

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 .

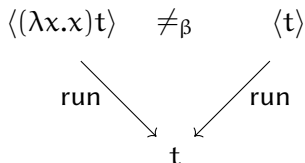
$$\langle f \ t_1 \ \sim \ ((\lambda x.x)\langle t_2 \rangle) \rangle \rightarrow \langle f \ t_1 \ t_2 \rangle$$

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.



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

$\text{match}\langle(\lambda x.x)(\lambda x.x)\rangle$ with $\text{App} \mapsto 1 \mid \text{Lam} \mapsto 2$

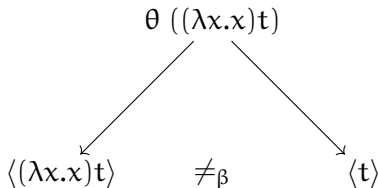


Example: creation of data *breaks* confluence

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

Otherwise...

If θ were this operator



Corollary

θ is not expressible in λ -calculus.

Frozen codes are mutable

- ▶ Substitution in frozen terms are allowed

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

Frozen codes are mutable

- ▶ Substitution in frozen terms are allowed

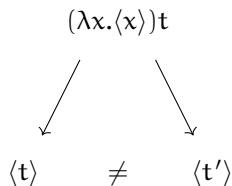
$$\langle t \rangle[x \rightarrow t'] = \langle t[x \rightarrow t'] \rangle$$

- ▶ Put a frozen term into another

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

Subtly

- ▶ Same confluence issue than reification
- ▶ Cannot plug terms which are not data.
- ▶ If $t \rightarrow t'$:



Usages IRL

- ▶ Partial evaluation optimisations: n-ary function can be specialized by inspecting their code: pow is the power function.
 - without specialization $\text{pow } 2 = \langle \lambda x. x \times \text{pow } 1 \rangle$
 - with specialization $\text{pow } 2 = \langle \lambda x. x \times x \times 1 \rangle$
- ▶ Dynamic code generation
- ▶ Code protection:
 - ▶ if $d = \widehat{\langle t_0 \rangle}$ is the encryption of $\langle t_0 \rangle$ and $\text{dec} : \widehat{\langle t \rangle} \mapsto \langle t \rangle$ is the decrypting function, $\text{run}(\text{dec } d) \rightarrow t_0$
 - ▶ Chained processus: $t_n = \text{run}(\text{dec } \widehat{\langle t_{n-1} \rangle})$

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: $\text{let} \langle x \rangle = t' \text{ in } t$
- ▶ 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: $\text{let} \langle x \rangle = t' \text{ in } t$
- ▶ Meta-redex: $(\text{let} \langle x \rangle = \langle t' \rangle \text{ in } t) \rightarrow t[x/t']$
- ▶ Typing rules:

$$\frac{\text{BOX} \quad \langle \Delta \rangle \vdash t : A}{\langle \Delta \rangle, \Gamma \vdash \langle t \rangle : \Box A} \quad \frac{\text{GVAR} \quad \langle x \rangle : A \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\text{LET} \quad \Gamma \vdash t' : \Box A \quad \langle x \rangle : A, \Gamma \vdash t : B}{\Gamma \vdash \text{let} \langle x \rangle = t' \text{ in } t : B}$$

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

Linear Logic

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

$$\begin{array}{c} \text{PROMOTION} \\ \frac{\langle \Delta \rangle \vdash t : A}{\langle \Delta \rangle, \Gamma \vdash \langle t \rangle : !A} \end{array} \quad \begin{array}{c} \text{DERELICTION} \\ \frac{\langle x \rangle : A \in \Gamma}{\Gamma \vdash x : A} \end{array} \quad \begin{array}{c} \text{LET} \\ \frac{\Gamma \vdash t' : !A \quad \langle x \rangle : A, \Gamma \vdash t : B}{\Gamma \vdash \text{let} \langle x \rangle = t' \text{ in } t : B} \end{array}$$

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

Example

- ▶ Writing:

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

Example

- ▶ Writing:

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

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\frac{\Gamma', \langle x \rangle : !A \vdash x : A}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{DERELICTION}}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{PROMOTION}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}$$

Example

- ▶ Writing:

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

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\Gamma', \langle x \rangle : !A \vdash x : A}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{DERELICTION}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A} \text{PROMOTION}$$

- ▶ Execution:

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

Example

- ▶ Writing:

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

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\frac{\Gamma', \langle x \rangle : !A \vdash x : A}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{DERELICTION}}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{PROMOTION}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}$$

- ▶ Execution:

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

Use this derivation:

$$\text{ABSTRACTION} \frac{\text{LET} \frac{\text{DERELICTION} \frac{\text{VAR} \frac{\Gamma, y : A \vdash y : A}{\Gamma, \langle y \rangle : !A \vdash y : A}}{\Gamma, x : !A \vdash \text{let } \langle y \rangle = x \text{ in } y : A}}{\Gamma \vdash \lambda x. \text{let } \langle y \rangle = x \text{ in } y : !A \rightarrow A}$$

Example

- ▶ Writing:

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

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\text{DERELICTION} \quad \frac{\text{PROMOTION} \quad \frac{\Gamma', \langle x \rangle : !A \vdash x : A}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}}$$

- ▶ Execution:

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

Use this derivation:

$$\text{ABSTRACTION} \frac{\text{LET} \frac{\text{DERELICTION} \quad \frac{\text{VAR} \quad \frac{\Gamma, y : A \vdash y : A}{\Gamma, \langle y \rangle : !A \vdash y : A}}{\Gamma, x : !A \vdash \text{let } \langle y \rangle = x \text{ in } y : A}}{\Gamma \vdash \lambda x. \text{let } \langle y \rangle = x \text{ in } y : !A \rightarrow A}}$$

⇒ Both use DERELICTION.

Example

- ▶ Writing:

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

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\Gamma', \langle x \rangle : !A \vdash x : A}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{DERELICTION PROMOTION}}{\Gamma \vdash \text{let } \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}$$

- ▶ Execution:

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

Use this derivation:

$$\text{ABSTRACTION} \frac{\text{LET} \frac{\text{DERELICTION} \frac{\text{VAR} \frac{}{\Gamma, y : A \vdash y : A}}{\Gamma, \langle y \rangle : !A \vdash y : A}}{\Gamma, x : !A \vdash \text{let } \langle y \rangle = x \text{ in } y : A}}{\Gamma \vdash \lambda x. \text{let } \langle y \rangle = x \text{ in } y : !A \rightarrow A}$$

⇒ 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!

$$\text{SOFTPROMOTION} \quad \frac{\Gamma \vdash t : A}{\langle \Gamma \rangle \vdash \langle t \rangle : !A}$$

Soft promotion

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

Idea

Compose DERELICTION and PROMOTION!

$$\frac{\text{SOFTPROMOTION} \quad \Gamma \vdash t : A}{\langle \Gamma \rangle \vdash \langle t \rangle : !A}$$

Consequences

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

$$\text{ABSTRACTION} \frac{\frac{\dots}{\Gamma \vdash \langle t' \rangle : !A} \quad \frac{\overline{\Gamma', x : A \vdash x : A} \text{VAR}}{\Gamma, \langle x \rangle : !A \vdash \langle x \rangle : !A} \text{SOFTPROMOTION}}{\Gamma \vdash \text{let} \langle x \rangle = \langle t' \rangle \text{ in } \langle x \rangle : !A}$$

Language behavior

$$\begin{array}{c} \text{VAR} \\ \hline \Gamma, x : A \vdash x : A \end{array} \qquad \begin{array}{c} \text{ABSTRACTION} \\ \Gamma, x : A \vdash t : B \\ \hline \Gamma \vdash \lambda x. t : A \rightarrow B \end{array} \qquad \begin{array}{c} \text{APPLICATION} \\ \Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash t' : A \\ \hline \Gamma \vdash t t' : B \end{array}$$

$$\begin{array}{c} \text{LET} \\ \Gamma \vdash t' : !A \quad \Gamma, \langle x \rangle : !A \vdash t : B \\ \hline \Gamma \vdash \text{let} \langle x \rangle = t' \text{ in } t : B \end{array} \qquad \begin{array}{c} \text{SOFTPROMOTION} \\ x_i : A_i \vdash t : B \\ \hline \Gamma, \langle x_i \rangle : !A_i \vdash \langle t \rangle : !B \end{array} \qquad \begin{array}{c} \text{DERELICTION} \\ \hline \Gamma, \langle x \rangle : !A \vdash x : A \end{array}$$

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

$$\begin{array}{c} \text{VAR} \\ \hline \Gamma, x : A \vdash x : A \end{array} \qquad \begin{array}{c} \text{ABSTRACTION} \\ \Gamma, x : A \vdash t : B \\ \hline \Gamma \vdash \lambda x. t : A \rightarrow B \end{array} \qquad \begin{array}{c} \text{APPLICATION} \\ \Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash t' : A \\ \hline \Gamma \vdash t t' : B \end{array}$$

$$\begin{array}{c} \text{LET} \\ \Gamma \vdash t' : !A \quad \Gamma, \langle x \rangle : !A \vdash t : B \\ \hline \Gamma \vdash \text{let} \langle x \rangle = t' \text{ in } t : B \end{array} \qquad \begin{array}{c} \text{SOFTPROMOTION} \\ x_i : A_i \vdash t : B \\ \hline \Gamma, \langle x_i \rangle : !A_i \vdash \langle t \rangle : !B \end{array} \qquad \begin{array}{c} \text{DERELICTION} \\ \hline \Gamma, \langle x \rangle : !A \vdash x : A \end{array}$$

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'

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

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:

$$\frac{}{(\lambda x.t) t' \xrightarrow{\text{CBN}} t[x/t']}$$

$$\frac{}{\text{let } \langle x \rangle = \langle t \rangle \text{ in } t_1 \xrightarrow{\text{CBN}} t_1[x/t]}$$

$$\frac{t_1 \xrightarrow{\text{CBN}} t'_1}{t_1 t_2 \xrightarrow{\text{CBN}} t'_1 t_2}$$

$$\frac{t_2 \xrightarrow{\text{CBN}} t'_2}{\text{let } \langle x \rangle = t_2 \text{ in } t_1 \xrightarrow{\text{CBN}} \text{let } \langle x \rangle = t'_2 \text{ in } t_1}$$

ASM₂ Machine: principle

A state

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

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

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

Extended Krivine Machine

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

$$t \sim d$$

- ▶ Soundness

$$\text{if } t \xrightarrow{\text{CBN}^*} v \\ \text{then } \langle d \mid \cdot \mid \cdot \mid \cdot \rangle \xrightarrow{*} \langle d' \mid \cdot \mid e \mid \mathbf{D} \rangle$$

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

- ▶ Completeness

$$\text{if } t \xrightarrow{\text{CBN}^*} \infty \\ \text{then } \langle d \mid \cdot \mid \cdot \mid \cdot \rangle \xrightarrow{*} \infty$$

Variables

- ▶ Variables have two roles

- ▶ reference:

$$\lambda x.x$$

- ▶ execution:

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

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

$$\frac{\text{C}_{\text{VAR}} \quad \Gamma(\mathbf{n}) : A}{\Gamma; \mathbf{T} \vdash \mathbf{n} \sim \text{fetch } \mathbf{n} : A}$$

$$\frac{\text{C}_{\text{RUN}} \quad \Gamma(\langle \mathbf{n} \rangle) : !A}{\Gamma; \mathbf{T} \vdash \mathbf{n} \sim \text{run } \mathbf{n} : A}$$

Examples (1/2)

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

		·	e	D
$\xrightarrow{\text{let}}$	$(\lambda \langle x \rangle . \langle x \rangle)_e$		e	D
$\xrightarrow{\langle t \rangle}$		·	r.e	D[r → t]
$\xrightarrow{\langle x \rangle}$	stop			

Examples (1/2)

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

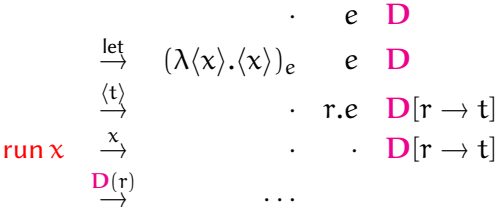
	·	e	D
$\xrightarrow{\text{let}}$	(λ⟨x⟩.⟨x⟩) _e	·	e D
$\xrightarrow{\langle t \rangle}$	·	r.e	D[r → t]
$\xrightarrow{\langle x \rangle}$	stop		

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

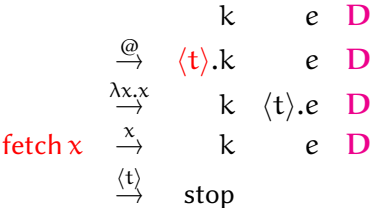
		⋮	⋮	⋮
		·	r.e	D[r → t]
run x	\xrightarrow{x}	·	·	D[r → t]
	$\xrightarrow{D(r)}$...		

Examples (2/2)

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



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



Correctness and decompilation

Correction of the compilation

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

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

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

Property

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

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