A taste of linear logic for staged computation

Hubert Godfroy Jean-Yves Marion

First GDRI-LL meeting
4 février 2016

Plan

Staged computation

A correct system

Soft modality

Low level correspondence

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.

Staged computation in everyday life

- Python
  - Strings are frozen codes.
  - `eval` launches execution of strings.
- Meta-OCaml
  - $\langle t \rangle$ are frozen codes.
  - pieces of unevaluated code can be inserted in frozen terms with $\sim$.

$$\langle f\ t_1\ \sim ((\lambda x.x)\langle t_2 \rangle)\rangle \rightarrow \langle f\ t_1\ t_2 \rangle$$

Syntactic mark for value

- $\langle t \rangle$ is a value *for any* t.
- $\langle (\lambda x.x) t \rangle$ and $\langle t \rangle$ are *not* β-equivalent…
- …but computations are frozen.

Syntactic mark for value

- $\langle t\rangle$ is a value *for any* t.
- $\langle(\lambda x.x)t\rangle$ and $\langle t\rangle$ are *not* β-equivalent…
- …but computations are frozen.

$$\begin{tikzcd}
\langle(\lambda x.x)t\rangle \arrow[dr, "\text{run}"'] & \neq_\beta & \langle t\rangle \arrow[dl, "\text{run}"] \\
& t &
\end{tikzcd}$$

Why is it important?

- Frozen codes are used to denote *data* like string.
- Pattern matching on data.
- Confluence should be preserved

Why is it important?

- Frozen codes are used to denote *data* like string.
- Pattern matching on data.
- Confluence should be preserved

Counter example

$$\begin{tikzcd} & \mathrm{match}\langle(\lambda x.x)(\lambda x.x)\rangle \text{ with App} \mapsto 1 \mid \text{Lam} \mapsto 2 \arrow[dl] \arrow[dr] & \\ 1 & & 2 \end{tikzcd}$$

Example: creation of data *breaks* confluence

It should not exist a reifying operator returning the code (data) of a term.

Otherwise…

If θ were this operator

$$\begin{tikzcd}
 & \theta\ ((\lambda x.x)t) \arrow[dl] \arrow[dr] & \\
\langle(\lambda x.x)t\rangle & \neq_\beta & \langle t\rangle
\end{tikzcd}$$

Corollary

θ is not expressible in λ-calculus.

Frozen codes are mutable

- Substitution in frozen terms are allowed

$$\langle t \rangle [x \to t'] = \langle t[x \to t'] \rangle$$

Frozen codes are mutable

- Substitution in frozen terms are allowed

$$\langle t\rangle[x \to t'] = \langle t[x \to t']\rangle$$

- Put a frozen term into another

$$\text{let}\langle x\rangle = \langle t'\rangle \text{ in } \langle t\rangle \quad \to \quad \langle t\rangle[x \to t'] \quad = \quad \langle t[x \to t']\rangle$$

Subtly

- Same confluence issue than reification
- Cannot plug terms which are note data.
- If $t \to t'$:

$$\begin{tikzcd}
 & (\lambda x.\langle x\rangle)t \arrow[dl] \arrow[dr] & \\
\langle t\rangle & \neq & \langle t'\rangle
\end{tikzcd}$$

Usages IRL

- Partial evaluation optimisations: n-ary function can be specialized by inspecting their code: pow is the power function.

without specialization	$\mathrm{pow}\ 2 = \langle \lambda x.x \times \mathrm{pow}\ 1 \rangle$
with specialization	$\mathrm{pow}\ 2 = \langle \lambda x.x \times x \times 1 \rangle$

- Dynamic code generation
- Code protection:
  - if $d = \widehat{\langle t_0 \rangle}$ is the encryption of $\langle t_0 \rangle$ and $\mathrm{dec} : \widehat{\langle t \rangle} \mapsto \langle t \rangle$ is the decrypting function, $\mathrm{run}(\mathrm{dec}\ d) \to t_0$
  - Chained processus: $t_n = \mathrm{run}(\mathrm{dec}\ \widehat{\langle t_{n-1} \rangle})$

Plan

Staged computation

A correct system

Soft modality

Low level correspondence

Modal Logic

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

Modal Logic

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

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:

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

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: $(\mathsf{let}\langle x \rangle = \langle t' \rangle \text{ in } t) \rightarrow t[x/t']$
- 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 $\langle \Gamma \rangle \overset{\text{def}}{=} \langle x_i \rangle : \,!A_i$.

Exemple

- Writing:

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

Exemple

- Writing:

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

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

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

Exemple

- Writing:

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

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:

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

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:

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

Use this derivation:

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

Exemple

- Writing:

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

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

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

- Execution:

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

Use this derivation:

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

⇒ Both use DERELICTION.

Exemple

- Writing:

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

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

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

- Execution:

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

Use this derivation:

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

$\Rightarrow$ Both use Dereliction.

Question

What is the logical meaning of Dereliction in this system?

Plan

Staged computation

A correct system

Soft modality

Low level correspondence

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!

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

Soft promotion

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

Idea

Compose Dereliction and Promotion!

$$\begin{array}{c}\textsc{SoftPromotion}\\ \dfrac{\Gamma \vdash t : A}{\langle \Gamma \rangle \vdash \langle t \rangle : !A}\end{array}$$

Consequences

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

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

Language behavior

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

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

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

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

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

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

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. $\lambda x.x$)

Plan

Staged computation

A correct system

Soft modality

Low level correspondence

Evaluation strategy

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

$$\frac{}{(\lambda x.t)\ t' \overset{\text{CBN}}{\rightarrow} t[x/t']} \qquad \frac{}{\mathsf{let}\langle x\rangle = \langle t\rangle\ \mathsf{in}\ t_1 \overset{\text{CBN}}{\rightarrow} t_1[x/t]}$$

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

ASM_2 Machine: principle

A state

$$\langle \mathrm{d} \mid \mathrm{k} \mid e \mid \mathbf{D} \rangle$$

- $\mathbf{D}$ a set of data (frozen terms)
- Stack k and environment e
- Code d being executed:

$$\mathrm{d} ::= \lambda.\mathrm{d} \mid \mathrm{d}\ \mathrm{d} \mid \langle \mathrm{d} \rangle \mid \text{let } \mathrm{d} \text{ in } \mathrm{d} \mid \text{run } \mathfrak{n} \mid \text{fetch } \mathfrak{n}$$

Extended Krivine Machine

	$\lambda.d$	$d'.k$	e	$\mathbf{D}$
$\rightarrow$	d	k	$d'.e$	$\mathbf{D}$
	$d\ d'$	k	e	$\mathbf{D}$
$\rightarrow$	d	$d'_e.k$	e	$\mathbf{D}$
	$\mathsf{fetch}\ n$	k	e	$\mathbf{D}$
$\xrightarrow{e[n]=d_{e'}}$	d	k	e'	$\mathbf{D}$
	$\mathsf{let}\ d\ \mathsf{in}\ d'$	k	e	$\mathbf{D}$
$\rightarrow$	d	$(\lambda\langle x\rangle.d')_e.k$	e	$\mathbf{D}$
	$\langle d\rangle$	$(\lambda\langle x\rangle.d')_{e'}.k$	e	$\mathbf{D}$
$\xrightarrow{r\ \text{fresh}}$	d'	k	$r.e'$	$\mathbf{D}[r \mapsto d[e, \mathbf{D}]]$
	$\mathsf{run}\ n$	k	e	$\mathbf{D}$
$\xrightarrow{e[n]=r}$	$\mathbf{D}(r)$	k	$\cdot$	$\mathbf{D}$

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:

$$t \rightsquigarrow d$$

- Soundness

$$\begin{array}{lccc} \text{if} & t & \xrightarrow{\text{CBN}*} & v \\ \text{then} & \langle d \mid \cdot \mid \cdot \mid \cdot \rangle & \xrightarrow{*} & \langle d' \mid \cdot \mid e \mid \mathbf{D} \rangle \end{array}$$

where $v = d'[e, \mathbf{D}]$.

- Completeness

$$\begin{array}{lccc} \text{if} & t & \xrightarrow{\text{CBN}*} & \infty \\ \text{then} & \langle d \mid \cdot \mid \cdot \mid \cdot \rangle & \xrightarrow{*} & \infty \end{array}$$

Variables

- Variables have two roles
    - reference:

$$\lambda x.x$$

    - execution:

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

- Indeterminism on variables can be mitigated with typing information.
- New compilation rules: $\Gamma \vdash t \sim d : A$ following typing system.
- Nothing changes but

$$\textsc{CVar}\ \frac{\Gamma(n) : A}{\Gamma; \mathbf{T} \vdash n \sim \mathsf{fetch}\ n : A} \qquad \textsc{CRun}\ \frac{\Gamma(\langle n\rangle) : !A}{\Gamma; \mathbf{T} \vdash n \sim \mathsf{run}\ n : A}$$

Examples (1/2)

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

$$\begin{array}{rrrl}
 & \cdot & e & \mathbf{D} \\
\xrightarrow{\text{let}} & (\lambda\langle x\rangle.\langle x\rangle)_e & e & \mathbf{D} \\
\xrightarrow{\langle t\rangle} & \cdot & r.e & \mathbf{D}[r \rightarrow t] \\
\xrightarrow{\langle x\rangle} & \text{stop} & &
\end{array}$$

Examples (1/2)

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

$$\begin{array}{rrrl}
 & \cdot & e & \mathbf{D} \\
\xrightarrow{\text{let}} & (\lambda\langle x\rangle.\langle x\rangle)_e & e & \mathbf{D} \\
\xrightarrow{\langle t\rangle} & \cdot & r.e & \mathbf{D}[r \to t] \\
\xrightarrow{\langle x\rangle} & \text{stop} & &
\end{array}$$

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

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

Examples (2/2)

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

$$\begin{array}{rcrrl}
 & & \cdot & e & \mathbf{D} \\
 & \xrightarrow{\text{let}} & (\lambda\langle x\rangle.\langle x\rangle)_e & e & \mathbf{D} \\
 & \xrightarrow{\langle t\rangle} & \cdot & r.e & \mathbf{D}[r \to t] \\
\text{run}\, x & \xrightarrow{x} & \cdot & \cdot & \mathbf{D}[r \to t] \\
 & \xrightarrow{\mathbf{D}(r)} & \ldots & &
\end{array}$$

Examples (2/2)

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

$$\begin{array}{rcrrl}
 & & \cdot & e & \mathbf{D} \\
 & \xrightarrow{\text{let}} & (\lambda\langle x\rangle.\langle x\rangle)_e & e & \mathbf{D} \\
 & \xrightarrow{\langle t\rangle} & \cdot & r.e & \mathbf{D}[r \to t] \\
\text{run } x & \xrightarrow{x} & \cdot & \cdot & \mathbf{D}[r \to t] \\
 & \xrightarrow{\mathbf{D}(r)} & \ldots & &
\end{array}$$

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

$$\begin{array}{rcrrl}
 & & k & e & \mathbf{D} \\
 & \xrightarrow{@} & \langle t\rangle.k & e & \mathbf{D} \\
 & \xrightarrow{\lambda x.x} & k & \langle t\rangle.e & \mathbf{D} \\
\text{fetch } x & \xrightarrow{x} & k & e & \mathbf{D} \\
 & \xrightarrow{\langle t\rangle} & \text{stop} & &
\end{array}$$

Correctness and decompilation

Correction of the compilation

If $t \sim d$ and t is well typed and closed

- $t \overset{\text{CBN}*}{\rightarrow} v \not\rightarrow$ then $\langle d \mid \cdot \mid \cdot \mid \cdot \rangle \xrightarrow{*} \langle d' \mid \cdot \mid e \mid \mathbf{D} \rangle$ and $v = d'[e, \mathbf{D}]$.
- $t \overset{\text{CBN}*}{\rightarrow} \infty$ then $\langle d \mid \cdot \mid \cdot \mid \cdot \rangle \xrightarrow{*} \infty$.

Decompilation

For all state $S = \langle d \mid k \mid e \mid \mathbf{D} \rangle$ decompiles following the rules:

$$\begin{array}{rcl} \mathcal{D}\langle d \mid k.(\lambda\langle x\rangle.d')_{e'} \mid e \mid \mathbf{D} \rangle & = & \text{let}\langle x \rangle = \mathcal{D}\langle d \mid k \mid e \mid \mathbf{D} \rangle \text{ in } d'[e', \mathbf{D}] \\ \mathcal{D}\langle d \mid k.d'_{e'} \mid e \mid \mathbf{D} \rangle & = & (\mathcal{D}\langle d \mid k \mid e \mid \mathbf{D} \rangle) d'[e', \mathbf{D}] \\ \mathcal{D}\langle d \mid \cdot \mid e \mid \mathbf{D} \rangle & = & d[e, \mathbf{D}] \end{array}$$

Property

$$S \to S' \quad \Rightarrow \quad \mathcal{D}(S) = \mathcal{D}(S') \vee \mathcal{D}(S) \to \mathcal{D}(S').$$

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