A taste of soft-linear logic for staged computation

Hubert Godfroy   Jean-Yves Marion

30 septembre 2015

Plan

Self-modification

Self-modifying program have the ability to

- generate others programs at runtime
  - Ex. just-in-time compilers
- execute data
  - Ex. `exec` function in python
- Match data against patterns
  - Ex. parser
- inspect its own code
  - Ex. integrated integrity checkers

Staged computation

- Paradigm where computation is split in stages
- Partial evaluation
- Run-time code generation
- Community uses tools like modal or temporal logic (MetaML)

R. Davies and F. Pfenning.
A modal analysis of staged computation.
Principles of programming languages, 1996.

R. Davies.
A temporal-logic approach to binding-time analysis.
Logic in Computer Science, pages 184–195, Jul 1996.

Modal Logic

- $\forall t, \langle t \rangle$ is in NF (data).

Modal Logic

- $\forall t, \langle t \rangle$ is in NF (data).
- Meta-binder: $\text{let}\langle x \rangle = t' \text{ in } t$
- Meta-redex: $(\text{let}\langle x \rangle = \langle t' \rangle \text{ in } t) \rightarrow t[x/t']$

Modal Logic

- $\forall t, \langle t \rangle$ is in NF (data).
- Meta-binder: $\mathsf{let}\langle x \rangle = t'$ in t
- Meta-redex: $(\mathsf{let}\langle x \rangle = \langle t' \rangle \text{ in } t) \to t[x/t']$
- Typing rules:

$$\textsc{Box}\ \frac{\langle \Delta \rangle \vdash t : A}{\langle \Delta \rangle, \Gamma \vdash \langle t \rangle : \Box A} \qquad \textsc{GVar}\ \frac{\langle x \rangle : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Let}\ \frac{\Gamma \vdash t' : \Box A \qquad \langle x \rangle : A, \Gamma \vdash t : B}{\Gamma \vdash \mathsf{let}\langle x \rangle = t' \text{ in } t : B}$$

where $\Gamma = x_i : A_i$, then $\langle \Gamma \rangle \overset{\text{def}}{=} \langle x_i \rangle : \Box A_i$.

Linear Logic

- $\forall t, \langle t \rangle$ is in NF (data).
- Meta-binder: $\mathsf{let}\langle x \rangle = t'$ in t
- Meta-redex: $(\mathsf{let}\langle x \rangle = \langle t' \rangle \text{ in } t) \to t[x/t']$
- Typing rules:

$$\textsc{Promotion}\ \frac{\langle \Delta \rangle \vdash t : A}{\langle \Delta \rangle, \Gamma \vdash \langle t \rangle : !A} \qquad \textsc{Dereliction}\ \frac{\langle x \rangle : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Let}\ \frac{\Gamma \vdash t' : !A \qquad \langle x \rangle : A, \Gamma \vdash t : B}{\Gamma \vdash \mathsf{let}\langle x \rangle = t' \text{ in } t : B}$$

where $\Gamma = x_i : A_i$, then $\langle \Gamma \rangle \stackrel{\text{def}}{=} \langle x_i \rangle : !A_i$.

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle t\rangle) \to \langle t[x/t']\rangle$$

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle t\rangle) \to \langle t[x/t']\rangle$$

Use this derivation (for $t = x$):

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \langle t'\rangle : !A} \qquad \dfrac{\overline{\Gamma', \langle x\rangle : !A \vdash x : A}\ \textsc{Dereliction}}{\Gamma, \langle x\rangle : !A \vdash \langle x\rangle : !A}\ \textsc{Promotion}}{\Gamma \vdash \mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle x\rangle : !A}\ \textsc{Abstraction}$$

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle t\rangle) \to \langle t[x/t']\rangle$$

  Use this derivation (for $t = x$):

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \langle t'\rangle : !A} \qquad \dfrac{\overline{\Gamma', \langle x\rangle : !A \vdash x : A}\ \textsc{Dereliction}}{\Gamma, \langle x\rangle : !A \vdash \langle x\rangle : !A}\ \textsc{Promotion}}{\Gamma \vdash \mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle x\rangle : !A}\ \textsc{Abstraction}$$

- Execution:

$$\mathsf{run} \overset{\text{def}}{=} \lambda x.\, \mathsf{let}\langle y\rangle = x \text{ in } y$$

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle t\rangle) \to \langle t[x/t']\rangle$$

Use this derivation (for $t = x$):

$$\textsc{Abstraction}\ \dfrac{\dfrac{\dots}{\Gamma \vdash \langle t'\rangle : !A} \qquad \dfrac{\overline{\Gamma', \langle x\rangle : !A \vdash x : A}\ \textsc{Dereliction}}{\Gamma, \langle x\rangle : !A \vdash \langle x\rangle : !A}\ \textsc{Promotion}}{\Gamma \vdash \mathsf{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle x\rangle : !A}$$

- Execution:

$$\mathsf{run} \overset{\text{def}}{=} \lambda x.\, \mathsf{let}\langle y\rangle = x \text{ in } y$$

Use this derivation:

$$\textsc{Abstraction}\ \dfrac{\textsc{Let}\ \dfrac{\textsc{Dereliction}\ \dfrac{\textsc{Var}\ \overline{\Gamma, y : A \vdash y : A}}{\Gamma, \langle y\rangle : !A \vdash y : A}}{\Gamma, x : !A \vdash \mathsf{let}\langle y\rangle = x \text{ in } y : A}}{\Gamma \vdash \lambda x.\, \mathsf{let}\langle y\rangle = x \text{ in } y : !A \to A}$$

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \ \mathsf{in}\ \langle t\rangle) \to \langle t[x/t']\rangle$$

Use this derivation (for $t = x$):

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \langle t'\rangle : !A} \qquad \dfrac{\dfrac{}{\Gamma', \langle x\rangle : !A \vdash x : A}\ \textsc{Dereliction}}{\Gamma, \langle x\rangle : !A \vdash \langle x\rangle : !A}\ \textsc{Promotion}}{\Gamma \vdash \mathsf{let}\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ \langle x\rangle : !A}\ \textsc{Abstraction}$$

- Execution:

$$\mathsf{run} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y$$

Use this derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : A \vdash y : A}\ \textsc{Var}}{\Gamma, \langle y\rangle : !A \vdash y : A}\ \textsc{Dereliction}}{\Gamma, x : !A \vdash \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y : A}\ \textsc{Let}}{\Gamma \vdash \lambda x.\, \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y : !A \to A}\ \textsc{Abstraction}$$

$\Rightarrow$ Both use Dereliction.

Exemple

- Writing:

$$(\mathsf{let}\langle x\rangle = \langle t'\rangle \ \mathsf{in}\ \langle t\rangle) \to \langle t[x/t']\rangle$$

Use this derivation (for $t = x$):

$$\textsc{Abstraction}\ \dfrac{\dfrac{\dots}{\Gamma \vdash \langle t'\rangle : !A} \qquad \dfrac{\dfrac{}{\Gamma', \langle x\rangle : !A \vdash x : A}\ \textsc{Dereliction}}{\Gamma, \langle x\rangle : !A \vdash \langle x\rangle : !A}\ \textsc{Promotion}}{\Gamma \vdash \mathsf{let}\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ \langle x\rangle : !A}$$

- Execution:

$$\mathsf{run} \stackrel{\text{def}}{=} \lambda x.\,\mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y$$

Use this derivation:

$$\textsc{Abstraction}\ \dfrac{\textsc{Let}\ \dfrac{\textsc{Dereliction}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma, y : A \vdash y : A}}{\Gamma, \langle y\rangle : !A \vdash y : A}}{\Gamma, x : !A \vdash \mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y : A}}{\Gamma \vdash \lambda x.\,\mathsf{let}\langle y\rangle = x\ \mathsf{in}\ y : !A \to A}$$

$\Rightarrow$ Both use DERELICTION.

Question

What is the logical meaning of DERELICTION in this system?

Plan

Soft promotion

DERELICTION after PROMOTION is used for writing, contrary to alone DERELICTION used to execute data.

Soft promotion

Dereliction after Promotion is used for writing, contrary to alone Dereliction used to execute data.

Idea

Compose Dereliction and Promotion!

$$\frac{\Gamma \vdash \mathrm{t} : A}{\langle\Gamma\rangle \vdash \langle\mathrm{t}\rangle : !A}\ \textsc{SoftPromotion}$$

Soft promotion

DERELICTION after PROMOTION is used for writing, contrary to alone DERELICTION used to execute data.

Idea

Compose DERELICTION and PROMOTION!

$$\frac{\Gamma \vdash t : A}{\langle \Gamma \rangle \vdash \langle t \rangle : !A}\ \textsc{SoftPromotion}$$

Consequences

- No more DERELICTION rule for writing
- DERELICTION used only for execution of data
- Ex.:

$$\textsc{Abstraction}\ \frac{\dfrac{\cdots}{\Gamma \vdash \langle t' \rangle : !A} \qquad \dfrac{\overline{\Gamma', x : A \vdash x : A}\ \textsc{Var}}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A}\ \textsc{SoftPromotion}}{\Gamma \vdash \mathsf{let}\langle x \rangle = \langle t' \rangle\ \mathsf{in}\ \langle x \rangle : !A}$$

Language behavior

$$\frac{}{\Gamma, x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \textsc{Abstraction} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B}\ \textsc{Application}$$

$$\frac{\Gamma \vdash t' : !A \qquad \Gamma, \langle x \rangle : !A \vdash t : B}{\Gamma \vdash \mathsf{let}\langle x \rangle = t'\ \mathsf{in}\ t : B}\ \textsc{Let} \qquad \frac{x_i : A_i \vdash t : B}{\Gamma, \langle x_i \rangle : !A_i \vdash \langle t \rangle : !B}\ \textsc{SoftPromotion} \qquad \frac{}{\Gamma, \langle x \rangle : !A \vdash x : A}\ \textsc{Dereliction}$$

Abilities

Properties inherited from languages inspired by modal logic.

- program generation (plug data into another)
- execution

Properties

- Language is confluent
- Type system has subject reduction property

Language behavior

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \textsc{Application}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t' : !A \qquad \Gamma, \langle x \rangle : !A \vdash t : B}{\Gamma \vdash \mathsf{let} \langle x \rangle = t'\ \mathsf{in}\ t : B} \qquad \textsc{SoftPromotion}\ \frac{x_i : A_i \vdash t : B}{\Gamma, \langle x_i \rangle : !A_i \vdash \langle t \rangle : !B} \qquad \textsc{Dereliction}\ \frac{}{\Gamma, \langle x \rangle : !A \vdash x : A}$$

Abilities

Properties inherited from languages inspired by modal logic.

- program generation (plug data into another)
- execution

Properties

- Language is confluent
- Type system has subject reduction property

Missing

- No reflexion possibilities
- No syntactic analysis of data

Dereliction as launch control (1/2)

Term typed without Dereliction cannot run data.

Corollary (non interference)

If $\Gamma \vdash C : A$ is derivable without Dereliction, then it exists C'

$$\forall x, t', C[x/\langle t' \rangle] \xrightarrow{*} C'[x/t'] \not\to .$$

Then operations allowed are only:

- Writing operations (substitution in others data)
- Data passing function (use data without knowing it is actually data : ex. $\lambda x.x$)

What is missing

- Type $!A \multimap !!A$ is forbidden.
- Ex.: $\text{let}\langle x\rangle = t \text{ in } \langle\langle x\rangle\rangle$ is rejected:

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash t : !A} \qquad \dfrac{\dfrac{?}{\Gamma', x : A \vdash \langle x\rangle : A}}{\Gamma, \langle x\rangle : !A \vdash \langle\langle x\rangle\rangle : !!A}\ \textsc{SoftPromotion}}{\Gamma \vdash \text{let}\langle x\rangle = t \text{ in } \langle\langle x\rangle\rangle : !!A}\ \textsc{Abstraction}$$

Ongoing solution

- Adding pattern $\langle x\rangle^n : !^n A$ in Γ
- Changing rules as follows, $\forall n \geqslant 1$:

$$\textsc{Dereliction}\ \dfrac{}{\Gamma, \langle x\rangle^n : !^n A \vdash x : A} \qquad \textsc{SoftPromotion}\ \dfrac{\langle x_i\rangle^{n-1} : !^{n-1} A_i \vdash t : B}{\langle x_i\rangle^{n} : !^{n} A_i \vdash \langle t\rangle : !B}$$

- Adding rule

$$\textsc{Digging}\ \dfrac{\langle x\rangle^{n+1} : !^{n+1} A, \Gamma \vdash t : B}{\langle x\rangle^{n} : !^{n} A, \Gamma \vdash t : B}$$

Example

$$
\textsc{Abstraction}\ \dfrac{\dfrac{\cdots}{\Gamma \vdash t : !A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'', x : A \vdash x : A}\,\textsc{Var}}{\Gamma', \langle x\rangle : A \vdash \langle x\rangle : !A}\,\textsc{SoftPromotion}}{\Gamma, \langle\langle x\rangle\rangle : !!A \vdash \langle\langle x\rangle\rangle : !!A}\,\textsc{SoftPromotion}}{\Gamma, \langle x\rangle : !A \vdash \langle\langle x\rangle\rangle : !!A}\,\textsc{Digging}}{\Gamma \vdash \mathsf{let}\langle x\rangle = t\ \mathsf{in}\ \langle\langle x\rangle\rangle : !!A}
$$

Drawbacks

- Dereliction is no more used only for running data.
- A term may have different type derivations.

Plan

Evaluation strategy

- Give low level interpretation of the language
- Simulate CBV strategy:

$$\frac{}{(\lambda x.t)\ v \overset{\text{CBV}}{\rightarrow} t[x/v]} \qquad \frac{}{\mathsf{let}\langle x\rangle = \langle t\rangle\ \mathsf{in}\ t_1 \overset{\text{CBV}}{\rightarrow} t_1[x/t]}$$

$$\frac{t_1 \overset{\text{CBV}}{\rightarrow} t_1'}{t_1\ t_2 \overset{\text{CBV}}{\rightarrow} t_1'\ t_2} \qquad \frac{t_2 \overset{\text{CBV}}{\rightarrow} t_2'}{v\ t_2 \overset{\text{CBV}}{\rightarrow} v\ t_2'}$$

$$\frac{t_2 \overset{\text{CBV}}{\rightarrow} t_2'}{\mathsf{let}\langle x\rangle = t_2\ \mathsf{in}\ t_1 \overset{\text{CBV}}{\rightarrow} \mathsf{let}\langle x\rangle = t_2'\ \mathsf{in}\ t_1}$$

Evaluation strategy

- Give low level interpretation of the language
- Simulate CBV strategy:

$$\frac{}{(\lambda x.t)\ v \overset{\text{CBV}}{\to} t[x/v]} \qquad \frac{}{\mathsf{let}\langle x\rangle = \langle t\rangle\ \mathsf{in}\ t_1 \overset{\text{CBV}}{\to} t_1[x/t]}$$

$$\frac{t_1 \overset{\text{CBV}}{\to} t_1'}{t_1\ t_2 \overset{\text{CBV}}{\to} t_1'\ t_2} \qquad \frac{t_2 \overset{\text{CBV}}{\to} t_2'}{v\ t_2 \overset{\text{CBV}}{\to} v\ t_2'}$$

$$\frac{t_2 \overset{\text{CBV}}{\to} t_2'}{\mathsf{let}\langle x\rangle = t_2\ \mathsf{in}\ t_1 \overset{\text{CBV}}{\to} \mathsf{let}\langle x\rangle = t_2'\ \mathsf{in}\ t_1}$$

Why CBV?

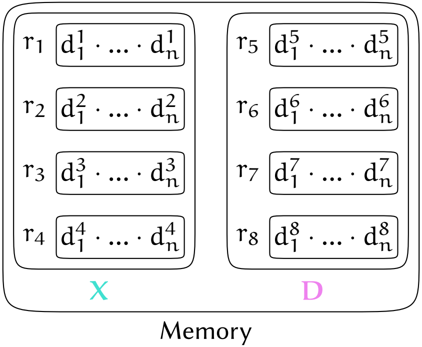
- Simple way to solve problem of redex $\mathsf{let}\langle x\rangle = \langle t'\rangle\ \mathsf{in}\ t$
- Already well studied abstract CBV machine (SECD)

ASM_2 Machine: principle

RASP-like machine for structure and SECD-like for instructions. A state $\langle d \mid k \mid e \mid \mathbf{X} \mid \mathbf{D} \rangle$ is composed by:

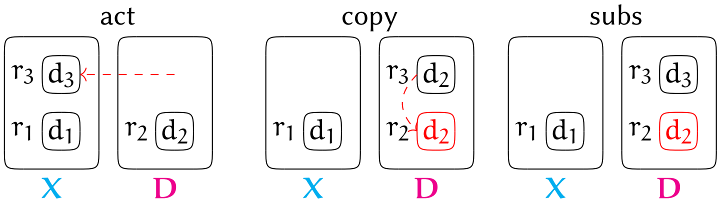
- Two set of registers $\mathbf{X}$ and $\mathbf{D}$: programs (executable and immutable) and data (non executable and mutable)
- A stack k and an environment e: it contains closure (d, e), data pointers $\langle r \rangle$ to $\mathbf{D}$, and program pointers r to $\mathbf{X}$.
- A code d being executed.

Data and programs are stored in registers.



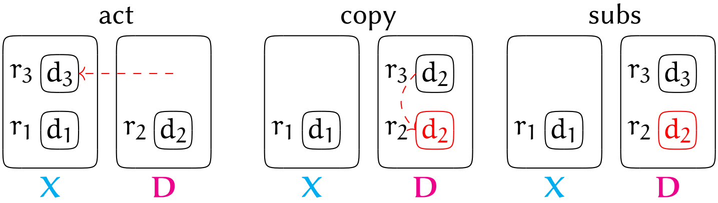
Abilities

SECD machine with…



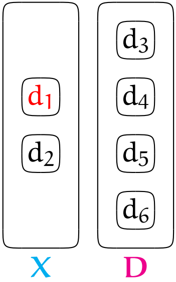
Abilities

SECD machine with...



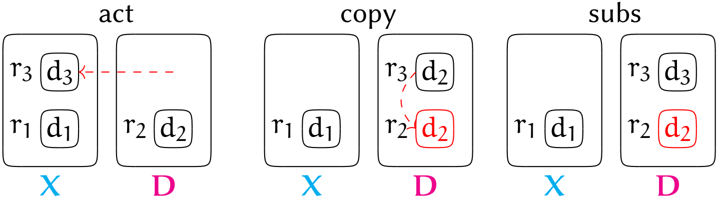
Example

If $d_1 =$ subs d_4; subs d_6; act d_6; run d_6



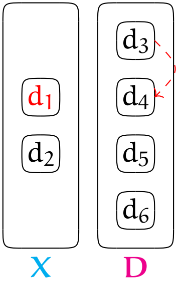
Abilities

SECD machine with...



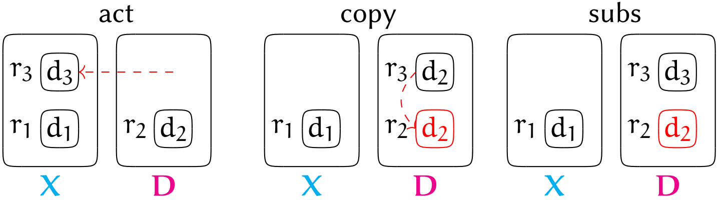
Example

If $d_1 = \text{subs } d_4; \text{subs } d_6; \text{act } d_6; \text{run } d_6$



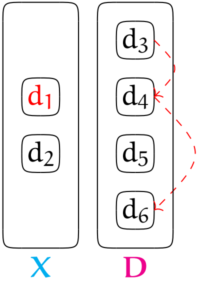
Abilities

SECD machine with…



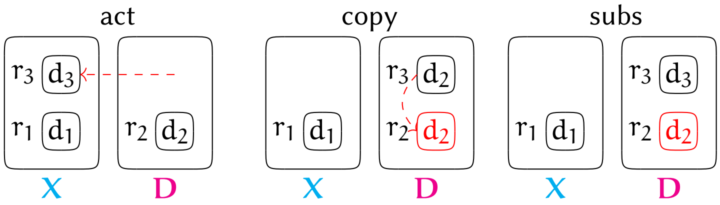
Example

If $d_1 = $ subs d_4; subs d_6; act d_6; run d_6



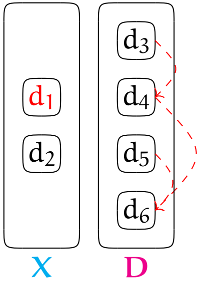
Abilities

SECD machine with...



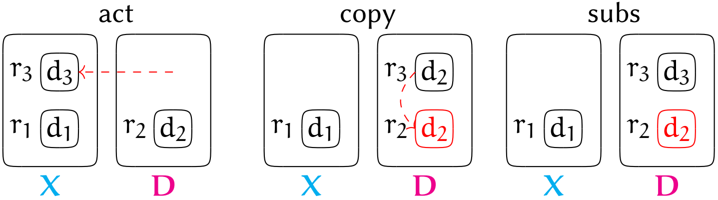
Example

If $d_1 = $ subs d_4; subs d_6; act d_6; run d_6



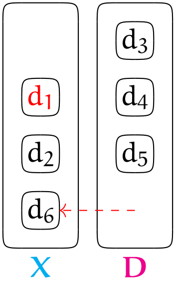
Abilities

SECD machine with...



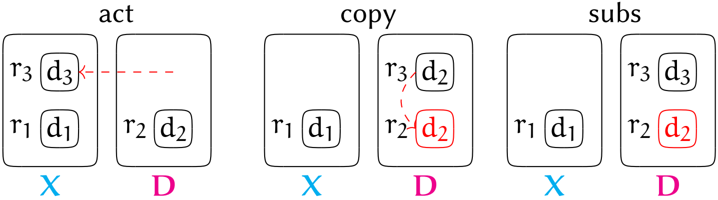
Example

If $d_1 =$ subs d_4; subs d_6; act d_6; run d_6



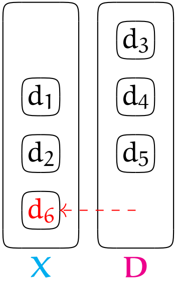
Abilities

SECD machine with…



Example

If $d_1 =$ subs d_4; subs d_6; act d_6; run d_6



Abilities

- Program and data live in the same world (program are data)
- Clear distinction between program and data
- Execution of data are made explicit

Abilities

- Program and data live in the same world (program are data)
- Clear distinction between program and data
- Execution of data are made explicit

Lacks

- Pattern matching on data
- Reflexivity

⇒ Same lacks than the high level language.

Compilation

Our will

- Compilation of t is given by a set of data T and an executable data d:
$$\mathsf{T} \vdash t \sim d.$$
- If $t \overset{CBv*}{\rightarrow} v$ and $\mathsf{T} \vdash t \sim d$, then it exists $\mathbf{X}', \mathbf{D}', v'$ such that
$$\langle d \cdot d' \mid k \mid e \mid \mathbf{X} \mid \mathbf{D} \rangle \overset{*}{\rightarrow} \langle d' \mid v'.k \mid e \mid \mathbf{X}' \mid \mathbf{D}' \rangle$$
with $\mathbf{X} \subset \mathbf{X}'$, $\mathbf{D} \subset \mathbf{D}'$ and $\mathsf{T} \in \mathbf{D}$
- If $t \overset{CBv*}{\rightarrow} \infty$ and $\mathsf{T} \vdash t \sim d$, then
$$\langle d \cdot d' \mid k \mid e \mid \mathbf{X} \mid \mathbf{D} \rangle \overset{*}{\rightarrow} \infty$$
with $\mathsf{T} \in \mathbf{D}$.
- Untyped then typed compilation

ASM_2 Machine: Semantics

- Usual SECD rules:

$$\begin{array}{lrrrcc} & \mathsf{push}(d').d & k & e & X & D \\ \to & d & (d', e).k & e & X & D \end{array}$$

$$\begin{array}{lrrrcc} & \mathsf{call}\,.d & v.(d', e').k & e & X & D \\ \to & d' & (d, e).k & v.e' & X & D \end{array}$$

$$\begin{array}{lrrrcc} & \mathsf{ret}\,.d & v.(d', e').k & e & X & D \\ \to & d' & v.k & e' & X & D \end{array}$$

- Usual compilation rules:

$$\textsc{CApp}\ \frac{\mathbf{T} \vdash t \sim d \qquad \mathbf{T} \vdash t' \sim d'}{\mathbf{T} \vdash t\ t' \sim d.d'.[\mathsf{call}]} \qquad \textsc{CAbs}\ \frac{\mathbf{T} \vdash t \sim d}{\mathbf{T} \vdash \lambda.t \sim [\mathsf{push}(d.[\mathsf{ret}])]}$$

Box

- Rule:

$$\begin{array}{rrrccl} & (\text{copy}\, r').d & k & e & \color{cyan}{X} & \color{magenta}{D} \\ \xrightarrow{\text{if } r'' \text{ is fresh}} & d & \langle r'' \rangle.k & e & \color{cyan}{X} & \color{magenta}{D}[r'' \mapsto \color{magenta}{D}(r')] \end{array}$$

$$\begin{array}{rrrccl} & \text{subs}\,.d & \langle r' \rangle.k & e & \color{cyan}{X} & \color{magenta}{D} \\ \to & \langle d & \langle r' \rangle.k & e & \color{cyan}{X} & \color{magenta}{D}[r' \mapsto \color{magenta}{D}(r')[e, \color{magenta}{D}]] \end{array}$$

- Compilation rule:

$$\text{CBox}\ \frac{\mathbf{T} \vdash t \sim d \qquad \mathbf{T}(r) = d.[\text{ret}]}{\mathbf{T} \vdash \langle t \rangle \sim [\text{copy}\, r].[\text{subs}]}$$

Let

- Rule:

$$\begin{array}{lllll} \text{act}\, n.d & k & e & \mathbf{X} & \mathbf{D} \\ \xrightarrow{\text{if } e(n) = \langle r' \rangle} \quad d & k & e[n \to r'] & \mathbf{X}[r' \mapsto \mathbf{D}(r')] & \mathbf{D} \end{array}$$

- Compilation rule:

$$\text{CLet}\ \frac{\mathsf{T} \vdash t \sim d \qquad \mathsf{T} \vdash t' \sim d'}{\mathsf{T} \vdash \text{let}\, t' \text{ in } t \sim [\text{push}([\text{act}\, 0].d.[\text{ret}])].d'.[\text{call}]}$$

Remark

- By default, a let activate the given data.
- Ex: $\text{let}\langle x \rangle = t \in fx\langle x \rangle$

Variables

- Variables have two roles
  - reference:

$$\lambda x.x$$

  - execution:

$$\text{let}\langle x\rangle = t \text{ in } x$$

- Compilation of variables:

$$\frac{\textsc{CVar}}{\mathbf{T} \vdash n \rightsquigarrow [\text{fetch}\, n]}$$

- Indeterminism of instruction:

$$\begin{array}{lrrccc} & (\text{fetch}\, n).d & k & e & X & D \\ \xrightarrow{\text{if } e(n) \neq r'} & d & e(n).k & e & X & D \\ & (\text{fetch}\, n).d & k & e & X & D \\ \xrightarrow{\text{if } e(n) = r'} & X(r') & (d, e).k & \cdot & X & D \end{array}$$

Examples (1/2)

Writing: let$\langle x \rangle = \langle t \rangle$ in $\langle x \rangle$

$$
\begin{array}{rrrll}
 & k & e & \mathbf{X} & \mathbf{D} \\
\xrightarrow{\langle t \rangle} & \langle r \rangle.k & e & \mathbf{X} & \mathbf{D}[r \rightarrow t] \\
\xrightarrow{\text{let}} & k & r.e & \mathbf{X}[r \rightarrow t] & \mathbf{D}[r \rightarrow t] \\
\xrightarrow{\langle x \rangle} & \langle r' \rangle.k & r.e & \mathbf{X}[r \rightarrow t] & \mathbf{D}[r \rightarrow t][r' \rightarrow t]
\end{array}
$$

Examples (1/2)

Writing: let⟨x⟩ = ⟨t⟩ in ⟨x⟩

$$\begin{array}{rrrll}
 & k & e & \mathbf{X} & \mathbf{D} \\
\xrightarrow{\langle t \rangle} & \langle r \rangle.k & e & \mathbf{X} & \mathbf{D}[r \to t] \\
\xrightarrow{\text{let}} & k & r.e & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\xrightarrow{\langle x \rangle} & \langle r' \rangle.k & r.e & \mathbf{X}[r \to t] & \mathbf{D}[r \to t][r' \to t]
\end{array}$$

Execution: let⟨x⟩ = ⟨t⟩ in x

$$\begin{array}{rrrll}
 & \vdots & \vdots & \vdots & \vdots \\
 & k & r.e & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\xrightarrow{x} & (\cdot, e).k & \cdot & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\xrightarrow{\mathbf{X}(r)} & \dots & & &
\end{array}$$

Examples (1/2)

Writing: let$\langle x\rangle = \langle t\rangle$ in $\langle x\rangle$

$$
\begin{array}{lrrrll}
 & & \mathsf{k} & \mathsf{e} & \mathbf{X} & \mathbf{D} \\
 & \xrightarrow{\langle t\rangle} & \langle r\rangle.\mathsf{k} & \mathsf{e} & \mathbf{X} & \mathbf{D}[r \to t] \\
\text{useless activation} & \xrightarrow{\text{let}} & \mathsf{k} & r.\mathsf{e} & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
 & \xrightarrow{\langle x\rangle} & \langle r'\rangle.\mathsf{k} & r.\mathsf{e} & \mathbf{X}[r \to t] & \mathbf{D}[r \to t][r' \to t]
\end{array}
$$

Execution: let$\langle x\rangle = \langle t\rangle$ in x

$$
\begin{array}{lrrll}
 & \vdots & \vdots & \vdots & \vdots \\
 & \mathsf{k} & r.\mathsf{e} & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\xrightarrow{x} & (\cdot, \mathsf{e}).\mathsf{k} & \cdot & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\xrightarrow{\mathbf{X}(r)} & \dots & & &
\end{array}
$$

Examples (2/2)

Execution: let$\langle x \rangle = \langle t \rangle$ in $\langle x \rangle$

$$
\begin{array}{rcrcll}
 & & k & e & \mathbf{X} & \mathbf{D} \\
 & \xrightarrow{\langle t \rangle} & \langle r \rangle.k & e & \mathbf{X} & \mathbf{D}[r \to t] \\
 & \xrightarrow{\text{let}} & k & r.e & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
\text{fetch } n & \xrightarrow{x} & (\cdot, e).k & \cdot & \mathbf{X}[r \to t] & \mathbf{D}[r \to t] \\
 & \xrightarrow{\mathbf{X}(r)} & \dots & & &
\end{array}
$$

Examples (2/2)

Execution: let$\langle x \rangle = \langle t \rangle$ in $\langle x \rangle$

$$
\begin{array}{rcrrll}
 & & k & e & X & D \\
 & \xrightarrow{\langle t \rangle} & \langle r \rangle.k & e & X & D[r \to t] \\
 & \xrightarrow{\text{let}} & k & r.e & X[r \to t] & D[r \to t] \\
\text{fetch } n & \xrightarrow{x} & (\cdot, e).k & \cdot & X[r \to t] & D[r \to t] \\
 & \xrightarrow{X(r)} & \ldots & & &
\end{array}
$$

Identity: $(\lambda x.x)\langle t \rangle$

$$
\begin{array}{rcrrll}
 & & k & e & X & D \\
 & \xrightarrow{\lambda x.x} & (x, e).k & e & X & D \\
 & \xrightarrow{\langle t \rangle} & \langle r \rangle.(\lambda x.x, e).k & e & X & D[r \to t] \\
 & \xrightarrow{@} & (\cdot, e).k & \langle r \rangle.e & X & D[r \to t] \\
\text{fetch } n & \xrightarrow{x} & \langle r \rangle.(\cdot, e).k & e & X & D[r \to t] \\
 & \xrightarrow{\text{ret}} & \langle r \rangle.k & e & X & D[r \to t]
\end{array}
$$

Dereliction as launch control (2/2)

- Indeterminism on variables can be mitigated with typing information.
- fetch n can be specialized in two new rules: fetch n and run n

$$
\begin{array}{lrrcrr}
 & (\mathsf{fetch}\,n).d & k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
\xrightarrow{\text{if } e(n) \neq r'} & d & e(n).k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
 & (\,\mathsf{fetch}\,n).d & k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
\xrightarrow{\text{if } e(n) = r'} & \color{cyan}{\mathbf{X}}(r') & (d, e).k & \cdot & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}}
\end{array}
$$

Dereliction as launch control (2/2)

- Indeterminism on variables can be mitigated with typing information.
- fetch $\mathfrak{n}$ can be specialized in two new rules: fetch $\mathfrak{n}$ and run $\mathfrak{n}$

$$
\begin{array}{rrrrrr}
 & (\text{fetch}\,\mathfrak{n}).d & k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
\to & d & e(\mathfrak{n}).k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
 & (\,\color{red}{\text{run}}\,\mathfrak{n}).d & k & e & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}} \\
\overset{\text{if } e(\mathfrak{n}) = r'}{\to} & \color{cyan}{\mathbf{X}}(r') & (d, e).k & \cdot & \color{cyan}{\mathbf{X}} & \color{magenta}{\mathbf{D}}
\end{array}
$$

Dereliction as launch control (2/2)

- Indeterminism on variables can be mitigated with typing information.
- fetch n can be specialized in two new rules: fetch n and run n

$$\begin{array}{rrrrcc} & (\mathsf{fetch}\,n).d & k & e & X & D \\ \to & d & e(n).k & e & X & D \\ & (\,\mathsf{run}\,n).d & k & e & X & D \\ \xrightarrow{\text{if } e(n) = r'} & X(r') & (d,e).k & \cdot & X & D \end{array}$$

- Activation can be lazily postoned

$$\textsc{CLet}\quad \frac{T \vdash t \sim d \qquad T \vdash t' \sim d'}{T \vdash \mathsf{let}\; t' \;\mathsf{in}\; t \sim [\mathsf{push}([\mathsf{act}\,0].d.[\mathsf{ret}])].d'.[\mathsf{call}]}$$

Dereliction as launch control (2/2)

- Indeterminism on variables can be mitigated with typing information.
- fetch n can be specialized in two new rules: fetch n and run n

$$\begin{array}{rrrccc} & (\text{fetch}\, n).d & k & e & X & D \\ \rightarrow & d & e(n).k & e & X & D \\ & (\,\text{run}\, n).d & k & e & X & D \\ \overset{\text{if } e(n) = r'}{\longrightarrow} & X(r') & (d, e).k & \cdot & X & D \end{array}$$

- Activation can be lazily postoned

CLet

$$\frac{T \vdash t \sim d \qquad T \vdash t' \sim d'}{T \vdash \text{let}\, t' \text{ in } t \sim [\text{push}(\qquad d.[\text{ret}])].d'.[\text{call}]}$$

Dereliction as launch control (2/2)

- Indeterminism on variables can be mitigated with typing information.
- fetch n can be specialized in two new rules: fetch n and run n

$$
\begin{array}{rrrrrr}
 & (\text{fetch}\, n).d & k & e & \mathcal{X} & \mathcal{D} \\
\to & d & e(n).k & e & \mathcal{X} & \mathcal{D} \\
 & (\,\text{run}\, n).d & k & e & \mathcal{X} & \mathcal{D} \\
\overset{\text{if } e(n) = r'}{\to} & \mathcal{X}(r') & (d, e).k & \cdot & \mathcal{X} & \mathcal{D}
\end{array}
$$

- Activation can be lazily postoned

CLet

$$
\frac{\mathsf{T} \vdash t \sim d \qquad \mathsf{T} \vdash t' \sim d'}{\mathsf{T} \vdash \text{let}\, t' \text{ in } t \sim [\text{push}(\qquad d.[\text{ret}])].d'.[\text{call}]}
$$

$\Rightarrow$ Use typing information

Dereliction as launch control (2/2)

- New compilation rules: $\Gamma; \mathbf{T} \vdash t \sim d : A$ following typing system.
- Nothing changes but

$$\textsc{CVar}\ \frac{\Gamma(n) : A}{\Gamma; \mathbf{T} \vdash n \sim [\mathrm{fetch}\, n] : A} \qquad \textsc{CRun}\ \frac{\Gamma(\langle n \rangle) : !A}{\Gamma; \mathbf{T} \vdash n \sim [\mathrm{act}\, n].[\mathrm{run}\, n] : A}$$

$\Rightarrow$ Correctness and completness of compilation is preserved.

Plan

Problem

- Given a reduction strategy, how to locally force reduction?

Example

How to write power function pow such that for all n, pow n is *completely* reduced, that is, for instance

$$\mathrm{pow}\,2 = \lambda x.x * x.$$

First try

- $\mathrm{pow} \stackrel{\mathrm{def}}{=} \mathrm{Fix}\, p : \mathbb{N} \to !(\mathbb{N} \to \mathbb{N}).$
  $\lambda n : \mathbb{N}.\, \mathrm{case}\, n\, \mathrm{with}$
  $|0 \to \langle \lambda x.1 \rangle$
  $|n + 1 \to \mathrm{let} \langle q \rangle = p\ n\ \mathrm{in}\ \langle \lambda x.x * (q\ x) \rangle$

Problem

- Given a reduction strategy, how to locally force reduction?

Example

How to write power function pow such that for all n, $\mathrm{pow}\, n$ is *completely* reduced, that is, for instance

$$\mathrm{pow}\, 2 = \lambda x.x * x.$$

First try

- $\mathrm{pow} \stackrel{\text{def}}{=} \mathrm{Fix}\, p : \mathbb{N} \to !(\mathbb{N} \to \mathbb{N}).$
  $\lambda n : \mathbb{N}.\, \mathrm{case}\, n\, \mathrm{with}$
  $|0 \to \langle \lambda x.1 \rangle$
  $|n + 1 \to \mathrm{let} \langle q \rangle = p\ n\ \mathrm{in}\ \langle \lambda x.x * (q\ x) \rangle$
- $\mathrm{pow}\, 2 \xrightarrow{*} \langle \lambda x.x * (\lambda x.x * (\lambda x.1) x) x \rangle$: not optimal

Problem

- Given a reduction strategy, how to locally force reduction?

Example

How to write power function pow such that for all n, $\mathrm{pow}\, n$ is *completely* reduced, that is, for instance

$$\mathrm{pow}\, 2 = \lambda x.x * x.$$

First try

- $\mathrm{pow} \stackrel{\text{def}}{=} \mathrm{Fix}\, p : \mathbb{N} \to !(\mathbb{N} \to \mathbb{N}).$
  $\lambda n : \mathbb{N}.\, \mathrm{case}\, n\, \mathrm{with}$
  $|0 \to \langle \lambda x.1 \rangle$
  $|n + 1 \to \mathrm{let} \langle q \rangle = p\ n\ \mathrm{in}\ \langle \lambda x.x * (q\ x) \rangle$
- $\mathrm{pow}\, 2 \xrightarrow{*} \langle \lambda x.x * (\lambda x.x * (\lambda x.1) x) x \rangle$: not optimal

A. Nanevski, F. Pfenning, and B. Pientka.
Contextual modal type theory.
Transactions on Computational Logic, February 2007.

(New type $A[\Gamma]$ for the new syntax $\langle t \rangle_{x_1,\ldots,x_n}$, with chained substitution.)

Suspended box

Idea

Enable reduction under special box.

Special operator $\mathsf{box}\,t$ such that:

- New redex:

$$\mathsf{box}\,t \to \langle t \rangle$$

- New typing rule:

$$\frac{\Gamma \vdash t : A}{\langle \Gamma \rangle \vdash \mathsf{box}\,t : !A}$$

- Evaluation under box:

$$\frac{t \xrightarrow{*} t'}{\mathsf{box}\,t \xrightarrow{*} \mathsf{box}\,t'}$$

Remark

- Subject reduction is preserved.
- Confluence is lost…
- …but every NF are β-equivalent for usual λ-calculus.

Usage

Second try

- $\mathsf{pow} \stackrel{\text{def}}{=} \mathsf{Fix}\, p : \mathbb{N} \to !(\mathbb{N} \to \mathbb{N}).$

$$
\begin{array}{l}
\lambda n : \mathbb{N}.\, \mathsf{case}\, n\, \mathsf{with} \\
\quad |0 \to \langle \lambda x.1 \rangle \\
\quad |n + 1 \to \mathsf{let} \langle q \rangle = p\ n\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let} \langle q' \rangle = \mathsf{box}\, q\ \mathsf{in} \\
\qquad\qquad\quad \langle \lambda x.x * (q'\ x) \rangle
\end{array}
$$

Usage

Second try

- $\mathsf{pow} \stackrel{\text{def}}{=} \mathsf{Fix}\, p : \mathbb{N} \to !(\mathbb{N} \to \mathbb{N}).$
  $\lambda n : \mathbb{N}.\, \mathsf{case}\, n\, \mathsf{with}$
  $|0 \to \langle \lambda x.1 \rangle$
  $|n + 1 \to \mathsf{let}\langle q \rangle = p\ n\ \mathsf{in}$
  $\mathsf{let}\langle q' \rangle = \mathsf{box}\, q\ \mathsf{in}$
  $\langle \lambda x.x * (q'\ x) \rangle$
- $\mathsf{pow}\, 2 \xrightarrow{*} \langle \lambda x.x * x * 1 \rangle$ but indeterministically

Conclusion

- High level language with self-modifying behaviors (writing & executing data)
- Dereliction is a data execution
- Meaningful compilation in low-level machine designed for self-modification
- Dereliction still corresponds to data execution

Still missing

- Reflexion & pattern matching
- A more deterministic partial evaluation
- Recover all the power of modal logic