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

- $f :: \alpha \to \alpha, \qquad \mathit{fix}\ f = f\ (\mathit{fix}\ f)$
- Example:

$$\begin{aligned} &\mathit{fact}\ 0 = \mathit{return}\ 1 \\ &\mathit{fact}\ n = \mathbf{do}\ \mathit{putStrLn}\ (\text{“}\texttt{Now computing at: }\text{”} \mathbin{+\!\!+} \mathit{show}\ n) \\ &\qquad\qquad\quad r \leftarrow \mathit{fact}\ (n - 1) \\ &\qquad\qquad\quad \mathit{return}\ (n \times r) \end{aligned}$$

- 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*

$$
\begin{aligned}
\textbf{do}\ & w \leftarrow \ \ldots.\ x\ \ldots..\\
& x \leftarrow\ ..\ w\ ..\ x\ ..\\
& \ldots
\end{aligned}
$$

- 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:

$$\begin{array}{l} \textbf{do}\ \mathit{f1} \leftarrow \mathit{inputField}\ (\mathit{fieldSize}\ 10) \\ \quad\ \ \mathit{f2} \leftarrow \mathit{inputField}\ (\mathit{fieldSize}\ 10) \\ \quad\ \ \mathit{submitButton}\ (\mathit{someAction}\ \mathit{f1}\ \mathit{f2}) \\ \quad\ \ \ldots \end{array}$$

- What if submit button has to come first?

$$\begin{array}{l} \textbf{do}\ \mathit{submitButton}\ (\mathit{someAction}\ \mathit{f1}\ \mathit{f2}) \\ \quad\ \ \mathit{f1} \leftarrow \mathit{inputField}\ (\mathit{fieldSize}\ 10) \\ \quad\ \ \mathit{f2} \leftarrow \mathit{inputField}\ (\mathit{fieldSize}\ 10) \\ \quad\ \ \ldots \end{array}$$

Forking threads

- $forkIO :: IO\ () \rightarrow IO\ \ ThreadId$
- Run two algorithms on the same input, first one to finish kills the other

$$\begin{aligned} tryBoth\ \ inp\ = \mathbf{do}\ & t1 \leftarrow forkIO\ \ (alg1\ \ inp\ \ t2) \\ & t2 \leftarrow forkIO\ \ (alg2\ \ inp\ \ t1) \\ & \ldots \end{aligned}$$

$$\begin{aligned} alg1\ \ inp\ \ t\ \ = \mathbf{do}\ & ..\ compute\ \ with\ \ inp\ \ldots \\ & killThread\ \ t \end{aligned}$$

Modeling circuits using monads

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

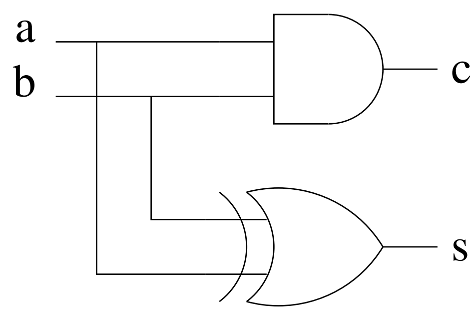
Basic idea

- Signals and Circuits, use abstraction:
  - *"Sig α"* to represent signals of type α
  - Monad *"C"* captures the underlying circuits semantics
- Basic components:

$$\begin{aligned}
&and &&:: Sig\ \ Bool \rightarrow Sig\ \ Bool \rightarrow C\ \ (Sig\ \ Bool)\\
&mux &&:: Sig\ \ Bool \rightarrow Sig\ \ \alpha \rightarrow Sig\ \ \alpha \rightarrow C\ \ (Sig\ \ \alpha)\\
&delay &&:: \alpha \rightarrow Sig\ \ \alpha \rightarrow C\ \ (Sig\ \ \alpha)
\end{aligned}$$

- *or*, *xor*, etc..

Half adder



$$halfAdd :: Sig\ Bool \rightarrow Sig\ Bool \rightarrow C\ (Sig\ Bool,\ Sig\ Bool)$$

$$\begin{aligned} halfAdd\ a\ b = \mathbf{do}\ & c \leftarrow and\ a\ b \\ & s \leftarrow xor\ a\ b \\ & return\ (c,\ s) \end{aligned}$$

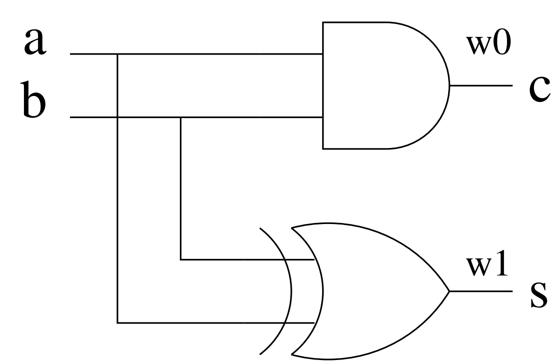
Using the half adder

- Simulation:

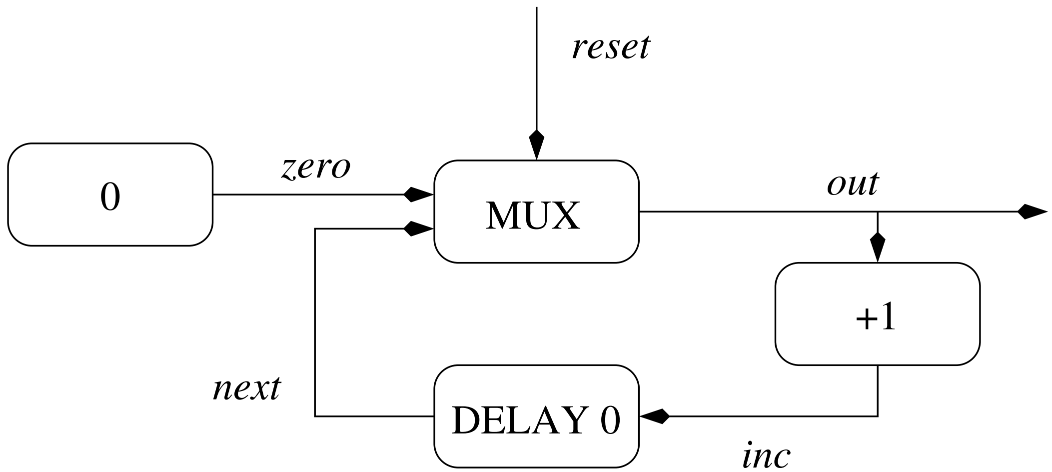
```
Main> halfAdd [True, True] [False, True]
([False,True],[True,False])
```

- NetList:

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



Feedback in circuits



$$
\begin{aligned}
&counter && :: Sig\ Bool \rightarrow C\ (Sig\ Int) \\
&counter\ reset && = \mathbf{do}\ next \leftarrow delay\ 0\ inc \\
& && \qquad inc \leftarrow add1\ out \\
& && \qquad out \leftarrow mux\ reset\ zero\ next \\
& && \qquad zero \leftarrow constant\ 0 \\
& && \qquad return\ out
\end{aligned}
$$

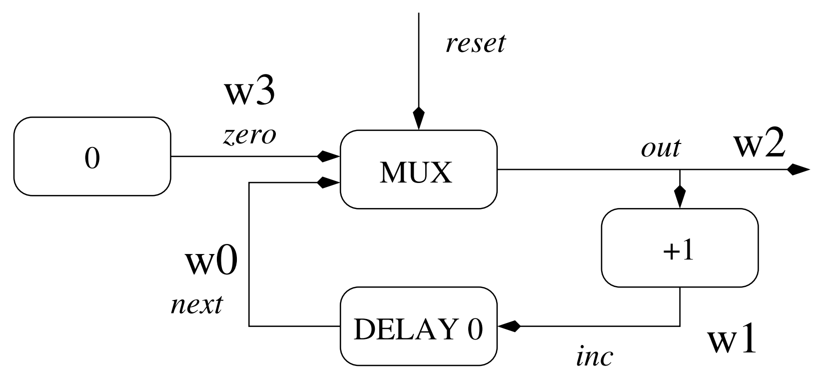
Using the counter

- Simulation:

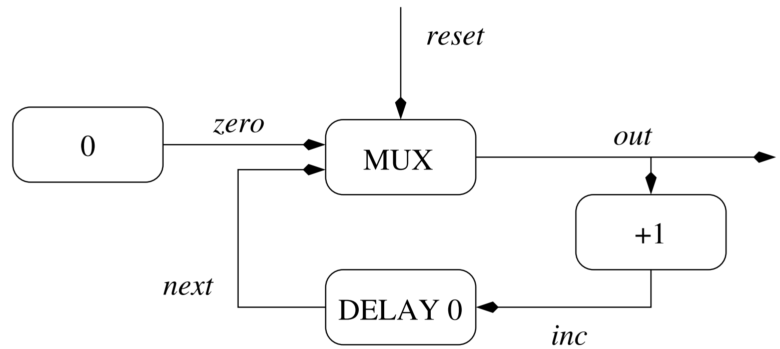
```
Main> counter [False, False, True, False, False, True, False]
[0,1,0,1,2,0,1]
```

- NetList:

```
Main> counter "reset"
w0 = delay 0 w1
w1 = add1 w2
w2 = mux reset w3 w0
w3 = constant 0
Result: w2
```

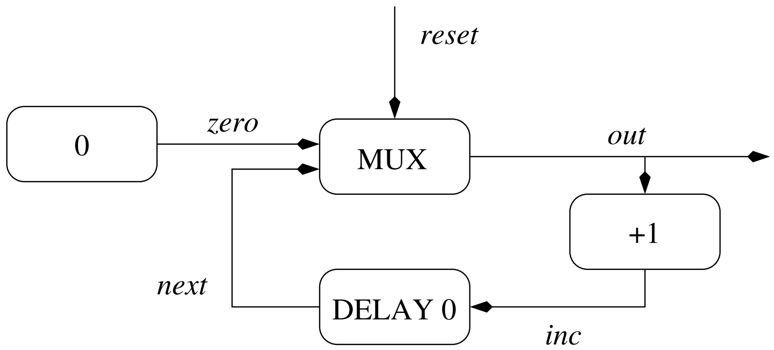


The problem



$$
\begin{array}{lll}
counter & :: & Sig\ Bool \rightarrow C\ (Sig\ Int) \\
counter\ reset & = & \mathbf{do}\ next \leftarrow delay\ 0\ inc \\
 & & \quad\ \ inc \leftarrow add1\ out \\
 & & \quad\ \ out \leftarrow mux\ reset\ zero\ next \\
 & & \quad\ \ zero \leftarrow constant\ 0 \\
 & & \quad\ \ return\ out
\end{array}
$$

The problem



$$
\begin{aligned}
&counter :: Sig\ Bool \rightarrow C\ (Sig\ Int)\\
&counter\ reset = \mathbf{do}\ next \leftarrow delay\ 0\ inc\\
&\qquad inc \leftarrow add1\ out\\
&\qquad out \leftarrow mux\ reset\ zero\ next\\
&\qquad zero \leftarrow constant\ 0\\
&\qquad return\ out
\end{aligned}
$$

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

$$\begin{array}{l} \textbf{let}\ x = 1 : y \\ \quad\ \ y = 2 : x \\ \textbf{in}\ x \end{array}$$

$$\begin{array}{l} \textbf{let}\ (x,\ y) = \\ \quad \mathit{fix}\ (\lambda(x,\ y).\ \textbf{let}\ x = 1 : y \\ \qquad\qquad\qquad\qquad\quad\ \ y = 2 : x \\ \qquad\qquad\qquad\qquad \textbf{in}\ (x,\ y)) \\ \textbf{in}\ x \end{array}$$

- Getting rid of recursive-let with *fix* and non-recursive let
- What if we have effects?

What we want

$$\begin{aligned}
&\mathit{mfix}\ (\lambda(\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero}).\\
&\qquad\qquad \mathbf{do}\ \mathit{next} \leftarrow \mathit{delay}\ 0\ \mathit{inc}\\
&\qquad\qquad\quad\ \ \mathit{inc} \leftarrow \mathit{add1}\ \mathit{out}\\
&\qquad\qquad\quad\ \ \mathit{out} \leftarrow \mathit{mux}\ \mathit{reset}\ \mathit{zero}\ \mathit{next}\\
&\qquad\qquad\quad\ \ \mathit{zero} \leftarrow \mathit{constant}\ 0\\
&\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero})\\
&\quad)\\
&\ \gg\!= \lambda(\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero}).\ \mathit{return}\ \mathit{out}
\end{aligned}$$

- mfix: the *value recursion* operator

$$\mathit{mfix} :: (\alpha \to m\ \alpha) \to m\ \alpha$$

- Compare: $\mathit{fix} :: (\alpha \to \alpha) \to \alpha$

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 *fix*

- An *mfix* for all monads?
- Recall: $\mathit{fix}\ f\ =\ f\ (\mathit{fix}\ f), \quad \mathit{fix}\ ::\ (\alpha \to \alpha) \to \alpha$
- Possible definition for *mfix*:

$$\begin{aligned} \mathit{mfix} \quad &::\ (\alpha \to m\ \alpha) \to m\ \alpha \\ \mathit{mfix}\ f\ &=\ \mathit{mfix}\ f \gg\!= f \\ &=\ \mathit{fix}\ (\lambda m.\ m \gg\!= f) \end{aligned}$$

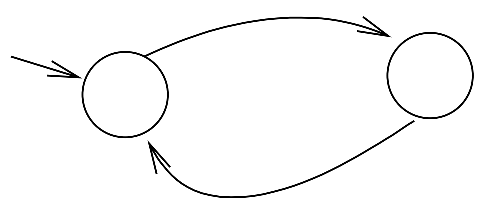
- $\mathit{fix}\, f = \bigsqcup \{\bot,\ f\ \bot,\ f\ (f\ \bot),\ f\ (f\ (f\ \bot)),\ ...\}$

mfix f = fix (λm. m $\gg=$ f)

- That is:

$$\begin{aligned} \mathit{mfix}\, f \;&=\; \bigsqcup \{\bot,\ \bot \gg= f,\ \bot \gg= f \gg= f, \\ &\qquad \bot \gg= f \gg= f \gg= f, \ldots\} \end{aligned}$$

- 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*



Example: State Monad

$$\textbf{type}\ \mathit{State} \quad = \ldots$$
$$\textbf{type}\ \mathit{ST}\ \alpha = \mathit{State} \to (\alpha,\ \mathit{State})$$

$$\begin{aligned}
\mathit{mfix} &:: (\alpha \to \mathit{ST}\ \alpha) \to \mathit{ST}\ \alpha \\
&:: (\alpha \to \mathit{State} \to (\alpha,\ \mathit{State})) \to \mathit{State} \to (\alpha,\ \mathit{State}) \\
\mathit{mfix}\ f &= \lambda s.\ \textbf{let}\ (a,\ s') = f\ a\ s \\
&\qquad\quad \textbf{in}\ (a,\ s')
\end{aligned}$$

Example: State Monad

$$\textbf{type}\ \mathit{State} \;=\; \ldots$$
$$\textbf{type}\ \mathit{ST}\ \alpha \;=\; \mathit{State} \rightarrow (\alpha,\ \mathit{State})$$

$$\begin{aligned}
\mathit{mfix} &:: (\alpha \rightarrow \mathit{ST}\ \alpha) \rightarrow \mathit{ST}\ \alpha \\
&:: (\alpha \rightarrow \mathit{State} \rightarrow (\alpha,\ \mathit{State})) \rightarrow \mathit{State} \rightarrow (\alpha,\ \mathit{State}) \\
\mathit{mfix}\ f &= \lambda s.\ \textbf{let}\ (a,\ s') = f\ a\ s \\
&\ \textbf{in}\ (a,\ s')
\end{aligned}$$

- State monad clearly separates values & effects
- Other monads are not that nice!
  - Maybe: $(\alpha \rightarrow \mathit{Maybe}\ \alpha) \rightarrow \mathit{Maybe}\ \alpha$
  - List: $(\alpha \rightarrow [\alpha]) \rightarrow [\alpha]$

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 *fix*

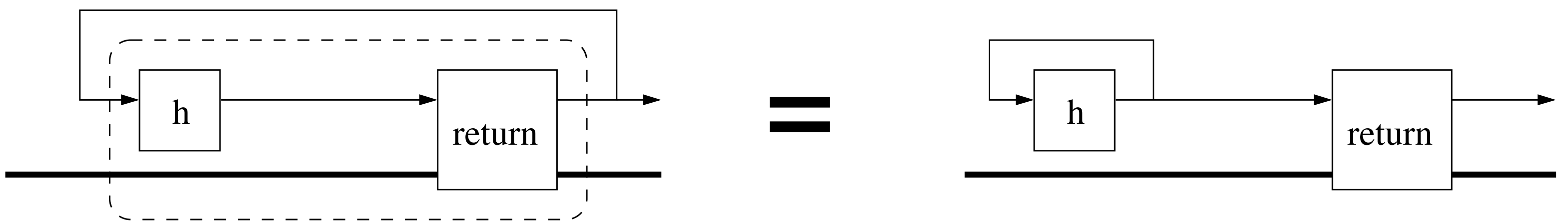
Outline

- 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



$$h :: \alpha \to \alpha$$

$$\mathit{mfix}\ (\mathit{return} \cdot h) = \mathit{return}\ (\mathit{fix}\ h)$$

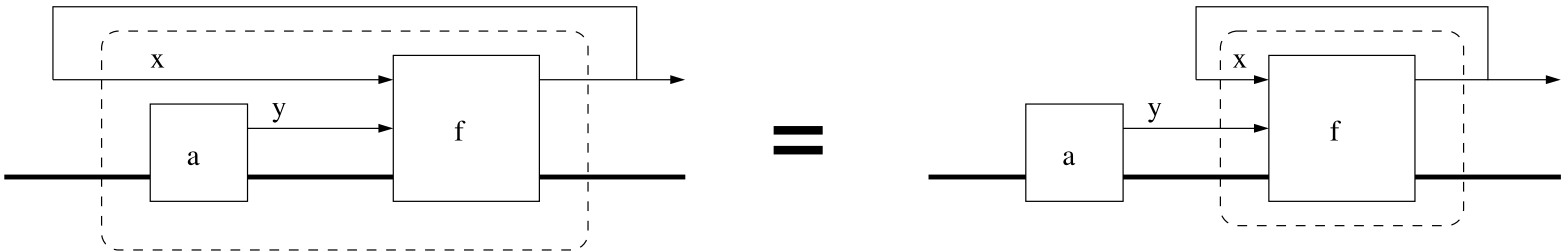
"Pure computations" have "pure fixed-points"

(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



$a :: m\ \tau \qquad\qquad f :: \sigma \to \tau \to m\ \sigma$

$$mfix\ (\lambda x.\ a \gg\!= \lambda y.\ f\ x\ y) = a \gg\!= \lambda y.\ mfix\ (\lambda x.\ f\ x\ y)$$

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...

		Strict	Pure	Left	Nest	Slide	Right
Identity		✔	✔	✔	✔	✔	✔
Maybe		✔	✔	✔	✔	✗	✗
Lists		✔	✔	✔	✔	✗	✗
State	$mfix_0$	✔	✔	✔	✔	✗	✗
	$mfix_i$	✔	✔	✔	✗	✗	✗
	$mfix_\omega$	✔	✔	✔	✔	✔	✔
Output		✔	✔	✔	✔	✔	✔
Environment		✔	✔	✔	✔	✔	✔
Continuations		?					

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

mdo-notation

- Compare:

$$
\begin{array}{l}
\mathit{mfix}\ (\lambda(\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero}).\\
\qquad\qquad \mathbf{do}\ \begin{array}{lll}
\mathit{next} & \leftarrow & \mathit{delay}\ 0\ \mathit{inc}\\
\mathit{inc} & \leftarrow & \mathit{add1}\ \mathit{out}\\
\mathit{out} & \leftarrow & \mathit{mux}\ \mathit{reset}\ \mathit{zero}\ \mathit{next}\\
\mathit{zero} & \leftarrow & \mathit{constant}\ 0\\
\multicolumn{3}{l}{\mathit{return}\ (\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero})}
\end{array}\\
\quad)\\
\ \gg\!=\ \lambda(\mathit{next},\ \mathit{inc},\ \mathit{out},\ \mathit{zero}).\ \mathit{return}\ \mathit{out}
\end{array}
$$

mdo-notation (cont.)

- To:

$$
\begin{aligned}
\textbf{do}\ & next \leftarrow delay\ 0\ \ inc \\
& inc \quad \leftarrow add1\ \ out \\
& out \quad \leftarrow mux\ \ reset\ \ zero\ \ next \\
& zero \leftarrow constant\ 0 \\
& return\ (next,\ out)
\end{aligned}
$$

The *MonadRec* class

$$\textbf{class}\ Monad\ m \Rightarrow MonadRec\ m\ \textbf{where}$$
$$\mathit{mfix} :: (\alpha \to m\ \alpha) \to m\ \alpha$$

- **mdo** to be available for all instances of *MonadRec*

Importance of Left Tightening

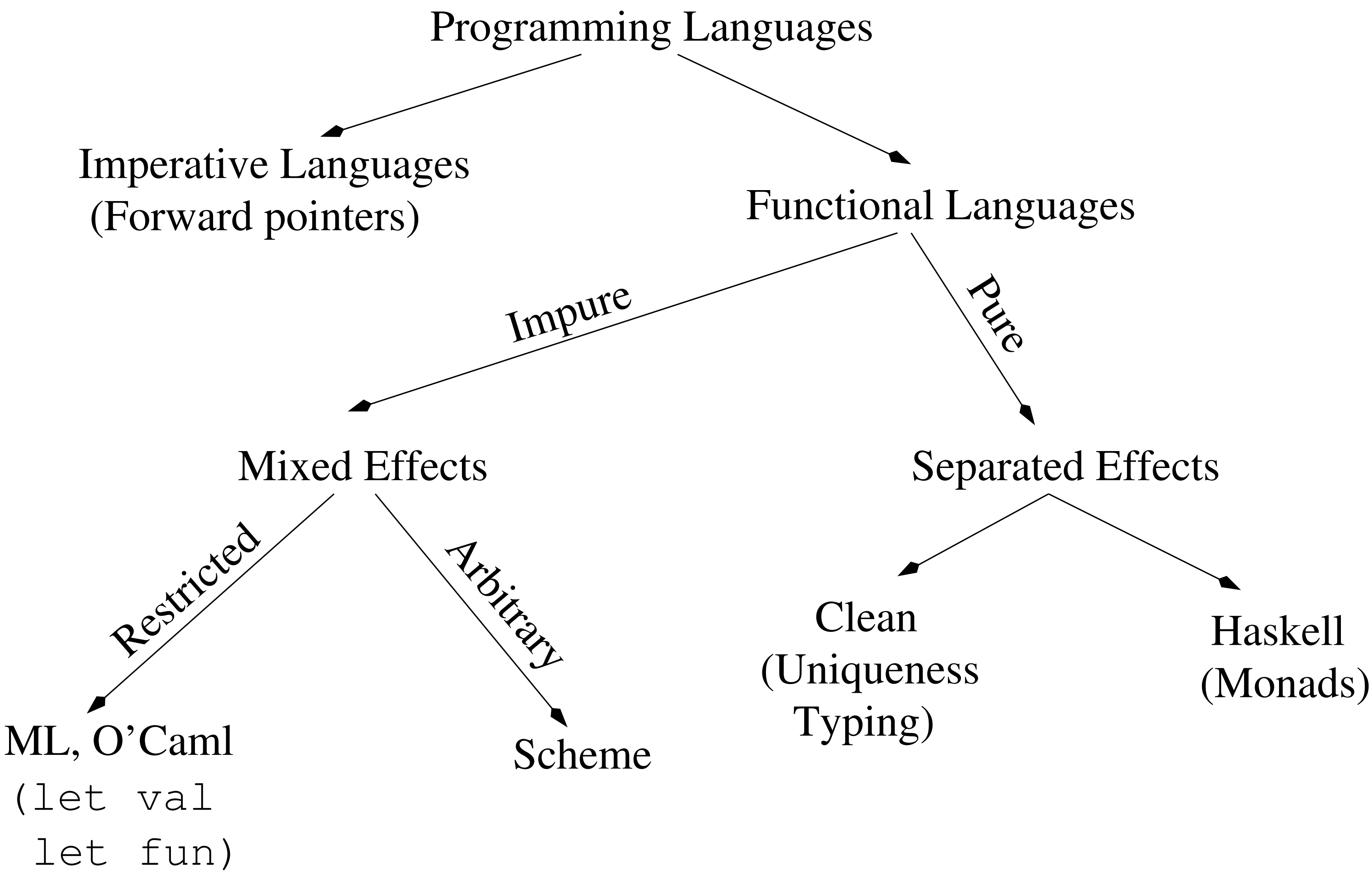
$$
\begin{array}{lll}
\textbf{mdo}\ x \leftarrow e_1 & & \textbf{do}\ x \leftarrow e_1 \\
\quad\quad\ \ y \leftarrow e_2 & \Longrightarrow & \quad\ \ \textbf{mdo}\ y \leftarrow e_2 \\
\quad\quad\ \ e_3 & & \quad\quad\quad\quad\ \ e_3
\end{array}
$$

- $x, y \notin FV(e_1)$
- If there is no recursion, *mfix* has no effect!
  - **mdo** is the same as **do** in that case
  - Backward compatibility

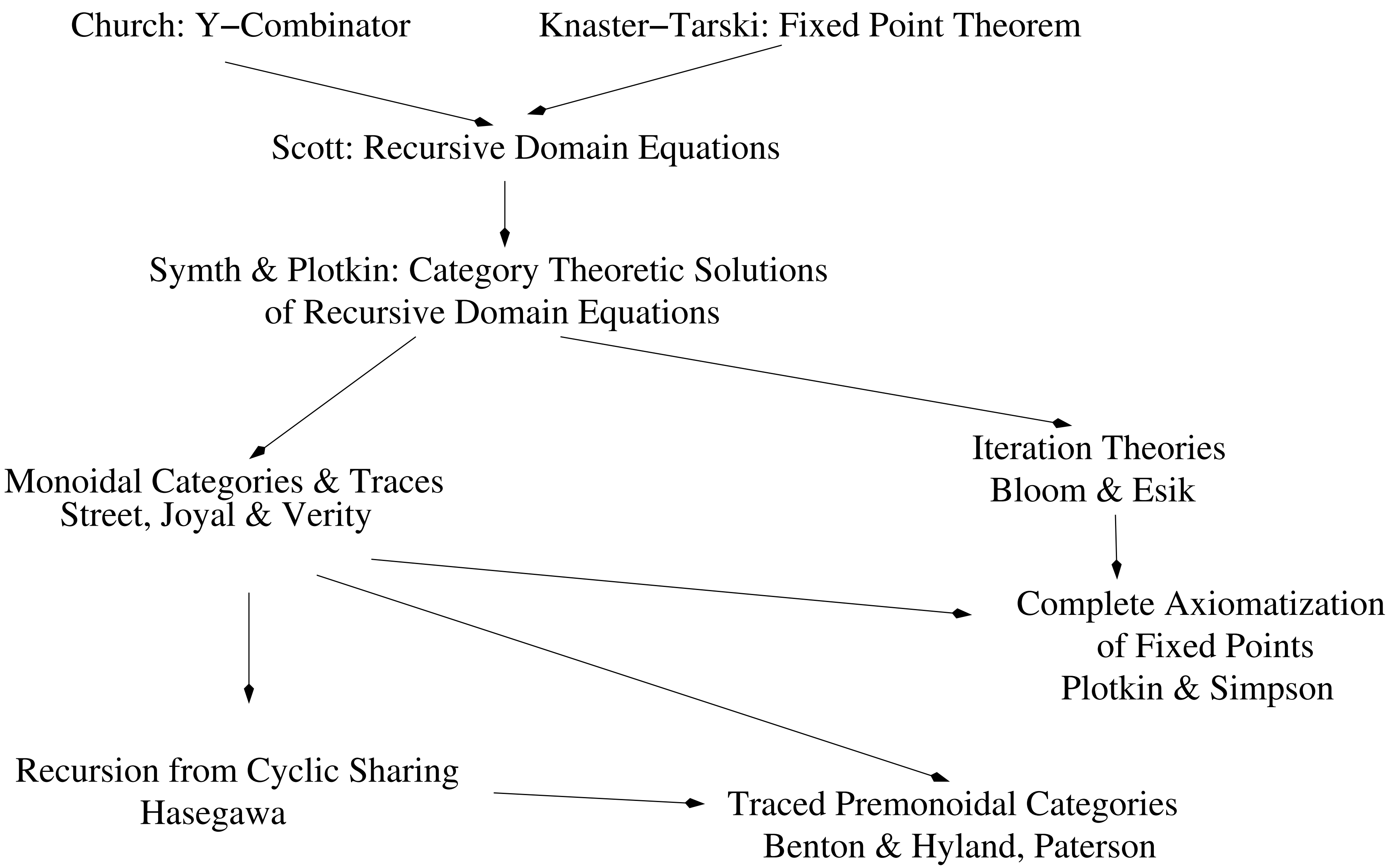
Outline

- 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

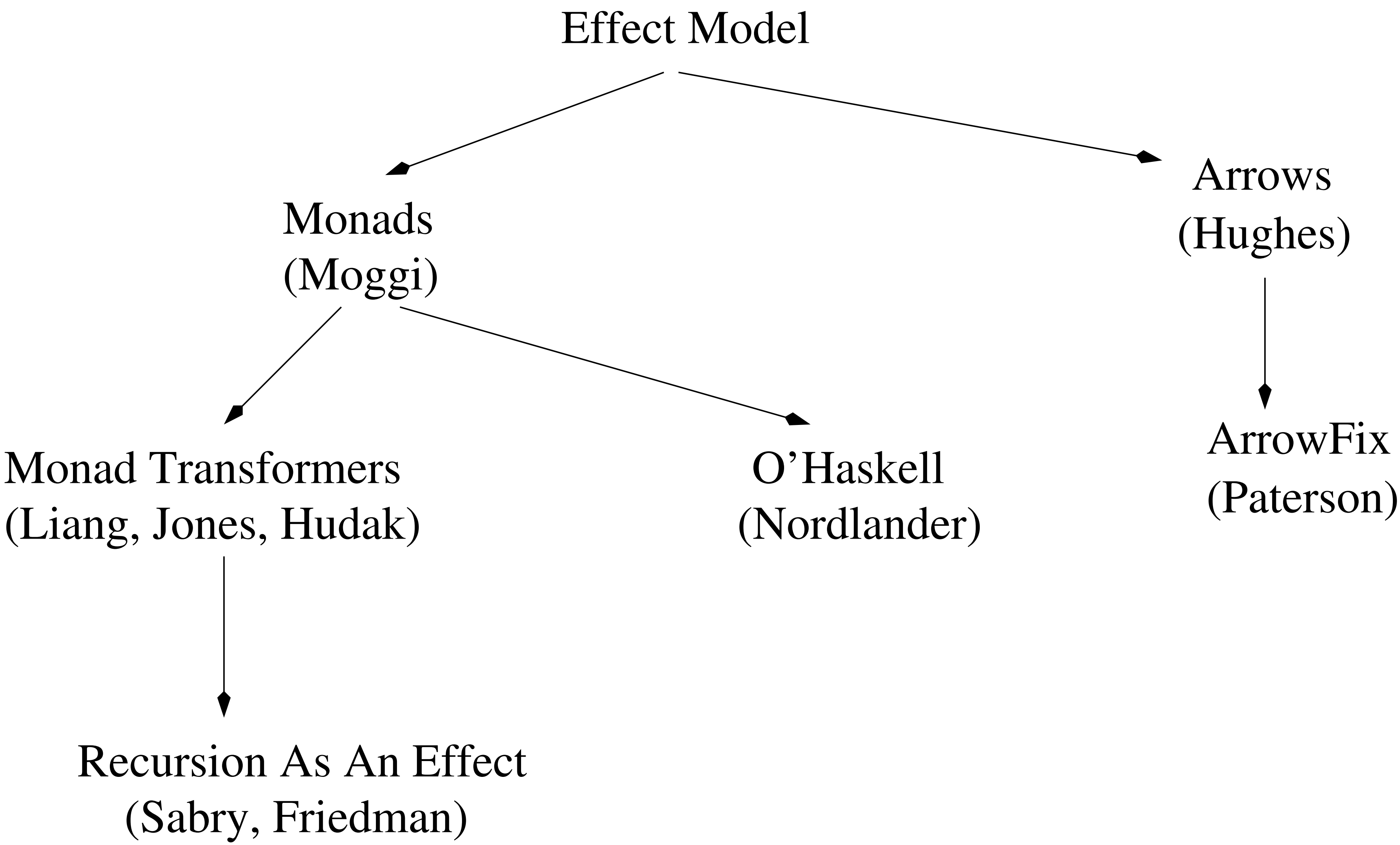
Effects


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

Fixed-points



Value Recursion



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 *fixIO*
  - Two level semantics
    - Top layer handles "functional" core
    - Bottom layer handles I/O
    - Clear interaction via reduction rules
  - Operational meaning of *fixIO* 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 embeded domain specific languages
  - Multiple interpretations

All heavily rely on monads, and recursion is inevitable