide how to separate the iterator from its caller. To this end, we introduce a delimiting operator called *run* to explicitly separate the *iterator* from its calling context. The operator *run* marks the boundary of an *iterator* and delimits the action of *yield*. The argument to *run* is an opaque computation that can *yield* and it returns a concrete data structure that the calling context can interact with. This immediately suggests a monadic encapsulation

for the effectful *iterator* computations with *yield* as the only effect operator of the monad. Since *run* marks the boundary of this effect, it can be used as the *run* operation that escapes the monad.

Indeed the imperative languages that we have considered here do have such a *run* delimiter, which is usually implicit, and is apparent in the fact that methods that *yield* are treated differently from methods that don't, requiring the former to be called in a special manner. In the C# and Ruby examples, the equivalent of *run* was hidden in the implementation of the loop construct and in JavaScript it was implicit in the iter1 = fibonacci () assignment. In C#, GetEnumerator() is the equivalent of *run* and it takes an opaque computation of type IEnumerable<T> to an interactive object of type IEnumerator<T>. The interface IEnumerator exposes methods for retrieving the current yielded value (Current) and for advancing the *iterator* computation (MoveNext). We can use GetEnumerator() explicitly as follows:

```
IEnumerator<int> iter = fibonacci().GetEnumerator();
while(iter.MoveNext()) {
  Console.WriteLine(iter.Current);
}
```

In C#, instead of using the implicit run hidden in loops, we can use the `GetEnumerator()` and the `IEnumerator` interface as an explicit run construct.

3 A Monadic *yield*

We are now ready to introduce the syntax of a monadic meta-language with *yield*. This language is Moggi's monadic metalanguage (MML) extended with the opaque monadic type *Yield i o r* and a type *Iterator i o r* for interacting with *iterators*:

$$\begin{array}{rcl} \textit{types}, t, i, o, r & = & b \mid t \to t \mid \textit{Iterator } i\ o\ r \mid \textit{Yield } i\ o\ r \\ \textit{expressions}, e & = & x \mid \lambda x.e \mid e_1\ e_2 \mid \textit{Result } e \mid \textit{Susp } e\ e \mid \textit{case } e\ e_1\ e_2 \\ & \mid & \textit{return } e \mid \textit{do } x \leftarrow e; e \mid \textit{yield } e \mid \textit{run } e \\ \textit{evaluation contexts}, E & = & \Box \mid E[\textit{do } x \leftarrow \Box; e] \end{array}$$

The set of expressions includes a pure subset consisting of variables, functions, applications, *Result*, *Susp*, and *case*, and a computational subset consisting of *return*, *do*, and *yield*. The operator *run* acts as the interface between pure and computational expressions. There are several possible type systems of varying expressive power (see Section 3.1). We present below a simple type system in which the types i and o are fixed for each iterator:

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textit{var} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2}\ \textit{lambda} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}\ \textit{application}$$

$$\frac{\Gamma \vdash e : r}{\Gamma \vdash \textit{Result } e : \textit{Iterator } i\ o\ r}\ \textit{result} \qquad \frac{\Gamma \vdash e_1 : o \quad \Gamma \vdash e_2 : i \to \textit{Iterator } i\ o\ r}{\Gamma \vdash \textit{Susp } e_1\ e_2 : \textit{Iterator } i\ o\ r}\ \textit{susp}$$

$$\frac{\Gamma \vdash e : \textit{Iterator } i\ o\ r \quad \Gamma \vdash e_1 : o \to (i \to \textit{Iterator } i\ o\ r) \to t \quad \Gamma \vdash e_2 : r \to t}{\Gamma \vdash \textit{case } e\ e_1\ e_2 : t}\ \textit{case}$$

$$\frac{\Gamma \vdash e : r}{\Gamma \vdash \textit{return } e : \textit{Yield } i\ o\ r}\ \textit{return} \qquad \frac{\Gamma \vdash e_1 : \textit{Yield } i\ o\ r' \quad \Gamma, x : r' \vdash e_2 : \textit{Yield } i\ o\ r}{\Gamma \vdash \textit{do } x \leftarrow e_1; e_2 : \textit{Yield } i\ o\ r}\ \textit{do}$$

$$\frac{\Gamma \vdash e : o}{\Gamma \vdash \textit{yield } e : \textit{Yield } i\ o\ i}\ \textit{yield} \qquad \frac{\Gamma \vdash e : \textit{Yield } i\ o\ r}{\Gamma \vdash \textit{run } e : \textit{Iterator } i\ o\ r}\ \textit{run}$$

As can be seen from the type system, the interactive type *Iterator i o r* is a particular sum type that, for ease of presentation, we have hardwired into the type system. The *case* construct is the eliminator for this type and constructors *Susp* and *Result* are the two introductions for this type.

Evaluation is specified as the pure relation $\rightarrow$ which can be applied non-deterministically in any subexpression and the monadic reduction $\mapsto$ which forces reductions to happen in a certain order.

$$\begin{array}{rcl@{\qquad\qquad}rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] & \langle do\ x \leftarrow e_1; e_2, E\rangle & \mapsto & \langle e_1, E[do\ x \leftarrow \Box; e_2]\rangle \\ case\ (Susp\ e_1\ e_2)\ f\ g & \rightarrow & f\ e_1\ e_2 & \langle do\ x \leftarrow return\ e_1; e_2, E\rangle & \mapsto & \langle e_2[e_1/x], E\rangle \\ case\ (Result\ e)\ f\ g & \rightarrow & g\ e & & & \end{array}$$

The above reductions are completely standard [MS04]. In particular, the relation $\mapsto$ is closed over the transitive closure of $\rightarrow$, i.e., an arbitrary number of $\rightarrow$ steps may be applied to the components of the configuration $\langle e, E\rangle$ for any $\mapsto$ step. The evaluation rule for *run* given below ties $\rightarrow$ and $\mapsto$ together and gives the semantics for the *yield* operation as delimited by *run*:

$$\begin{array}{rcl@{\qquad}rcl} run\ e & \rightarrow & Result\ e' & if\ \langle e, \Box\rangle & \mapsto^* & \langle return\ e', \Box\rangle \\ run\ e & \rightarrow & Susp\ e'\ (\lambda x.run\ E[return\ x]) & if\ \langle e, \Box\rangle & \mapsto^* & \langle yield\ e', E\rangle \end{array}$$

A *run* expression evaluates the subexpression using any number of pure or monadic steps. If that evaluation terminates with a normal *return* then the returned value is also the final answer of the *run* expression. However if that evaluation encounters a *yield*, a new suspension is built. The suspension includes e', the output of the *yield* operation that is communicated to the caller, and an *iterator* which can be called to resume the suspended computation. An important distinction between *yield* of the imperative languages presented and our functional semantics is that resuming a suspended *iterator* does not destroy it, unlike a call to `MoveNext()`. In other words, the term $(\lambda x.run\ E[return\ x])$ is indeed a pure function that may be called any number of times. The same behavior can be achieved by cloning the `IEnumerator` and this is a topic occasionally discussed on various Internet message boards. We discuss the issue of the linear use of continuations and their resultant expressive power in Sections 3.2 and 5.

3.1 Haskell Implementations

To make the abstract semantics more relevant to implementers, we present two monadic implementations of *yield* in Haskell. These have the advantage of being immediately usable and the structure of the monads suggest implementation approaches for language implementers. Further, embedding the *yield* type system into Haskell serves as a proof of its soundness by appeal to the soundness of Haskell's type system.

Frame Grabbing Implementation: One way to implement *yield* would be to use a *yield* occurrence as a trigger which starts the accumulation of pending stack frames. This accumulation stops when an explicit *run* statement is encountered. Below is the monadic realization of this which achieves the desired semantics. Note that the interface of the *iterator* and the opaque computation type *Yield* are the same in this monad.

```
data Iterator i o r = Result r
     | Susp o (i -> Iterator i o r)
type Yield = Iterator

yield :: o -> Yield i o i
yield v = Susp v return
```

```
instance Monad (Yield i o) where
  return = Result
  (Result v) >>= f = f v
  (Susp v k) >>= f = Susp v (\x -> (k x)>>=f)

run :: Yield i o r -> Iterator i o r
run = id
```

In many of the languages discussed in the introduction, *yield* suspends exactly the current method and the continuations captured by it may be described as *one-frame* or *lexically delimited* continuations. To implement such a version of *yield*, the frame-grabbing implementation only needs a calling convention or a compiler macro that can be used to suspend multiple stack frame by re-yielding from one frame to

the other, i.e., by translating call sites, such as "`func()`", that don't have an explicit *run*, to ones that yield explicitly, "**`foreach`**`(o` **`in`** `func())` **`yield`** `o`". While this implementation is inefficient due to the explicit frame-by-frame unwinding of the stack, it is interesting since it can be readily implemented in most languages. This is similar to *nested iterator methods* [JMPS05].

CPS Implementation: As suggested before, *yield* has close connection to continuations, and techniques studied for implementing continuations [Dan00, KBD98, HDI94, CHO99, BWD96] carry over as techniques for implementing *yield*.

We can implement *yield* by instantiating the conventional continuation monad. The usual `Cont` monad has type $(a \to r) \to r$ and here we adapt it by choosing the answer type to be *Iterator i o b*. The result type *b* of the *iterator* can be quantified since composition in the monad is independent of the final result. The input and output types (*i* and *o*) of the *iterator* are fixed for the duration of computation. The resulting CPS type *Yield i o r* allows for a completely standard definition of **`return`** and >>=, for which *yield* simply builds the suspension (*Susp*) and *run* passes the initial continuation (*Result*).

```
newtype Yield i o r = Yield {unY :: (forall b. (r -> (Iterator i o b)) -> (Iterator i o b))}

instance Monad (Yield i o) where
  return e          =   Yield (\k -> k e)
  (Yield e) >>= f   =   Yield (\k -> e (\v -> (unY (f v)) k))
yield :: o -> Yield i o i                          run :: Yield i o r -> Iterator i o r
yield v = Yield (Susp v)                           run (Yield e) = e Result
```

In the usual monadic representation for delimited continuations such as [DJS07], delimiters `reset` or `pushPrompt` do not escape the monad. Although the iterator escapes the monad, it encapsulates the continuation in a pure value which forbids the control effect from leaking.

Other type systems: Another type system for *yield* results from relaxing the restriction of having *i* and *o* types fixed for the lifetime of the *iterator* computation, thereby letting each *yield* invocation chose a specific type. We do this by turning *i* and *o* into parametric types thereby allowing $yield :: o\ a \to Yield\ i\ o\ (i\ a)$. This type system may be realized by both the frame-grabbing implementation or the CPS implementation discussed previously by merely replacing type definitions. This requires the GHC `-XExplicitForAll` extension. For instance, the *iterator* type and the CPS type would be:

```
data Iterator i o r  = Result r | forall a . Susp (o a) ((i a) -> Iterator i o r)
data Yield i o r = Yield { unY :: (forall b . (r -> (Iterator i o b)) -> (Iterator i o b))}
yield :: o a -> Yield i o (i a)
run   :: Yield i o r -> Iterator i o r
```

While the the parametric *yield* may be overkill for most applications, they are interesting in the context of encoding *shift-reset* as we will see in Section 4.

3.2 Examples

In this section we will start with simple examples of using *yield* that are readily expressible in existing languages, moving on to some that have no direct equivalent in any language that supports *yield* today.

As a first example, we express a simple depth-first tree traversal function as an *iterator*. As expected, the function `depthWalk` only captures the traversal logic; it does not prescribe what to do with the yielded values.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

depthWalk :: Tree a -> Yield b a (Tree b)
```

```
depthWalk (Node l r) =  do l' <- depthWalk l
                           r' <- depthWalk r
                           return (Node l' r')
depthWalk (Leaf a)   =  do b <- yield a
                           return (Leaf b)
```

The function depthWalk recursively traverses a binary tree and yields each leaf value. It uses the input values to reconstruct a tree that has the same structure as the original one, with new leaf values. *Iterators* like depthWalk abstract an operation that we want to perform on the elements of a data structure *i.e.*, they characterize the visitor pattern [GHJV95]. For simple recursive data types without cycles, they can be automatically derived based on the recursion template [JL03].

We can now write algorithms that rely on depth-first tree traversal. Here renum renumbers the leaves of a Tree **Int** by adding 1 to each leaf.

```
renum (Susp n k) = renum (k (n+1))
renum (Result t) = t
```

We can test this as follows:

```
*Main> let tr = Node (Leaf 10)(Leaf 20)
*Main> renum (run (depthWalk tr))
Node (Leaf 11) (Leaf 21)
```

Here the tree "Node (Leaf 10)(Leaf 20)" is used to create the iterator "depthWalk tr." The context renum is insulated by run and uses the values produced by the iterator. The iteration performed by renum is a common pattern and we can abstract it in a Ruby/C#-like loop construct:

```
loop :: (o -> i) -> Yield i o r -> r
loop f m = each (run m)
    where
      each (Susp v k) = each (k (f v))
      each (Result r) = r

renum = loop succ
```

The loop body may also be parametrized by an arbitrary monad allowing it to throw exceptions, break out of iteration, accumulate state, etc. We omit the straightforward definitions of such combinators.

We get a solution to the classic *same fringe* problem, by replacing the renum function above, with one that compares leaf values of two trees.

```
same (Result _) (Result _)            = True
same (Susp a ka) (Susp b kb) | a == b = same (ka a) (kb b)
same _ _                              = False

test>  same (run (depthWalk tree1)) (run (depthWalk tree2))
```

As a slight twist on *same fringe*, consider the *swap fringe* problem here where the objective is to not compare but to swap the leaves of two trees, of possibly dissimilar structure as long as they have the same number of leaf values:

```
swap (Susp a ka) (Susp b kb) = swap (ka b) (kb a)
swap (Result t1') (Result t2') = (t1', t2')
swap _ _ = error "Unequal number of leaves"
```

This last example uses the *iterators* input, output and return capabilities and the ability to operate over more than one *iterator* at a time. This is already not naturally expressible in any of the existing *yield* implementations without resorting to assignments or some more complex encoding. Further, it does not use any continuation twice. This is a simple example that shows that there is some expressiveness to be

gained by just adding these abilities to a language's *yield* implementation without adding non-linearly-used first-class delimited continuations with indefinite extent.

4 Delimited Continuations

The general folklore is that continuations are more expressive than all the *yield* variants commonly found in current languages. Indeed, in blogs and message boards discussing these various languages, one commonly finds statements expressing the limited nature of the relevant *yield* compared to "true" continuations. We can make these statements more precise by formalizing the connection between our generalized *yield* and the traditional *shift* and *reset* for delimited continuations: we can give encodings of *yield-run* in terms of the traditional *shift-reset* and vice versa. The following definitions realize *yield* and *run* in terms of the traditional control operators:

$$\begin{array}{rcl} \mathit{run}\ e & \equiv & \mathit{reset}\ (\mathit{do}\ x \leftarrow e; \mathit{return}\ (\mathit{Result}\ x)) \\ \mathit{yield}\ e & \equiv & \mathit{shift}\ (\lambda k.\mathit{return}\ (\mathit{Susp}\ e\ k)) \end{array}$$

A *run* expression corresponds to a control delimiter. The body e is evaluated under this delimiter and its value is wrapped in the constructor *Result* before it is returned. A *yield* expression simply captures the continuation up to the closest delimiter and immediately returns this continuation as part of a suspension.

The only complexity for the reverse encoding arises from the fact that the continuation captured by *yield* is only accessible through the suspension object to the context of the *iterator*. Hence it is necessary to have a little "interpreter" to recursively unwind the iterator one suspension at a time in order to extract the underlying delimited continuation.

$$\begin{array}{rcl} \mathit{shift}\ e & \equiv & \mathit{yield}\ e \\ \mathit{reset}\ e & \equiv & \mathit{interp}\ (\mathit{run}\ e) \end{array} \qquad \begin{array}{rcl} \mathit{interp}\ \mathit{iter} & = & \mathit{case}\ \mathit{iter} \\ & & (\lambda f\ k.\ \mathit{reset}\ (f\ (\lambda i.\mathit{interp}\ (k\ i)))) \\ & & (\lambda r.r) \end{array}$$

It is possible to prove that the encodings above (in both directions) properly implement the desired operators. In order to make the proof accessible, we present the semantics of *shift-reset* in the same monadic style as the semantics for *yield*/*run*. The interesting rules are the following:

$$\begin{array}{rcl} \mathit{reset}\ e & \rightarrow & e' \\ \mathit{reset}\ e & \rightarrow & \mathit{reset}\ (e'(\lambda x.\mathit{reset}\ E[\mathit{return}\ x])) \end{array} \qquad \begin{array}{rcl} \mathit{if}\ \langle e, \Box\rangle & \mapsto^* & \langle \mathit{return}\ e', \Box\rangle \\ \mathit{if}\ \langle e, \Box\rangle & \mapsto^* & \langle \mathit{shift}\ e', E\rangle \end{array}$$

It is a straightforward exercise to check that the translation of each pair of operators is consistent with the semantics of the other pair. We omit the straightforward calculations.

Operators *shift-reset* and *yield* have multiple type systems. The choice of type system for one set of operators affects types affordable by the other via translation. While the full relationship between the type systems of these operators needs further investigation, some connections can be made. The simple type system for *yield* allows for the following *shift-reset* with $\mathit{shift} : ((a \rightarrow \mathit{ans}) \rightarrow \mathit{SR}\ a\ \mathit{ans}\ \mathit{ans}) \rightarrow \mathit{SR}\ a\ \mathit{ans}\ a$ and $\mathit{reset} : \mathit{SR}\ a\ \mathit{ans}\ \mathit{ans} \rightarrow \mathit{ans}$.

```
type SR i ans r =
  Yield i ((i -> ans) -> C i ans ans) r
data C i ans r = C { unC :: SR i ans r }

shift::((a->ans)->SR a ans ans)->SR a ans a
shift e = yield (C . e)
```

```
reset :: SR a ans ans -> ans
reset e = interp (run e)
    where
      interp (Susp f k) =
        reset $ unC $ f $ \i -> interp (k i)
      interp (Result r) = r
```

The main restriction of this system is that it fixes both the answer type *ans* and the input type a of the continuations. The fact that the input type is fixed is an artifact of the fact that i and o are fixed for

an *iterator*. With the more expressive parametric type system for *yield* this restriction can be removed allowing for *shift* : $((a \to ans) \to SR\ ans\ ans) \to SR\ ans\ a$ and *reset* : $SR\ ans\ ans \to ans$.

```
type SR ans r = Yield In (Out ans) r
data In a = In { unIn :: a }
data Out ans a = Out (a -> ans) -> SR ans ans

shift :: ((a -> ans) -> SR ans ans)->SR ans a
shift e = (yield (Out e)) >>= (return . unIn)
```

```
reset :: SR ans ans -> ans
reset e = interp (run e)
     where
       interp (Susp (Out f) k) =
         reset $ f $ \i -> interp (k (In i))
       interp (Result r) = r
```

Here we have a fixed answer type *ans* but continuations of any input type can be captured. Handling full answer type polymorphism as in the system developed by Asai and Kameyama [AK07] is a topic of future work.

5 Conclusion

Motivating Delimited Continuations. In the first paper to introduce delimited continuations, Felleisen [Fel88] motivated the control operators $\mathcal{F}$ and prompt #, using a tree walking example. That example separates two processes using a prompt: a process that traverses a recursive data structure and produces the leaves as it encounters them; and a second process that consumes the leaves as it receives them. The expression used to process the leaf values is $e = \lambda l.(\mathcal{F}\ (\lambda k.(cons\ l\ (\lambda\_.\#\ (k\ nil)))))$

This is essentially an implementation of an output-only *yield* operator: indeed each time the process yields it produces the pair of the output value and a continuation that is resumable by "_." Although Felleisen does not isolate this pattern into a separate control operator, it is intriguing that programming with delimited continuations was first demonstrated by encoding a *yield* combinator and then programming with *yield*.

Generalized Coroutines. In a recent study of coroutines, de Moura et al. [dMI09] stress the lack of a single standard formalization of coroutines and study many constructs finally introducing a *yield*-like operator for an imperative language. The semantics of their operator involves a global store and stateful *iterators* which obscure the control aspects of the operator. Looking at their specification, one can sense some connection to continuation-based operators, but the entangled state effects prevent them from formalizing operational equivalence with delimited continuations.

Restricting the Power of Yield. Based on the introductory discussion, we can identify several variants of *yield* in the spirit of Berdine et. al. [BORT02] that are stylized uses of full delimited continuations, but less expressive than the full *yield*. An "internal-only yield" can be used only in the context of a `foreach`-like loop, where the iterators can do input, output and return and is much like the *yield* of Ruby where the continuation is never exposed. Alternately an "externalized *yield*" allows *iterators* that can be decoupled from `foreach` loops and can allow the use of multiple *iterators* at the same time giving us an operator that allows us to express the *swap leaves* problem but is still restricted to one-shot continuations. Another generalization that avoids continuations reuse is "*yield* with an explicit delimiter" where the control action is not delimited by method boundaries, but by an explicit *run* operation, thereby allowing *yield* to suspend multiple method frames, which — as noted in Section 3.1 — can be implemented in most languages by a frame accumulating approach.

The differences in *yield*'s various semantics and its connection to continuations, coroutines and closures have been the topic of much debate in various online forums, `Lambda the Ultimate` [HE06], `comp.lang.ruby`, `python-list` and more. Our formalization clarifies many of these intuitions and bridges

practical language design and foundational research into continuations. In a larger context, the significance of *yield* can be stated as follows: it is an operator that brings delimited continuations to the mainstream; it is intuitive enough and appealing enough that many programming languages have adopted it in one form or another; it is natural enough to have been rediscovered many times in different contexts; its restricted (linearly-used) versions are still useful and their expressiveness deserves further study.

Acknowledgments. We would like to thank Michael D. Adams, Kyle Ross and Simon Peyton-Jones for many helpful discussions.

References

[AK07] K. Asai and Y. Kameyama. Polymorphic delimited continuations. *APLAS*, 2007.

[BMS05] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in Cω : The Power is in the Dot! In *ECOOP 2005 - Object-Oriented Programming*. Springer Berlin / Heidelberg, 2005.

[BORT02] Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 2002.

[BWD96] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *PLDI*, pages 99–107, 1996.

[CHO99] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation Strategies for first-class Continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[Dan00] Olivier Danvy. Formalizing Implementation Strategies for first-class Continuations. In *ESOP*, 2000.

[DJS07] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 2007.

[dMI09] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), 2009.

[ECM99] ECMA. 262: ECMAScript Language Specification. *ECMA*, 1999.

[ECM06] ECMA. 334: C# language specification. *ECMA*, June 2006.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190, 1988.

[FH85] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *POPL*, pages 245–254, 1985.

[Fla06] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 2006.

[GG83] R. E. Griswold and M. T. Griswold. *The Icon programming language*. Prentice-Hall, 1983.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[HDI94] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1), 1994.

[HE06] Dave Herman and Brendan Eich. Specifying ECMAScript via ML. `http://lambda-the-ultimate.org/node/1784`, 2006.

[JL03] Simon L. Peyton Jones and Ralf Lämmel. Scrap your boilerplate. In *APLAS*, 2003.

[JMPS05] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[KBD98] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *Lisp and Symbolic Computation*, 10(3), 1998.

[KGK+07] D. König, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning, 2007.

[Lis93] Barbara Liskov. A History of CLU. In *HOPL Preprints*, pages 133–147, 1993.

[MOSS96] Stephan Murer, Stephen M. Omohundro, David Stoutamire, and Clemens A. Szyperski. Iteration Abstraction in Sather. *ACM Trans. Program. Lang. Syst.*, 18(1), 1996.

[Moz06] Mozilla.org. Javascript 1.7. `https://developer.mozilla.org/en/New_in_JavaScript_1.7`, 2006.

[MS04] Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. *ITA*, 38(4), 2004.

[MSB03] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, 2003.

[Omo91] Stephen M. Omohundro. The Sather Language and Libraries. In *TOOLS*, pages 439–440, 1991.

[Sym03] Don Syme. F# homepage. `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/`, 2003.

[TH00] D. Thomas and A. Hunt. *Programming Ruby: the Pragmatic Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[vRD03] G. van Rossum and F. L. Drake. *Python Language Reference Manual*. Network Theory, 2003.

[vRE05] G. van Rossum and P. Eby. PEP 342: Coroutines via Enhanced Generators. Python Software Foundation, 2005.

[YvR01] K. P. Yee and G. van Rossum. PEP 234: Iterators. Python Software Foundation, 2001.