Observing Functional Logic Computations

or:

CSy: The Curry Object Observation System

Bernd Braßel Olaf Chitil Frank Huch Michael Hanus

Christian-Albrechts-
Universität
zu Kiel

University
of Kent
at Canterbury

COSy — Lazy + Logic = Hard Debugging

Curry: a lazy functional logic language (extension of Haskell)

Lazy (demand-driven) evaluation complicates debugging:

- execution trace does not match program text
- some terms are not evaluated
- print statements (for testing) might change program execution

COOSy — Lazy + Logic = Hard Debugging

Curry: a lazy functional logic language (extension of Haskell)

Lazy (demand-driven) evaluation complicates debugging:

- execution trace does not match program text
- some terms are not evaluated
- print statements (for testing) might change program execution

More problems by logic programming features:

- non-deterministic computations with multiple results
- instantiation of logic variables influences computation order
- information about bindings is relevant

COOSy — Lazy + Logic = Hard Debugging

Curry: a lazy functional logic language (extension of Haskell)

Lazy (demand-driven) evaluation complicates debugging:

- execution trace does not match program text
- some terms are not evaluated
- print statements (for testing) might change program execution

More problems by logic programming features:

- non-deterministic computations with multiple results
- instantiation of logic variables influences computation order
- information about bindings is relevant

COOSy:

➜ a relatively simple approach to help debugging

➜ extension of HOOD (Haskell observation debugger [Gill'01])

COOSy is easy

COOSy is easy to use:

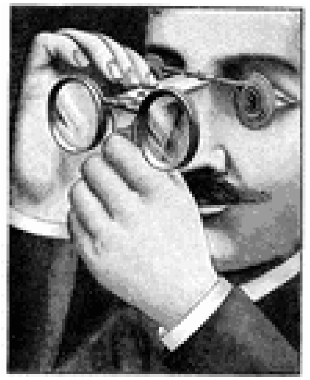
1. Import module Observe

2. Observe computed value of some expression e by

$$(observe\ observeType\ Label\ e)$$

   (many observeTypes are predefined; see later)

3. Start graphical COOSy interface, execute program,
   look at *observation protocol*



COOSy — An Example

The following program contains a bug:

```
max x y | x < y = y
        | x > y = x

maxList = foldl max 0

main = maxList [1,7,3,2,6,7,8]
```

Evaluate `main` ⇝ *no solution*

COOSy — An Example

The following program contains a bug:

```
max x y | x < y = y
        | x > y = x

maxList = foldl max 0

main = maxList [1,7,3,2,6,7,8]
```

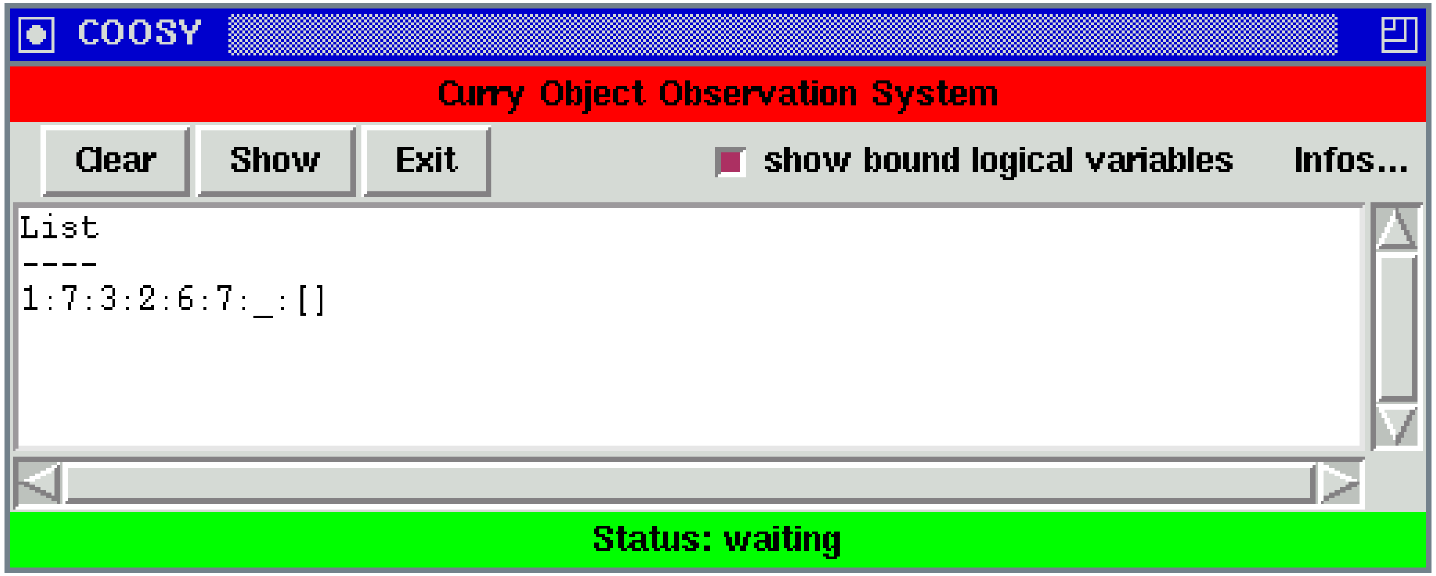
Evaluate `main` $\leadsto$ *no solution*

First debugging approach: observe the list

```
import Observe
...
main = maxList (observe (oList oInt) "List"
                [1,7,3,2,6,7,8] )
```

CO🔍Sy — Graphical Interface

Result of this example:



"_" ≈ this element has not been evaluated

COOSy — Protocol for Functional Observations

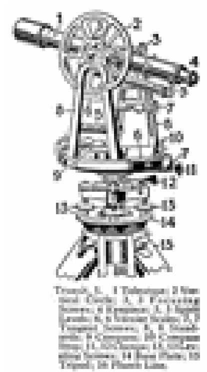
observe records uniquely numbered events:

Demand Event ≈ a value is demanded:

Format:	Demand		argument	number	parent
Example:	Demand		1	24	23

Value Event ≈ a value has been computed:

Format:	Value	value	arity	number	parent
Example:	Value	“4”	0	25	24



COOSy — Protocol for Functional Observations

observe records uniquely numbered events:

Demand Event ≈ a value is demanded:

Format:	Demand	argument	number	parent
Example:	Demand	1	24	23

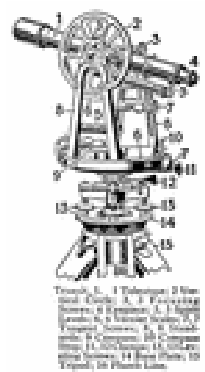
Value Event ≈ a value has been computed:

Format:	Value	value	arity	number	parent
Example:	Value	“4”	0	25	24

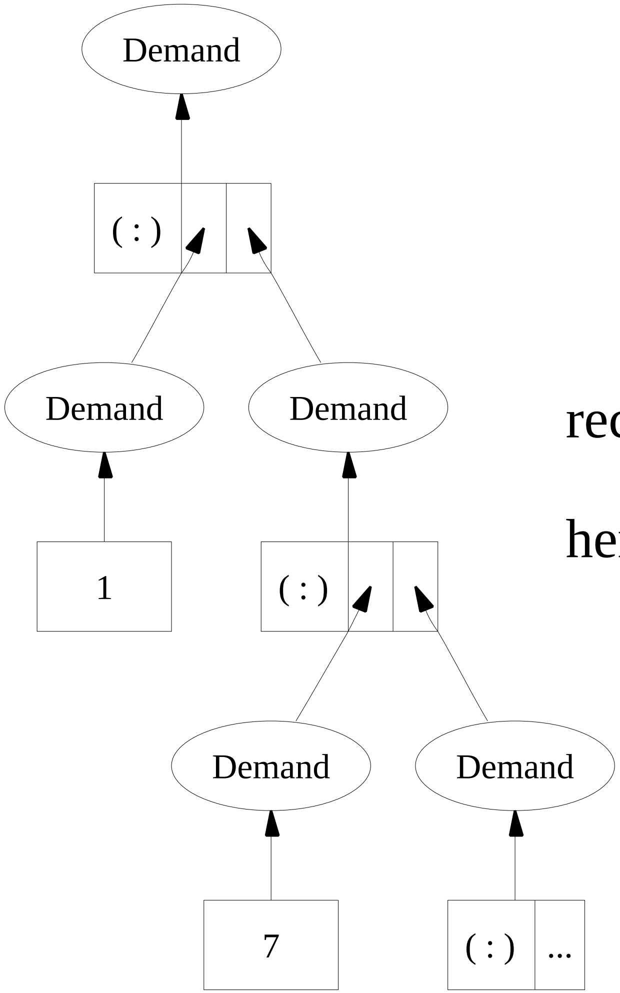
Chain of parent nodes $\leadsto$ complete data structure:

_ Value with reference r (arity > 0) but no Demand with parent r: argument not demanded

! Demand with reference r but not Value with parent r: value was demanded but not computed (failure, interrupt)

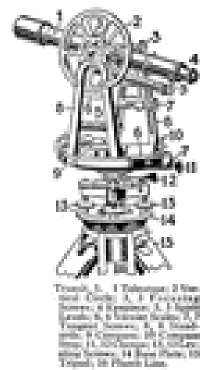


COOSy — Reconstructing the Data Structure



reconstruct data structure from parent references

here: beginning of list `1:7:[...]`



COOSy — The Observables

One can observe

- data structures: standard observation types are derivable from general patterns

```
data Nat    = O | S Nat
oNat O      = o0 "O" O
oNat (S x) = o1 oNat S "S" x
```

COOSy — The Observables

One can observe

- data structures: standard observation types are derivable from general patterns

```
data Nat    = O | S Nat
oNat O      = o0 "O" O
oNat (S x) = o1 oNat S "S" x
```

- functions:

```
oMax    = oInt ~> oInt ~> oInt
maxList = foldl (observe oMax "max" max) 0
```

`~>` is predefined infix operator of module Observe

COOSy — The Observables

One can observe

- data structures: standard observation types are derivable from general patterns

```
data Nat     = O | S Nat
oNat O       = o0 "O" O
oNat (S x) = o1 oNat S "S" x
```

- functions:

```
oMax     = oInt ~> oInt ~> oInt
maxList = foldl (observe oMax "max" max) 0
```

`~>` is predefined infix operator of module Observe

- non-deterministic branches (see below)

- bindings of logic variables (see below)

COOSy — Observing Functions

Proceed with debugging the initial example:

```
max x y | x < y = y
        | x > y = x
oMax    = oInt ~> oInt ~> oInt
maxList = foldl (observe oMax  "max" max) 0
main = maxList [1,7,3,6,7,8]
```

COOSy — Observing Functions

Proceed with debugging the initial example:

```
max x y | x < y = y
        | x > y = x
oMax    = oInt ~> oInt ~> oInt
maxList = foldl (observe oMax  "max" max) 0
main = maxList [1,7,3,6,7,8]
```

Observation protocol of a function:

argument/result pairs computed during program execution

```
{  ! _  -> !
,  7 7  -> !
,  7 6  -> 7
,  7 3  -> 7
,  1 7  -> 7
,  0 1  -> 1}
```

COOSy — Non-Determinism

We want to observe also non-deterministic operations:

```
coin = O                          plus O     x = x
coin = S O                        plus (S x) y = S (plus x y)
main = plus 0 coin
```

COOSy — Non-Determinism

We want to observe also non-deterministic operations:

```
coin = O                          plus O     x = x
coin = S O                        plus (S x) y = S (plus x y)
main = plus 0 coin
main = (observe (oNat ~> oNat ~> oNat) " + " plus) 0 coin
```

COOSy — Non-Determinism

We want to observe also non-deterministic operations:

```
coin = O                          plus O     x = x
coin = S O                        plus (S x) y = S (plus x y)
main = plus 0 coin
main = (observe (oNat ~> oNat ~> oNat) " + " plus) 0 coin
```

Desired observation:

```
 +
---
{ O O -> O }
{ O (S O) -> S O }
```

But: current chaining of events provides not enough information

COSy — Non-Determinism

We want to observe also non-deterministic operations:

```
coin = O                          plus O     x = x
coin = S O                        plus (S x) y = S (plus x y)
main = plus O coin
main = (observe (oNat ~> oNat ~> oNat) " + " plus) O coin
```

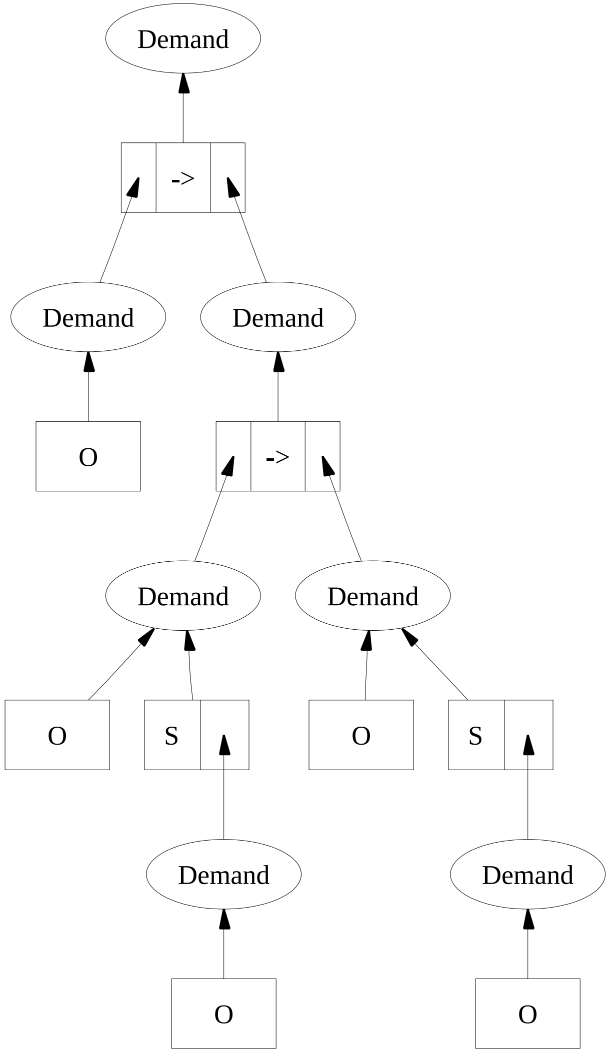
Desired observation:

```
 +
---
{ O O -> O }
{ O (S O) -> S O }
```

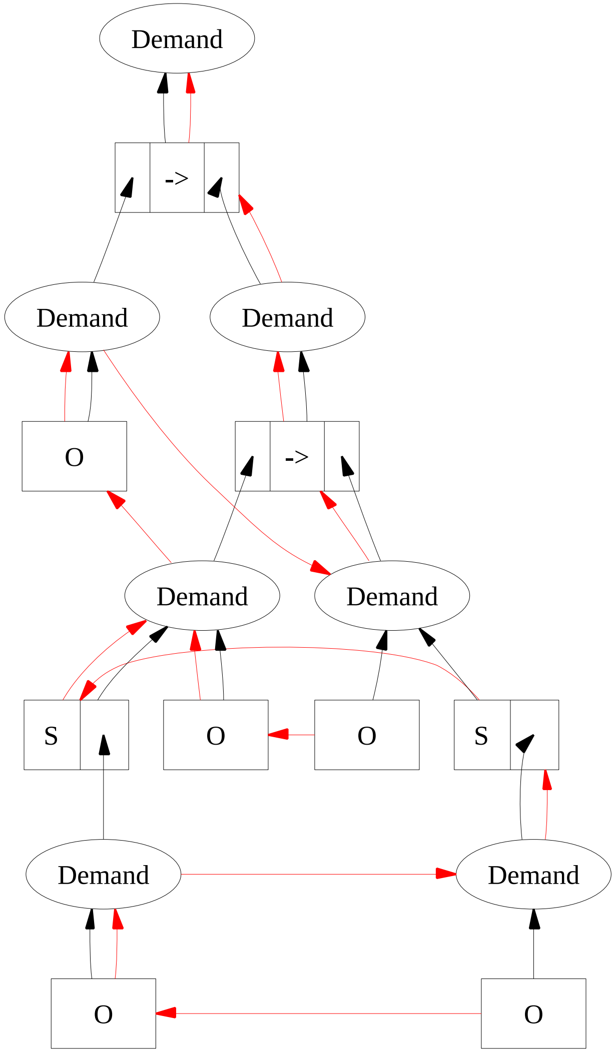
But: current chaining of events provides not enough information

Predecessor: number of event occurred just before in same branch

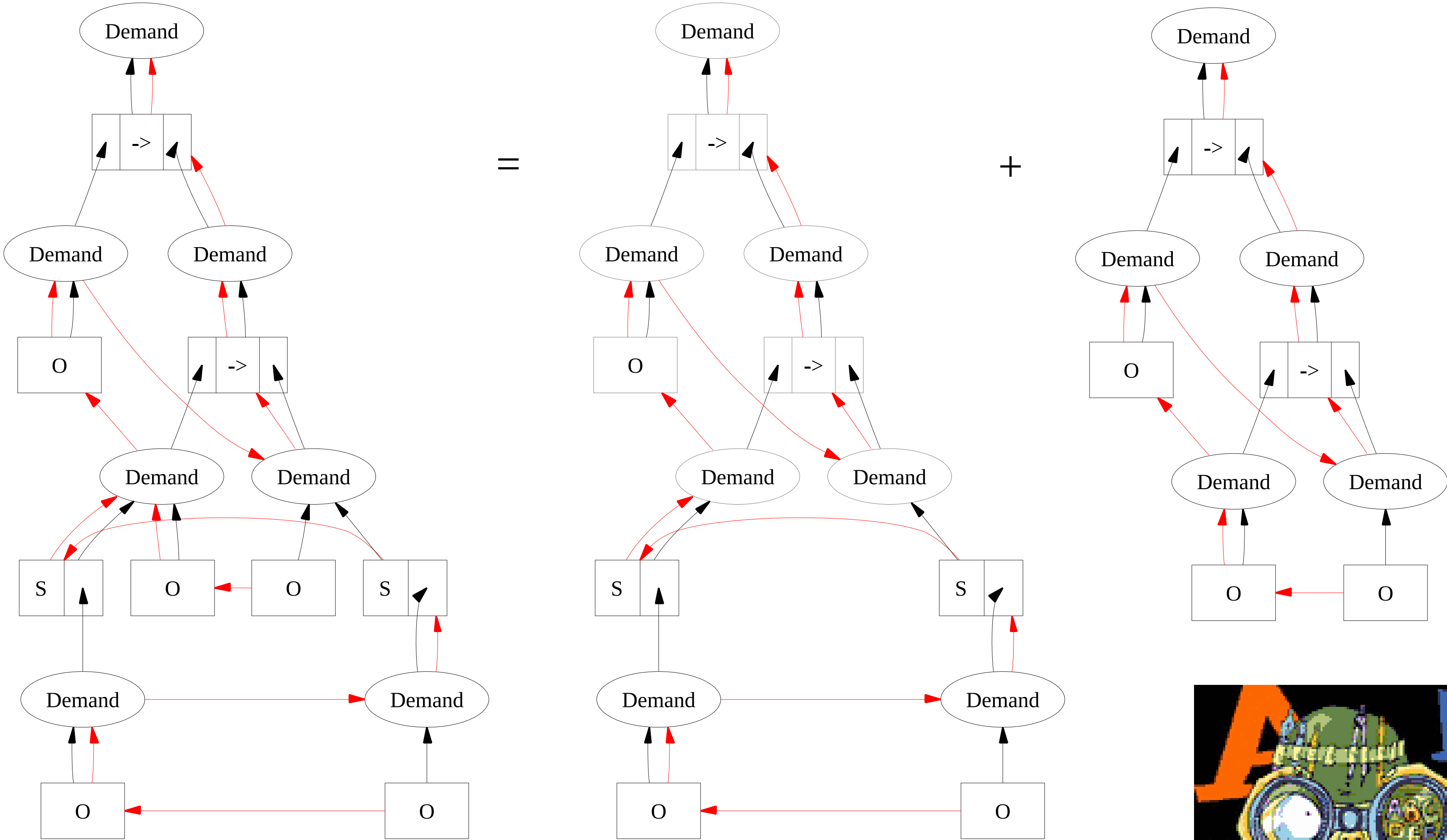
COOSy — Predecessor Chaining



COOSy — Predecessor Chaining



COOSy — Predecessor Chaining



COOSy — Logic Variables

We want to observe also logic variables:

```
plus O     x = x
plus (S x) y = S (plus x y)
main | plus                   x  y =:= S(S(S O))
     = (x,y)   where x,y free
```

⇝ four results: (0,3), (1,2), (2,1), (3,0)

COOSy — Logic Variables

We want to observe also logic variables:

```
plus O     x = x
plus (S x) y = S (plus x y)
main | plus (observe oNat "x" x) y =:= S(S(S O))
     = (x,y)   where x,y free
```

⇝ four results: (0,3), (1,2), (2,1), (3,0)

COOSy — Logic Variables

We want to observe also logic variables:

```
plus O     x = x
plus (S x) y = S (plus x y)
main | plus (observe oNat "x" x) y =:= S(S(S O))
     = (x,y)   where x,y free
```

⇝ four results: (0,3), (1,2), (2,1), (3,0)

Observation for logic variable x:

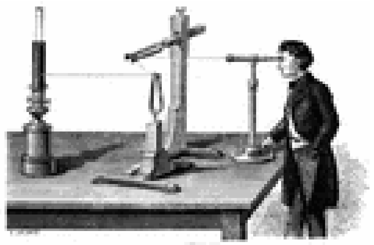
```
(?/O)
(?/(S ?/O))
(?/(S ?/(S ?/O)))
(?/(S ?/(S ?/(S ?/O))))
```

COOSy — How it Works

Problem: observation should not influence lazy evaluation

Solution: observe yields head constructor and further observe calls for arguments

Pattern: observe s (C $x_1 \ldots x_n$) = C (observe 1 $x_1 \ldots$ observe n x_n)



COOSy — How it Works

Problem: observation should not influence lazy evaluation

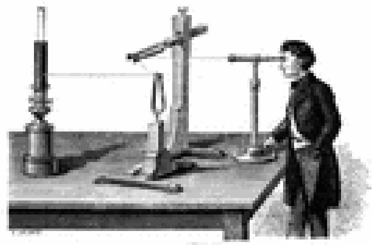
Solution: observe yields head constructor and further observe calls for arguments

Pattern: observe s (C $x_1 \ldots x_n$) = C (observe 1 $x_1 \ldots$ observe n x_n)

Problem: some terms are not evaluated

Solution: distinguish between Demand and Value events

Pattern: observe x = record Demand >> compute hnf x >>= record Value



COOSy — How it Works

Problem: observation should not influence lazy evaluation

Solution: observe yields head constructor and further observe calls for arguments

Pattern: observe s (C $x_1 \ldots x_n$) = C (observe 1 $x_1 \ldots$ observe n x_n)

Problem: some terms are not evaluated

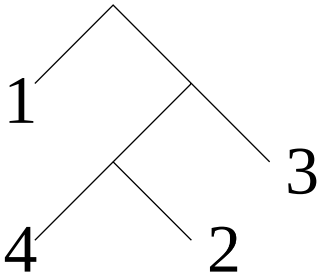
Solution: distinguish between Demand and Value events

Pattern: observe x = record Demand >> compute hnf x >>= record Value

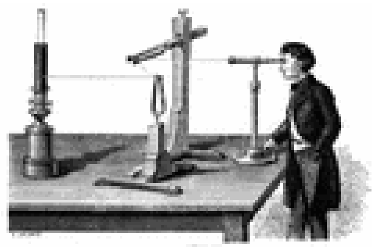
Problem: associating subterms to computation branches

Solution: reference chain to predecessor of a branch

Pattern? such information is usually not accessible!



computation order can be arbitrary
(e.g., fair computation of all branches)



COOSy — Computing Predecessor Chain

First solution: observe contains a logic variable as further parameter

- bindings visible in complete computation branch
- different computation branches — different bindings

COOSy — Computing Predecessor Chain

First solution: observe contains a logic variable as further parameter

- bindings visible in complete computation branch
- different computation branches — different bindings

Remaining problem: when should this logic variable be bound?

- multiple bindings $\leadsto$ failure of computation
- future branches not predictable $\leadsto$ *new* logic variables necessary

COOSy — Computing Predecessor Chain

First solution: observe contains a logic variable as further parameter

- bindings visible in complete computation branch
- different computation branches — different bindings

Remaining problem: when should this logic variable be bound?

- multiple bindings ⇝ failure of computation
- future branches not predictable ⇝ *new* logic variables necessary

Solution:

- variable will be bound to an open-ended list, every step records reference at the end (by instantiation)
- find end of list by (unsafe) “test of logic variable” (`isVar`)

COOSy — Observing Bindings

Final problem:

how can we observe concrete bindings of logic variables?

bindings are performed later (in other parts of the programs)

COOSy — Observing Bindings

Final problem:

how can we observe concrete bindings of logic variables?

bindings are performed later (in other parts of the programs)

Solution:

create a concurrent constraint:

this constraint suspends on logic variable and is activated on binding

Pattern:

```
if isVar x then
   spawnConstraint (x =:= observe oNat "x" x)
else ...
```

COOSy Conclusions

COOSy extends Haskell's HOOD to include

- non-determinism
- logic variables

COOSy Conclusions

COOSy extends Haskell's HOOD to include

- non-determinism
- logic variables

All desired properties of HOOD are still valid:

- simple approach
- specific observation, no big log files

COOSy Conclusions

COOSy extends Haskell's HOOD to include

- non-determinism
- logic variables

All desired properties of HOOD are still valid:

- simple approach
- specific observation, no big log files

Advantages for debugging functional logic programs:

- no difficult requirements on underlying implementation:
  functions used in module Observe: standard or easy to provide
- graphical interface ⇒ easy to use
- debug "batch" programs (e.g., web applications)

COOSy Further Work

Automatic generation of observe types $\surd$

Distribute tool and test its usability

Printing of partial information in parallel to exeuction of observed program