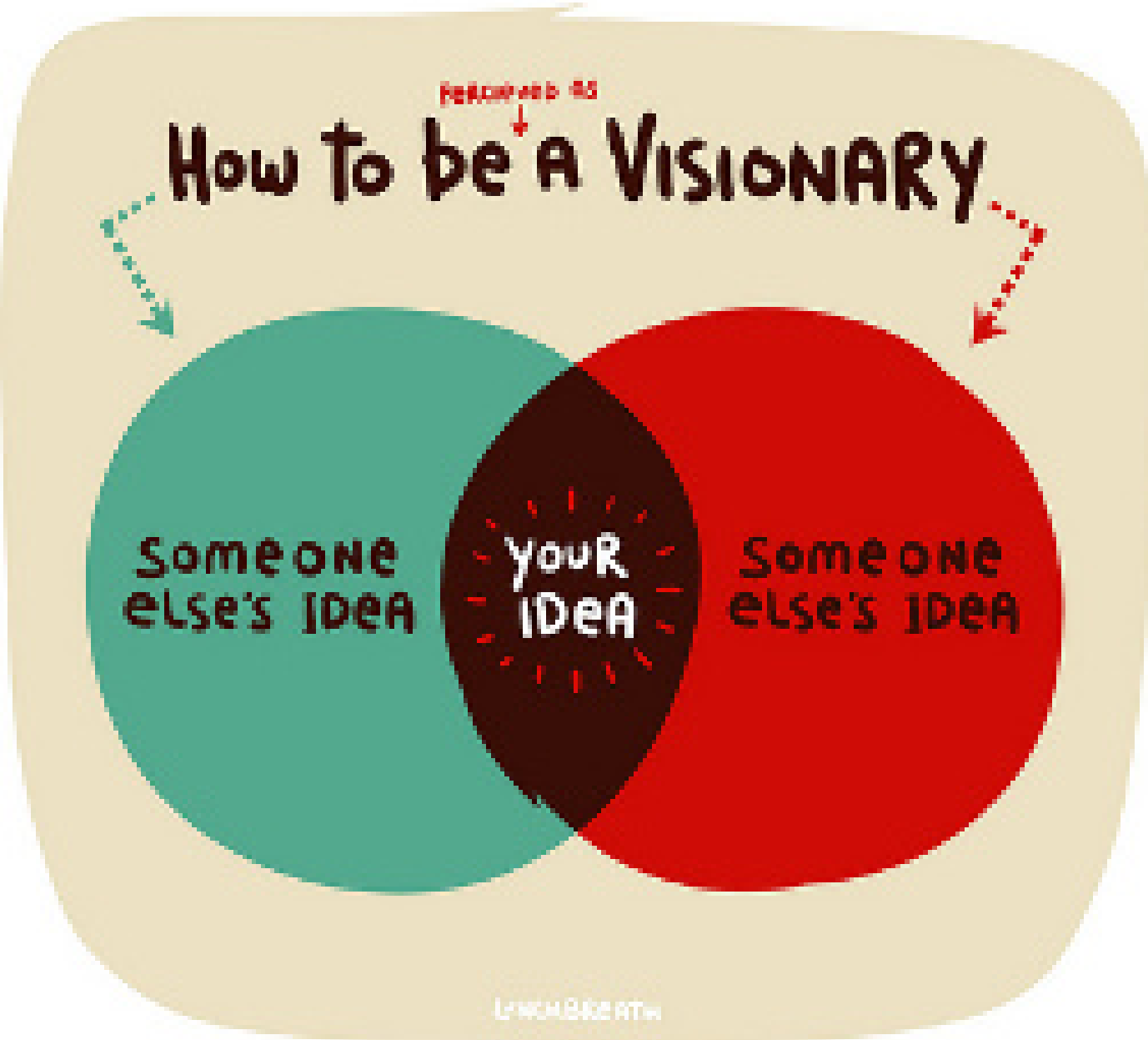
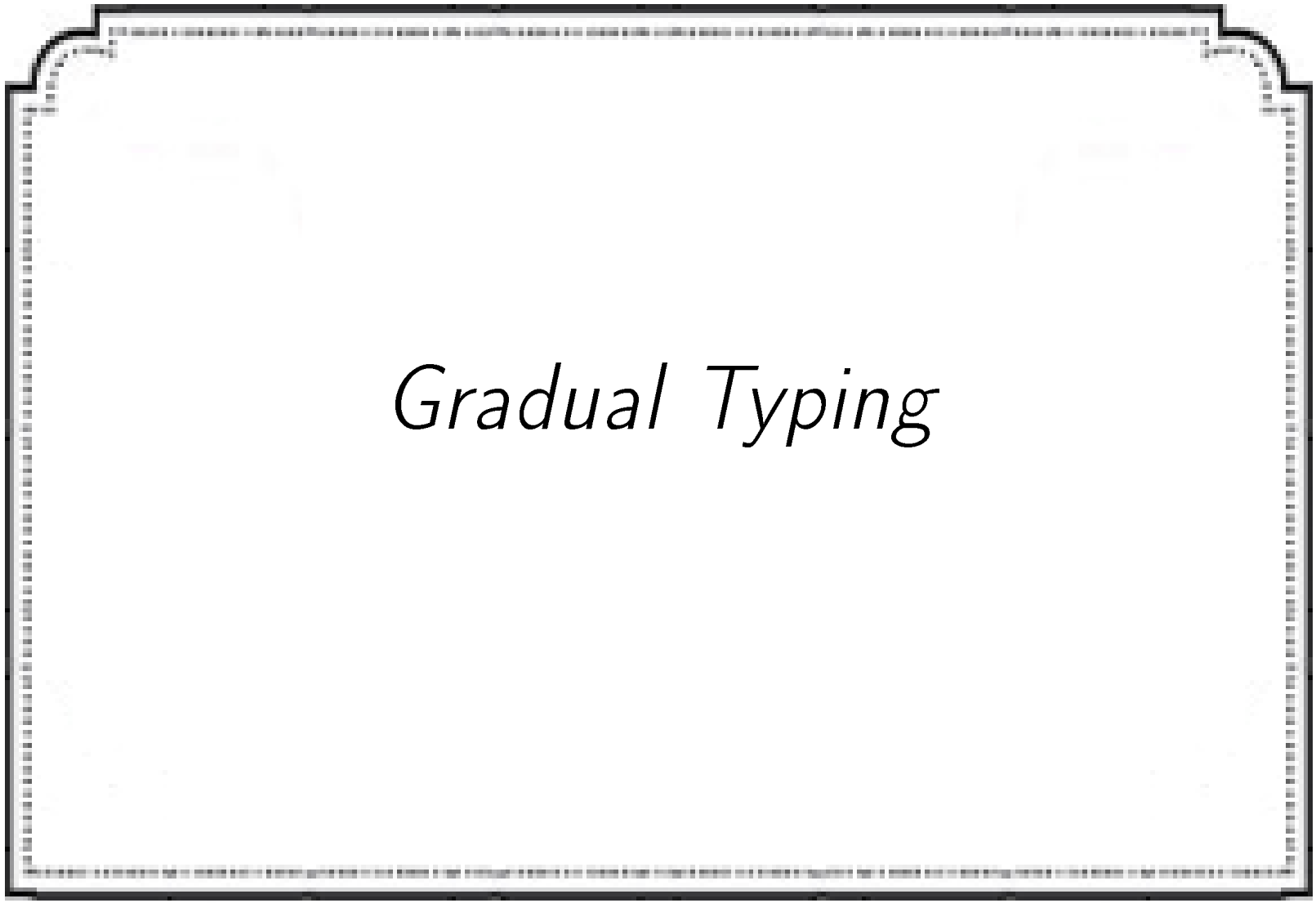
Deriving Interpretations of the Gradually-Typed Lambda Calculus

Álvaro García-Pérez Pablo Nogueira Ilya Sergey

San Diego, January 2014


How to be A VISIONARY
SOMEONE ELSE'S IDEA
YOUR IDEA
SOMEONE ELSE'S IDEA


Gradual Typing

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \rightarrow T \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\ e
\end{array}
$$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \rightarrow T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell
\end{array}
$$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_1} 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \rightarrow T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \rightarrow c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Int}! \rangle 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \rightarrow T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \rightarrow c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* 4$

$\langle \texttt{Bool} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_1} 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* 4$

$\langle \texttt{Bool} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Int}! \rangle 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* 4$

$\langle \texttt{Bool}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Core language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ of the gradually-typed lambda calculus (Siek, Garcia and Taha 2009)

$$
\begin{array}{rcl}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \rightarrow T \mid \texttt{Dyn} \\
e & ::= & k \mid op\ e \mid \texttt{if}\ cnd\ thn\ els \mid \\
 & & x \mid \lambda x : T.e \mid e\,e \mid \langle S \Leftarrow T \rangle^{\ell} e \mid \texttt{Blame}\ \ell \mid \langle c \rangle e \\
c & ::= & \iota \mid T! \mid T?^{\ell} \mid c \rightarrow c \mid c\,;c \mid \texttt{Fail}^{\ell}
\end{array}
$$

$\langle \texttt{Int}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* 4$

$\langle \texttt{Bool}?^{\ell_2} \rangle \langle \texttt{Int}! \rangle 4 \longmapsto^* \texttt{Blame}\ \ell_2$

(Siek, Garcia, and Taha 2009). Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium Systems*.

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

$$(\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x))\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_3} 1)$$

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

$$(\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x))\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_3} 1)$$

$$\longmapsto^* \quad \texttt{Blame}\ \ell_?$$

Who should be blamed?

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

$$(\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x))\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_3} 1)$$

$$\longmapsto^* \quad \texttt{Blame}\ \ell_?$$

Who should be blamed?

Two axis:

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

$$(\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x))\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_3} 1)$$

$$\longmapsto^* \quad \texttt{Blame}\ \ell_?$$

Who should be blamed?

Two axis:

- When should a cast be checked? (*eager* **E** or *lazy* **L** error detection).

$$\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x)$$

Type consistency relation: *deep* versus *shallow*.

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

$$(\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x))\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{\ell_3} 1)$$

$$\longmapsto^* \quad \texttt{Blame}\ \ell_?$$

Who should be blamed?

Two axis:

- When should a cast be checked? (*eager* **E** or *lazy* **L** error detection).

  $$\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x)$$

  Type consistency relation: *deep* versus *shallow*.
- How should functions be upcast? (*downcast* **D** or *upcast-downcast* **UD** blame tracking).

  $$\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \to \texttt{Dyn} \rangle^{\ell_2} \langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle^{\ell_1} (\lambda x : \texttt{Bool}.x)$$

  Subtyping relation: arrow types are subtypes $\texttt{Dyn} \to \texttt{Dyn}$.

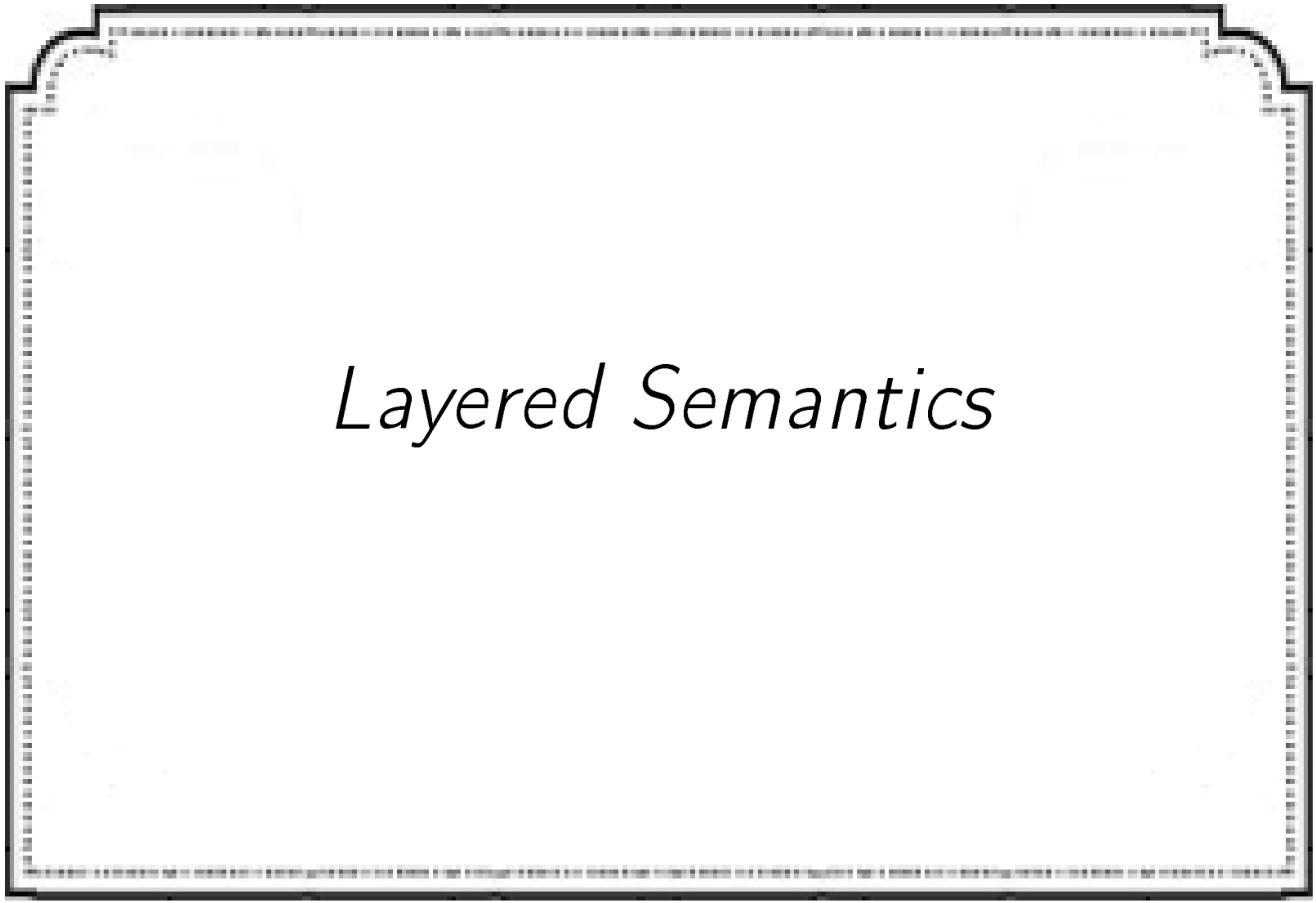
Choice of dynamic semantics (Siek, Garcia and Taha 2009)

ED	EUD
LD	LUD

Choice of dynamic semantics (Siek, Garcia and Taha 2009)

ED	EUD
LD	LUD

The eager-downcast semantics (**ED**) provides thorough error detection (deep consistency) and intuitive blame tracking (failure is produced by downcast).


Layered Semantics

Semantics for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ (Siek, Garcia and Taha 2009)

The reduction semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is *layered*:

$$(\lambda x : T.e)v \longrightarrow [x/v]e \qquad (\beta)$$

$$op\ n \longrightarrow \delta(op, n) \qquad (\delta)$$

$$\texttt{if}\ k\ e_1\ e_2 \longrightarrow \begin{cases} e_1 & \text{if } k = \texttt{true} \\ e_2 & \text{if } k = \texttt{false} \end{cases} \qquad (\text{If})$$

$$\langle c\rangle v \longrightarrow \langle c'\rangle v \quad \text{if } c \longmapsto_{\text{ED}} c' \qquad (\text{StepCst})$$

$$\langle \iota\rangle s \longrightarrow s \qquad (\text{IdCst})$$

$$\langle d\rangle\langle \overline{c}\rangle s \longrightarrow \langle \overline{c}\,; d\rangle s \qquad (\text{CmpCst})$$

$$\langle \tilde{c} \rightarrow \tilde{d}\rangle s\, v \longrightarrow \langle \tilde{d}\rangle(s\,\langle \tilde{c}\rangle v) \qquad (\text{AppCst})$$

$$\langle \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailCst})$$

$$\langle (\tilde{c} \rightarrow \tilde{d})\,; \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailFC})$$

Semantics for $\lambda^{\langle\cdot\rangle}_{\to}$ (Siek, Garcia and Taha 2009)

The reduction semantics of $\lambda^{\langle\cdot\rangle}_{\to}$ is *layered*:

$$(\lambda x : T.e)v \longrightarrow [x/v]e \qquad (\beta)$$

$$op\ n \longrightarrow \delta(op, n) \qquad (\delta)$$

$$\texttt{if}\ k\ e_1\ e_2 \longrightarrow \begin{cases} e_1 & \text{if } k = \texttt{true} \\ e_2 & \text{if } k = \texttt{false} \end{cases} \qquad (\text{If})$$

$$\langle c\rangle v \longrightarrow \langle c'\rangle v \quad \text{if } c \longmapsto_{\mathsf{ED}} c' \qquad (\text{StepCst})$$

$$\langle \iota\rangle s \longrightarrow s \qquad (\text{IdCst})$$

$$\langle d\rangle\langle \overline{c}\rangle s \longrightarrow \langle \overline{c}\,; d\rangle s \qquad (\text{CmpCst})$$

$$\langle \tilde{c} \to \tilde{d}\rangle s\, v \longrightarrow \langle \tilde{d}\rangle(s\,\langle \tilde{c}\rangle v) \qquad (\text{AppCst})$$

$$\langle \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailCst})$$

$$\langle (\tilde{c} \to \tilde{d})\,; \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailFC})$$

Semantics for **ED** (Siek, Garcia and Taha 2009)

$$I_1!\,;I_2?^\ell \longrightarrow_{\mathsf{ED}} \langle\!\langle I_2 \Leftarrow I_1 \rangle\!\rangle^\ell \qquad \text{(InOut)}$$

$$(\tilde{c}_1 \to \tilde{c}_2)\,;(\tilde{d}_1 \to \tilde{d}_2) \longrightarrow_{\mathsf{ED}} ((\tilde{d}_1\,;\tilde{c}_1) \to (\tilde{c}_2\,;\tilde{d}_2)) \qquad \text{(Arr)}$$

$$\iota\,;\hat{c} \longrightarrow_{\mathsf{ED}} \hat{c} \qquad \text{(IdL)}$$

$$\hat{c}\,;\iota \longrightarrow_{\mathsf{ED}} \hat{c} \qquad \text{(IdR)}$$

$$\mathtt{Fail}^\ell\,;\hat{c} \longrightarrow_{\mathsf{ED}} \mathtt{Fail}^\ell \qquad \text{(FailCo)}$$

$$I!\,;\mathtt{Fail}^\ell \longrightarrow_{\mathsf{ED}} \mathtt{Fail}^\ell \qquad \text{(InFail)}$$

$$(\mathtt{Fail}^\ell \to \hat{d}) \longrightarrow_{\mathsf{ED}} \mathtt{Fail}^\ell \qquad \text{(FailL)}$$

$$(\tilde{c} \to \mathtt{Fail}^\ell) \longrightarrow_{\mathsf{ED}} \mathtt{Fail}^\ell \qquad \text{(FailR)}$$


The Goal

Proving correction of existing implementations (Siek and Garcia 2012)

```
fun interp cast apply (e, env)
    = let val recur = interp cast apply
      in (case (e, env)
           of (CON c, env) => V (VCON c)
            | (IND n, env) => V (List.nth (env, n))
            | (PRIM (oper, e), env) => (case recur (e, env)
                                         of (V (VCON c)) => V (VCON (delta (oper, c)))
                                          | (blame as BLAME _) => blame)
            | (IFTE (cnd, thn, els), env)
              => (case recur (cnd, env)
                   of (V (VCON TRUE))    => recur (thn, env)
                    | (V (VCON FALSE))   => recur (els, env)
                    | (blame as BLAME _) => blame)
            | (APP (e1, e2), env)
              => (case recur (e1, env)
                   of (V v1)             => (case recur (e2, env)
                                              of (V v2)             => apply (v1, v2)
                                               | (blame as BLAME _) => blame)
                    | (blame as BLAME _) => blame)
            | (LAM (t1, e), env) => V (VPROC (fn v => recur (e, v :: env)))
            | (CAST (e, t1, t2, l), env) => (case recur (e, env)
                                              of (V v)  => cast (v, t1, t2, l)
                                               | (blame as BLAME _) => blame))
    end
```

(Siek and Garcia 2012). Interpretations of the gradually-typed lambda calculus. In *Proceedings of the Workshop on Scheme and Functional Programming*.

Proving correction of existing implementations (Siek and Garcia 2012)

```
fun interp cast apply (e, env)
    = let val recur = interp cast apply
      in (case (e, env)
           of (CON c, env) => V (VCON c)
            | (IND n, env) => V (List.nth (env, n))
            | (PRIM (oper, e), env) => (case recur (e, env)
                                         of (V (VCON c)) => V (VCON (delta (oper, c)))
                                          | (blame as BLAME _) => blame)
            | (IFTE (cnd, thn, els), env)
              => (case recur (cnd, env)
                   of (V (VCON TRUE))    => recur (thn, env)
                    | (V (VCON FALSE))   => recur (els, env)
                    | (blame as BLAME _) => blame)
            | (APP (e1, e2), env)
              => (case recur (e1, env)
                   of (V v1)             => (case recur (e2, env)
                                              of (V v2)             => apply (v1, v2)
                                               | (blame as BLAME _) => blame)
                    | (blame as BLAME _) => blame)
            | (LAM (t1, e), env) => V (VPROC (fn v => recur (e, v :: env)))
            | (CAST (e, t1, t2, l), env) => (case recur (e, env)
                                              of (V v)  => cast (v, t1, t2, l)
                                               | (blame as BLAME _) => blame))
      end
```

(Siek and Garcia 2012). Interpretations of the gradually-typed lambda calculus. In *Proceedings of the Workshop on Scheme and Functional Programming*.

(Siek and Garcia 2012)

Conjecture (correctness of coercion composition)

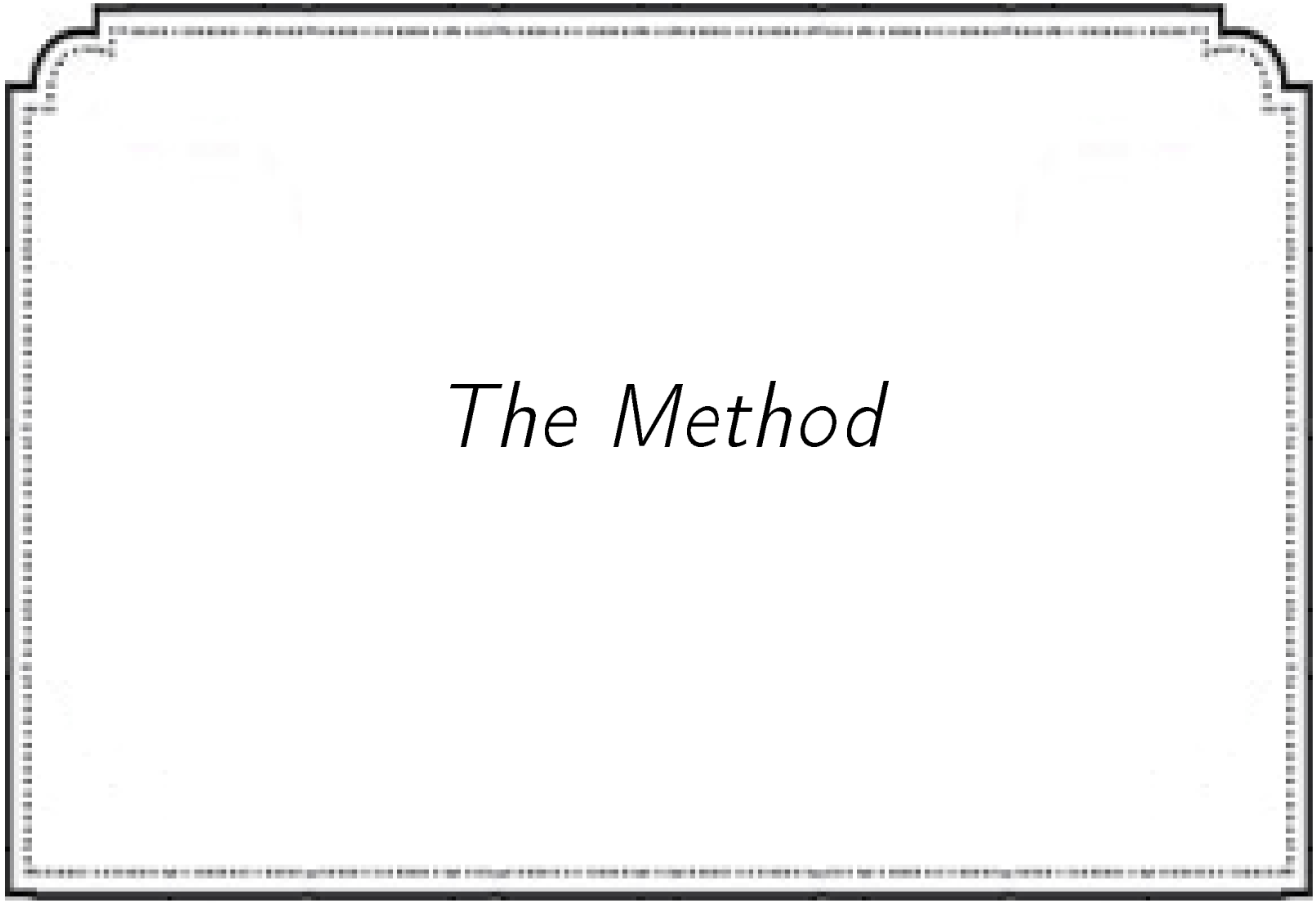
Given two well-typed coercions in normal form, $\hat{c}_1$ and $\hat{c}_2$, we have

$$\texttt{seq\_ed}(\hat{c}_1, \hat{c}_2) = \hat{c}_3 \text{ and } (\hat{c}_1 ; \hat{c}_2) \longmapsto^*_{\mathbf{ED}} \hat{c}_3$$

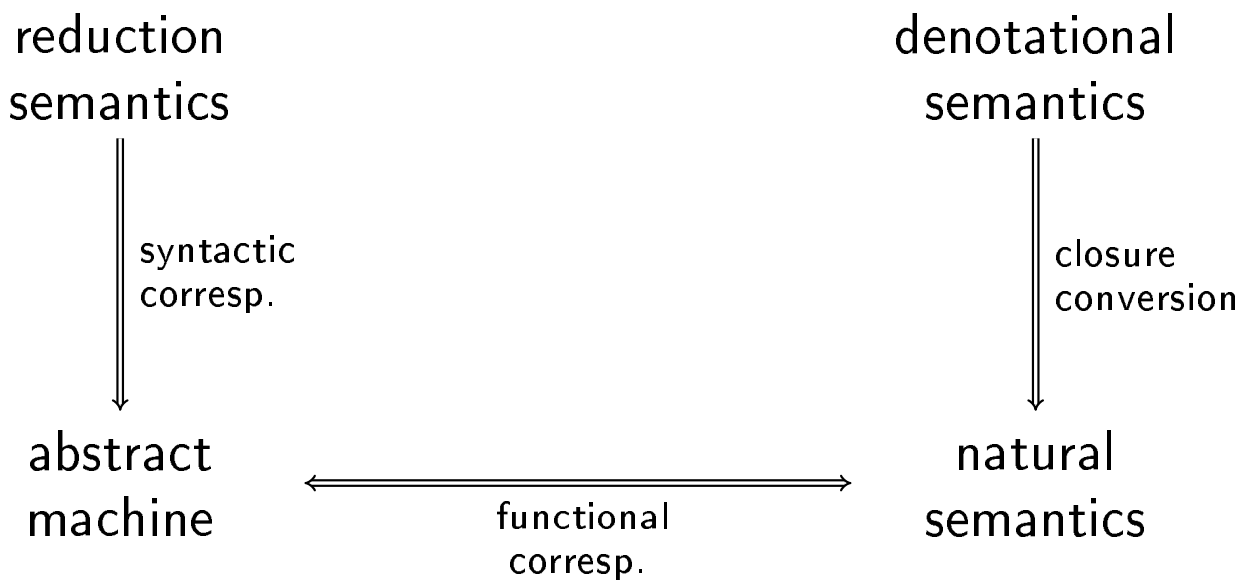
Conjecture (agreement with $\lambda^{\langle\cdot\rangle}_{\rightarrow}$)

For any well-typed program e, we have

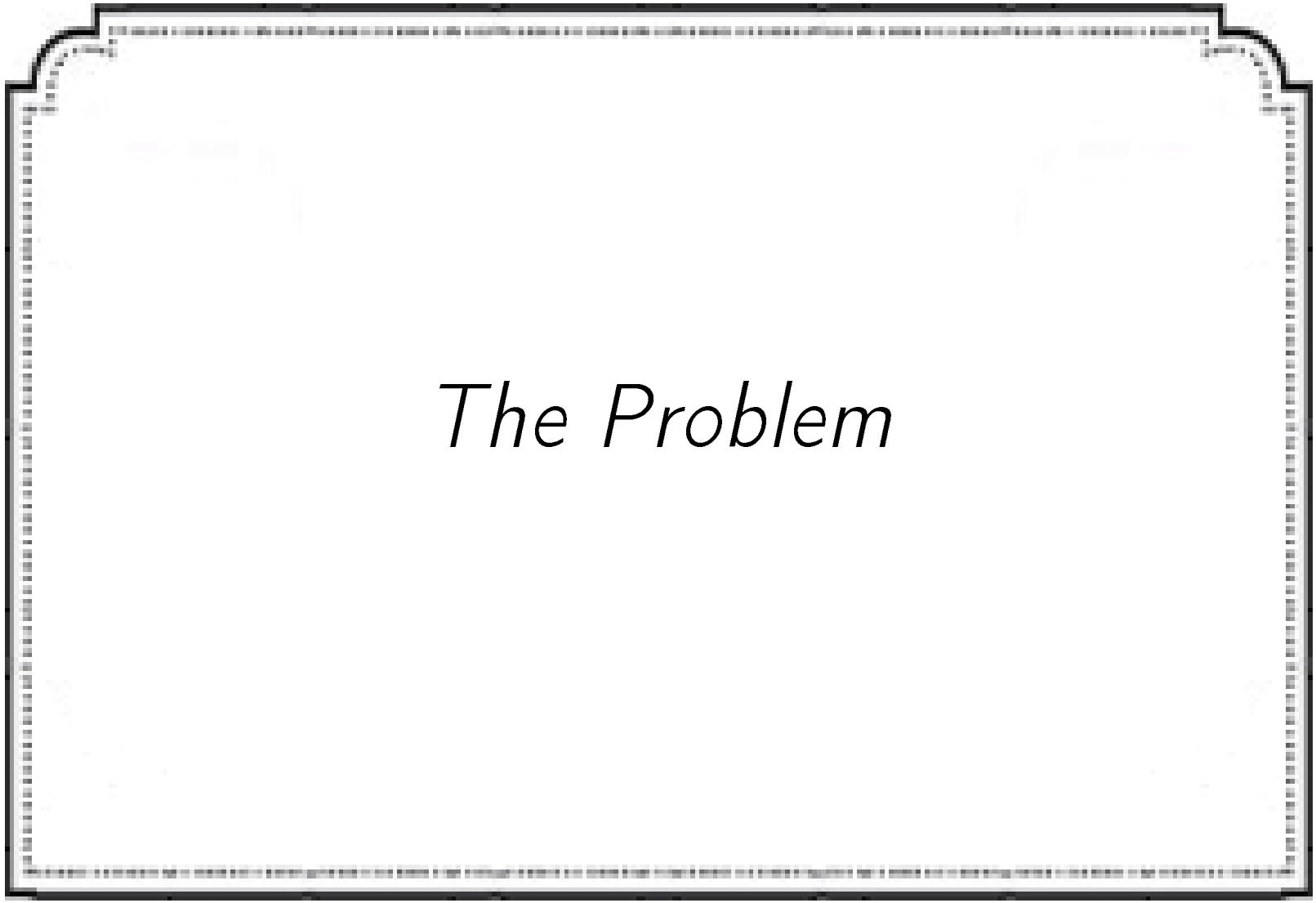
$$\texttt{interp\_ed}(e) = v \text{ iff } e \longmapsto^* v.$$


The Method

Inter-derivation of semantic artefacts (Danvy 2008)



(Danvy 2008). Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference Programming,(ICFP'08)*.


The Problem

What about other dynamic semantics?

$$(\lambda x : T.e)v \longrightarrow [x/v]e \qquad (\beta)$$

$$op\ n \longrightarrow \delta(op, n) \qquad (\delta)$$

$$\texttt{if}\ k\ e_1\ e_2 \longrightarrow \begin{cases} e_1 & \text{if } k = \texttt{true} \\ e_2 & \text{if } k = \texttt{false} \end{cases} \qquad (\text{If})$$

$$\langle c \rangle v \longrightarrow \langle c' \rangle v \quad \text{if } c \longmapsto_{\mathsf{ED}} c' \qquad (\text{StepCst})$$

$$\langle \iota \rangle s \longrightarrow s \qquad (\text{IdCst})$$

$$\langle d \rangle \langle \overline{c} \rangle s \longrightarrow \langle \overline{c}\,; d \rangle s \qquad (\text{CmpCst})$$

$$\langle \tilde{c} \to \tilde{d} \rangle s\, v \longrightarrow \langle \tilde{d} \rangle (s\, \langle \tilde{c} \rangle v) \qquad (\text{AppCst})$$

$$\langle \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailCst})$$

$$\langle (\tilde{c} \to \tilde{d})\,; \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailFC})$$

What about other dynamic semantics?

$$(\lambda x : T.e)v \longrightarrow [x/v]e \qquad (\beta)$$

$$op\ n \longrightarrow \delta(op, n) \qquad (\delta)$$

$$\texttt{if}\ k\ e_1\ e_2 \longrightarrow \begin{cases} e_1 & \text{if } k = \texttt{true} \\ e_2 & \text{if } k = \texttt{false} \end{cases} \qquad (\text{If})$$

$$\langle c \rangle v \longrightarrow \langle c' \rangle v \quad \text{if } c \longmapsto\!\!\!\!\!\rightarrow_{\text{xx}} c' \qquad (\text{StepCst})$$

$$\langle \iota \rangle s \longrightarrow s \qquad (\text{IdCst})$$

$$\langle d \rangle \langle \overline{c} \rangle s \longrightarrow \langle \overline{c}\,; d \rangle s \qquad (\text{CmpCst})$$

$$\langle \tilde{c} \to \tilde{d} \rangle s\, v \longrightarrow \langle \tilde{d} \rangle (s\, \langle \tilde{c} \rangle v) \qquad (\text{AppCst})$$

$$\langle \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailCst})$$

$$\langle (\tilde{c} \to \tilde{d})\,; \texttt{Fail}^{\ell} \rangle s \longrightarrow \texttt{Blame}\ \ell \qquad (\text{FailFC})$$

Inter-derivation for modular artefacts?

- Normal order strategy in pure λ-calculus:
  Reasoning on the shape invariant of the **layered** continuation stack (García-Pérez and Nogueira 2013).
- Eval-readback machine of (Curien 1993):
  Using **2-level CPS** to separate the eval from the readback stage (Danvy, Millikin and Munk 2013).

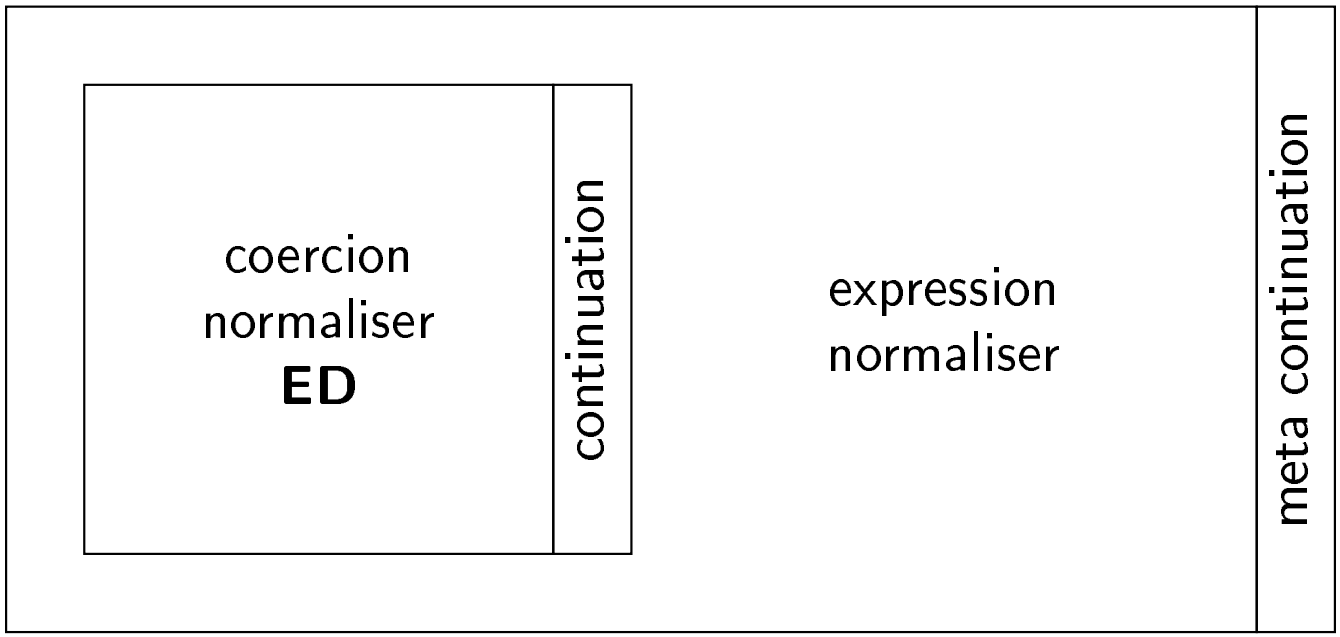
(García-Pérez and Nogueira 2013). A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. In *Symposium on Partial Evaluation and Program Manipulation (PEPM'13)*.

(Danvy, Millikin, and Munk 2013). A correspondence between full normalization by reduction and full normalization by evaluation. *A scientific meeting in honor of Pierre-Louis Curien*.

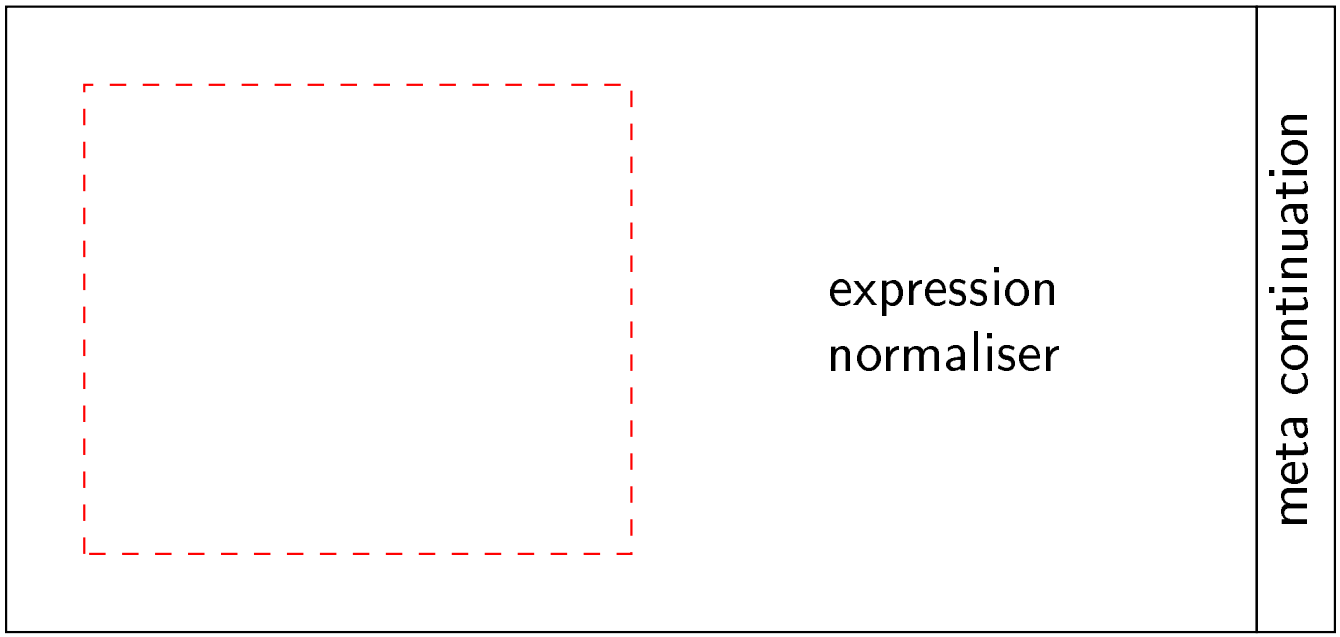
The Solution

Layered  2-CPS

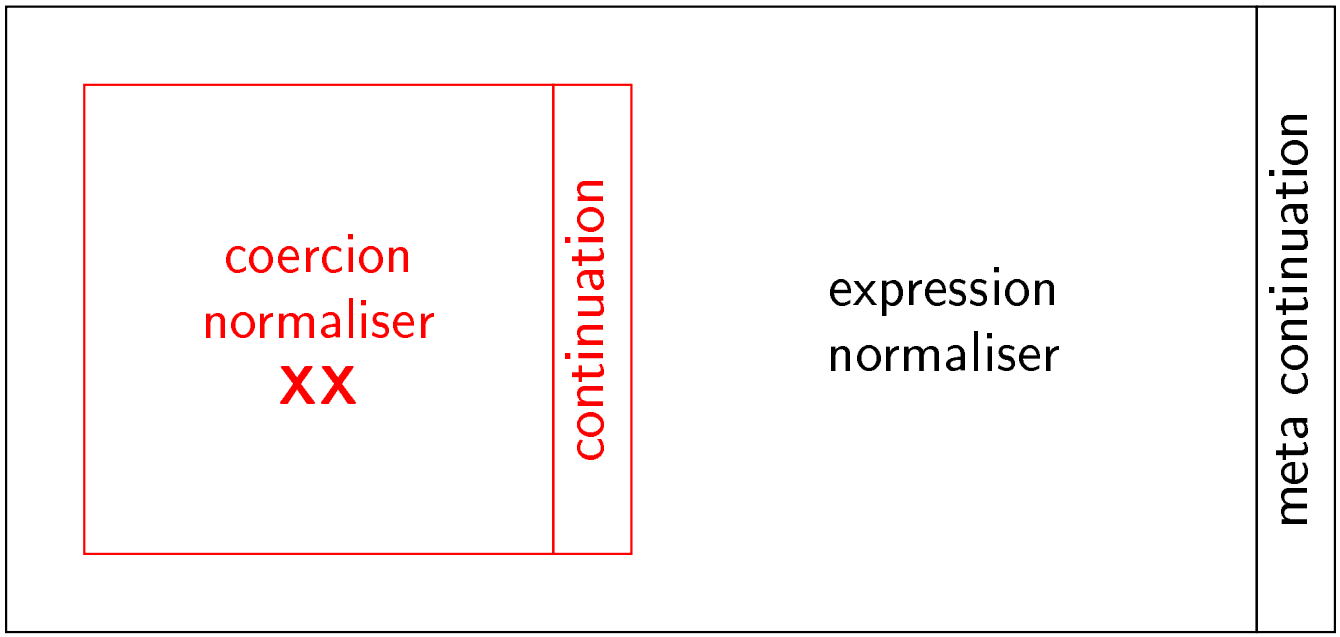
2-Level CPS artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$



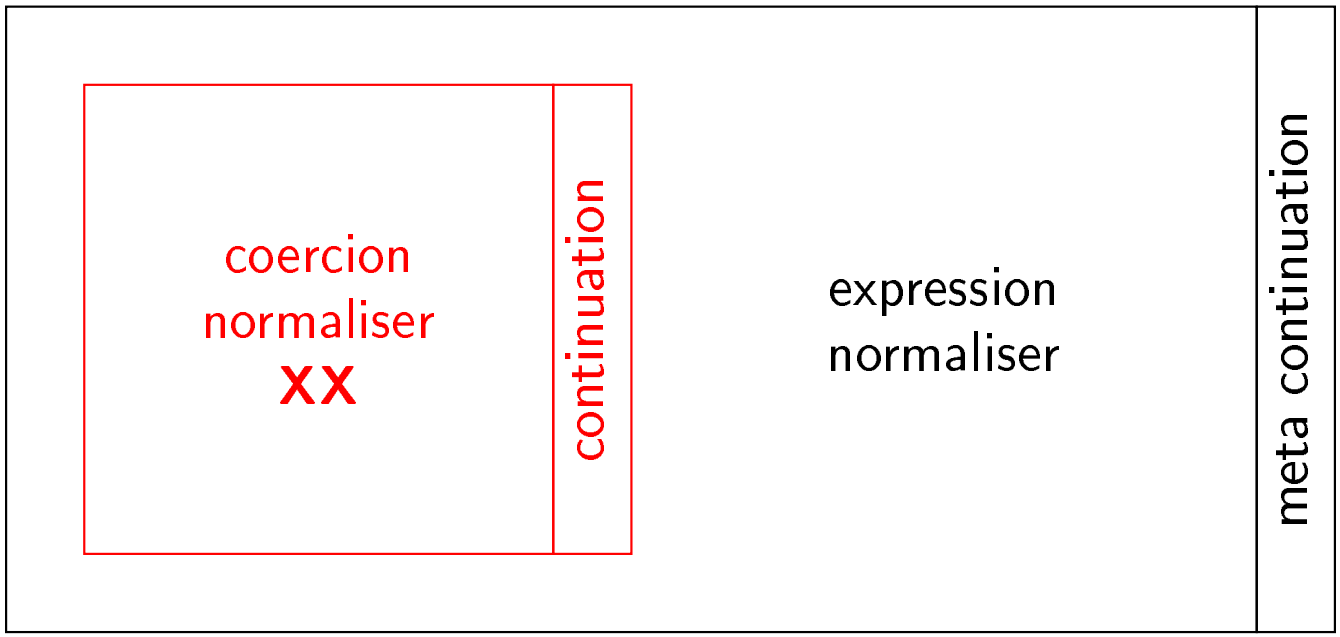
2-Level CPS artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$



2-Level CPS artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$

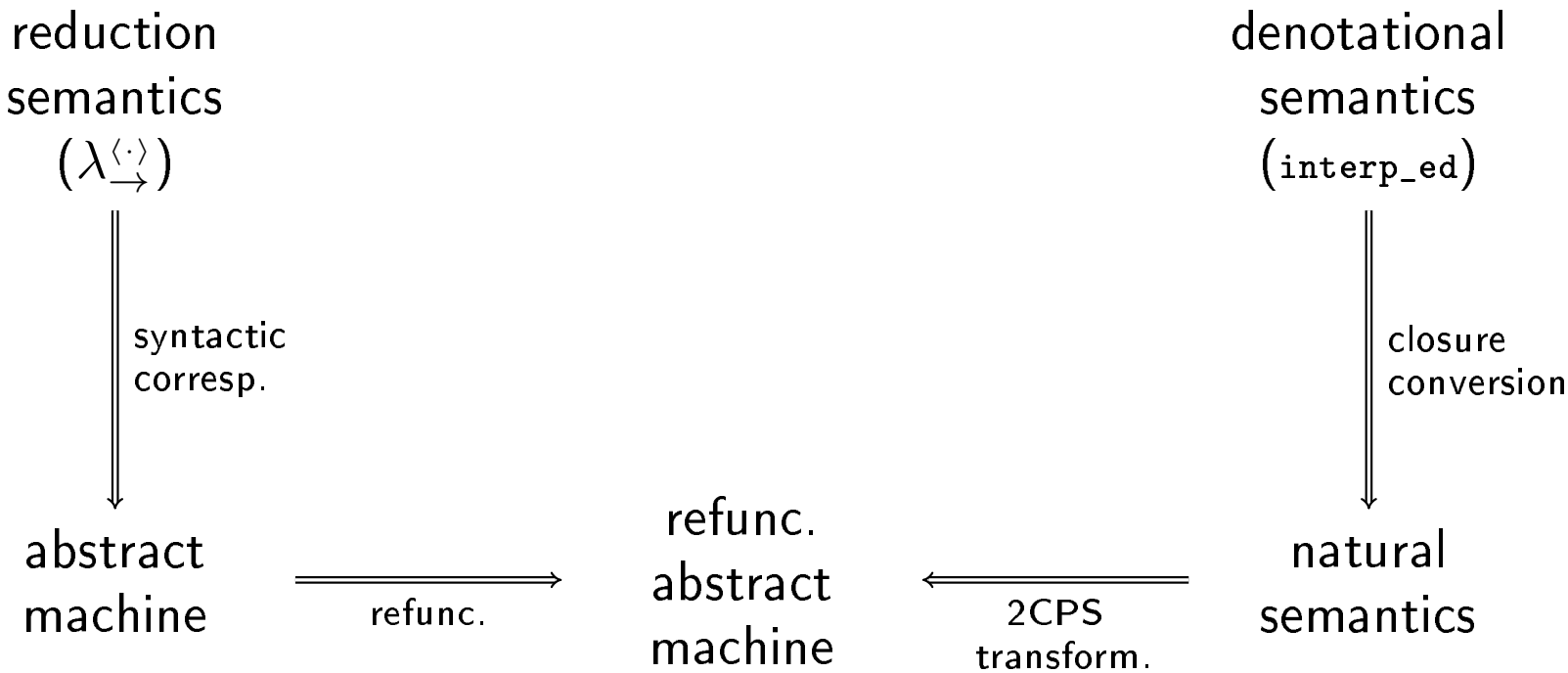


2-Level CPS artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$

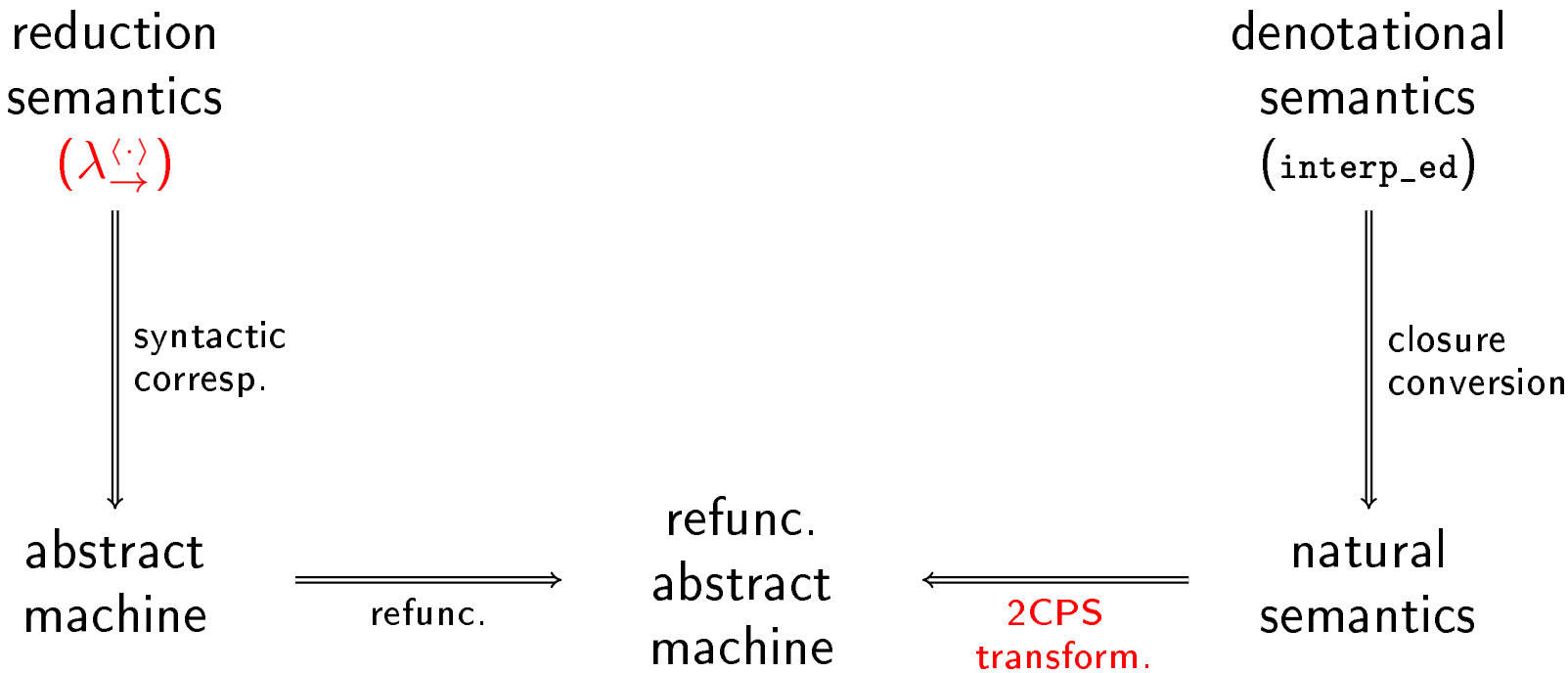


Allows the inter-derivation to be modular in the dynamic semantics!

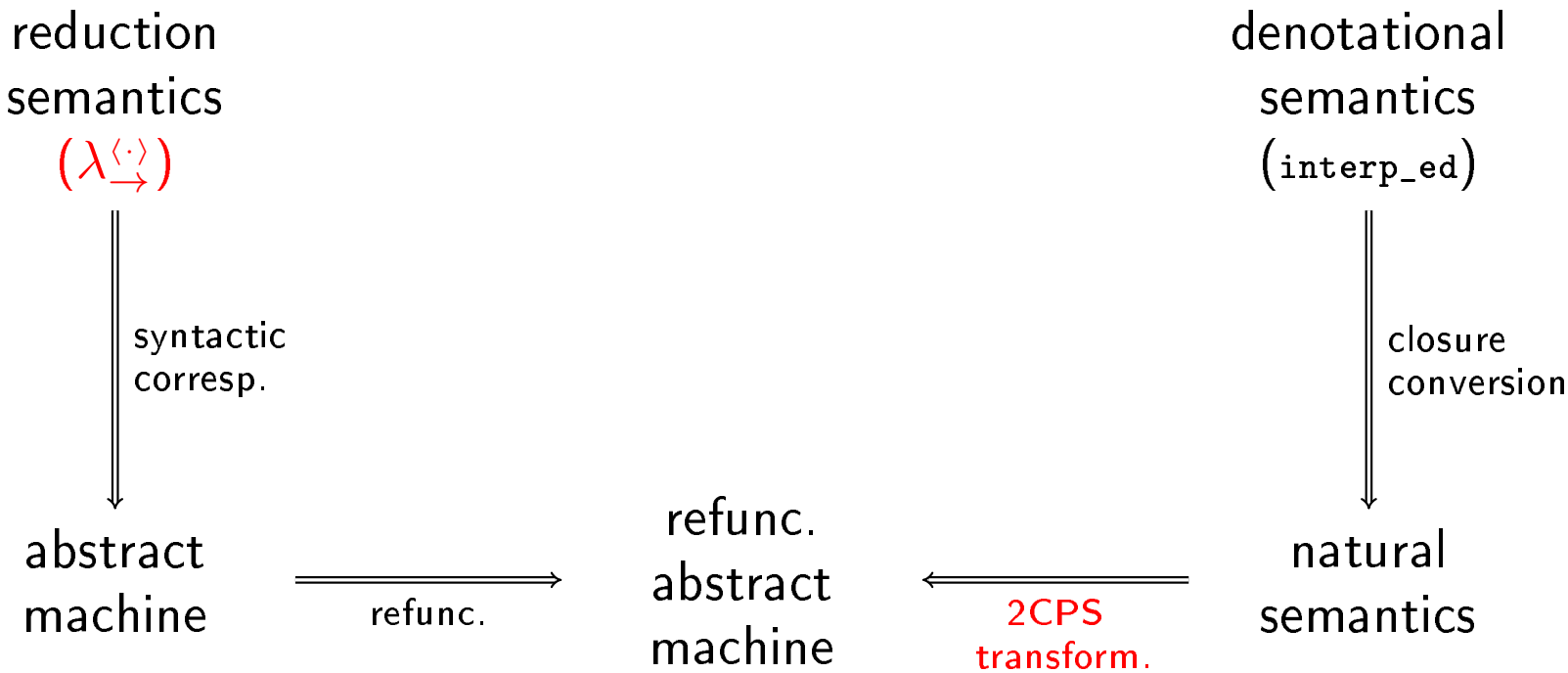
Derivation path for 2-CPS artefacts



Derivation path for 2-CPS artefacts



Derivation path for 2-CPS artefacts



Constructive proof of the correctness conjectures!

Contributions

- We prove correctness of Siek and Garcia's implementation for the **ED** semantics by program transformation, *i.e.*, *inter-derivation of semantic artefacts*.
- We showcase the use of *2-level continuation passing style* (2CPS) (Danvy, Millikin and Munk 2013) to make the operational semantics modular on the coercion's dynamic semantics.
- Correctness for other choices of dynamic semantics can be proven by reusing the machinery.
- We introduce the gradually-typed calculus of closures $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, which simulates $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

On-line implementation at
`http://babel.ls.fi.upm.es/~agarcia/papers/Gradual`

Conclusions

- Implementations of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ are proven correct.
- Inter-derivation is an effective but underused technique.
- Layered semantics are everywhere:
  - Ahead Machine (Paolini and Ronchi della Roccca 1999).
  - Strong Reduction (Grégorie and Leroy 2002).
  - Hybrid strategies (Sestoft 2002).
  - Full-reducing Krivine Abstract Machine (Crégut 2007) (García-Pérez, Nogueira, Moreno-Navarro 2013).
  - Outermost reduction (Danvy and Johansenn 2013).

  Inter-derivation works for them!
- 2CPS techniques introduce conceptual overhead, but they deliver a high return in some settings.

Conclusions

- Implementations of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ are proven correct.
- Inter-derivation is an effective but underused technique.
- Layered semantics are everywhere:
  - Ahead Machine (Paolini and Ronchi della Roccca 1999).
  - Strong Reduction (Grégorie and Leroy 2002).
  - Hybrid strategies (Sestoft 2002).
  - Full-reducing Krivine Abstract Machine (Crégut 2007) (García-Pérez, Nogueira, Moreno-Navarro 2013).
  - Outermost reduction (Danvy and Johansenn 2013).

  Inter-derivation works for them!
- 2CPS techniques introduce conceptual overhead, but they deliver a high return in some settings.

References I

Crégut, P. (2007).
Strongly reducing variants of the Krivine abstract machine.
Higher-Order and Symbolic Computation, 20(3):209–230.

Curien, P.-L. (1993).
Categorical Combinators, Sequential Algorithms and Functional Programming.
Progress in Theoretical Computer Science. Birkhaüser.

Danvy, O. (2008).
Defunctionalized interpreters for programming languages.
In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*.

Danvy, O. and Johannsen, J. (2013).
From outermost reduction semantics to abstract machine.
In *Pre-Proceedings of 23rd Symposium on Logic-Based Program Synthesis and Transformation*, pages 145–160.

Danvy, O., Millikin, K., and Munk, J. (2013).
A correspondence between full normalization by reduction and full normalization by evaluation.
A scientific meeting in honor of Pierre-Louis Curien.

References II

García-Pérez, Á. and Nogueira, P. (2013).
A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers.
In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM*.

García-Pérez, A., Nogueira, P., and Moreno-Navarro, J. J. (2013).
Deriving the *Full-Reducing* krivine machine from the small-step operational semantics of normal order.
In *Proceedings of the 15th International Symposium on Principles and Practice of Declarative Programming*.

Grégoire, B. and Leroy, X. (2002).
A compiled implementation of strong reduction.
In *Proceedings of International Conference on Functional Programming*, pages 235–246.

Paolini, L. and Ronchi della Rocca, S. (1999).
Call-by-value solvability.
Informatique Théorique et Applications, 33(6):507–534.

Sestoft, P. (2002).
Demonstrating lambda calculus reduction.
In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 420–435. Springer.

References III

Siek, J. G. and Garcia, R. (2012).
Interpretations of the gradually-typed lambda calculus.
In *Proceedings of the Workshop on Scheme and Functional Programming*.

Siek, J. G., Garcia, R., and Taha, W. (2009).
Exploring the design space of higher-order casts.
In *Proceedings of the 18th European Symposium on Programming Languages and Systems*.

Gradually-typed lambda calculus ($\lambda^{?}_{\rightarrow}$) (Siek and Taha 2006)

$$
\begin{array}{rcl}
T & ::= & * \mid ? \mid T \rightarrow T \\
e & ::= & k \mid x \mid \lambda x : T.e \mid e\, e
\end{array}
$$

Translation for the **ED** semantics

$$
\begin{array}{rcll}
\langle\!\langle B \Leftarrow B \rangle\!\rangle^{\ell} & = & \iota & \\
\langle\!\langle B_2 \Leftarrow B_1 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \text{if } B_1 \neq B_2 \\
\langle\!\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & \iota & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow B \rangle\!\rangle^{\ell} & = & B! & \\
\langle\!\langle B \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & B?^{\ell} & \\
\langle\!\langle T_1 \to T_2 \Leftarrow B \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \\
\langle\!\langle B \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \\
\langle\!\langle T_1 \to T_2 \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathit{mkArr}(\langle\!\langle S_1 \Leftarrow T_1 \rangle\!\rangle^{\ell}, \langle\!\langle T_2 \Leftarrow S_2 \rangle\!\rangle^{\ell}) & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & S_1 \to S_2! & \\
\langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & T_1 \to T_2?^{\ell} & \\
\mathit{mkArr}(\mathtt{Fail}^{\ell}, \hat{c}_2) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \mathtt{Fail}^{\ell}) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \hat{c}_2) & = & \hat{c}_1 \to \hat{c}_2 & \text{otherwise}
\end{array}
$$

Higher-order casts

Not possible to check all the casts immediately:

$$\langle \texttt{Int} \to \texttt{Int} \Leftarrow \texttt{Int} \to \texttt{Dyn} \rangle \\ (\lambda x : \texttt{Int}.\ \texttt{if}\ (x > 0)\ \texttt{then}\ (\langle \texttt{Dyn} \Leftarrow \texttt{Bool} \rangle \texttt{true})\ \texttt{else}\ (\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle 2))$$

Therefore, casts could be...

- ...*safe* (respect subtyping).
- ...*unsafe* (might fail at runtime).
- ...*inadmissible* (don't respect static typing).

Existing Implementations (Siek and Garcia 2012)

```
fun seq_ed (ID, c2)                                        = c2
  | seq_ed (c1, ID)                                        = c1
  | seq_ed (INJ t1, PROJ (t2, l))                          = mk_coerce_ed (t1, t2, l)
  | seq_ed (ARR (c11, c12), ARR (c21, c22))                = mk_arrow_eager (seq_ed (c21, c11),
                                                                             seq_ed (c12, c22))
  | seq_ed (FAIL l, c2)                                    = FAIL l
  | seq_ed (INJ t, FAIL l)                                 = FAIL l
  | seq_ed (SEQ (c11, c12), c2)                            = seq_ed (c11, seq_ed (c12, c2))
  | seq_ed (PROJ (t, l), SEQ (ARR (c21, c22), c23)) = SEQ (PROJ (t, l), c2)
  | seq_ed (c1, SEQ (c21, c22))                            = seq_ed (seq_ed (c1, c21), c22)
  | seq_ed (c1, c2)                                        = SEQ (c1, c2))

fun apply_coercion_ed (c, v)
    = (case v of (VCOER (c1, v1)) => mk_cast_eager (seq_ed (c1, c), v1)
               | _                => mk_cast_eager (c, v))

fun apply_cast_ed (v, t1, t2, l) = apply_coercion_ed (mk_coerce_ed (t1, t2, l), v)

fun apply_ed (VCOER (ARR (c1, c2), f1), v)
    = (case apply_coercion (c1, v)
        of (V v1)    => (case recur (f1, v1) of (V v2)    => apply_coercion (c2, v2)
                                              | (BLAME l) => BLAME l)
         | (BLAME l) => BLAME l)
  | apply_ed (VPROC proc, v)
    = proc v
```

Existing Implementations (Siek and Garcia 2012)

```
fun seq_ed (ID, c2)                                        = c2
  | seq_ed (c1, ID)                                        = c1
  | seq_ed (INJ t1, PROJ (t2, l))                          = mk_coerce_ed (t1, t2, l)
  | seq_ed (ARR (c11, c12), ARR (c21, c22))                = mk_arrow_eager (seq_ed (c21, c11),
                                                                             seq_ed (c12, c22))
  | seq_ed (FAIL l, c2)                                    = FAIL l
  | seq_ed (INJ t, FAIL l)                                 = FAIL l
  | seq_ed (SEQ (c11, c12), c2)                            = seq_ed (c11, seq_ed (c12, c2))
  | seq_ed (PROJ (t, l), SEQ (ARR (c21, c22), c23)) = SEQ (PROJ (t, l), c2)
  | seq_ed (c1, SEQ (c21, c22))                            = seq_ed (seq_ed (c1, c21), c22)
  | seq_ed (c1, c2)                                        = SEQ (c1, c2))

fun apply_coercion_ed (c, v)
    = (case v of (VCOER (c1, v1)) => mk_cast_eager (seq_ed (c1, c), v1)
               | _                => mk_cast_eager (c, v))

fun apply_cast_ed (v, t1, t2, l) = apply_coercion_ed (mk_coerce_ed (t1, t2, l), v)

fun apply_ed (VCOER (ARR (c1, c2), f1), v)
    = (case apply_coercion (c1, v)
        of (V v1)    => (case recur (f1, v1) of (V v2)    => apply_coercion (c2, v2)
                                              | (BLAME l) => BLAME l)
         | (BLAME l) => BLAME l)
  | apply_ed (VPROC proc, v)
    = proc v
```

2-Level CPS

```
datatype cont = C0
              | C1 of coercion * cont
              | C2 of normal * cont
              | C3 of normal * cont
              | C4 of normal * cont
              | C5 of coercion * cont
              | C6 of normal * cont

datatype meta_cont = MC0
                   | MC1 of operator * meta_cont
                   | MC2 of closure * closure * meta_cont
                   | MC3 of closure * meta_cont
                   | MC4 of value * meta_cont
                   | MC5 of coercion * meta_cont
```

2-Level CPS

```
datatype coer_redex = CIDL of normal
                    | CIDR of coercion
                    | CINOUT of typ * typ * label
                    | CARR of normal * normal
                    | CFAILCO of label
                    | CINFAIL of label
                    | CFAILL of label
                    | CFAILR of label
datatype coer_decomp = DEC of coer_redex * cont
                     | RES of normal
datatype exp_redex = EBETA of exp * env * value
                   | EDELTA of operator * constant
                   | EIFTE of constant * value * value
                   | ESTEPCST of coercion * simp_val
                   | EIDCST1 of value
                   | ECMPCST of coercion * normal * simp_val
                   | EAPPCST of normal * simp_val * value
                   | EFAILCST of label
                   | EFAILFC of label
datatype exp_decomp = MDEC of exp_redex * meta_cont
                    | MRES of result
```

2-Level CPS

```
datatype coer_redex = CIDL of normal
                    | CIDR of coercion
                    | CINOUT of typ * typ * label
                    | CARR of normal * normal
                    | CFAILCO of label
                    | CINFAIL of label
                    | CFAILL of label
                    | CFAILR of label
datatype coer_decomp = DEC of coer_redex * cont
                     | RES of normal
datatype exp_redex = EBETA of exp * env * value
                   | EDELTA of operator * constant
                   | EIFTE of constant * value * value
                   | ESTEPCST of coercion * simp_val
                   | EIDCST1 of value
                   | ECMPCST of coercion * normal * simp_val
                   | EAPPCST of normal * simp_val * value
                   | EFAILCST of label
                   | EFAILFC of label
datatype exp_decomp = MDEC of exp_redex * meta_cont
                    | MRES of result
```

2-Level CPS

```
datatype coer_redex = CIDL of normal
                    | CIDR of coercion
                    | CINOUT of typ * typ * label
                    | CARR of normal * normal
                    | CFAILCO of label
                    | CINFAIL of label
                    | CFAILL of label
                    | CFAILR of label
datatype coer_decomp = DEC of coer_redex * cont
                     | RES of normal
datatype exp_redex = EBETA of exp * env * value
                   | EDELTA of operator * constant
                   | EIFTE of constant * value * value
                   | ESTEPCST of coer_decomp * simp_val
                   | EIDCST1 of value
                   | ECMPCST of coercion * normal * simp_val
                   | EAPPCST of normal * simp_val * value
                   | EFAILCST of label
                   | EFAILFC of label
datatype exp_decomp = MDEC of exp_redex * meta_cont
                    | MRES of result
```

Gradually-typed calculus of closures

$$
\begin{array}{lrcl}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{cl}) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \textbf{prim}\ op\ \mathsf{cl} \mid \textbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl}
\end{array}
$$

$$((\lambda x : \mathtt{Dyn}.x)(\langle \mathtt{Int!} \rangle 1))[\epsilon]$$

Gradually-typed calculus of closures

$$
\begin{array}{lrcl}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{cl}) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \textbf{prim}\ op\ \mathsf{cl} \mid \textbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl}
\end{array}
$$

$$
\begin{array}{rl}
 & ((\lambda x : \mathtt{Dyn}.x)(\langle \mathtt{Int!} \rangle 1))[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot (\langle \mathtt{Int!} \rangle 1)[\epsilon]
\end{array}
$$

Gradually-typed calculus of closures

$$
\begin{array}{llll}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{cl}) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \textbf{prim}\ op\ \mathsf{cl} \mid \textbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl}
\end{array}
$$

$$
\begin{array}{ll}
 & ((\lambda x : \mathtt{Dyn}.x)(\langle \mathtt{Int!} \rangle 1))[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot (\langle \mathtt{Int!} \rangle 1)[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot \langle \mathtt{Int!} \rangle (1[\epsilon])
\end{array}
$$

Gradually-typed calculus of closures

$$
\begin{array}{llll}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{cl}) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \textbf{prim}\ op\ \mathsf{cl} \mid \textbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl}
\end{array}
$$

$$
\begin{array}{ll}
 & ((\lambda x : \mathtt{Dyn}.x)(\langle \mathtt{Int!} \rangle 1))[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot (\langle \mathtt{Int!} \rangle 1)[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot \langle \mathtt{Int!} \rangle (1[\epsilon]) \\
\longmapsto_\rho & \textit{delayed beta contraction} \\
 & x[(x \mapsto \langle \mathtt{Int!} \rangle (1[\epsilon])) : \epsilon]
\end{array}
$$

Gradually-typed calculus of closures

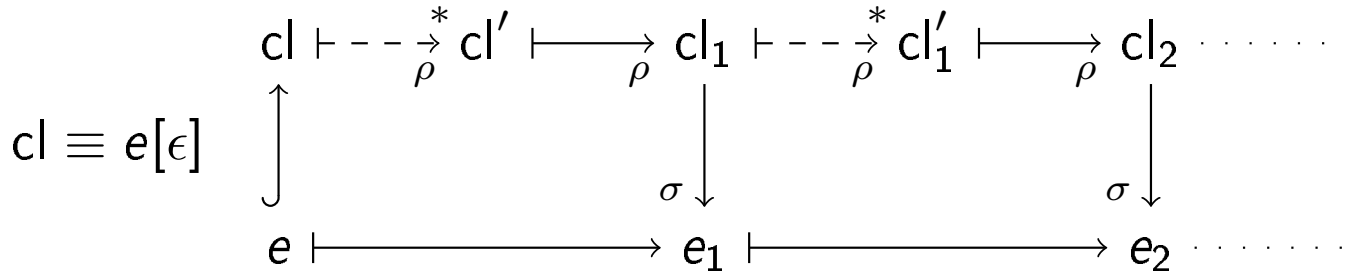
$$
\begin{array}{llll}
\text{environments} & \rho & ::= & \epsilon \mid (x \mapsto \mathsf{cl}) : \rho \\
\text{closures} & \mathsf{cl} & ::= & e[\rho] \mid \textbf{prim}\ op\ \mathsf{cl} \mid \textbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl}
\end{array}
$$

$$
\begin{array}{ll}
 & ((\lambda x : \mathtt{Dyn}.x)(\langle \mathtt{Int!} \rangle 1))[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot (\langle \mathtt{Int!} \rangle 1)[\epsilon] \\
\longmapsto_\rho & \textit{ephemeral expansion} \\
 & (\lambda x : \mathtt{Dyn}.x)[\epsilon] \cdot \langle \mathtt{Int!} \rangle (1[\epsilon]) \\
\longmapsto_\rho & \textit{delayed beta contraction} \\
 & x[(x \mapsto \langle \mathtt{Int!} \rangle (1[\epsilon])) : \epsilon] \\
\longmapsto_\rho & \textit{on-demand substitution} \\
 & \langle \mathtt{Int!} \rangle (1[\epsilon])
\end{array}
$$

Gradually-typed calculus of closures

$$
\begin{array}{rcll}
(op\ e)[\rho] & \longrightarrow_\rho & \textbf{prim}\ op\ (e[\rho]) & (\mathsf{Prim}_\rho) \\
(\texttt{if}\ e_1\ e_2\ e_3)[\rho] & \longrightarrow_\rho & \textbf{if}\ (e_1[\rho])\ (e_2[\rho])\ (e_3[\rho]) & (\mathsf{IfTE}_\rho) \\
(e_1\ e_2)[\rho] & \longrightarrow_\rho & (e_1[\rho]) \cdot (e_2[\rho]) & (\mathsf{App}_\rho) \\
(\langle c \rangle e)[\rho] & \longrightarrow_\rho & \langle c \rangle (e[\rho]) & (\mathsf{Coer}_\rho) \\
 & & & \\
x[\rho] & \longrightarrow_\rho & \mathsf{cl} \quad \text{if } (x \mapsto \mathsf{cl}) \in \rho & (\mathsf{Var}_\rho) \\
 & & & \\
e[\rho] \cdot \mathsf{v} & \longrightarrow_\rho & e[(x \mapsto \mathsf{v}) : \rho] & (\beta_\rho) \\
\textbf{prim}\ op\ (n[\rho]) & \longrightarrow_\rho & (\delta(op, n))[\epsilon] & (\delta_\rho) \\
\textbf{if}\ (k[\rho])\ \mathsf{cl}_1\ \mathsf{cl}_2 & \longrightarrow_\rho & \begin{cases} \mathsf{cl}_1 & \text{if } k = \texttt{true} \\ \mathsf{cl}_2 & \text{if } k = \texttt{false} \end{cases} & (\mathsf{If}_\rho) \\
\langle c_1 \rangle \mathsf{v} & \longrightarrow_\rho & \langle c_2 \rangle \mathsf{v} \quad \text{if } c_1 \longmapsto_{\mathsf{ED}} c_2 & (\mathsf{StepCst}_\rho) \\
\langle \iota \rangle \mathsf{v} & \longrightarrow_\rho & \mathsf{v} & (\mathsf{IdCst}_\rho) \\
\langle d \rangle \langle \overline{c} \rangle \mathsf{s} & \longrightarrow_\rho & \langle \overline{c}\,;\, d \rangle \mathsf{s} & (\mathsf{CmpCst}_\rho) \\
(\langle \tilde{c} \to \tilde{d} \rangle \mathsf{s}) \cdot \mathsf{v} & \longrightarrow_\rho & \langle \tilde{d} \rangle (\mathsf{s} \cdot \langle \tilde{c} \rangle \mathsf{v}) & (\mathsf{AppCst}_\rho) \\
\langle \texttt{Fail}^\ell \rangle \mathsf{s} & \longrightarrow_\rho & (\texttt{Blame}\ \ell)[\epsilon] & (\mathsf{FailCst}_\rho) \\
\langle (\tilde{c} \to \tilde{d})\,;\texttt{Fail}^\ell \rangle \mathsf{s} & \longrightarrow_\rho & \texttt{Blame}\ \ell & (\mathsf{FailFC})
\end{array}
$$

Gradually-typed calculus of closures



$$
\begin{array}{rcl}
\sigma(k[\rho]) & = & k \\
\sigma((op\ e)[\rho]) & = & op\ \sigma(e[\rho]) \\
\sigma(x[\rho]) & = & \begin{cases} \sigma(\mathsf{cl}) & \text{if } (x \mapsto \mathsf{cl}) \in \rho \\ x & \text{otherwise} \end{cases} \\
\sigma((\lambda x : T.e)[\rho]) & = & \lambda x : T.\sigma(e[\rho]) \\
\sigma((e_1\ e_2)[\rho]) & = & \sigma(e_1[\rho])\,\sigma(e_2[\rho]) \\
\sigma((\langle c \rangle e)[\rho]) & = & \langle c \rangle(\sigma(e[\rho])) \\
\sigma(\langle\langle c \rangle\rangle \mathsf{cl}) & = & \langle c \rangle(\sigma(\mathsf{cl})) \\
\sigma((\texttt{Blame}\ \ell)[\rho]) & = & \texttt{Blame}\ \ell
\end{array}
$$