Unrestricted pure call-by-value recursion

Johan Nordlander, Magnus Carlsson, Andy Gill

ML'08

LULEÅ UNIVERSITY OF TECHNOLOGY

Recursive bindings

```
let  x1 = e1
     x2 = e2
in ...
```

Haskell (& friends):

e_1 and e_2 can be any kind of expression

e_1 and e_2 can have any type

SML (& friends):

e_1 and e_2 must be syntactic values

e_1 and e_2 must only have function type

Goal of this work: lift both restrictions for call-by-value!

Examples

Uncontroversial:

(a) `f = \n . if n==0 then 1 else n * f (n-1)`

Not of function type:

(b) `f = 1 : 2 : 3 : f`

Not a value:

(c)
```
f = g f
    where g = \h. \n. if n==0 then 1 else n * h (n-1)
```

But note:

- (a) and (b) are basically the same memory initialization problem
- (c) could evaluate to (a) without caring what `f` is

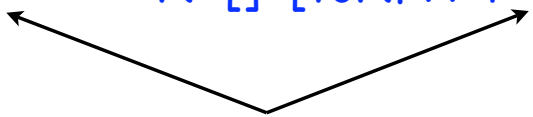
Example

Converting regular expressions to NFAs:

```
data RegExp = Lit Char
            | Seq RegExp RegExp
            | Star RegExp

data NFA    = N [(Char,NFA)] [NFA]
            | Accept

toNFA (Lit c) n       = N [(c,n)] []
toNFA (Seq r1 r2) n   = toNFA r1 (toNFA r2 n)
toNFA (Star r) n      = n'
          where n'    = N [] [toNFA r n', n]
```



Supporting unrestricted cbv recursion

- Our idea:
  - "resolve addresses dynamically like a 2-pass assembler"
  - "let address placeholders be valid function arguments"
- Semantically: evaluate in the presence of free variables
- Our contribution:
  1. A simple reduction semantics
  2. A lightweight implementation technique
- Not addressed:
  - Static detection of ill-founded recursion
  - Static identification of non-inductive data

Language

expressions	$e ::= \lambda x.e \mid x \mid e\ e' \mid \textbf{let}\ b\ \textbf{in}\ e$
bindings	$b ::= x = e \mid b,b' \mid 0$
values	$v ::= \lambda x.e$
value bindings	$\Gamma ::= x = v \mid \Gamma, \Gamma' \mid 0$
weak values	$w ::= v \mid x$

$$b,(b',b'') \equiv (b,b'),b''$$

$$b,0 \equiv b \equiv 0,b$$

Always assume bound variables don't overlap

Evaluate in the presence of a Γ (the heap)

Evaluation

Beta $\Gamma \vdash (\lambda x.e)\ w \rightarrow [w/x]e$

Var $\Gamma, x=v \vdash x \rightarrow v$

Nest $$\frac{\Gamma+E \vdash e \rightarrow e'}{\Gamma \vdash E[e] \rightarrow E[e']}$$

Merge $\Gamma \vdash E[\textbf{let}\ \Gamma'\ \textbf{in}\ w] \rightarrow (E+\Gamma')[w]$

$E ::= []\ e \mid (\lambda x.e)\ [] \mid \textbf{let}\ \Gamma, x=[], b\ \textbf{in}\ e \mid \textbf{let}\ \Gamma\ \textbf{in}\ []$

Nesting & merging

Extending a heap with the local bindings of a context (rule Nest):

$$\Gamma + (\textbf{let } \Gamma', x=[], b \textbf{ in } e) = \Gamma, \Gamma'$$
$$\Gamma + (\textbf{let } \Gamma' \textbf{ in } []) = \Gamma, \Gamma'$$
$$\Gamma + ([]\ e) = \Gamma$$
$$\Gamma + ((\lambda x.e)\ []) = \Gamma$$

Extending a context with a local heap (rule Merge):

$$(\textbf{let } \Gamma, x=[], b \textbf{ in } e) + \Gamma' = \textbf{let } \Gamma, \Gamma', x=[], b \textbf{ in } e$$
$$(\textbf{let } \Gamma \textbf{ in } []) + \Gamma' = \textbf{let } \Gamma, \Gamma' \textbf{ in } []$$
$$([]\ e) + \Gamma' = \textbf{let } \Gamma' \textbf{ in } []\ e$$
$$((\lambda x.e)\ []) + \Gamma' = \textbf{let } \Gamma' \textbf{ in } (\lambda x.e)\ []$$

Evaluation examples

$$\Gamma, x=v \vdash (\lambda y.y)\ x \xrightarrow{\text{Var}} (\lambda y.y)\ v \xrightarrow{\text{Beta}} v$$

$$\Gamma, x=v \vdash (\lambda y.y)\ x \xrightarrow{\text{Beta}} x \xrightarrow{\text{Var}} v$$

$$\Gamma \vdash (\lambda y.y)\ (\textbf{let}\ x=v\ \textbf{in}\ x) \xrightarrow{\text{Merge}} \textbf{let}\ x=v\ \textbf{in}\ (\lambda y.y)\ x$$

$$\Gamma, f=\lambda x.f\ x \vdash f\ w \xrightarrow{\text{Var}} (\lambda x.f\ x)\ w \xrightarrow{\text{Beta}} f\ w \rightarrow \ldots$$

$$\Gamma, g=\lambda h.\lambda x.h\ x \vdash \textbf{let}\ f = \boxed{g\ f}\ \textbf{in}\ f\ w \xrightarrow{\text{Var}}$$
$$\textbf{let}\ f = \boxed{(\lambda h.\lambda x.h\ x)\ f}\ \textbf{in}\ f\ w \xrightarrow{\text{Beta}}$$
$$\textbf{let}\ f = \lambda x.f\ x\ \textbf{in}\ f\ w \rightarrow \ldots$$

Confluence

... up to referential equivalence

Define: $\Gamma, x=v, \Gamma' \vdash x = v$

Lift to an equivalence relation on expressions

Theorem: If $\Gamma \vdash e \rightarrow e_1$ and $\Gamma \vdash e \rightarrow e_2$

then $\Gamma \vdash e_1 \rightarrow^* e_1'$ and $\Gamma \vdash e_2 \rightarrow^* e_2'$ such that $\Gamma \vdash e_1' = e_2'$

Theorem (referential transparency):
Reduction preserves referential equivalence

Extensions

Records:

$$e ::= \ldots \mid \{ s_i = e_i \} \mid e.s$$

$$v ::= \ldots \mid \{ s_i = w_i \}$$

$$E ::= \ldots \mid \{ s_i = []_i \} \mid [].s$$

Sel $\quad \Gamma \vdash \{ s_i = w_i \}.s_j \rightarrow w_j$

Algebraic datatypes and primitive types follow same pattern

Record examples

$\Gamma, x=\{hd=7, tl=x\} \vdash x.tl.hd \xrightarrow{Var}$

$\{hd=7, tl=x\}.tl.hd \xrightarrow{Sel}$

$x.hd \xrightarrow{Sel} 7$

$\Gamma, f=\lambda y.\lambda z.\{hd=y, tl=z\} \vdash$ **let** x = f 7 x **in** e $\xrightarrow{Var}$

let x = $(\lambda y.\lambda z.\{hd=y, tl=z\})$ 7 x **in** e $\xrightarrow{Beta}$

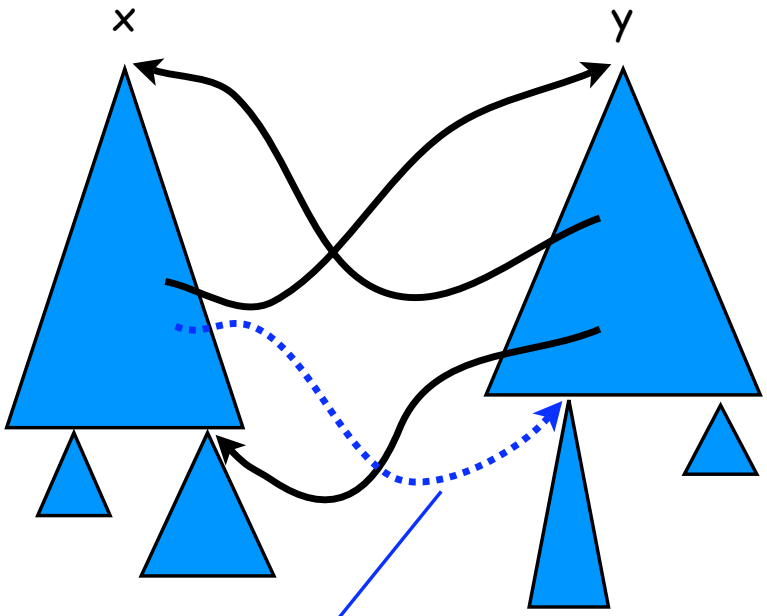
let x = {hd=7, tl=x} **in** e

$\Gamma \vdash$ **let** x={hd=7, tl=y}, y={hd=x.hd, tl=x} **in** e $\xrightarrow{Sel}$

let x={hd=7, tl=y}, y={hd=7, tl=x} **in** e

Mutually recursive data

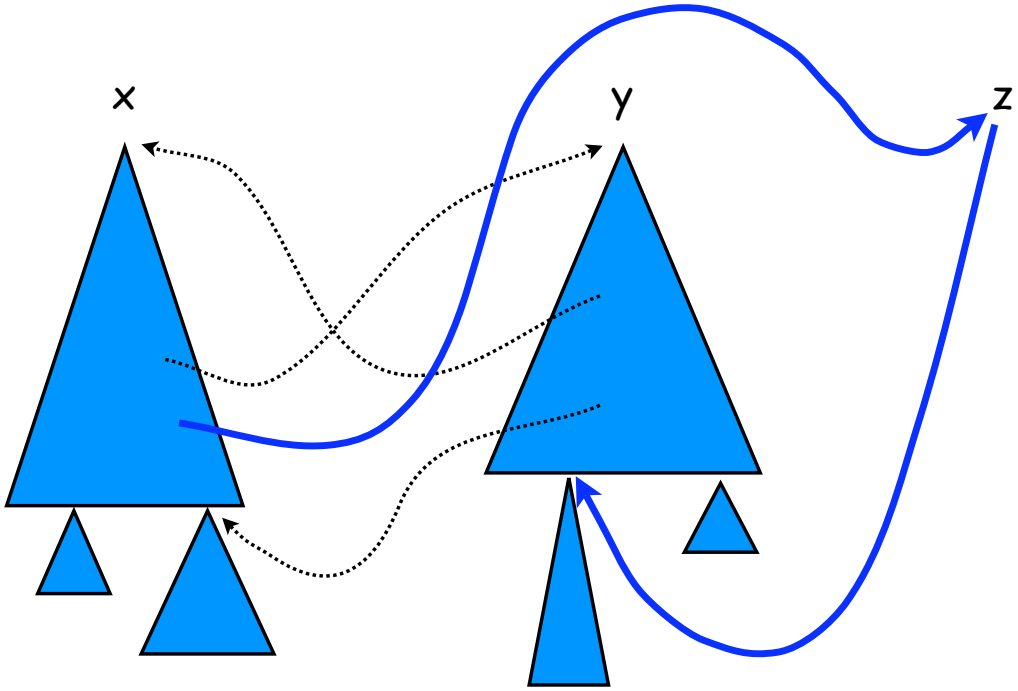
x = f … y … y.s
y = g … x … x.s



Ill-defined (needs to destruct y before y exists)

Interesting workaround

x = f … y … z
y = g … x … x.s
z = y.s



Delayed selection!

Implementation

- Heap-bound variables are pointers
- Pointer dereferencing implements rule Var
- Strategy: only dereference when necessary (not in args)
- Core challenge: how represent pointers that can't be dereferenced (variables in scope, but absent in Γ)
- Solution: use illegal but still distinct addresses
  - Odd addresses, or
  - addresses pointing outside allocated heap.
  - Keep track of next unused illegal address using a stack-like counter

Implementation

$[\![\ \textbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \textbf{in}\ e_{n+1}\ ]\!] =$

$\tau_1\ x_1 = \xi_1;$

...

$\tau_n\ x_n = \xi_n;$

$x_1 = [\![\ e_1\ ]\!];\ \ \mathrm{subst}(\theta_1, x_1);$

...

$x_n = [\![\ e_n\ ]\!];\ \ \mathrm{subst}(\theta_n, x_1, \ldots, x_n);$

$\textbf{return}\ [\![\ e_{n+1}\ ]\!]$

where $\xi_1, \ldots \xi_n$ are fresh illegal addresses
and $\theta_i = [\ x_1/\xi_1, \ldots, x_i/\xi_i\ ]$

Function subst

subst(θ, x_1, ..., x_k)

- destructively applies θ to each root x_1, ..., x_k
- requires garbage collection infrastructure (scalar/pointer distinction, node layout) but not tied to a particular *GC*
- one visited bit per node (not shared with *GC*)
- several optimizations possible...

Note dependency on pure evaluation:
if the RHS e_i could have side effects, subst would generally have to traverse the whole heap

Implementation performance

- A trade-off between cyclic structure building cost and cost for data access
- Our choice: zero data access cost; c.f. C translation:
  - `[ x.s ]` = `x->s`
  - `[ case x of ... ]` = `switch (x->tag) { ... }`
  - `[ x arg ]` = `x->code(x, arg)`
- Cost for building cyclic data = cost of subst calls
- With optimizations: just one subst traversal per dependency graph cycle
- Note: the longer a cyclic structure lives, the cheaper any initial subst calls become

Related approaches

- Hirschowitz, Leroy & Wells (PPDP'03)
  - Allocate empty top nodes (must know size statically)
  - Copy actual results to pre-allocated nodes
  - Requires separate well-foundedness analysis
- Boudol & Zimmer (FICS'02)
  - Always access recursive values through an indirection
  - Bind to exception initially, update final value in one step
- Syme (Electronic Notes in Theoretical CS, 2006)
  - delay RHS exprs, force LHS vars where they appear
  - no direct cyclic data, but order independence

Conclusion

- A reduction semantics and an implementation technique for unrestricted (w.r.t. type & syntax) cbv recursion
- Simple, referentially transparent, extensible semantics
- Implementation uses illegal addresses & subst traversals, takes all cost at data construction (zero access cost)
- Moderate cost of subst depends on purity of RHS exprs
- Future directions:
  - Static detection of ill-definedness & non-termination
  - Relaxed dynamics: delayed selection...

A bigger example

Combinator parsers using applicative functors:

`accept :: [Char] -> P Char`	accept one char from given set
`return :: a -> P a`	succeed without consuming input
`($$) :: P (a->b) -> P a -> P b`	sequential parser composition
`($+) :: P a -> P a -> P a`	alternative parser composition

Example of use:

```
data Exp = EOp Var Op Exp | EVar
data Var = V Char
data Op = O Char
pExp = return EOp $$ pVar $$ pOp $$ pExp
     $+ return EVar $$ pVar
     $+ parens pExp
pVar = return V $$ accept ['a'..'z']
pOp  = return O $$ accept ['+','-','*','/']
```

Combinator implementation

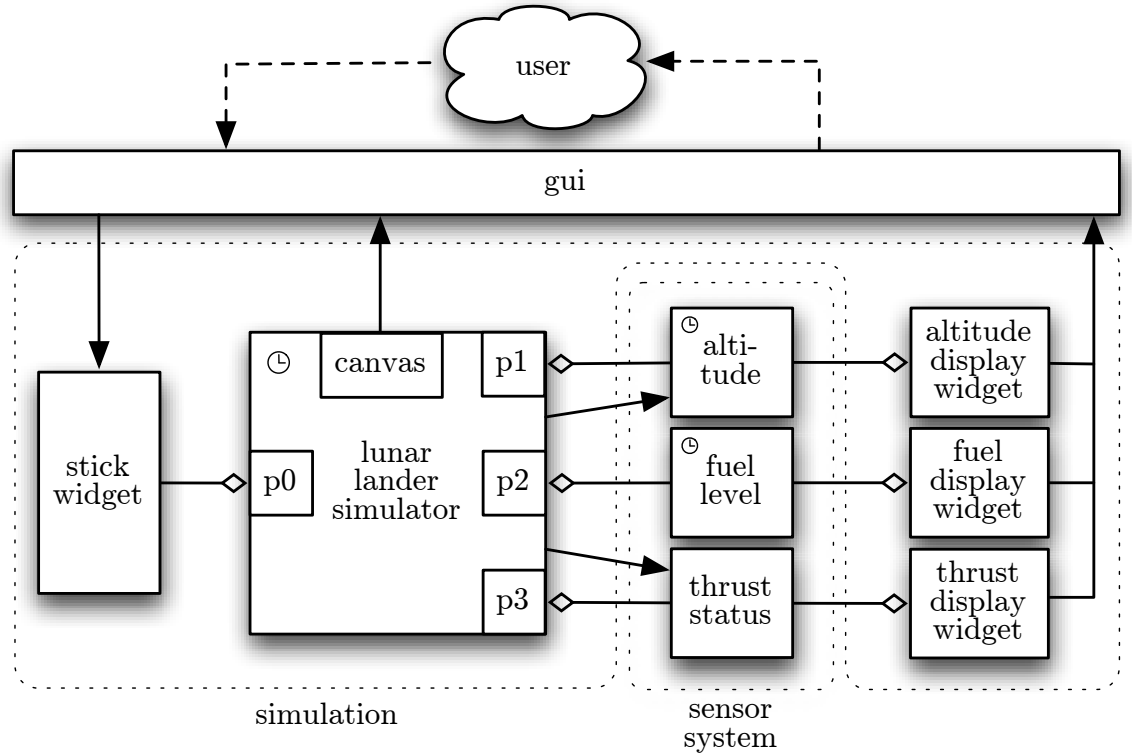
Self-optimizing during "startup" (Swierstra & Duponchel):

```
type P a    = (Maybe a,  [(Char, String -> (String, Maybe a))]

return a    = (Just a, [])
accept cs   = (Nothing, [(c,\s->(s,Just c)) | c <- cs ])
fp  $$  ap  =  (empty, nonempty) where
    empty  =  case fst fp of
                  Nothing -> Nothing
                  Just f -> case fst ap of
                                Nothing -> Nothing
                                Just a -> Just (f a)
    nonempty  =  combineSeq  fp  ap
p1  $+  p2   =  ...
```

Another example

A lunar lander simulator in Timber




```
game gui = class
      (simStart, stick, p1, p2, p3, d1, d2, d3) = new simulation gui echoI thrustI
      (sysStart, echoI, thrustI)                = new sensorSys p1 p2 p3 d1 d2 d3
      result (stick, action simStart; sysStart)
```