Extended recursive definitions in call-by-value languages

Tom Hirschowitz — ENS Lyon

Xavier Leroy — INRIA Rocquencourt

J. B. Wells — Heriot-Watt University

Recursive definitions

```
let rec x = e[x] in ...
```

```
let rec x = e[x,y] and y = f[x,y] in ...
```

Under call-by-name or lazy evaluation:
r.h.s. are arbitrary expressions
evaluation is equivalent to on-demand unrolling `e[e[e[...]]]`.

Under call-by-value:
r.h.s. are traditionally restricted to be syntactic values $\lambda x. e$

Extended call-by-value recursive definitions

Can we do call-by-value evaluation of more general recursive definitions? Example:

```
let F = λg. λx. ... g ...

let rec g = F g
```

This should evaluate like `let rec g = λx. ... g ...`

Applications: some object encodings; recursive modules; CBV mixin modules.

Challenge: evaluate r.h.s. exactly once and in a predictable order. (No lazy evaluation.)

Application 1: recursive ML modules

```
module rec A =
  struct type t = Leaf of int | Node of ASet.t
         let compare t1 t2 = ...
  end

and ASet = Set.Make(A)
```

Most practical examples involve a functor application in a r.h.s.

After type erasure, compiles to a `let rec` involving function applications in r.h.s.

Application 2: call-by-value mixin modules

(Hirschowitz, Leroy, 2002-2004)

Mixin modules = modules with holes (deferred components).

```
mixin M = mix
    import y : int -> int
    export y = λx. ...y...
    export z = y 0
end

module A = close(M)
```

```
let M = {

  y = λy.λx. ...y...
  z = λy. y 0
}

let A =
  let rec y = M.y y
  and z = M.z y
  in { y = y; z = z }
```

Compilation schemes for extended recursion

```
let rec x = e[x,y] and y = f[x,y] in ...
```

Scheme's `letrec`: update variables (via references)

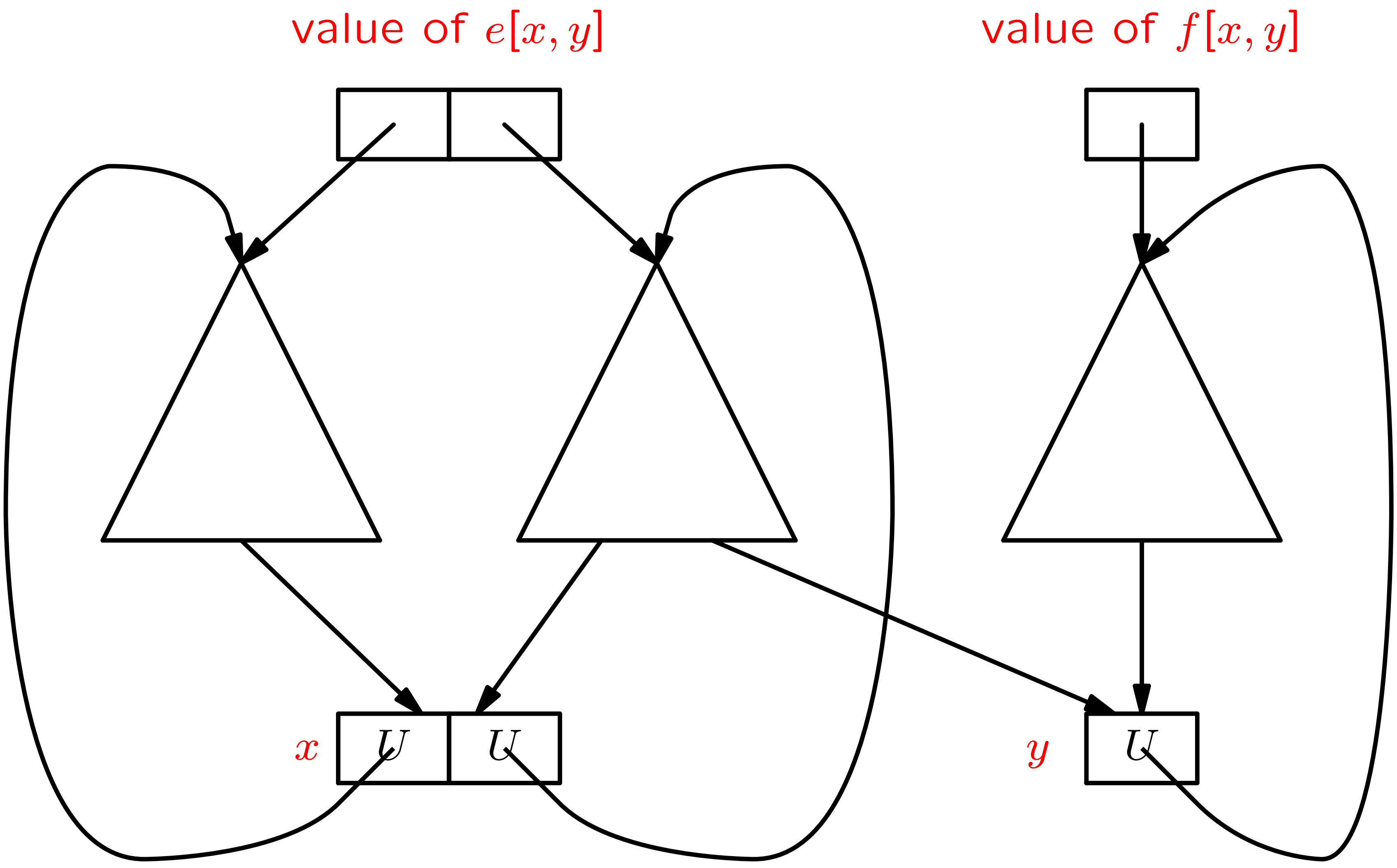
```
let rx = ref Undef and ry = ref Undef in
let x = e[!rx,!ry] and y = f[!rx,!ry] in
rx := x; ry := y;
...
```

The "in-place update trick" (Cousineau et al):
update memory blocks in place

```
let x = newblock(2, Undef) and y = newblock(1, Undef) in
update(x, e[x, y]);
update(y, f[x, y]);
...
```

(Need to guess in advance the memory size of the values of the r.h.s.)

In-place update in action



A store-less reduction semantics for in-place update

Capture the expressiveness and limitations of the in-place update scheme without using a store.

A strategy imposed on Ariola & Blom's five fundamental equivalences for recursive definitions.

Use evaluation contexts to impose a deterministic, CBV evaluation order.

Add size annotations on definitions and an abstract notion of value size.

The semantics get stuck exactly when the in-place update scheme would lead to using the undefined value.

Ariola and Blom's equational theory

Lifting:

$$C[\texttt{letrec}\ b\ \texttt{in}\ e]$$
$$\approx\ \texttt{letrec}\ b\ \texttt{in}\ C[e]$$

Internal merging:

$$\texttt{letrec}\ b_1, x = (\texttt{letrec}\ b_2\ \texttt{in}\ e), b_3\ \texttt{in}\ f$$
$$\approx\ \texttt{letrec}\ b_1, b_2, x = e, b_3\ \texttt{in}\ f$$

External merging:

$$\texttt{letrec}\ b_1\ \texttt{in}\ \texttt{letrec}\ b_2\ \texttt{in}\ f$$
$$\approx\ \texttt{letrec}\ b_1, b_2\ \texttt{in}\ f$$

External substitution:

$$\texttt{letrec}\ b_1, x = e, b_2\ \texttt{in}\ C[x]$$
$$\approx\ \texttt{letrec}\ b_1, x = e, b_2\ \texttt{in}\ C[e]$$

Internal substitution:

$$\texttt{letrec}\ b_1, x = e, b_2, y = C[x], b_3\ \texttt{in}\ f$$
$$\approx\ \texttt{letrec}\ b_1, x = e, b_2, y = C[e], b_3\ \texttt{in}\ f$$

Formalization of a compilation scheme

The in-place update scheme = a compilation scheme from extended `letrec` to an imperative language with updateable blocks.

Prove the correctness of this compilation scheme: if e reduces to a value, or reduces infinitely, or gets stuck, then so does its translation.

(The proof is not quite a simulation argument and is surprisingly difficult.)

Static typing of extended recursive definitions

Extended recursive definitions can get stuck at run-time:

```
let rec x = x + 1          let rec x = f 0 and f = λy.1
```

Use a type system to guarantee that this does not happen.

Boudol's type system: annotated function types $\xrightarrow{1}$ (non-strict function) and $\xrightarrow{0}$ (possibly strict).

`let rec x = f x` allowed iff `f` : $\tau_1 \xrightarrow{1} \tau_2$.

Hirschowitz's generalization: function types $\xrightarrow{n}$ where n is the "delay" between passing the parameter and actually using it.

Dreyer's effect system: track the "effect" of using the value of a recursively-defined identifier.

Future work

These type systems are expressive but too complex to be exposed to the programmer.
$\rightarrow$ For recursive modules, find coarser, simpler type system.