Value Recursion in Monadic Computations

Levent Erkök
OGI School of Science and Engineering, OHSU

Advisor: John Launchbury

June 24th, 2002
Outline

- Recursion and effects
- Motivating examples
- Value recursion operators
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- Summary, future work and conclusions
Recursion

- Two important uses of recursion:
  - As a control structure
    * Recursive functions
  - As a means for creating cyclic data structures
    * Streams, self referential data

- Semantics of recursion is well understood
  - Extensively studied since 60’s
  - Modeled by least fixed-points
Effects

- Another fundamental programming technique
  - I/O is inevitable
  - Mutable variables, exceptions, non-determinism, ...

- Semantics of effects
  - Traditional semantics: Hoare logic
  - Monadic semantics: Moggi
The question

*How do we model recursion in the presence of effects?*

- Two different notions of recursion
  - The usual unfolding semantics
  - Value recursion
Unfolding recursion repeats effects

- \( f :: \alpha \to \alpha, \quad \text{fix } f = f (\text{fix } f) \)
- Example:
  \[
  \text{fact } 0 = \text{return } 1 \\
  \text{fact } n = \text{do } \text{putStrLn} \left( \text{“Now computing at: ” } ++ \text{show } n \right) \\
  \quad r \leftarrow \text{fact} \ (n - 1) \\
  \quad \text{return} \ (n \times r)
  \]
- Sample run
  Main> fact 3 \\
  Now computing at: 3 \\
  Now computing at: 2 \\
  Now computing at: 1 \\
  6
Value recursion

- An alternative notion when recursion is only over *values*
  - The result of a monadic action is recursively defined

- Effects should neither be *lost* nor *duplicated* but *preserved*

  ```
  do w ← .... x ..... 
  x ← .. w .. x .. 
  ...
  ```

- Arises most frequently in embedded domain specific languages
  - Recursion in the meta-language is not sufficient to express recursion in the object-language
Outline

- Recursion and effects
- **Motivating examples**
- Value recursion operators
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- Summary, future work and conclusions
Monadic GUI libraries

- Order determines screen layout:

  ```
  do f1 ← inputField (fieldSize 10)
  f2 ← inputField (fieldSize 10)
  submitButton (someAction f1 f2)
  ...
  ```

- What if submit button has to come first?

  ```
  do submitButton (someAction f1 f2)
  f1 ← inputField (fieldSize 10)
  f2 ← inputField (fieldSize 10)
  ...
  ```
Forking threads

- $forkIO :: IO () \rightarrow IO \text{ ThreadId}$

- Run two algorithms on the same input, first one to finish kills the other

\[
\text{tryBoth } \text{inp} = \text{do} \quad t1 \leftarrow forkIO \ (\text{alg1 } \text{inp} \ t2) \\
    t2 \leftarrow forkIO \ (\text{alg2 } \text{inp} \ t1) \\
    \ldots
\]

\[
\text{alg1 } \text{inp} \ t = \text{do} \quad \ldots \text{ compute with } \text{inp} \ldots \\
    \text{killThread } t
\]
Modeling circuits using monads

- Lava, Hawk
- Multiple interpretations
- From the same description, just change the monad to
  - Simulate
  - Dump a netlist description
  - ...

11/50
Basic idea

- Signals and Circuits, use abstraction:
  - “Sig α” to represent signals of type α
  - Monad “C” captures the underlying circuits semantics

- Basic components:
  
  \[
  \begin{align*}
  \text{and} & : \text{Sig} \: \text{Bool} \rightarrow \text{Sig} \: \text{Bool} \rightarrow C \ (\text{Sig} \ \text{Bool}) \\
  \text{mux} & : \text{Sig} \ : \text{Bool} \rightarrow \text{Sig} \ \alpha \rightarrow \text{Sig} \ \alpha \rightarrow C \ (\text{Sig} \ \alpha) \\
  \text{delay} & : \alpha \rightarrow \text{Sig} \ \alpha \rightarrow C \ (\text{Sig} \ \alpha)
  \end{align*}
  \]

- or, xor, etc..
Half adder

\[
\text{halfAdd} :: \text{Sig Bool} \rightarrow \text{Sig Bool} \rightarrow C (\text{Sig Bool}, \text{Sig Bool})
\]

\[
\text{halfAdd} \ a \ b = \text{do} \ c \leftarrow \text{and} \ a \ b \\
\quad s \leftarrow \text{xor} \ a \ b \\
\quad \text{return} \ (c, s)
\]
Using the half adder

- Simulation:

Main> halfAdd [True, True] [False, True] ([False,True],[True,False])

- NetList:

Main> halfAdd "a" "b"
w0 = and a b
w1 = xor a b
Result: (w0, w1)
Feedback in circuits

counter :: Sig Bool → C (Sig Int)

counter reset = do next ← delay 0 inc
               inc ← add1 out
               out ← mux reset zero next
               zero ← constant 0
               return out
Using the counter

- Simulation:
  Main> counter [False, False, True, False, False, True, False] [0,1,0,1,2,0,1]

- NetList:
  Main> counter "reset"
  w0 = delay 0 w1
  w1 = add1 w2
  w2 = mux reset w3 w0
  w3 = constant 0
  Result: w2
counter :: Sig Bool → C (Sig Int)

counter reset = do next ← delay 0 inc
   inc ← add1 out
   out ← mux reset zero next
   zero ← constant 0
   return out
The problem

\[
\text{counter} :: \text{Sig Bool} \rightarrow C (\text{Sig Int})
\]

\[
\text{counter reset} = \text{do next} \leftarrow \text{delay 0 inc}
\]

\[
\text{inc} \leftarrow \text{add1 out}
\]

\[
\text{out} \leftarrow \text{mux reset zero next}
\]

\[
\text{zero} \leftarrow \text{constant 0}
\]

\[
\text{return out}
\]

How to make the do-notation recursive?
Recursion at object-level and meta-level

Recursion in the meta-language is not sufficient to express recursion in the object-language

- Recall: usual recursion repeats effects
- We don’t want circuit elements to be recreated by the fixed-point computation!
- Recursion is only over the values
Making the do-notation recursive

- Recall how recursive let works

```plaintext
let (x, y) =

  let x = 1 : y
  y = 2 : x
in x

  fix (λ(x, y). let x = 1 : y
        y = 2 : x
   in (x, y))
in x
```

- Getting rid of recursive-let with `fix` and non-recursive let

- What if we have effects?
What we want

\[ mfix \ (\lambda (\text{next, inc, out, zero}). \]

\[
\begin{align*}
do & \text{ next} \leftarrow \text{delay} \ 0 \ \text{inc} \\
& \text{inc} \leftarrow \text{add1} \ \text{out} \\
& \text{out} \leftarrow \text{mux} \ \text{reset} \ \text{zero} \ \text{next} \\
& \text{zero} \leftarrow \text{constant} \ 0 \\
& \text{return} \ (\text{next, inc, out, zero}) \\
\end{align*}
\]

\[ \gg= \ \lambda (\text{next, inc, out, zero}). \ \text{return} \ \text{out} \]

- \textit{mfix}: the \textit{value recursion} operator
  \[
mfix :: (\alpha \to m \alpha) \to m \alpha
\]
- Compare: \textit{fix} :: (\alpha \to \alpha) \to \alpha
Outline

- Recursion and effects
- Motivating examples
- **Value recursion operators**
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- Summary, future work and conclusions
Generalizing \textit{fix}

• An \textit{mfix} for all monads?

• Recall: \textit{fix} \( f = f \ (\textit{fix} \ f) \), \quad \textit{fix} :: (\alpha \to \alpha) \to \alpha

• Possible definition for \textit{mfix}:

\[
mfix :: (\alpha \to m \alpha) \to m \alpha
\]

\[
mfix \ f = mfix \ f \gg f
\]

\[
= fix (\lambda m. \ m \gg f)
\]

• \textit{fix} \( f = \bigcup \{\bot, \ f \ \bot, \ f \ (f \ \bot), \ f \ (f \ (f \ \bot)), \ldots\} \)


\( mfix \ f = \mathbf{fix} \ (\lambda m. \ m \gg f) \)

- That is:

\[
mfix \ f = \bigsqcup \{ \bot, \ \bot \gg f, \ \bot \gg f \gg f, \ldots \}
\]

- Would diverge whenever \( \gg \) is left-strict

- Computing the fixed point over both effects and values!
  - But we want to compute it only over the values

\[\begin{array}{c}
\scriptstyle \xymatrix{\bullet \\
\bullet}
\end{array}\]
Example: State Monad

\[
\textbf{type} \ State = \ldots
\]
\[
\textbf{type} \ ST \ \alpha = State \to (\alpha, \ State)
\]

\[
mfix :: (\alpha \to ST \alpha) \to ST \alpha
\]
\[
:: (\alpha \to State \to (\alpha, \ State)) \to State \to (\alpha, \ State)
\]

\[
mfix \ f = \lambda s. \ \textbf{let} \ (a, \ s') = f \ a \ s \in (a, \ s')
\]
Example: State Monad

\textbf{type} \ State = ...

\textbf{type} \ ST \ \alpha = \ State \to (\alpha, \ State)

\textbf{mfix} :: (\alpha \to ST \ \alpha) \to ST \ \alpha

:: (\alpha \to State \to (\alpha, \ State)) \to State \to (\alpha, \ State)

\textbf{mfix} \ f = \lambda s. \ let \ (a, s') = f \ a \ s

\textbf{in} \ (a, s')

- State monad clearly separates values & effects
- Other monads are not that nice!
  - \textbf{Maybe}: (\alpha \to Maybe \ \alpha) \to Maybe \ \alpha
  - \textbf{List}: \ (\alpha \to [\alpha]) \to [\alpha]
Our Approach

- No generic solution, find individual instances
- Hypothesize expected properties of value recursion operators
- Study instances to verify properties
- Make a classification
  - identify important cases
  - identify cases when a recursive do-notation is feasible
- Relate these properties to those of $\text{fix}$
Outline

- Recursion and effects
- Motivating examples
- Value recursion operators
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- Summary, future work and conclusions
Basic Properties

- Strict functions
  - If $f$ is a strict function, $mfix f$ should be $\perp$
- Converse strictness
  - $mfix f$ should be $\perp$ only when $f$ is strict
- Purity
  - If there are no effects, $mfix$ should behave just like $fix$
Purity

$h :: \alpha \rightarrow \alpha$

$mfix \ (return \cdot h) = return \ (fix \ h)$

“Pure computations” have “pure fixed-points”
(If there are no effects, $mfix$ is just $fix$)
Tightening properties

- Left tightening
  - A *preceeding* non-interfering computation can be pulled out of a recursive loop

- Right tightening
  - A *succeeding* non-interfering computation can be pulled out of a recursive loop
Left tightening

\[ a :: m \tau \quad f :: \sigma \rightarrow \tau \rightarrow m \sigma \]

\[ mfix (\lambda x. a \gg= \lambda y. f x y) = a \gg= \lambda y. mfix (\lambda x. f x y) \]

Pulling a non-interfering computation out of the recursive loop: Tighten the loop from left
Yet others...

- Nesting property
  - Simultaneous and pointwise fixed points coincide

- Parametricity properties, if $s$ is strict:
  - $g \cdot s = map\ s \cdot f \rightarrow map\ s\ (mfix\ f) = mfix\ g$
  - Similar to: $g \cdot s = s \cdot f \rightarrow s\ (fix\ f) = fix\ g$

- Sliding:
  - $mfix\ (map\ h \cdot f) = map\ h\ (mfix\ (f \cdot h))$
  - Similar to: $fix\ (h \cdot f) = h\ (fix\ (f \cdot h))$

- etc...
<table>
<thead>
<tr>
<th></th>
<th>Strict</th>
<th>Pure</th>
<th>Left</th>
<th>Nest</th>
<th>Slide</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Maybe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Lists</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mfix_0$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$mfix_i$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$mfix_\omega$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Output</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Environment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Continuations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□</td>
</tr>
</tbody>
</table>
Outline

- Recursion and effects
- Motivating examples
- Value recursion operators
- Properties
- **The recursive do-notation**
- Related work: How do we fit in?
- Summary, future work and conclusions
• Compare:

\[ mfix \ (\lambda(next, \ inc, \ out, \ zero). \]

\[
\textbf{do} \quad next \leftarrow \text{delay} \ 0 \ \text{inc} \\
inc \leftarrow \text{add1} \ \text{out} \\
out \leftarrow \text{mux} \ \text{reset} \ \text{zero} \ \text{next} \\
\text{zero} \leftarrow \text{constant} \ 0 \\
\text{return} \ (\text{next, inc, out, zero}) \\
) \\
\Longrightarrow \ \lambda(next, \ inc, \ out, \ zero). \ \text{return} \ \text{out} \]
mdo-notation (cont.)

- To:

  ```
  do next ← delay 0 inc
      inc ← add1 out
      out ← mux reset zero next
      zero ← constant 0
      return (next, out)
  ```
The *MonadRec* class

```haskell
class Monad m => MonadRec m where
  mfix :: (a -> m a) -> m a
```

- **mdo** to be available for all instances of *MonadRec*
Importance of Left Tightening

\[
\begin{align*}
\mathbf{mdo} & \quad x \leftarrow e_1 \\
& \quad y \leftarrow e_2 \quad \Rightarrow \\
& \quad e_3 \\
\mathbf{do} & \quad x \leftarrow e_1 \\
& \quad \mathbf{mdo} \quad y \leftarrow e_2 \\
& \quad e_3
\end{align*}
\]

- \( x, y \not\in FV(e_1) \)
- If there is no recursion, \( mfix \) has no effect!
  - \( \mathbf{mdo} \) is the same as \( \mathbf{do} \) in that case
  - Backward compatibility
Outline

- Recursion and effects
- Motivating examples
- Value recursion operators
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- Summary, future work and conclusions
Effects

- Programming Languages
  - Imperative Languages (Forward pointers)
  - Functional Languages
    - Impure
    - Pure
      - Mixed Effects
        - Restricted
          - ML, O’Caml
            (let val
             let fun)
        - Arbitrary
          - Scheme
      - Separated Effects
        - Clean (Uniqueness Typing)
        - Haskell (Monads)
Fixed-points

- Church: Y-Combinator
- Knaster-Tarski: Fixed Point Theorem
- Scott: Recursive Domain Equations
  - Symth & Plotkin: Category Theoretic Solutions of Recursive Domain Equations
    - Monoidal Categories & Traces
      - Street, Joyal & Verity
    - Iteration Theories
      - Bloom & Esik
    - Complete Axiomatization of Fixed Points
      - Plotkin & Simpson
- Recursion from Cyclic Sharing
  - Hasegawa
- Traced Premonoidal Categories
  - Benton & Hyland, Paterson
Outline

- Recursion and effects
- Motivating examples
- Value recursion operators
- Properties
- The recursive do-notation
- Related work: How do we fit in?
- **Summary, future work and conclusions**
Summary

- Search for a generic $mfix$
- Properties, both expected and derived
- Study of monads
  - Identity, exceptions (maybe), non-determinism (list), state, environment, output, trees, fudgets, I/O ...
- Embeddings
  - Preservation of properties through embeddings of monads
Summary (cont.)

- Transformers
  - Obtaining a new $mfix$ by transforming an old one
- The mdo-notation
  - Typing
  - Pragmatics
    * Repeated variables
    * Let-generators (monomorphic)
  - Translation algorithm
  - Implementation in February 2001 release of Hugs
Summary (cont.)

- The IO monad and $\text{fix}IO$
  - Two level semantics
    * Top layer handles “functional” core
    * Bottom layer handles I/O
    * Clear interaction via reduction rules
  - Operational meaning of $\text{fix}IO$ clarified
Summary (cont.)

- Relation to other axiomatizations
  - arrowFix
  - “traced premonoidal categories”
  - They are cleaner, but limited applicability
    * **OK:** State (lazy), environment, output
    * **But not:** Exceptions, lists, strict state, IO, tree, fudgets, ...
Summary (cont.)

- Examples, case studies
  - Circuit simulation
  - Bird’s replaceMin problem
  - Sorting networks, GUI layout problem
  - Interpreters
  - Doubly-linked circular lists with stateful nodes
  - Logical variables
Future work

- Practical:
  - Support for `mdo` in all Haskell systems
  - Opportunities in other paradigms
  - More monads...

- Theoretical:
  - Semantics of `fixIO` needs more work (parametricity)
  - A more precise “categorical” account via traces
  - A precise analysis for the continuation monad
Conclusions

- **Theory:** value recursion operators form an interesting class
  - Making the interaction between effects and recursion clear is important

- **Practice:** Work on `mdo` provides necessary syntactic support in Haskell
  - Lava and Hawk can really use it
  - More in the spirit of Haskell:
    - `let` is recursive, why not `do`?
Conclusions (cont.)

- Future of functional programming
  - Lazy imperative programming
  - Semantics and implementation of embedded domain specific languages
  - Multiple interpretations

*All heavily rely on monads, and recursion is inevitable*