Lazy Modules

Keiko Nakata
Institute of Cybernetics, Tallinn

WG2.8 meeting, 2010

Constrained lazy initialization for modules

Plan of my talk

- Lazy initialization in practice
- Constrained laziness

  Or, hybrid strategies between call-by-value and call-by-need
- Models for several strategies with varying laziness
  - Compilation scheme from the source syntax to the target languages
  - Target languages, variations of Ariola and Felleisen's cyclic call-by-need calculi with state in the style of Hieb and Felleisen.

Lazy initialization

Traditionally ML modules are initialized in call-by-value.

- Predictable initialization order is good for having arbitrary side-effects.
- Theoretically all modules, including libraries, are initialized at startup time.

Practice has shown lazy initialization may be interesting.

- Dynamically linked shared libraries, plugins
- Lazy file initialization in F#
- Lazy class initialization in Java and F#
- Alice ML
- OSGi, NetBeans (through bundles)
- Eclipse

Why not lazy initialization for recursive modules?

But how much laziness we want?

All these available implementations combine call-by-value and laziness in the presence of side-effects.

Controlled uses of lazy initialization for recursion

Syme proposed *initialization graphs*, by introducing lazy initialization in a controlled way, to allow for more recursive initialization patterns in a ML-like language.

$$\begin{array}{l}
\mathit{let\ rec}\ x_0 = a_0\ \ldots\ x_n = a_n\ \mathit{in}\ a \\
\Rightarrow \\
\mathit{let}\ (x_0, .., x_n) = \\
\quad \mathit{let\ rec}\ x_0 =\ \mathit{lazy}\ a'_0\ \ldots\ x_n =\ \mathit{lazy}\ a'_n \\
\quad \mathit{in}\ (\mathit{force}\ x_0; ...; \mathit{force}\ x_n) \\
\mathit{in}\ a
\end{array}$$

Support for relaxed recursive initialization patterns is important for interfacing with external OO libraries, e.g., GUI APIs.

Picklers API

```
type Channel   (* e.g. file stream *)
type α Mrshl
val marshal: α Mrshl → α ∗ Channel → unit
val unmarshal: α Mrshl → Channel →α
val optionMarsh: α Mrshl →(option α) Mrshl
val pairMrshl: α Mrshl ∗ β Mrshl → (α ∗ β) Mrshl
val listMrshl: α Mrshl → (α list) Mrshl
val innerMrshl: (α → β) ∗ (β → α) → α Mrshl → β Mrshl
val intMrshl : int Mrshl
val stringMrshl: string Mrshl
val delayMrshl: (unit → α Mrshl) → α Mrshl

//  let delayMrshl p =
//  { marshal = (λ x → (p ()).marshal x);
//     unmarshal = (λ y → (p ()).unmarshal y)}
```

Pickler for binary trees

```
type t = option (t ∗ int ∗ t)
let mrshl =
  optionMrshl (pairMrshl mrshl (pairMrshl intMrshl mrshl))
```

Cannot evaluate in call-by-value.

Pickler for binary trees with initialization graphs

```
type t = option (t ∗ int ∗ t)
let mrshl =
  optionMrshl (pairMrshl mrshl0 (pairMrshl intMrshl mrshl0))
and mrshl0 = delayMrshl(λ().mrshl)
```

```
implemented as
let (mrshl, mrshl0) =
  let rec mrshl =
    lazy (optionMrshl (pairMrshl mrshl0 (pairMrshl intMrshl mrshl0)))
  and mrshl0 = lazy (delayMrshl(λ().mrshl))
in (force mrshl, force mrsh0)
```

where the library provides

```
val delayMrshl: (unit → α Mrshl) → α Mrshl
```

```
let delayMrshl p =
{ marshal = (λ x → (p ()).marshal x);
  unmarshal = (λ y → (p ()).unmarshal y)}
```

MakeSet functor with picklers

```
module Set =
functor (Ord: sig
  type t val compare: t → t → bool val mrshl : t Mrshl end) →
struct
  type elt = Ord.t
  type t = option (t ∗ elt ∗ t)
  ...
  let mrshl =
    optionMrshl (pairMrshl mrshl0 (pairMrshl Ord.mrshl mrshl0))
  and mrshl0 = delayMrshl (λ().mrshl)
end
```

Picklers for Folder and Folders

```
module Folder =
struct
  type file = int ∗ string
  let fileMrshl = pairMrshl (intMrshl, stringMrshl)
  let filesMrshl = listMrshl filMrshl
  type t = { files: file list; subfldrs: Folders.t }
  let mkFldr x y = { files = x; subfldrs = y }
  let destFldr f = (f.files, f.subfldrs)
  let fldrInnerMrshl(f, g) =
    innerMrshl (mkFldr, destFldr) (pairMrshl(f,g))
  let mrshl =
    fldrInnerMrshl(filesMrshl, delayMrshl(λ(). Folders.mrshl))
  let initFldr = unmarshal mrshl “/home/template/initfldr.txt”
end

and Folders = Set(Folder)
```

Expressivity, predictability, simplicity, stability

Can we find a happy compromise between call-by-value and call-by-need?

- interesting recursive initialization patterns, i.e., expressivity
- predictable initialization order
    - when side effects are produced
    - in which order side effects are produced
- simple implementation
- stability of success of the initialization
  (ongoing work towards formal results)

Model

Model for investigating the design space.

- target languages,
  variations of the cyclic call-by-need calculus equipped with array primitives
- compilation scheme
  from the source syntax into target languages

Five strategies with different degrees of laziness are examined, inspired by strategies of existing languages (Moscow ML, F#, Java).

Inclusion between strategies in a pure setting.

Call-by-need strategy à la F#

1. Evaluation of a module is delayed until the module is accessed for the first time. In particular, a functor argument is evaluated lazily when the argument is used.
2. All the members of a structure, excluding those of substructures, are evaluated at once from-top-to-bottom order on the first access to the structure
3. A member of a structure is only accessible after all the core field of the structure have been evaluated.

Examples

Call-by-need strategy à la F#

$$\begin{aligned}\{\ & F = \Lambda X.\{\ c = \mathit{print}\ \text{“}\mathit{bye}\text{”};\ \};\\ & M = F(\{\ c = \mathit{print}\ \text{“}\mathit{hello}\text{”};\ \});\\ & c = M.c;\ \}\end{aligned}$$

prints “*bye*”.

$$\begin{aligned}\{\ & F = \Lambda X.\{\ c_1 = X.c;\ \ c_2 = \mathit{print}\ \text{“}\mathit{bye}\text{”};\ \};\\ & M = F(\{\ c = \mathit{print}\ \text{“}\mathit{hello}\text{”};\ \});\\ & c = M.c_2;\ \}\end{aligned}$$

prints “*hello bye*”.

Target language λ_{need} for call-by-need modules

Expr.	a	$::=$	$x \mid \lambda x.a \mid a_1\ a_2 \mid (a, \ldots) \mid a.n$
		$\mid$	let rec d in $a \mid \{r, \ldots\} \mid a!n \mid \langle x \rangle$
References	r	$::=$	$x \mid \lambda\_.x$
Dereferences	$\sharp x$	$::=$	$x \mid \langle x \rangle !n$
Values	v	$::=$	$\lambda x.a \mid (v, \ldots) \mid \langle x \rangle \mid \{r, \ldots\}$
Definitions	d	$::=$	$x = a$ and $\ldots$
Configurations	c	$::=$	$d \vdash a$
Lift contexts	L	$::=$	$[]\ a \mid (\ldots, v, [], a, \ldots) \mid [].n \mid []!n$
Nested lift cnxt.	N	$::=$	$[] \mid L[N]$
Lazy evalu. cnxt	K	$::=$	$d \vdash N$
		$\mid$	$x' = N$ and $d^*[x, x']$ and $d \vdash N'[\sharp x]$
Dependencies	$d[x, x']$	$::=$	$x = N[\sharp x']$
		$\mid$	$d[x, x'']$ and $x'' = N[\sharp x']$

Reduction rules for λ_{need}

$$
\begin{array}{lrcl}
\beta_{need}: & (\lambda x.a)\ a' & \underset{\text{need}}{\longrightarrow} & \text{let rec } x = a' \text{ in } a \\
prj: & (\ldots, v_n, \ldots).n & \underset{\text{need}}{\longrightarrow} & v_n \\
lift: & L[\text{let rec } d \text{ in } a] & \underset{\text{need}}{\longrightarrow} & \text{let rec } d \text{ in } L[a] \\
cxt: & K[a] & \underset{\text{need}}{\longmapsto} & K[a'] \ \ if\ a \underset{\text{need}}{\longrightarrow} a' \\
deref: & K[x] & \underset{\text{need}}{\longmapsto} & K[v] \ \ if\ x = v \in K \\
arr_{need}: & K[\langle x \rangle ! n] & \underset{\text{need}}{\longmapsto} & K[(r, \ldots).n] \\
 & & & if\ x = \{r, \ldots\} \in K \\
alloc: & d \vdash \text{let rec } d' \text{ in } a & \underset{\text{need}}{\longmapsto} & d \text{ and } d' \vdash a
\end{array}
$$

$$
\begin{array}{ll}
alloc\text{-}env: & x' = (\text{let rec } d \text{ in } a) \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x] \\
 & \underset{\text{need}}{\longmapsto} \ d \text{ and } x' = a \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x]
\end{array}
$$

$$
\begin{array}{llcl}
acc: & x = a & \in & d \vdash N \ \ if\ x = a \in d \\
acc\text{-}env: & x = a & \in & x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] \\
 & & & if\ x = a \in d
\end{array}
$$

Example of λ_{need}reductions

$$
\begin{array}{lll}
 & \vdash \text{let rec } x = (\lambda y.y)\ (\lambda y.y) \text{ in } x & \\
\underset{\text{need}}{\longmapsto} & x = (\lambda y.y)\ (\lambda y.y) \vdash x & \text{by } \mathit{alloc} \\
\underset{\text{need}}{\longmapsto} & x = (\text{let rec } y = \lambda y.y \text{ in } y) \vdash x & \text{by } \beta_{need} \\
\underset{\text{need}}{\longmapsto} & y = \lambda y.y \text{ and } x = y \vdash x & \text{by } \mathit{alloc\text{-}env} \\
\underset{\text{need}}{\longmapsto} & y = \lambda y.y \text{ and } x = \lambda y'.y' \vdash x & \text{by } \mathit{deref} \\
\underset{\text{need}}{\longmapsto} & y = \lambda y.y \text{ and } x = \lambda y'.y' \vdash \lambda y''.y'' & \text{by } \mathit{deref}
\end{array}
$$

Example of λ_{need}reductions

$$
\begin{array}{lll}
 & \vdash \text{let rec } x = (\lambda y.\lambda y'.y)\ x \text{ in } x\ (\lambda x'.x') & \\
\underset{\text{need}}{\longmapsto} & x = (\lambda y.\lambda y'.y)\ x \vdash x\ (\lambda x'.x') & \text{by } \mathit{alloc} \\
\underset{\text{need}}{\longmapsto} & x = (\text{let rec } y = x \text{ in } \lambda y'.y) \vdash x\ (\lambda x'.x') & \text{by } \beta_{need} \\
\underset{\text{need}}{\longmapsto} & y = x \text{ and } x = \lambda y'.y \vdash x\ (\lambda x'.x') & \text{by } \mathit{alloc\text{-}env} \\
\underset{\text{need}}{\longmapsto} & y = x \text{ and } x = \lambda y'.y \vdash (\lambda y_1.y)\ (\lambda x'.x') & \text{by } \mathit{deref} \\
\underset{\text{need}}{\longmapsto} & y = x \text{ and } x = \lambda y'.y \vdash \text{let rec } y_1 = \lambda x'.x' \text{ in } y & \text{by } \beta_{need} \\
\underset{\text{need}}{\longmapsto} & y = x \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash y & \text{by } \mathit{alloc} \\
\underset{\text{need}}{\longmapsto} & y = \lambda y_2.y \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash y & \text{by } \mathit{deref} \\
\underset{\text{need}}{\longmapsto} & y = \lambda y_2.y \text{ and } x = \lambda y'.y \text{ and } y_1 = \lambda x'.x' \vdash \lambda y_3.y & \text{by } \mathit{deref}
\end{array}
$$

Target language λ_{need} for call-by-need modules (cont.)

$$
\begin{array}{llcl}
\textit{Expr.} & a & ::= & x \mid \lambda x.a \mid a_1\ a_2 \mid (a, \ldots) \mid a.n \\
 & & \mid & \text{let rec } d \text{ in } a \mid \{r, \ldots\} \mid a!n \mid \langle x \rangle \\
\textit{References} & r & ::= & x \mid \lambda\_.x \\
\textit{Dereferences} & \sharp x & ::= & x \mid \langle x \rangle !n \\
\textit{Values} & v & ::= & \lambda x.a \mid (v, \ldots) \mid \langle x \rangle \mid \{r, \ldots\} \\
\textit{Definitions} & d & ::= & x = a \text{ and } \ldots \\
\textit{Lift contexts} & L & ::= & [\,]\ a \mid (\ldots, v, [\,], a, \ldots) \mid [\,].n \mid [\,]!n \\
\textit{Nested lift cnxt.} & N & ::= & [\,] \mid L[N] \\
\textit{Lazy evalu. cnxt} & K & ::= & d \vdash N \\
 & & \mid & x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] \\
\textit{Dependencies} & d[x, x'] & ::= & x = N[\sharp x'] \\
 & & \mid & d[x, x''] \text{ and } x'' = N[\sharp x']
\end{array}
$$

Reduction rules for λ_{need}

$$\begin{array}{lrcl}
\beta_{need} : & (\lambda x.a)\ a' & \underset{\text{need}}{\longrightarrow} & \text{let rec } x = a' \text{ in } a \\
prj : & (\ldots, v_n, \ldots).n & \underset{\text{need}}{\longrightarrow} & v_n \\
lift : & L[\text{let rec } d \text{ in } a] & \underset{\text{need}}{\longrightarrow} & \text{let rec } d \text{ in } L[a] \\
cxt : & K[a] & \underset{\text{need}}{\longmapsto} & K[a'] \ \ if\ a \underset{\text{need}}{\longrightarrow} a' \\
deref : & K[x] & \underset{\text{need}}{\longmapsto} & K[v] \ \ if\ x = v \in K \\
arr_{need} : & K[\langle x \rangle !n] & \underset{\text{need}}{\longmapsto} & K[(r, \ldots).n] \\
 & & & if\ x = \{r, \ldots\} \in K \\
alloc : & d \vdash \text{let rec } d' \text{ in } a & \underset{\text{need}}{\longmapsto} & d \text{ and } d' \vdash a
\end{array}$$

$$\begin{array}{ll}
alloc\text{-}env : & x' = (\text{let rec } d \text{ in } a) \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x] \\
 & \underset{\text{need}}{\longmapsto} \ d \text{ and } x' = a \text{ and } d^*[x, x'] \text{ and } d' \vdash N[\sharp x]
\end{array}$$

$$\begin{array}{llcl}
acc : & x = a & \in & d \vdash N \ \ if\ x = a \in d \\
acc\text{-}env : & x = a & \in & x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] \\
 & & & if\ x = a \in d
\end{array}$$

Reduction rules for λ_{need}

$$
\begin{array}{lrcl}
\beta_{need} : & (\lambda x.a)\ a' & \underset{\mathrm{need}}{\longrightarrow} & \mathrm{let\ rec}\ x = a'\ \mathrm{in}\ a \\
prj : & (\ldots, v_n, \ldots).n & \underset{\mathrm{need}}{\longrightarrow} & v_n \\
lift : & L[\mathrm{let\ rec}\ d\ \mathrm{in}\ a] & \underset{\mathrm{need}}{\longrightarrow} & \mathrm{let\ rec}\ d\ \mathrm{in}\ L[a] \\
cxt : & K[a] & \underset{\mathrm{need}}{\longmapsto} & K[a']\ \ if\ a \underset{\mathrm{need}}{\longrightarrow} a' \\
deref : & K[x] & \underset{\mathrm{need}}{\longmapsto} & K[v]\ \ if\ x = v \in K \\
arr_{need} : & K[\langle x \rangle ! n] & \underset{\mathrm{need}}{\longmapsto} & K[(r, \ldots).n] \\
 & & & if\ x = \{r, \ldots\} \in K
\end{array}
$$

```
let get (a : ('a Lazy.t) array) n =
  for i = 0 to Array.length a - 1 do Lazy.force a.(i) done;
  Lazy.force a.(n)
```

λ_{need} with state

$$\begin{array}{llcl}
\textit{Expr.} & a & ::= & x \mid \lambda x.a \mid a_1\ a_2 \mid (a, \ldots) \mid a.n \\
 & & \mid & \text{let rec } d \text{ in } a \mid \{r, \ldots\} \mid a!n \mid \langle x \rangle \\
 & & \mid & \text{set! } x\ a \\
\textit{References} & r & ::= & x \mid \lambda\_.x \\
\textit{Dereferences} & \sharp x & ::= & x \mid \langle x \rangle !n \\
\textit{Values} & v & ::= & \lambda x.a \mid (v, \ldots) \mid \langle x \rangle \mid \{r, \ldots\} \\
\textit{Definitions} & d & ::= & x = a \text{ and } \ldots \\
\textit{Lift contexts} & L & ::= & []\ a \mid (\ldots, v, [], a, \ldots) \mid [].n \mid []!n \\
 & & \mid & \text{set! } x\ [] \\
\textit{Nested lift cnxt.} & N & ::= & [] \mid L[N] \\
\textit{Lazy evalu. cnxt} & K & ::= & d \vdash N \\
 & & \mid & x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x] \\
\textit{Dependencies} & d[x, x'] & ::= & x = N[\sharp x'] \\
 & & \mid & d[x, x''] \text{ and } x'' = N[\sharp x']
\end{array}$$

Reduction rules for set! in λ_{need}

$$\textit{set}: \quad x = a \text{ and } d \vdash N[\text{set! } x\ v] \underset{\text{need}}{\longmapsto} x = v \text{ and } d \vdash N[v]$$

$$\textit{set-env}: \quad x'' = a \text{ and } x' = N[\text{set! } x''\ v] \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x]$$
$$\underset{\text{need}}{\longmapsto} x'' = v \text{ and } x' = N[v] \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x]$$

Syntax for *Osan*

Module expressions	E	::=	$\{(X)\ f\} \mid p \mid \Lambda X.E \mid E_1(E_2)$
Definitions	f	::=	$\epsilon \mid M = E; f \mid c = e; f$
Module paths	p	::=	$X \mid M \mid p.n$
Core expressions	e	::=	$c \mid p.n \mid \ldots$

Example

Syntax for *Osan*

```
{ (X)
  Tree = { (X_t)
    add = λ t. match t with (i, f) => i + X.Forest.add f; };
  Forest = { (X_f)
    add = λ f. match f with [] => 0 | t :: f' => Tree.add t + X_f.add f'; };}
```

Translation from *Osan* to λ_{need}

$$
\begin{array}{llll}
\mathit{str}: & \mathit{Tr}_N(\{(X)\ f\})_\rho & = & \\
& \text{let rec } x = \langle x' \rangle \text{ and } x' = \mathit{TrFld}_N(f:\ \epsilon)_{\rho[X\mapsto x]} \text{ in } \langle x' \rangle & & \\
\mathit{mfld}: & \mathit{TrFld}_N(M = E; f:\ r, \ldots)_\rho & = & \\
& \text{let rec } x = \mathit{Tr}_N(E)_\rho \text{ in } \mathit{TrFld}_N(f:\ r, \ldots, \lambda\_.x)_{\rho[M\mapsto x]} & & \\
\mathit{cfld}: & \mathit{TrFld}_N(c = e; f:\ r, \ldots)_\rho & = & \\
& \text{let rec } x = \mathit{TrC}_N(e)_\rho \text{ in } \mathit{TrFld}_N(f:\ r, \ldots, x)_{\rho[c\mapsto x]} & & \\
\mathit{strbody}: & \mathit{TrFld}_N(\epsilon:\ r, \ldots)_\rho & = & \{r, \ldots\} \\
\mathit{vpath}: & \mathit{TrC}_N(p.n)_\rho & = & \mathit{Tr}_N(p)_\rho!n \\
\mathit{mpath}: & \mathit{Tr}_N(p.n)_\rho & = & (\mathit{Tr}_N(p)_\rho!n)\ I \\
\mathit{mvar}: & \mathit{Tr}_N(X)_\rho & = & \rho(X) \\
\mathit{funct}: & \mathit{Tr}_N(\Lambda X.E)_\rho & = & \lambda x.\mathit{Tr}_N(E)_{\rho[X\mapsto x]} \\
\mathit{app}: & \mathit{Tr}_N(E_1(E_2))_\rho & = & \mathit{Tr}_N(E_1)_\rho\ \mathit{Tr}_N(E_2)_\rho \\
\mathit{mname}: & \mathit{Tr}_N(M)_\rho & = & \rho(M) \\
\mathit{cname}: & \mathit{Tr}_N(c)_\rho & = & \rho(c)
\end{array}
$$

Example of compilation

$$\begin{array}{ll}\{ & M = \{\ c_1 = \mathit{print}\ \text{“}\mathit{good}\text{”};\ \ c_2 = \mathit{print}\ \text{“}\mathit{bye}\text{”};\ \};\\ & c_1 = \mathit{print}\ \text{“}\mathit{hello}\text{”};\\ & c_2 = M.c_1;\ \}\end{array}$$

```
let rec $x = \langle x' \rangle$
and $x' =$
  let rec $m =$
    let rec $x_1 = \langle x'_1 \rangle$
    and $x'_1 =$
      let rec $c'_1 = print$ “$good$” in  let rec $c'_2 = print$ “$bye$” in $\{c'_1, c'_2\}$ in
    $\langle x'_1 \rangle$ in
  let rec $c_1 = print$ “$hello$” in
  let rec $c_2 = m!1$ in
  $\{\lambda\_.m, c_1, c_2\}$ in
$x!3$
```

Assessment

Call-by-need

- △ interesting recursive initialization patterns, i.e., expressivity
- ✓ predictable initialization order
- ✓ simple implementation
- ✓ stability of success of the initialization (in a pure setting)

Assessment cont.

Call-by-need

- One may take fixpoints of functors.

$$
\begin{array}{l}
\{\ F = \Lambda Y.\{\ g = \mathit{fun\ if}\ i = 0\ \mathit{then\ true\ else}\ i = 1\ \mathit{then\ false} \\
\qquad\qquad\qquad\qquad \mathit{else}\ Y.g\ (i-1);\ \}; \\
\quad M = \{(X)\ \ M' = F(X.M');\ \};\ \}
\end{array}
$$

- Self variables are strict.

$$
\begin{array}{l}
\{\ F = \Lambda Y.\{\ g = \mathit{fun\ if}\ i = 0\ \mathit{then\ true\ else}\ i = 1\ \mathit{then\ false} \\
\qquad\qquad\qquad\qquad \mathit{else}\ Y.g\ (i-1); \\
\qquad\qquad c = g\ 2\ \}; \\
\quad M = \{(X)\ \ M' = F(X.M');\ \}; \\
\quad c = M.M'.c;\ \}
\end{array}
$$

Lazy-field strategy à la Java

Variations

We may allow a member of a structure to be accessed when it has been evaluated, but before evaluation of all the members of the structure is completed.

Target language λ_{lazy} for lazy-filed modules

Expr.	a	$::=$	$x \mid \lambda x.a \mid a_1\ a_2 \mid (a, \dots) \mid a.n$
		$\mid$	$\text{let rec } d \text{ in } a$
		$\mid$	$\{r, \dots\} \mid \{\!\{r, \dots\}\!\} \mid a!n \mid \langle x \rangle$
References	r	$::=$	$x \mid \lambda\_.x$
Dereferences	$\sharp x$	$::=$	$x \mid \langle x \rangle!n$
Values	v	$::=$	$\lambda x.a \mid (v, \dots) \mid \langle x \rangle \mid \{r, \dots\}$
		$\mid$	$\{\!\{r, \dots\}\!\}$
Definitions	d	$::=$	$x = a \text{ and } \dots$
Lift contexts	L	$::=$	$[]\ a \mid (\dots, v, [], a, \dots) \mid [].n \mid []!n$
Nested lift cnxt.	N	$::=$	$[] \mid L[N]$
Lazy evalu. cnxt	K	$::=$	$d \vdash N$
		$\mid$	$x' = N \text{ and } d^*[x, x'] \text{ and } d \vdash N'[\sharp x]$
Dependencies	$d[x, x']$	$::=$	$x = N[\sharp x']$
		$\mid$	$d[x, x''] \text{ and } x'' = N[\sharp x']$

Reduction rules for λ_{lazy}

init : $x = a$ and $d \vdash N[\langle x \rangle ! n] \underset{\text{lazy}}{\longmapsto} x = a'$ and $d \vdash N[(r, \ldots).n]$
where $a = \{\!\{ r, \ldots \}\!\}$ and $a' = \{ r, \ldots \}$

init-env : $x'' = a$ and $x' = N[\langle x'' \rangle ! n]$ and $d^{*}[x, x']$ and $d \vdash N'[\sharp x]$
$\underset{\text{lazy}}{\longmapsto} x'' = a'$ and $x' = N[(r, \ldots).n]$ and $d[x, x']$ and $d \vdash N'[\sharp x]$
where $a = \{\!\{ r, \ldots \}\!\}$ and $a' = \{ r, \ldots \}$

arr_{lazy} : $K[\langle x \rangle ! n] \underset{\text{lazy}}{\longmapsto} K[(r_1, \ldots, r_n).n]$
if $x = \{ r_1, \ldots, r_n, r_{n+1} \ldots \} \in K$

Assessment

Lazy-field

- ✓ interesting recursive initialization patterns, i.e., expressivity
- ✓ predictable initialization order
- ✓ simple implementation
- \- stability of success of the initialization

Assessment cont.

Lazy-field

$$
\begin{array}{l}
\{(X) \\
\quad M = \{\ c_1 = 1;\ \ c_2 = X.N.c_2\ \}; \\
\quad N = \{\ c_1 = M.c_1;\ \ c_2 = \ \ 2;\}; \}
\end{array}
$$

If *M* is forced first then the evaluation is successful, but if *N* is forced first then the evaluation fails due to unsound initialization.

Modest-field strategy

Variations

We may initialize members as much as necessary,
or initialize members from the top to the member accessed.

$$\mathit{arr}_{\mathit{modest}} : \quad K[\langle x \rangle ! n] \quad \underset{\text{modest}}{\longmapsto} \quad \begin{array}{l} K[(r_1, \ldots, r_n).n] \\ \mathit{if}\ x = \{r_1, \ldots, r_n, r_{n+1}, \ldots\} \in K \end{array}$$

Assessment

Modest-field

- ✓ interesting recursive initialization patterns, i.e., expressivity
- \- predictable initialization order
- ✓ simple implementation
- ✓ stability of success of the initialization in a pure setting

Assessment cont.

Modest-field

$$
\begin{array}{l}
\{\ M = \{(X) \\
\quad c_1 = \mathit{print}\ 1; \\
\quad M_1 = \{\ c_1 = \mathit{print}\ 2;\ \ c_2 = X.M_2.c;\ \ c_3 = \mathit{print}\ 3;\ \}; \\
\quad c_2 = \mathit{print}\ 4; \\
\quad M_2 = \{\ c = \mathit{print}\ 5;\ \}; \\
\quad c_3 = \mathit{print}\ 6;\ \}; \\
\ \ c = M.M_1.c_3;\ \}
\end{array}
$$

“1 4 6 2 5 3” is printed in the call-by-need and lazy-field strategies.

“1 2 4 5 3” is printed in the modest-field strategy.

Some technical results

Proposition

Call-by-value $\subseteq$ *Call-by-need* $\subseteq$ *Lazy-field* $\subseteq$ *Modest-field* $\subseteq$ *Fully-lazy*

Proof.

By going through natural semantics. ☐

Ongoing work

- Introduction of bundles.
  - Initialize bundles with the call-by-need strategy, but modules with the modest-field strategy.
- A framework to talk about stability of success of the initialization.