A calculus of expandable stores
Continuation-and-environment-passing style translations

Hugo Herbelin\textsuperscript{1} \hfill Étienne Miquey\textsuperscript{2}

\textsuperscript{1}Inria
Université de Paris, CNRS, IRIF

\textsuperscript{2}CNRS
ÉNS Paris-Saclay, Inria, LSV

LICS 2020
A computational wonderland

The \(\lambda\)-calculus

One calculus to rule them all

A very nice abstraction:
- Turing-complete
- different evaluation strategies
- different type systems
- pure and effectful computations

Operational semantics through abstract machines
\(\mapsto\) SECD (Landin), KAM (Krivine), CEK (Felleisen and Friedman), ZINC (Leroy)...

Continuation-passing style (CPS) translations allow to abstract the machine again.
- specify an evaluation strategy
- make explicit the control flow
- induce a type translation \(\equiv\) syntactic model
\(\mapsto\) allowing to transfer logical properties from the target calculus
A computational wonderland

The $\lambda$-calculus

One calculus to rule them all

A very nice abstraction:
- Turing-complete
- different type systems
- different evaluation strategies
- pure and effectful computations

Operational semantics through abstract machines
↬ SECD (Landin), KAM (Krivine), CEK (Felleisen and Friedman), ZINC (Leroy)…

Continuation-passing style (CPS) translations allow to abstract the machine again.
- specify an evaluation strategy
- make explicit the control flow
- induce a type translation ≡ syntactic model
↬ allowing to transfer logical properties from the target calculus
A computational wonderland

The $\lambda$-calculus

A very nice abstraction:

- Turing-complete
- different type systems
- different evaluation strategies
- pure and effectful computations

Operational semantics through abstract machines

⇒ SECD (Landin), KAM (Krivine), CEK (Felleisen and Friedman), ZINC (Leroy)...
A computational wonderland

The $\lambda$-calculus

One calculus to rule them all

A very nice abstraction:

- Turing-complete
- different type systems
- different evaluation strategies
- pure and effectful computations

Operational semantics through abstract machines

$\leadsto$ SECD (Landin), KAM (Krivine), CEK (Felleisen and Friedman), ZINC (Leroy)...

Continuation-passing style (CPS) translations allow to abstract the machine again.

- specify an evaluation strategy
- make explicit the control flow
- induce a type translation $\equiv$ syntactic model

$\leadsto$ allowing to transfer logical properties from the target calculus
In praise of laziness

**Call-by-need** evaluation strategy:
- evaluates arguments of functions only when needed
  - as in *call-by-name*
- shares the evaluations across all places where they are needed
  - as in *call-by-value*

In short:

demand-driven computations + memoization

Many benefits, used in Haskell (by default) or Coq (tactic, kernel).

Trickier and historically less studied than CbName/CbValue.
In praise of laziness

**Call-by-need** evaluation strategy:
- evaluates arguments of functions only when needed
  ⇨ *as in call-by-name*
- shares the evaluations across all places where they are needed
  ⇨ *as in call-by-value*

In short:

**demand-driven** computations + **memoization**

Many benefits, used in **Haskell** (by default) or **Coq** (tactic, kernel).

Trickier and historically less studied than CbName/CbValue.
In praise of laziness

**Call-by-need** evaluation strategy:
- evaluates arguments of functions only when needed
  ↪ *as in* call-by-name
- shares the evaluations across all places where they are needed
  ↪ *as in* call-by-value

In short:

**demand-driven** computations + memoization

Many benefits, used in Haskell (by default) or Coq (tactic, kernel).

Trickier and historically less studied than CbName/CbValue.
Computing with global environments

Standard abstract machines use **local environments** and closures:

### Krivine Abstract Machine (CbName)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Term</th>
<th>Type</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tu \star S \star E$</td>
<td></td>
<td>$t \star (u, E) \cdot S \star E$</td>
<td>$\rightarrow_c$</td>
<td></td>
</tr>
<tr>
<td>$\lambda x.t \star (u, E') \cdot S \star E$</td>
<td></td>
<td>$t \star S \star E[x := (u, E')]$</td>
<td>$\rightarrow_\beta$</td>
<td></td>
</tr>
<tr>
<td>$x \star S \star E[x := (t, E')]E''$</td>
<td></td>
<td>$t \star S \star E'$</td>
<td>$\rightarrow_s$</td>
<td></td>
</tr>
</tbody>
</table>

**Call-by-need** requires a **global environment** to share computations.

### Milner Abstract Machine (CbName)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Term</th>
<th>Type</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tu \star \pi \star \tau$</td>
<td></td>
<td>$t \star u \cdot \pi \star \tau$</td>
<td>$\rightarrow_c$</td>
<td></td>
</tr>
<tr>
<td>$\lambda x.t \star u \cdot \pi \star \tau$</td>
<td></td>
<td>$t \star \pi \star \tau[x := u]$</td>
<td>$\rightarrow_\beta$</td>
<td></td>
</tr>
<tr>
<td>$x \star \pi \star \tau[x := t] \tau'$</td>
<td></td>
<td>$\tilde{t}^\alpha \star \pi \star \tau[x := t] \tau'$</td>
<td>$\rightarrow_s$</td>
<td></td>
</tr>
</tbody>
</table>

**Globality** requires to explicitly handle addresses or a **renaming process**.
Computing with global environments

Standard abstract machines use **local environments** and closures:

### Krivine Abstract Machine (CbName)

| t u ⋆ S ⋆ E | →_c | t ⋆ (u, E) ⋆ S ⋆ E |
| λx.t ⋆ (u, E’) ⋆ S ⋆ E | →_β | t ⋆ S ⋆ E[x ::= (u, E’)] |
| x ⋆ S ⋆ E[x ::= (t, E’)]E’’ | →_s | t ⋆ S ⋆ E’ |

Call-by-need requires a **global environment** to share computations.

### Milner Abstract Machine (CbName)

| tu ⋆ π ⋆ τ | →_c | t ⋆ u ⋆ π ⋆ τ |
| λx.t ⋆ u ⋆ π ⋆ τ | →_β | t ⋆ π ⋆ τ[x ::= u] |
| x ⋆ π ⋆ τ[x ::= t]τ’ | →_s | i’α ⋆ π ⋆ τ[x ::= t]τ’ |

Globality requires to explicitly handle addresses or a **renaming process**.
Computing with global environments

Standard abstract machines use **local environments** and closures:

### Krivine Abstract Machine (CbName)

\[
\begin{align*}
&tu \star S \star E \quad \rightarrow_c \quad t \star (u, E) \cdot S \star E \\
&\lambda x.t \star (u, E') \cdot S \star E \\n&x \star S \star E[x := (t, E')]E'' \quad \rightarrow_s \quad t \star S \star E'
\end{align*}
\]

### Milner Abstract Machine (CbName)

\[
\begin{align*}
&tu \star \pi \star \tau \quad \rightarrow_c \quad t \star u \cdot \pi \star \tau \\
&\lambda x.t \star u \cdot \pi \star \tau \\
&x \star \pi \star \tau[x := t]\tau' \quad \rightarrow_s \quad \overline{t}^\alpha \star \pi \star \tau[x := t]\tau'
\end{align*}
\]

Call-by-need requires a **global environment** to share computations.

Globality requires to explicitly handle addresses or a **renaming process**.
A thorn in the side

A lost paradise?
- Abstract machines with global environments
- By-need abstract machines
  - ❯ Sestoft’s machine, Accattoli, Barenbaum and Mazza’s Merged MAD
- ❧ Typed continuation-and-environment passing style translation?

Several difficulties to handle:
- How should control and environments interact?
- Can we soundly type environments?
- ... while accounting for extensibility?
- How to avoid name clashes?
This paper

Our goal
Typed continuation-and-environment-passing style (CEPS) translations

\( \mapsto \) \textit{i.e. understand how to soundly CEPS translate calculi with global environments}

Contribution

- We introduce \( F_\Upsilon \), a \textit{generic} calculus used as the target of CEPS translations, which features:
  - a data type for \textit{typed stores}
  - \textit{explicit coercions} witnessing store extensions

- We use it to implement simply-typed CEPS translations for:
  - \checkmark call-by-need
  - \checkmark call-by-name
  - \checkmark call-by-value
This paper

Our goal

Typed continuation-and-environment-passing style (CEPS) translations

\[ \text{i.e. understand how to soundly CEPS translate calculi with global environments} \]

Contribution

- We introduce \( F_\Upsilon \), a generic calculus used as the target of CEPS translations, which features:
  - a data type for typed stores
  - explicit coercions witnessing store extensions

- We use it to implement simply-typed CEPS translations for:
  - call-by-need
  - call-by-name
  - call-by-value
This paper

Our goal
Typed continuation-and-environment-passing style (CEPS) translations

⇒ i.e. understand how to soundly CEPS translate calculi with global environments

Contribution
- We introduce $F_\Upsilon$, a generic calculus used as the target of CEPS translations, which features:
  - a data type for typed stores
  - explicit coercions witnessing store extensions

Generic?
We aim at isolating the key ingredients necessary to the definition of well-typed CEPS translations.
This paper

Our goal

Typed continuation-and-environment-passing style (CEPS) translations

⇒ *i.e. understand how to soundly CEPS translate calculi with global environments*

Contribution

- We introduce $F_\Upsilon$, a *generic* calculus used as the target of CEPS translations, which features:
  - a data type for *typed stores*
  - *explicit coercions* witnessing store extensions

- We use it to implement simply-typed CEPS translations for:
  ✓ call-by-need ✓ call-by-name ✓ call-by-value
Continuation-and-environment passing style translations

Towards typed translations
Question

What should be the semantics of a control operator in presence of a shared memory?

\[
\begin{align*}
&\text{let } a = \text{catch}_k (\text{fun } k \Rightarrow (\text{Id , fun } x \Rightarrow \text{throw } k x)) \\
&f = \text{fst } a \\
&q = \text{snd } a \\
&\text{in } f q (\text{Id , Id })
\end{align*}
\]

Okasaki, Lee & Tarditi ’93:

What does not force the effect is shared.

- \( q \) shared
- \( f \) recomputed

\( \Rightarrow \text{loops...} \)
### Question

What should be the semantics of a control operator in presence of a shared memory?

#### Backtrack and Laziness

<table>
<thead>
<tr>
<th>Method:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. sequent calculus</td>
</tr>
<tr>
<td>2. abstract machine</td>
</tr>
<tr>
<td>3. (untyped) CPS translation</td>
</tr>
<tr>
<td>4. realizability interpretation</td>
</tr>
</tbody>
</table>

#### Ariola et al. ’12:

Nothing is shared inside an effect

- $f$ recomputed
- $q$ recomputed

⇒ *returns $(Id, Id)$ ✓*

#### Okasaki, Lee & Tarditi ’93:

What does *not* force the effect is shared.

- $q$ shared
- $f$ recomputed

⇒ *loops…*
Backtrack and laziness

Theorem

Ariola et al.’s semantics is typable, normalizing and consistent.

\[
\text{let } a = \text{catch}_k \ (\text{fun } k \Rightarrow (\text{Id}, \text{fun } x \Rightarrow \text{throw } k \ x)) \\
f = \text{fst} \ a \\
q = \text{snd} \ a \\
in \ f \ q \ (\text{Id}, \text{Id})
\]

Okasaki, Lee & Tarditi ’93:

What does not force the effect is shared.

- \(q\) shared
- \(f\) recomputed

\[\Rightarrow \text{loops}…\]

Ariola et al. ’12:

Nothing is shared inside an effect

- \(f\) recomputed
- \(q\) recomputed

\[\Rightarrow \text{returns (Id,Id)}\ ✔\]

Method:

1. sequent calculus
2. abstract machine
3. (untyped) CPS translation
4. realizability interpretation
Intuitions

(Analyzing Ariola et al. ’12)

Sequent calculus:

\[\langle t \parallel e \rangle \tau\]

Term  Context  Environment

Syntax

<table>
<thead>
<tr>
<th>Terms</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>e ::= E</td>
</tr>
<tr>
<td>Weak val.</td>
<td>V ::= v \mid x</td>
</tr>
<tr>
<td>Strong val.</td>
<td>v ::= \lambda x.t</td>
</tr>
<tr>
<td>Environments</td>
<td>τ ::= ε \mid τ[x := t] \mid τ[α := E]</td>
</tr>
<tr>
<td>Commands</td>
<td>c ::= \langle t \parallel e \rangle</td>
</tr>
<tr>
<td>Forcing cont.</td>
<td>F ::= t \cdot E</td>
</tr>
<tr>
<td>Catchable cont.</td>
<td>E ::= F \mid α \mid µ[x].\langle x \parallel F \rangle \tau</td>
</tr>
</tbody>
</table>
Intuitions

(Analyzing Ariola et al. ’12)

Sequent calculus:

\[ \langle t \parallel e \rangle_\tau \]

Term  Context  Environment

Syntax

<table>
<thead>
<tr>
<th>Terms</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>Contexts</td>
</tr>
<tr>
<td>t, u ::= V</td>
<td>e ::= E</td>
</tr>
<tr>
<td>Weak val.</td>
<td>Catchable cont.</td>
</tr>
<tr>
<td>V ::= v x</td>
<td>E ::= F</td>
</tr>
<tr>
<td>Strong val.</td>
<td>Forcing cont.</td>
</tr>
<tr>
<td>v ::= \lambda x.t</td>
<td>F ::= t \cdot E</td>
</tr>
</tbody>
</table>

Environments  τ ::= ε | τ[x := t] | τ[\alpha := E]

Commands        c ::= \langle t \parallel e \rangle
Intuitions

(Analyzing Ariola et al. ’12)

Syntax

<table>
<thead>
<tr>
<th>Terms</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>$t, u ::= V \mid \mu\alpha.c$</td>
</tr>
<tr>
<td>Weak val.</td>
<td>$V ::= v \mid x$</td>
</tr>
<tr>
<td>Strong val.</td>
<td>$\nu ::= \lambda x.t \mid k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environments</th>
<th>Catchable cont.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau ::= \epsilon \mid \tau[x := t] \mid \tau[\alpha := E]$</td>
<td></td>
</tr>
</tbody>
</table>

| Commands | $c ::= \langle t \parallel e \rangle$ |

Lazy reduction:

(Lazy storage) $\langle t \parallel \tilde{\mu}x.c \rangle \tau \rightarrow c\tau[x := t]$

(Catch) $\langle \mu\alpha.c \parallel E \rangle \tau \rightarrow c\tau[\alpha := E]$

(Lookup) $\langle x \parallel F \rangle \tau[x := t] \tau' \rightarrow \langle t \parallel \tilde{\mu}[x].\langle x \parallel F \rangle \tau' \rangle \tau$

(Forced eval.) $\langle V \parallel \tilde{\mu}[x].\langle x \parallel F \rangle \tau' \rangle \tau \rightarrow \langle V \parallel F \rangle \tau[x := V] \tau'$

$\langle \lambda x.t \parallel u \cdot E \rangle \tau \rightarrow \langle u \parallel \tilde{\mu}x.\langle t \parallel E \rangle \rangle \tau$
Intuitions

(Analyzing Ariola et al. ’12)

Sequent calculus:

\[
\langle t \parallel e \rangle \tau
\]

\[\downarrow \quad \uparrow \quad \downarrow\]

Term Context Environment

Untyped CEPS:

\[
\llbracket \langle t \parallel e \rangle \tau \rrbracket \simeq \llbracket e \rrbracket_e [\tau]_\tau [t]_t
\]

environment passing
continuation passing
Intuitions

(Analyzing Ariola et al. ’12)

Untyped CEPS:

\[
\llbracket \langle t \parallel e \rangle \tau \rrbracket \simeq \llbracket e \rrbracket_e \llbracket \tau \rrbracket_\tau \llbracket t \rrbracket_t
\]

\[
\begin{align*}
\llbracket \bar{\mu} x. c \rrbracket_e & := \lambda \tau t. [c]_c \tau [x := t] \\
\llbracket E \rrbracket_e & := \lambda \tau t. t \tau [E]_E \\
\llbracket \mu \alpha . c \rrbracket_t & := \lambda \tau E. ([c]_c \tau)[E/\alpha] \\
\llbracket V \rrbracket_t & := \lambda \tau E. E \tau [V]_v \\
\llbracket \bar{\mu} [x]. \langle x \parallel F \rangle \tau' \rrbracket_E & := \lambda \tau V. V \tau [x := V][\tau']_\tau [F]_f \\
\llbracket F \rrbracket_E & := \lambda \tau V. V \tau [F]_f \\
\llbracket x \rrbracket_v & := \lambda \tau F. \tau (x) \tau (\lambda \tau V. V \tau [x := V][\tau']_\tau [F]_f) \\
\llbracket \lambda x.t \rrbracket_v & := \lambda \tau F. F \tau (\lambda u \tau E. [t]_t \tau [x := u] E) \\
\llbracket u \cdot E \rrbracket_f & := \lambda \tau v. v [t]_t \tau [E]_E
\end{align*}
\]
Typing the CEPS: guidelines

\[
\begin{align*}
\llbracket (t \parallel e) \tau \rrbracket & \simeq \llbracket e \rrbracket_e \llbracket \tau \rrbracket_\tau \llbracket t \rrbracket_t \\
\text{environment passing} & \quad \text{continuation passing}
\end{align*}
\]
Typing the CEPS: guidelines

\[ \langle t \parallel e \rangle_\tau \simeq [e]_e [\tau]_\tau [t]_t \]

Step 1 - Continuation-passing part

\[ \Gamma \vdash_t t : A \]

\[ \Downarrow \]

\[ [\Gamma] \vdash [t]_t : [A]_t \]
Typing the CEPS: guidelines (1/4)

\[
\begin{align*}
\llbracket (t \parallel e) \tau \rrbracket & \simeq \llbracket e \rrbracket_\tau \llbracket \tau \rrbracket \llbracket t \rrbracket_t \\
\text{environment passing} & \quad \text{continuation passing}
\end{align*}
\]

Step 1 - Continuation-passing part

\[
\begin{align*}
\llbracket A \rrbracket_e \equiv & \quad \llbracket A \rrbracket_t \rightarrow \bot \\
\llbracket A \rrbracket_t \equiv & \quad \llbracket A \rrbracket_E \rightarrow \bot \\
\llbracket A \rrbracket_E \equiv & \quad \llbracket A \rrbracket_V \rightarrow \bot \\
\llbracket A \rrbracket_V \equiv & \quad \llbracket A \rrbracket_F \rightarrow \bot \\
\llbracket A \rightarrow B \rrbracket_V \equiv & \quad \llbracket A \rrbracket_t \rightarrow \llbracket B \rrbracket_E \rightarrow \bot \\
\end{align*}
\]

\[\mu x. c \rrbracket_e = \lambda t. \llbracket c \rrbracket_c \]

\[\mu \alpha. c \rrbracket_t = \lambda \alpha. \llbracket c \rrbracket_c \]

\[\vdash \text{In comparison, for call-by-name/call-by-value we would only have 4/3 layers.}\]
Typing the CEPS: guidelines

$$\left[\langle t \parallel e \rangle \tau \right] \simeq \left[ e \right]_e \left[ \tau \right]_\tau \left[ t \right]_t$$

environment passing
continuation passing

Step 2- Environment-passing part

$$\Gamma \vdash_t t : A$$

$$\vdash [t]_t : [\Gamma] \rightarrow [A]_t$$
Typing the CEPS: guidelines

\[[<t \parallel e>\tau] \simeq [e]_e [\tau]_\tau [t]_t\]

environment passing

continuation passing

Step 2- Environment-passing part

\[\Gamma \vdash_t t : A\]

\[\downarrow\]

\[\vdash [t]_t : [\Gamma] \triangleright_t A\]
Typing the CEPS: guidelines

(2/4)

\[ [\langle t \parallel e \rangle \tau] \simeq [e]_e [\tau]_\tau [t]_t \]

environment-passing

continuation-passing

Step 2 - Environment-passing part

\[ \Gamma \vdash_t t : A \]

\[ \downarrow \]

\[ \vdash [t]_t : [\Gamma] \rightarrow [\Gamma] \triangleright E A \rightarrow \perp \]
Typing the CEPS: guidelines

\[
\begin{align*}
\left[\langle t \parallel e \rangle \tau \right] & \simeq \left[ e \right]_e \left[ \tau \right]_\tau \left[ t \right]_t \\
\text{environment passing} & \\
\text{continuation passing}
\end{align*}
\]

Step 2- Environment-passing part

\[
\begin{array}{c}
\Gamma \vdash t : A \\
\downarrow \\
\vdash \left[ t \right]_t : \left[ \Gamma \right] \rightarrow \left( \left[ \Gamma \right] \rightarrow \left[ \Gamma \right] \uplus A \rightarrow \bot \right) \rightarrow \bot
\end{array}
\]
Typing the CEPS: guidelines (2/4)

\[
\left[\left\langle t \mid e \right\rangle \tau \right] \simeq [e]_e \left[\tau\right]_e \left[t\right]_t
\]

environment passing

continuation passing

Step 2- Environment-passing part

\[
\begin{align*}
\left[\Gamma\right] \triangleright_e A & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_t A \rightarrow \perp \\
\left[\Gamma\right] \triangleright_t A & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_E A \rightarrow \perp \\
\left[\Gamma\right] \triangleright_E A & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_V A \rightarrow \perp \\
\left[\Gamma\right] \triangleright_V A & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_F A \rightarrow \perp \\
\left[\Gamma\right] \triangleright_F A & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_V A \rightarrow \perp \\
\left[\Gamma\right] \triangleright_V A \rightarrow B & \triangleq \left[\Gamma\right] \rightarrow \left[\Gamma\right] \triangleright_t A \rightarrow \left[\Gamma\right] \triangleright_E B \rightarrow \perp
\end{align*}
\]
Step 3 - Extension of the environment

A possible reduction scheme:

$t$ is needed \hspace{1cm} \langle x \parallel F \rangle \tau_1 [x := t] \tau_2
Step 3 - Extension of the environment

A possible reduction scheme:

\[ t \text{ is needed} \quad \langle x \parallel F \rangle_{\tau_1} [x := t]_{\tau_2} \]
\[ \rightarrow \langle t \parallel \tilde{\mu}[x].\langle x \parallel F \rangle_{\tau_2} \rangle_{\tau_1} \]
Typing the CEPS: guidelines

(3/4)

Step 3 - Extension of the environment

A possible reduction scheme:

\[
\begin{align*}
\text{t is needed} & \quad \langle x \parallel F \rangle \tau_1[x := t] \tau_2 \\
\text{evaluation of t} & \quad \rightarrow \langle t \parallel \tilde{\mu}[x] \rangle . \langle x \parallel F \rangle \tau_2 \tau_1 \\
\text{t produces a value} & \quad \rightarrow^* \langle V \parallel \tilde{\mu}[x] \rangle . \langle x \parallel F \rangle \tau_2 \tau_1 \tau' 
\end{align*}
\]
Step 3 - Extension of the environment
A possible reduction scheme:

- \( t \) is needed
  \[ \langle x \| F \rangle_{\tau_1} [x := t]_{\tau_2} \]
  \[ \rightarrow \langle t \| \tilde{\mu}[x].x \| F \rangle_{\tau_2} \tau_1 \]

- Evaluation of \( t \)
  \[ \rightarrow^* \langle V \| \tilde{\mu}[x].x \| F \rangle_{\tau_2} \tau_1 \tau'[x := V]_{\tau_2} \]

- \( t \) produces a value
- \( V \) is stored

Key idea:

\[ \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket \triangleright_t A \text{ should be compatible with any extension of } \llbracket \Gamma \rrbracket \]
Typing the CEPS: guidelines (3/4)

Step 3 - Extension of the environment

Key idea:

\([t]_t : [\Gamma] \triangleright_t A\) should be compatible with any extension of \([\Gamma]\)

Store subtyping:

\(\Gamma' \ll: \Gamma\)

\[
\begin{array}{cccccc}
B_1 & B_2 & A_1 & B_3 & A_2 & B_4 \\
\end{array}
\ll:
\begin{array}{cc}
A_1 & A_2 \\
\end{array}
\]
Step 3 - Extension of the environment

Key idea:

\[ [t]_t : \Gamma \triangleright_t A \text{ should be compatible with any extension of } [\Gamma] \]

Store subtyping:

\[ \Gamma' <: \Gamma \]

Translation:

\[ \Gamma \vdash_t t : A \]

\[ \downarrow \]

\[ \vdash [t]_t : [\Gamma] \rightarrow [\Gamma] \triangleright_E A \rightarrow \bot \]
Typing the CEPS: guidelines (3/4)

Step 3 - Extension of the environment

Key idea:

\[ [t]_t : [\Gamma] \triangleright_t A \text{ should be compatible with any extension of } [\Gamma] \]

Store subtyping:

\[ \Gamma' <: \Gamma \]

Translation:

\[ \Gamma \vdash_t t : A \]

\[ \downarrow \]

\[ \vdash [t]_t : \forall \Upsilon <: [\Gamma] . \Upsilon \rightarrow \Upsilon \triangleright E A \rightarrow \bot \]
Typing the CEPS: guidelines (3/4)

Step 3 - Extension of the environment

Key idea:

\[[t]_t : [\Gamma] \triangleright_t A\] should be compatible with any extension of \([\Gamma]\)

Store subtyping:

\[\Gamma' <: \Gamma\]

Translation:

\[\Gamma \vdash_t t : A\]

\[\vdash [t]_t : \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow (\forall \Upsilon' <: \Upsilon. \Upsilon' \triangleright \Upsilon A \rightarrow \bot) \rightarrow \bot\]
Step 3 - Extension of the environment

Key idea:

\[ [t]_t : [\Gamma] \triangleright_t A \text{ should be compatible with any extension of } [\Gamma] \]

Store subtyping:

\[ \Gamma' <: \Gamma \]

Translation:

\[
\begin{align*}
[\Gamma] \triangleright_e A & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_t A \rightarrow \bot \\
[\Gamma] \triangleright_t A & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_E A \rightarrow \bot \\
[\Gamma] \triangleright_E A & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_V A \rightarrow \bot \\
[\Gamma] \triangleright_V A & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_F A \rightarrow \bot \\
[\Gamma] \triangleright_F A & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_V A \rightarrow \bot \\
[\Gamma] \triangleright_v A \rightarrow B & \triangleq \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_t A \rightarrow \Upsilon \triangleright_E B \rightarrow \bot 
\end{align*}
\]
Step 4 - Avoiding name clashes

Ariola et al. work implicit relies on $\alpha$-renaming on-the-fly.

$\Rightarrow$ incompatible with the CEPS translation
Step 4 - Avoiding name clashes

Ariola et al. work implicit relies on $\alpha$-renaming on-the-fly.

⇒ incompatible with the CEPS translation

Here, we use De Bruijn levels both:

- in the source:

\[
\frac{\Gamma(n) = (x_n : T)}{\Gamma \vdash V \ x_n : T} \quad \langle x_n \parallel F \rangle \tau[x_n := t] \tau \quad \frac{n = |\tau|}{\langle t \parallel \tilde{\mu}[x_n].\langle x_n \parallel F \rangle \tau' \rangle \tau} \\
\langle V \parallel \tilde{\mu}[x_i].\langle x_i \parallel F \rangle \tau' \rangle \tau \quad \frac{n = |\tau|}{\langle V \parallel \uparrow^n_i F \rangle \tau[x_n := V] \uparrow^n_i \tau'}
\]
Step 4 - Avoiding name clashes

Ariola et al. work implicit relies on $\alpha$-renaming on-the-fly. 

$\uparrow$ incompatible with the CEPS translation

Here, we use De Bruijn levels both:

- and the target:

\[
x_0 : A, \alpha_1 : B \perp, x_2 : C \vdash_t t : D
\]

\[
\downarrow
\]

\[
\vdash [t]_t : A, B \perp, C \triangleright_t D
\]
Step 4 - Avoiding name clashes

Here, we use De Bruijn levels both:

- and the target:

\[
\begin{align*}
x_0 &: A, \alpha_1 : B^\perp, x_2 &: C \vdash_t t : D
\end{align*}
\]

\[
\downarrow
\]

\[
\vdash [t]_t : A, B^\perp, C \triangleright_t D
\]

...where we use coercions \(\sigma : \Gamma' <: \Gamma\) to witness store extension and keep track of De Bruijn:
A calculus of expandable stores

Introducing $F_Y$
The motto

System $F_\Upsilon$ defines a *parametric* target for CEPS translations

Each CEPS translation can be divided in three blocks:
- a source calculus and its type system
- a syntax for stores and coercions
- the target calculus, an instance of $F_\Upsilon$
Principles

The motto

System $F_\Upsilon$ defines a *parametric* target for CEPS translations

Each CEPS translation can be divided in three blocks:

1. a **source calculus** and its type system
   ⊳ *Here, simply-typed calculi*
2. a syntax for **stores** and **coercions**
3. the **target calculus**, an instance of $F_\Upsilon$
The motto

System $F_\gamma$ defines a *parametric* target for CEPS translations

Each CEPS translation can be divided in three blocks:

1. a **source calculus** and its type system
2. a syntax for **stores and coercions**
3. the **target calculus**, an instance of $F_\gamma$
The motto

System $F_Y$ defines a *parametric* target for CEPS translations

Each CEPS translation can be divided in three blocks:

1. a *source calculus* and its type system
2. a syntax for *stores and coercions*
3. the *target calculus*, an instance of $F_Y$
In this paper, we only use **lists** to represent stores:

### Source types

\[ A ::= X \mid A \to B \]

### Store types

\[ \Upsilon ::= Y \mid \emptyset \mid \Upsilon, F \mid \Upsilon; \Upsilon' \]

### Stores

\[ \tau ::= \delta \mid [] \mid \tau[t] \mid \tau; \tau' \]

\[ \vdash \tau : \Upsilon' \triangleright_{\tau} \Upsilon \]

"Appended to a store of type \( \Upsilon' \), the store \( \tau \) is of type \( \Upsilon \)."

\[ \Gamma \vdash [] : \emptyset \triangleright_{\tau} \emptyset \]
\[ \Gamma \vdash [t] : \Upsilon_0 \triangleright_{\tau} T \]
\[ \Gamma \vdash \tau : \Upsilon_0 \triangleright_{\tau} \Upsilon \]
\[ \Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) \triangleright_{\tau} \Upsilon' \]
\[ \Gamma \vdash \tau; \tau' : \Upsilon_0 \triangleright_{\tau} \Upsilon; \Upsilon' \]

**Remark**

- Type of a store = list of source types
- How these types are translated = \( \triangleright_{\tau} \) = *parameter* of the target
In this paper, we only use lists to represent stores:

**Source types**
\[ A ::= X | A \rightarrow B \]

**Store types**
\[ \Upsilon ::= Y | \emptyset | \Upsilon, F | \Upsilon; \Upsilon' \]

**Stores**
\[ \tau ::= \delta | [] | \tau[t] | \tau; \tau' \]

\[ \vdash \tau : \Upsilon' \rightarrow_\tau \Upsilon \]

“Appended to a store of type \( \Upsilon' \), the store \( \tau \) is of type \( \Upsilon \).”

| \( \Gamma \vdash [] : \emptyset \rightarrow_\tau \emptyset \) | \( \Gamma \vdash [t] : \Upsilon_0 \rightarrow_\tau T \) | \( \Gamma \vdash \tau : \Upsilon_0 \rightarrow_\tau \Upsilon \) | \( \Gamma \vdash \tau : (\Upsilon_0 ; \Upsilon) \rightarrow_\tau \Upsilon' \) | \( \Gamma \vdash \tau ; \tau' : \Upsilon_0 \rightarrow_\tau \Upsilon ; \Upsilon' \) |

**Remark**

\( \text{type of a store} = \text{list of source types} \)

\( \text{how these types are translated} = \rightarrow_\tau = \text{parameter of the target} \)
In this paper, we only use \textbf{lists} to represent stores:

\begin{itemize}
  \item \textbf{Source types} \quad A ::= X \mid A \rightarrow B \quad F ::= A \mid A^\perp
  \item \textbf{Store types} \quad \Upsilon ::= Y \mid \emptyset \mid \Upsilon, F \mid \Upsilon; \Upsilon'
  \item \textbf{Stores} \quad \tau ::= \delta \mid [] \mid \tau[t] \mid \tau; \tau'
\end{itemize}

\[
\vdash \tau : \Upsilon' \triangleright_{\tau} \Upsilon
\]

“Appended to a store of type \( \Upsilon' \), the store \( \tau \) is of type \( \Upsilon \).”

\[
\begin{array}{c}
\Gamma \vdash [] : \emptyset \triangleright_{\tau} \emptyset \\
\Gamma \vdash [t] : \Upsilon_0 \triangleright_{\tau} T \\
\Gamma \vdash \tau : \Upsilon_0 \triangleright_{\tau} \Upsilon \\
\Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) \triangleright_{\tau} \Upsilon'
\end{array}
\]

Remark
\begin{itemize}
  \item \textit{type of a store} = \textit{list of source types}
  \item \textit{how these types are translated} = \triangleright = \textit{parameter} of the target
\end{itemize}
Coercions

Explicit witnesses of list inclusions:

1. Base case

\[ \Gamma \vdash \varepsilon : \emptyset <: \emptyset^{(\varepsilon)} \]

2. Local identity

\[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \quad \Gamma \vdash \sigma^+ : (\Upsilon', F) <:(\Upsilon, F)^{(<_:+)} \]

3. Strict extension

\[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \quad \Gamma \vdash \mathcal{F} \sigma : (\Upsilon', F) <:\Upsilon^{(<:_\mathcal{F})} \]

Example:

\[ \vdash \mathcal{F}((\mathcal{F}\varepsilon)^+) : T_0, T, U, T_1 <: T, U \]

Remark: this corresponds to the function

\[ \begin{align*}
0 & \mapsto \rightarrow \\
1 & \mapsto \rightarrow \\
2 & \mapsto \rightarrow \\
\end{align*} \]

17/ 20
Coercions

Explicit witnesses of list inclusions:

1. Base case
   \[ \Gamma \vdash \varepsilon : \emptyset <: \emptyset \]

2. Local identity
   \[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \quad \frac{\Gamma \vdash \sigma^+ : (\Upsilon', F) <: (\Upsilon, F)}{\Gamma \vdash \uparrow \sigma : (\Upsilon', F) <: \Upsilon} \]

3. Strict extension
   \[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \quad \frac{\Gamma \vdash \uparrow \sigma : (\Upsilon', F) <: \Upsilon}{\Gamma \vdash \uparrow (\uparrow \varepsilon)^+: T_0, T, U, T_1 <: T, U} \]

Example:
Coercions

**Explicit witnesses of list inclusions:**

1. **Base case**

   \[ \Gamma \vdash \varepsilon : \emptyset <: \emptyset \]

2. **Local identity**

   \[ \Gamma \vdash \sigma : \gamma' <: \gamma \]
   \[
   \frac{\Gamma \vdash \sigma^+ : (\gamma', F) <: (\gamma, F)}{\Gamma \vdash \uparrow \sigma : (\gamma', F) <: \gamma} \]

3. **Strict extension**

   \[ \Gamma \vdash \sigma : \gamma' <: \gamma \]
   \[
   \frac{\Gamma \vdash \uparrow \sigma : (\gamma', F) <: \gamma}{\Gamma \vdash \uparrow \sigma : (\gamma', F) <: \gamma} \]

**Example:**

\[ \vdash \uparrow ((\uparrow \varepsilon)^{++}) : T_0, T, U, T_1 <: T, U \]
Coercions

 Explicit witnesses of list inclusions:

1. Base case

   \[ \Gamma \vdash \varepsilon : \emptyset <: \emptyset^{(\varepsilon)} \]

2. Local identity

   \[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon^{(\sigma)} \quad \frac{\Gamma \vdash \sigma^+ : (\Upsilon', F) <: (\Upsilon, F)^{(<:+)}}{\Gamma \vdash \sigma^+ : (\Upsilon', F) <: (\Upsilon, F)^{(<:+)}} \]

3. Strict extension

   \[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon^{(\sigma)} \quad \frac{\Gamma \vdash \Upsilon \vdash \uparrow \sigma : (\Upsilon', F) <: \Upsilon^{(\uparrow \sigma)}}{\Gamma \vdash \uparrow \sigma : (\Upsilon', F) <: \Upsilon^{(\uparrow \sigma)}} \]

Example:

\[ \vdash \uparrow ((\uparrow \varepsilon)^{+++}) : T_0, T, U, T_1 <: T, U \]

Remark: this corresponds to the function

\[ \bullet 0 \mapsto 1 \quad \bullet 1 \mapsto 2 \quad \bullet 2 \mapsto 4 \]
Coercions

Explicit witnesses of list inclusions:

1. Base case

\[ \Gamma \vdash \varepsilon : \emptyset <: \emptyset \]  

2. Local identity

\[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \]  
\[ \Gamma \vdash \sigma^+ : (\Upsilon', F) <: (\Upsilon, F) \]  

3. Strict extension

\[ \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \]  
\[ \Gamma \vdash \uparrow\sigma : (\Upsilon', F) <: \Upsilon \]  

Example:

\[ \vdash \uparrow(\uparrow(\varepsilon)^{++}) : T_0, T, U, T_1 <: T, U \]

Remark: this corresponds to the function

- \(0 \mapsto 1\)
- \(1 \mapsto 2\)
- \(2 \mapsto 4\)
System $F_{\Upsilon}$

In broad lines

System F extended with stores and coercions$^1$

$^1$Actually, false advertizing, the situation is more involved.
System $F_\Upsilon$

**Syntax:**

*Store type* $\Upsilon$ + *Stores* $\tau$ + *Coercions* $\sigma$ +

**Types**

$T ::= X \mid T \rightarrow U \mid \Upsilon' <: \Upsilon \rightarrow T \mid \Upsilon \triangleright_\tau \Upsilon' \rightarrow T \mid \forall Y. T$

**Terms**

$t ::= k \mid x \mid \lambda x.t \mid t u \mid \lambda s.t \mid t \sigma \mid \lambda \delta.t \mid t \tau \mid \lambda Y.t \mid t \Upsilon$

| split $\tau$ at $n$ along $\sigma : \Upsilon' <: \Upsilon$ as $(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)$ in $t$ |
System $F_{\Upsilon}$

Syntax: 

Store type $\Upsilon$ + Stores $\tau$ + Coercions $\sigma$ + 

Types 

$T ::= X \mid T \rightarrow U \mid \Upsilon' <: \Upsilon \rightarrow T \mid \Upsilon_{\sigma \tau} \Upsilon' \rightarrow T \mid \forall Y.T$

Terms 

$t ::= k \mid x \mid \lambda x.t \mid t u \mid \lambda s.t \mid t \sigma \mid \lambda \delta.t \mid t \tau \mid \lambda Y.t \mid t \Upsilon$

| split $\tau$ at $n$ along $\sigma: \Upsilon' <: \Upsilon$ as $(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)$ in $t$

Intuitively, split allows to look in $\Upsilon'$ for the term expected at position $n$ in $\Upsilon$ using $\sigma : \Upsilon' <: \Upsilon$:
# System $F_\gamma$

**Syntax:**

<table>
<thead>
<tr>
<th>Types</th>
<th>$T ::= X \mid T \rightarrow U \mid \gamma' &lt;: \gamma \rightarrow T \mid \gamma \triangleright_\tau \gamma' \rightarrow T \mid \forall Y.T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>$t ::= k \mid x \mid \lambda x.t \mid t u \mid \lambda s.t \mid t \sigma \mid \lambda \delta.t \mid t \tau \mid \lambda Y.t \mid t \gamma$</td>
</tr>
<tr>
<td></td>
<td>$\mid \text{split } \tau \text{ at } n \text{ along } \sigma : \gamma' &lt;: \gamma \text{ as } (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1) \text{ in } t$</td>
</tr>
</tbody>
</table>

Intuitively, **split** allows to look in $\gamma'$ for the term *expected at position n in $\gamma$* using $\sigma : \gamma' <: \gamma$:

![Diagram showing the split operation](image-url)
System $F_\Upsilon$

**Syntax:**

- **Store type** $\Upsilon$  +  **Stores** $\tau$  +  **Coercions** $\sigma$  +

**Types**

$T ::= X | T \to U | \Upsilon' <: \Upsilon \to T | \Upsilon \triangleright_\tau \Upsilon' \to T | \forall Y.T$

**Terms**

$t ::= k | x | \lambda x.t | t u | \lambda s.t | t \sigma | \lambda \delta.t | t \tau | \lambda Y.t | t \Upsilon$

| split $\tau$ at $n$ along $\sigma : \Upsilon' <: \Upsilon$ as $(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)$ in $t$

Three kinds of reductions:

- split
- normalization of coercions
- usual $\beta$-reduction

We have:

**Properties**

1. Reduction preserves typing  
   *(Subject reduction)*
2. Typed terms normalize  
   *(Normalization)*

Shallow embedding in Coq: https://gitlab.com/emiquey/fupsilon
Examples

In the paper, we take advantage of the genericity of $F_\Upsilon$:

$$\Gamma \vdash t : \Upsilon_0 \triangleright T$$

$$\Gamma \vdash [t] : \Upsilon_0 \triangleright_\tau T$$

to define well-typed CEPS for simply-typed calculi:

✓ call-by-need  ✓ call-by-name  ✓ call-by-value

These translations exactly follow the intuitions we saw before:

negative translation  Kripke-style forcing

Remark: we could also consider System F as source calculus, by changing the notion of source types.
In the paper, we take advantage of the genericity of $F_\Upsilon$:

\[
\frac{\Gamma \vdash t : \Upsilon_0 \triangleright T}{\Gamma \vdash [t] : \Upsilon_0 \triangleright_{\tau} T}
\]

to define well-typed CEPS for simply-typed calculi:

✓ call-by-need  
✓ call-by-name  
✓ call-by-value

These translations exactly follow the intuitions we saw before:

negative translation  
Kripke-style forcing

Remark: we could also consider System F as source calculus, by changing the notion of source types.
Examples

In the paper, we take advantage of the genericity of $F_\Upsilon$:

$$\Gamma \vdash t : \Upsilon_0 \rightarrow T$$

$$\Gamma \vdash [t] : \Upsilon_0 \tau \rightarrow T$$

to define well-typed CEPS for simply-typed calculi:

✓ call-by-need
✓ call-by-name
✓ call-by-value

These translations exactly follow the intuitions we saw before:

continuation-passing + environment-passing

negative translation Kripke-style forcing

Remark: we could also consider System F as source calculus, by changing the notion of source types.
Examples

In the paper, we take advantage of the genericity of $F_{\gamma}$:

$$\Gamma \vdash t : \gamma_0 \quad \Rightarrow T$$

$$\Gamma \vdash [t] : \gamma_0 \triangleleft_\tau T$$

Thus, we define well-typed CEPS for simply-typed calculi:

✓ call-by-need  ✓ call-by-name  ✓ call-by-value

These translations exactly follow the intuitions we saw before:

continuation-passing + environment-passing

negative translation  Kripke-style forcing

Remark: we could also consider System F as source calculus, by changing the notion of source types.
Conclusion

We isolated the **key ingredients** for well-typed CEPS:

1. terms to represent and manipulate **typed stores**,  
2. explicit **coercions** to witness store extensions.

\( F_\Upsilon \) has the benefits of being **parametric**:

- suitable for CEPS with different evaluation strategies  
- compatible with different sources/type systems.  
- compatible with different implementation of stores
Conclusion

We isolated the **key ingredients** for well-typed CEPS:

1. terms to represent and manipulate **typed stores**,  
2. explicit **coercions** to witness store extensions.

$F_\gamma$ has the benefits of being **parametric**:

- suitable for CEPS with different evaluation strategies  
- compatible with different sources/type systems.  
- compatible with different implementation of stores
Conclusion

From a logical viewpoint:

CEPS ≅ Kripke forcing interleaved with a negative translation

Connection between **forcing and environment** already known:

![Diagram](attachment:Diagram.png)
Open questions / further work

1. Towards well-typed compilation transformations for lazily-evaluated calculi? (cf. MetaCoq project)

2. Exact expressiveness of $F_\Sigma$?

3. Type translation as a modality?
Conclusion

Open questions / further work

1. Towards well-typed compilation transformations for lazily-evaluated calculi? (cf. MetaCoq project)

2. Exact expressiveness of $F_Y$?

3. Type translation as a modality?
Open questions / further work

1. Towards well-typed compilation transformations for lazily-evaluated calculi? (cf. MetaCoq project)

2. Exact expressiveness of $F_\Upsilon$?

3. Type translation as a modality?

\[ \triangleright t A \text{ is a function : store type } \mapsto \text{ type} \]
Open questions / further work

1. Towards well-typed compilation transformations for lazily-evaluated calculi? (cf. MetaCoq project)

2. Exact expressiveness of $F_{\Upsilon}$?

3. Type translation as a modality?

\[ \cdot \triangleright_t A \text{ is a function : store type } \mapsto \text{ type} \]

\[ \square \mathcal{F} \triangleq \Upsilon \mapsto \forall \Upsilon' <: \Upsilon. \Upsilon' \rightarrow (\mathcal{F} \Upsilon') \rightarrow \bot \]

\[ \cdot \triangleright_t A = \square (\cdot \triangleright_E A) = \square (\square (\cdot \triangleright_V A)) = ... \]
Thank you for your attention.