Yield : Mainstream Delimited Continuations

Roshan P. James and Amr Sabry

Indiana University, Bloomington

30th May, 2011

The Yield Operator

- The yield operator is beginning to surface in many mainstream languages in recent years: Ruby, Python, C#, JavaScript, F#, etc.
- What sorts of programs are written with it?
- Does it have an interesting formal semantics?
- Informally, *yield* is used to suspend the execution of a procedure and resume it later. That feels like continuations, but what is the formal connection?

Our work

1. An overview of several existing *yield* operators.
   - Origins in CLU and Icon. Popularized by Ruby in recent times.
   - Features of *yield* vary slightly from one language to the other.
   - Detailed language comparison.
   - Motivate *yield* using only Ruby and C# for this talk.
2. Extrapolate a unified set of features based on several *yield* operators. We'll look at the resulting operator formally.
3. Examine its connection to continuations.

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
▶def f(x) x=2
   y1 = yield (x+1)
   y2 = yield (x+2)
   return (y1+y2)
 end

 sum = f(2) {|a|
   if a > 100
    break
   elsif isPrime(a)
    1
   else
    0
   end
 }
 print sum
```

Ruby : Yield

Iterators

```
def f(x)
▶ y1 = yield (x+1)  x=2
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

►sum = f(2) {|a| a=3
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
▶ if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
▶ elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
▶ y1 = yield (x+1)     y1=1
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
► y2 = yield (x+2) x=2
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

▶sum = f(2) {|a|     a=4
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
 }
 print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
▶ elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
▶ else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
▶ y2 = yield (x+2)      y2=0
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
► return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
▶end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

▶sum = f(2) {|a|        sum=1
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
print sum
```

Ruby : Yield

Iterators

```
def f(x)
  y1 = yield (x+1)
  y2 = yield (x+2)
  return (y1+y2)
end

sum = f(2) {|a|
  if a > 100
   break
  elsif isPrime(a)
   1
  else
   0
  end
}
▶print sum
```

C#: Tree walking

Depth First Traversal

```
IEnumerable<int> treeWalk(Tree tr) {
   if(tr.isLeaf())
     yield return tr.leafValue();
   else {
     foreach(int v in treeWalk(tr.leftBranch()))
       yield return v;

     foreach(int v in treeWalk(tr.rightBranch()))
       yield return v;
   }
}
```

- foreach loops for consuming iterators.

C#: Using Multiple Iterators

Same fringe

```
bool sameFringe(Tree tr1, Tree tr2) {
  IEnumerator<int> w1 = treeWalk(tr1).GetEnumerator();
  IEnumerator<int> w2 = treeWalk(tr2).GetEnumerator();
  for(bool b1 = w1.MoveNext(), b2 = w2.MoveNext();
      b1 && b2;
      b1 = w1.MoveNext(), b2 = w2.MoveNext()) {
    if(w1.Current != w2.Current) return false;
  }
  return (b1 == b2);
}
```

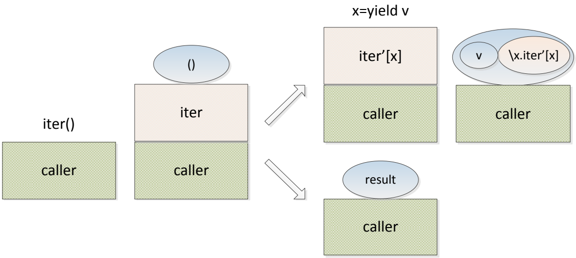
- First class access to iterators as objects.

Our *yield*

We design our generalized yield that is inspired by yield in these languages.

1. *yield* suspends functions, yielding outputs values.
2. Functions can still return values. Return values are different from yielded values.
3. *yield* can return inputs that are supplied by its calling context.
4. Suspended functions are first class objects. Suspended functions don't have to be resumed.

Yield encapsulates a delimited continuation

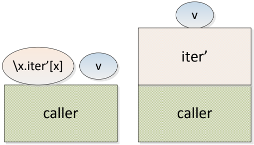


Two situations:

1. *yield* produces a pair of values : the yielded value and a function that denotes the suspended function.
   - This "suspended function" needs an input value to resume computation.
2. On termination a function produces a result value.

Resuming a suspended iterator

- The iterator's stack frame is recreated:



- The control action is limited to the iterator.

Formal Semantics I

- $E[\mathit{yield}\ v] \mapsto (v, \lambda x.E[x])$

Formal Semantics I

- $E[yield\ v] \mapsto (v, \lambda x.E[x])$
- Problem: How far does E go in the yield rule?
  - Current frame (implicit delimiter)?
  - Let's have an explicit delimiter : *run*.

Formal Semantics I

- $E[yield\ v] \mapsto (v, \lambda x.E[x])$

- Problem: How far does E go in the yield rule?
  - Current frame (implicit delimiter)?
  - Let's have an explicit delimiter : *run*.

- $E'[run\ E[yield\ v]] \mapsto E'[(v, \lambda x.run\ E[x])]$
- *run* is part of the captured iterator.

Formal Semantics I

- $E[yield\ v] \mapsto (v, \lambda x.E[x])$
- Problem: How far does E go in the yield rule?
  - Current frame (implicit delimiter)?
  - Let's have an explicit delimiter : *run*.
- $E'[run\ E[yield\ v]] \mapsto E'[(v, \lambda x.run\ E[x])]$
- *run* is part of the captured iterator.
- Calling the captured iterator:
  $E'[(\lambda x.run\ (E[x]))\ v] \mapsto E'[run\ (E[v])]$
- Here $\lambda x.run\ E[x]$ is a function — it can be called again.

Formal Semantics I

- $E[yield\ v] \mapsto (v, \lambda x.E[x])$

- Problem: How far does E go in the yield rule?
  - Current frame (implicit delimiter)?
  - Let's have an explicit delimiter : *run*.

- $E'[run\ E[yield\ v]] \mapsto E'[(v, \lambda x.run\ E[x])]$
- *run* is part of the captured iterator.
- Calling the captured iterator:
  $E'[(\lambda x.run\ (E[x]))\ v] \mapsto E'[run\ (E[v])]$
- Here $\lambda x.run\ E[x]$ is a function — it can be called again.

- The context outside the delimiter might receive a pair or a final answer:
  $E'[run\ v] \mapsto E'[v]$

Formal Semantics I

- $E[yield\ v] \mapsto (v, \lambda x.E[x])$

- Problem: How far does E go in the yield rule?
  - Current frame (implicit delimiter)?
  - Let's have an explicit delimiter : *run*.

- $E'[run\ E[yield\ v]] \mapsto E'[(v, \lambda x.run\ E[x])]$
- *run* is part of the captured iterator.
- Calling the captured iterator:
  $E'[(\lambda x.run\ (E[x]))\ v] \mapsto E'[run\ (E[v])]$
- Here $\lambda x.run\ E[x]$ is a function — it can be called again.

- The context outside the delimiter might receive a pair or a final answer:
  $E'[run\ v] \mapsto E'[v]$
- We use a sum type to distinguish the two cases:
  $Iterator\ i\ o\ r = Result\ r \mid Susp\ o\ (i \rightarrow Iterator\ i\ o\ r)$

Abstract the semantics

- Monads to explore various implementations, translations.

Abstract the semantics

- Monads to explore various implementations, translations.
- To ensure separate order of evaluation from the control effect.

Abstract the semantics

- Monads to explore various implementations, translations.
- To ensure separate order of evaluation from the control effect.
- Encapsulate the *yield* effect within the scope of *run*.

Yield Monad I

- Monadic type is parameterized by fixed input/output types.
- $M\ a \Longrightarrow Yield\ i\ o\ a$.

Monadic language: Syntax

- New value constructors *Susp* and *Result*.
- *yield* and its delimiter *run*.

Syntax

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

Monadic language: Syntax

- New value constructors *Susp* and *Result*.
- *yield* and its delimiter *run*.

Syntax

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

Monadic language: Syntax

- New value constructors *Susp* and *Result*.
- *yield* and its delimiter *run*.

Syntax

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

Monadic language: Types

Types

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

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

- *yield* is an effectful function, $\mathit{yield} : o \rightarrow i$.
- *run* reifies a computation into an interactive data structure, $\mathit{run} : \mathit{Yield}\ i\ o\ r \rightarrow \mathit{Iterator}\ i\ o\ r$.
- In Haskell syntax : $\mathit{Iterator}\ i\ o\ r = \mathit{Result}\ r \mid \mathit{Susp}\ o\ (i \rightarrow \mathit{Iterator}\ i\ o\ r)$

Monadic Semantics : pure and monadic evaluation

- Completely standard except for our monadic type *Yield i o*
- *case* is an elimination form for *Iterator i o r*

Evaluation rules

Pure Evaluation:

$$\begin{array}{rcl} (\lambda x.e)\ e' & \rightarrow & e[e'/x] \\ case\ (Susp\ e_1\ e_2)\ f\ g & \rightarrow & f\ e_1\ e_2 \\ case\ (Result\ e)\ f\ g & \rightarrow & g\ e \end{array}$$

Monadic (sequenced) evaluation:

$$\begin{array}{rcl} \langle do\ x \leftarrow e_1; e_2, E\rangle & \mapsto & \langle e_1, E[do\ x \leftarrow \square; e_2]\rangle \\ \langle return\ e_1, E[do\ x \leftarrow \square; e_2]\rangle & \mapsto & \langle e_2[e_1/x], E\rangle \end{array}$$

Types

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

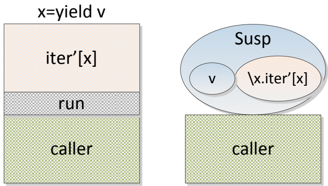
$$\frac{\Gamma \vdash e : Iterator\ i\ o\ r \quad \Gamma \vdash e_1 : o \rightarrow (i \rightarrow Iterator\ i\ o\ r) \rightarrow t \quad \Gamma \vdash e_2 : r \rightarrow t}{\Gamma \vdash case\ e\ e_1\ e_2 : t}\ case$$

Monadic Semantics : Yield and Run

Evaluation rules

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

- *run* tags the "pure" return value : produces a *Result*
- *yield* captures the delimited continuation : produces a *Susp*



Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3;\ return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \square \rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \square; return\ (a * 2) \rangle$

Example

- $\mathit{let}\ v = \mathit{run}\ (\mathit{do}\ a \leftarrow \mathit{yield}\ 3; \mathit{return}\ (a * 2))$
  $\mathit{in}\ \mathit{case}\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle \mathit{do}\ a \leftarrow \mathit{yield}\ 3; \mathit{return}\ (a * 2), \Box \rangle$
    $\mapsto \langle \mathit{yield}\ 3, \mathit{do}\ a \leftarrow \Box; \mathit{return}\ (a * 2) \rangle$
  - This creates a pair of a value and a continuation:
    $\mathit{run}\ (\mathit{do}\ a \leftarrow \mathit{yield}\ 3; \mathit{return}\ (a * 2))$
    $\rightarrow \mathit{Susp}\ 3\ (\lambda x.\mathit{run}\ (\mathit{do}\ a \leftarrow \mathit{return}\ x; \mathit{return}\ (a * 2)))$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \square \rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \square; return\ (a * 2) \rangle$
  - This creates a pair of a value and a continuation:
    $run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
    $\rightarrow Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$

- Thus we have:
  $\rightarrow let\ v = Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \square\rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \square; return\ (a * 2)\rangle$
  - This creates a pair of a value and a continuation:
    $run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
    $\rightarrow Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
- Thus we have:
  $\rightarrow let\ v = Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
- $\rightarrow case\ (Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2))))$
  $(\lambda xk.k\ (x * x))(\lambda x.x + 1)$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \square\rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \square; return\ (a * 2)\rangle$
  - This creates a pair of a value and a continuation:
    $run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
    $\rightarrow Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
- Thus we have:
  $\rightarrow let\ v = Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
- $\rightarrow case\ (Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2))))$
  $(\lambda xk.k\ (x * x))(\lambda x.x + 1)$
- Resuming the continuation:
  $\rightarrow^*\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))\ (3 * 3)$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \Box\rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \Box; return\ (a * 2)\rangle$
  - This creates a pair of a value and a continuation:
    $run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
    $\rightarrow Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
- Thus we have:
  $\rightarrow let\ v = Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
- $\rightarrow case\ (Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2))))$
  $(\lambda xk.k\ (x * x))(\lambda x.x + 1)$
- Resuming the continuation:
  $\rightarrow^*\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))\ (3 * 3)$
- $\rightarrow run\ (do\ a \leftarrow return\ 9; return\ (a * 2))$

Example

- $let\ v = run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
  - Let us focus on reductions inside the monad:
    $\langle do\ a \leftarrow yield\ 3; return\ (a * 2), \square\rangle$
    $\mapsto \langle yield\ 3, do\ a \leftarrow \square; return\ (a * 2)\rangle$
  - This creates a pair of a value and a continuation:
    $run\ (do\ a \leftarrow yield\ 3; return\ (a * 2))$
    $\rightarrow Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
- Thus we have:
  $\rightarrow let\ v = Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))$
  $in\ case\ v\ (\lambda xk.k(x * x))\ (\lambda x.x + 1)$
- $\rightarrow case\ (Susp\ 3\ (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2))))$
  $(\lambda xk.k\ (x * x))(\lambda x.x + 1)$
- Resuming the continuation:
  $\rightarrow^* (\lambda x.run\ (do\ a \leftarrow return\ x; return\ (a * 2)))\ (3 * 3)$
- $\rightarrow run\ (do\ a \leftarrow return\ 9; return\ (a * 2))$
- $\rightarrow^* Result\ 18$

Delimited Continuations

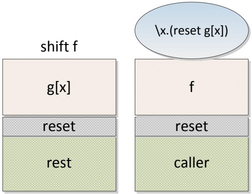
- We now have a formal semantics for *yield*.
- What is the connection to delimited continuations?
- Compare to *shift-reset*.

Delimited Continuations

Operational Semantics

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

- Monadic semantics for *shift*-reset in the same style.



Encoding *yield* using *shift-reset*

Translations

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

Operational Semantics

yield

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

shift-reset

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

Encoding *yield* using *shift-reset*

Translations

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

Operational Semantics

yield

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

shift-reset

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

Encoding *yield* using *shift-reset*

Translations

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

Operational Semantics

yield

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

shift-reset

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

Encoding *shift-reset* in terms of *yield*

Translations

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

- *reset* is represented by *run* along with a simple interpreter for the *Iterator i o r* type.

Operational Semantics

yield

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

shift-reset

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

Encoding *shift-reset* in terms of *yield*

Translations

$$\begin{array}{rcl} \textit{shift } e & \equiv & \textit{yield } e \\ \textit{reset } e & \equiv & \textit{interp } (\textit{run } e) \end{array}$$

$$\begin{array}{rcl} \textit{interp iter} & = & \textit{case iter} \\ & & (\lambda f\ k.\ \textit{reset}\ (f\ (\lambda i.\textit{interp}\ (k\ i)))) \\ & & (\lambda r.r) \end{array}$$

- *reset* is represented by *run* along with a simple interpreter for the *Iterator i o r* type.

Operational Semantics

yield

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

shift-reset

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

Encoding *shift-reset* in terms of *yield*

Translations

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

- *reset* is represented by *run* along with a simple interpreter for the *Iterator i o r* type.

Operational Semantics

yield

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

shift-reset

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

Encoding *shift-reset* in terms of *yield*

Translations

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

- *reset* is represented by *run* along with a simple interpreter for the *Iterator i o r* type.

Operational Semantics

yield

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

shift-reset

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

What about types?

- Operational correspondence has been established.
- The choice *yield* type system influences what types we can assign to *shift-reset*.

shift-reset type system I

- Fixed answer type, 'ans'.
- Fixed argument type for the continuations, 'arg'.
- $M\ a \Longrightarrow SR\ arg\ ans\ a$.

Typed implementation of *shift-reset*

$$\frac{\Gamma \vdash e : ((arg \to ans) \to SR\ arg\ ans\ ans) \to SR\ arg\ ans\ arg}{\Gamma \vdash shift\ e : SR\ arg\ ans\ arg} \qquad \frac{\Gamma \vdash e : SR\ arg\ ans\ ans}{\Gamma \vdash run\ e : ans}$$

- Corresponds to fixed input/output interface in the *yield* type system.

shift-reset type system I

- Fixed answer type, 'ans'.
- Fixed argument type for the continuations, 'arg'.
- $M\ a \Longrightarrow SR\ arg\ ans\ a$.

Typed implementation of *shift-reset*

$$\frac{\Gamma \vdash e : ((arg \to ans) \to SR\ arg\ ans\ ans) \to SR\ arg\ ans\ arg}{\Gamma \vdash shift\ e : SR\ arg\ ans\ arg} \qquad \frac{\Gamma \vdash e : SR\ arg\ ans\ ans}{\Gamma \vdash run\ e : ans}$$

- Corresponds to fixed input/output interface in the *yield* type system.

Yield Monad II

- $M\ a \Longrightarrow Yield\ i\ o\ a$.
- Input and output types are parametric types.
- Previously $yield : o \rightarrow i$.
- And now $yield : o\ a \rightarrow i\ a$ for any type a.

Parametric *yield*

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

shift-reset type system II

- $M\ a \Longrightarrow SR\ ans\ a$.
- Fixed answer type, 'ans'.
- The continuation can take any argument type.

shift-reset

$$\frac{\Gamma \vdash e : ((a \rightarrow ans) \rightarrow SR\ ans\ ans) \rightarrow SR\ ans\ a}{\Gamma \vdash shift\ e : SR\ ans\ a} \qquad \frac{\Gamma \vdash e : SR\ ans\ ans}{\Gamma \vdash run\ e : ans}$$

- Answer type polymorphism?

Main result

Our yield/run can macro-express shift/reset and vice-versa.

Operators yield/run naturally encapsulate the control effect.

The Yield Operator : Practice

- Mainstream languages have shied away from giving users access to continuations or control operators.
  - Informally they argue that:
    - this has limited utility.
    - they are too complex for the majority of users to grok
    - they add to the complexity of the language implementation.
  - May be its time to revisit this situation.
    - *yield* is used pervasively in languages like Ruby as a control abstraction.
    - *yield* is already implemented in various forms in many languages.
  - Continuations can be made available to mainstream programming: we just have to change the way they are packaged.

The Yield Operator : Theory

- May be the change in packaging is of theoretical interest too:
  - This form of exposing control already suggests several natural restrictions on the power of the continuations.
  - Many of these restrictions result in simpler implementation models:
    - One-shot continuations for example fall out very naturally from this model.
  - The full type theoretic implications of this new form of control are not fully understood.
    - We have seen two type systems that have straightforward operational interpretations.
    - Are there more?
    - How does this reflect on the answer-type polymorphism and related issues?
    - Implications for proof-theoretic uses of delimited conitnuations?

Questions?

Thank you!

shift-reset: Haskell code

Typed implementation of *shift-reset*

```
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
```

Parametric version of *shift-reset*: Haskell code

shift-reset reloaded

```
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
```

Ruby : Input Values

Fold

```
def fold(list, acc)
  list.each{|v|
    acc = yield v,acc
  }
  acc
end

sum = fold([1,2,3], 0) {|x, y|
  x + y
}
puts "Sum of list #{sum}"
```

Output:

```
Sum of list 6
```