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

Hugo Herbelin\textsuperscript{1}  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

\textit{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 \textit{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 \(\equiv\) \textit{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

$\rightarrow$ 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
  $\rightarrow$ 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
\[ \mapsto \textit{SECD (Landin), KAM (Krivine), CEK (Felleisen and Friedman), ZINC (Leroy)} \]

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

The $\lambda$-calculus

One calculus to rule them all

A very nice abstraction:

- Turing-complete
- different evaluation strategies
- different type systems
- 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
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)**

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

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

**Milner Abstract Machine (CbName)**

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

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)**

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

- $\rightarrow_c \quad t \star (u, E) \cdot S \star E$
- $\rightarrow_\beta \quad t \star S \star E[x ::= (u, E')]$
- $\rightarrow_s \quad t \star S \star E'$

**Milner Abstract Machine (CbName)**

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

- $\rightarrow_c \quad t \star u \cdot \pi \star \tau$
- $\rightarrow_\beta \quad t \star \pi \star \tau[x ::= u]$
- $\rightarrow_s \quad \tilde{\tau}^{\alpha} \star \pi \star \tau[x ::= t]\tau'$

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

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)

\[
\begin{align*}
& t u \quad \star \quad S \quad \star \quad E \quad \rightarrow_{c} \quad t \star (u, E) \cdot S \star E \\
& \lambda x. t \star (u, E') \cdot S \star E \quad \rightarrow_{\beta} \quad t \star S \star E[x ::= (u, E')]
\end{align*}
\]

\[
\begin{align*}
& x \quad \star \quad S \quad \star \quad E[x ::= (t, E')]E'' \quad \rightarrow_{s} \quad t \star S \star E'
\end{align*}
\]

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

### Milner Abstract Machine (CbName)

\[
\begin{align*}
& tu \quad \star \quad \pi \quad \star \quad \tau \quad \rightarrow_{c} \quad t \star u \cdot \pi \star \tau \\
& \lambda x. t \star u \cdot \pi \star \tau \quad \rightarrow_{\beta} \quad t \star \pi \star \tau[x ::= u]
\end{align*}
\]

\[
\begin{align*}
& x \quad \star \quad \pi \quad \star \quad \tau[x ::= t]\tau' \quad \rightarrow_{s} \quad \overline{t^\alpha} \star \pi \star \tau[x ::= t]\tau'
\end{align*}
\]

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

\(\xrightarrow{\text{i.e.}} \) 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:
  - \(\checkmark\) call-by-need
  - \(\checkmark\) call-by-name
  - \(\checkmark\) 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

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

\(\Rightarrow\) \textit{loops}...
Backtrack and laziness

Question

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

\begin{verbatim}
let a = catch_k (fun k \Rightarrow (Id , fun x \Rightarrow throw k x))
f = fst a
q = snd a
in f q (Id , Id )
\end{verbatim}

Okasaki, Lee & Tarditi ’93:

What does not force the effect is shared.

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

\( \Rightarrow \) loops...

Ariola et al. ’12:

Nothing is shared inside an effect

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

\( \Rightarrow \) returns \((Id,Id)\)
Backtrack and laziness

**Theorem**

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

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

Okasaki, Lee & Tarditi ‘93:

What does not force the effect is shared.

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

\(\rightarrow\) loops...

Ariola et al. ’12:

Nothing is shared inside an effect

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

\(\rightarrow\) returns \((\text{Id,Id})\)

**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>Contexts</td>
</tr>
<tr>
<td>$t, u ::= V \mid \mu \alpha.c$</td>
<td>$e ::= E \mid \tilde{\mu}x.c$</td>
</tr>
<tr>
<td>Weak val. $V ::= v \mid x$</td>
<td>Catchable cont. $E ::= F \mid \alpha \mid \tilde{\mu}[x].\langle x \parallel F \rangle \tau$</td>
</tr>
<tr>
<td>Strong val. $v ::= \lambda x.t \mid k$</td>
<td>Forcing cont. $F ::= t \cdot E \mid \kappa$</td>
</tr>
</tbody>
</table>

Environments $\tau ::= \epsilon \mid \tau[x := t] \mid \tau[\alpha := E]$ |

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

(Analyzing Ariola et al. ’12)

Sequent calculus:

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

Term \hspace{1cm} Context \hspace{1cm} Environment

Syntax

<table>
<thead>
<tr>
<th>Terms</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>$t, u ::= V \mid \mu x. c$</td>
</tr>
<tr>
<td>Weak val.</td>
<td>$V ::= v \mid x$</td>
</tr>
<tr>
<td>Strong val.</td>
<td>$v ::= \lambda x. t \mid k$</td>
</tr>
<tr>
<td>Contexts</td>
<td>$e ::= E \mid \tilde{\mu} x. c$</td>
</tr>
<tr>
<td>Catchable cont.</td>
<td>$E ::= F \mid \alpha \mid \tilde{\mu}[x]. \langle x \parallel F \rangle \tau$</td>
</tr>
<tr>
<td>Forcing cont.</td>
<td>$F ::= t \cdot E \mid \kappa$</td>
</tr>
</tbody>
</table>

Environments

$\tau ::= \epsilon \mid \tau[x := t] \mid \tau[\alpha := E]$  

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>

**Environments**

$\tau ::= \varepsilon \mid \tau[x := t] \mid \tau[\alpha := E]$  

**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:

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

\[ \text{Term} \quad \text{Context} \quad \text{Environment} \]

Untyped CEPS:

\[ \llbracket \langle t \parallel e \rangle \tau \rrbracket \approx \llbracket e \rrbracket_e \llbracket \tau \rrbracket_{\tau} \llbracket t \rrbracket_t \]

\[ \text{environment passing} \quad \text{continuation passing} \]
Intuitions

(Analyzing Ariola et al. ’12)

Untyped CEPS:

\[
\begin{align*}
[[\langle t \parallel e \rangle \tau]] & \simeq [[e]]_e \, [[\tau]]_\tau \, [[t]]_t \\
\end{align*}
\]

\begin{align*}
[[\bar{\mu}x.c]]_e & := \lambda \tau t.([[c]]_c \, \tau [x := t]) \\
[[E]]_e & := \lambda \tau t. t \, \tau [[E]]_E \\
[[\mu \alpha . c]]_t & := \lambda \tau E.([[c]]_c \, \tau)[E/\alpha] \\
[[V]]_t & := \lambda \tau E. E \, \tau [[V]]_v \\
[[\bar{\mu}[x].\langle x \parallel F \rangle \tau']]_E & := \lambda \tau V. V \, \tau [x := V] [[\tau']]_\tau \, [[F]]_f \\
[[F]]_E & := \lambda \tau V. V \, \tau [[F]]_f \\
[[x]]_v & := \lambda \tau F. \tau (x) \, \tau (\lambda \tau V. V \, \tau [x := V] \tau' [[F]]_f) \\
[[\lambda x.t]]_v & := \lambda \tau F. F \, \tau (\lambda u \tau E. [[t]]_t \, \tau [x := u] \, E) \\
[[u \cdot E]]_f & := \lambda \tau v. v \, [[t]]_t \, \tau [[E]]_E
\end{align*}

\]
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
Typing the CEPS: guidelines (1/4)

\[
\begin{align*}
\mathbf{e} \left\{ \tau \right\} \& \mathbf{t} \left\{ \tau \right\} & \simeq & \mathbf{e} \left\{ \tau \right\} \mathbf{t} \left\{ \tau \right\} \\
\text{environment passing} & & \text{continuation passing}
\end{align*}
\]

Step 1 - Continuation-passing part

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

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

environment passing

continuation passing

Step 1 - Continuation-passing part

\[ \llbracket A \rrbracket_e \triangleq \llbracket A \rrbracket_t \rightarrow \bot \]
\[ \llbracket A \rrbracket_t \triangleq \llbracket A \rrbracket_E \rightarrow \bot \]
\[ \llbracket A \rrbracket_E \triangleq \llbracket A \rrbracket_V \rightarrow \bot \]
\[ \llbracket A \rrbracket_V \triangleq \llbracket A \rrbracket_F \rightarrow \bot \]
\[ \llbracket A \rightarrow B \rrbracket_V \triangleq \llbracket A \rrbracket_t \rightarrow \llbracket B \rrbracket_E \rightarrow \bot \]

\[ \llbracket \bar{\mu}x.c \rrbracket_e = \lambda t.\llbracket c \rrbracket_c \]
\[ \llbracket \mu\alpha.c \rrbracket_t = \lambda\alpha.\llbracket c \rrbracket_c \]

\( \equiv \) In comparison, for call-by-name/call-by-value we would only have 4/3 layers.
Typing the CEPS: guidelines (2/4)

\[
\begin{align*}
\llangle t \parallel e \rrangle \tau & \simeq [e]_e [\tau]_\tau [t]_t \\
\text{environment passing} & \quad \text{continuation passing}
\end{align*}
\]

**Step 2- Environment-passing part**

\[
\Gamma \vdash_t t : A
\]

\[
\downarrow
\]

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

Step 2 - Environment-passing part

\[ \Gamma \vdash_t t : A \]

\[ \downarrow \]

\[ \vdash [t]_t : [[\Gamma]]_{t} A \]
Typing the CEPS: guidelines

Step 2- Environment-passing part

$$\tau = e_{\tau} [\tau]_{\tau} \# t$$

**environment passing**

**continuation passing**

\[ \Gamma \vdash t : A \]

\[ \uparrow \]

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

(2/4)

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

environment passing

continuation passing

Step 2- Environment-passing part

\[
\begin{align*}
\Gamma \vdash_t t &: A \\
\downarrow \\
\downarrow
\end{align*}
\]

\[
\vdash \left[t\right]_t : [\Gamma] \rightarrow ([\Gamma] \rightarrow [\Gamma] \triangleright \vee A \rightarrow \bot) \rightarrow \bot
\]
Typing the CEPS: guidelines (2/4)

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

environment passing

continuation passing

Step 2- Environment-passing part

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

A possible reduction scheme:

\[
\langle x \| F \rangle \tau_1[x := t] \tau_2
\]

\( t \) is needed
typing the CEPS: guidelines

(3/4)

Step 3 - Extension of the environment

A possible reduction scheme:

$t$ is needed
\[ \langle x \parallel F \rangle_{\tau_1} [x := t]_{\tau_2} \]
evaluation of $t$
\[ \rightarrow \langle t \parallel \tilde{\mu}[x]. \langle x \parallel F \rangle_{\tau_2} \rangle_\tau_1 \]
Step 3 - Extension of the environment

A possible reduction scheme:

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

A possible reduction scheme:

- **t is needed**
  \[ \langle x \parallel F \rangle \tau_1[x := t] \tau_2 \]
  \[ \rightarrow \langle t \parallel \tilde{\mu}[x].(x \parallel F)\tau_2 \rangle \tau_1 \]

- **evaluation of t**
  \[ \rightarrow^* \langle V \parallel \tilde{\mu}[x].(x \parallel F)\tau_2 \rangle \tau_1 [\tau'] \]

- **t produces a value**
  \[ \rightarrow \langle V \parallel F \rangle \tau_1 \tau'[x := V] \tau_2 \]

- **V is stored**

**Key idea:**

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

$$\Gamma' <: \Gamma$$

$$\begin{array}{ccccccc}
B_1 & B_2 & A_1 & B_3 & A_2 & B_4 \\
\end{array} <: 
\begin{array}{ccccccc}
A_1 & A_2 \\
\end{array}$$
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:

\[
\Gamma \vdash_t t : A
\]

\[
\Downarrow
\]

\[
\vdash [t]_t : [\Gamma] \rightarrow [\Gamma] \triangleright_E A \rightarrow \bot
\]
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:

\[ \Gamma \vdash_t t : A \]

\[ \vdash [t]_t : \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow \Upsilon \triangleright_E A \rightarrow \bot \]
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:

\[\Gamma \vdash_t t : A\]

\[\vdash [t]_t : \forall \Upsilon <: [\Gamma]. \Upsilon \rightarrow (\forall \Upsilon' <: \Upsilon. \Upsilon' \triangleright \Upsilon A \rightarrow \bot) \rightarrow \bot\]

(reminiscent of Kripke forcing)
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:

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

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

⇒ incompatible with the CEPS translation
Step 4 - Avoiding name clashes

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

$\Leftarrow$ incompatible with the CEPS translation

Here, we use De Bruijn levels both:

- in the source:

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

Ariola et al. work implicit relies on $\alpha$-renaming on-the-fly.  
\[ \leadsto \text{incompatible with the CEPS translation} \]

Here, we use De Bruijn levels both:

- and the target:

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

\[ \downarrow \]

\[
\vdash [t]_t : A, B \perp, C \triangleright_t D
\]
**Typing the CEPS: guidelines (4/4)**

**Step 4 - Avoiding name clashes**

Here, we use **De Bruijn levels** both:

- and the target:

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

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

\[
\vdash [t]_t : A, B^\perp, C \triangleright_t D
\]
A calculus of expandable stores

Introducing $F_\Upsilon$
Principles

The motto

System $F_\gamma$ 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 $F_\gamma$
Principles

The motto

System $F_\Upsilon$ defines a \textit{parametric} target for CEPS translations

Each CEPS translation can be divided in three blocks:

1. a \textbf{source calculus} and its type system
   
   ↴ \textit{Here, simply-typed calculi}

2. a syntax for \textit{stores and coercions}

3. the \textbf{target calculus}, an instance of $F_\Upsilon$
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
2. a syntax for *stores and coercions*
3. the target calculus, an instance of $F_{\Upsilon}$
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
2. a syntax for *stores and coercions*
3. the *target calculus*, an instance of $F_\Upsilon$
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**

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

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

Remark

- **type of a store** = list of source types
- how these types are translated = \( \triangleright \) = *parameter* of the target
In this paper, we only use **lists** to represent stores:

<table>
<thead>
<tr>
<th>Source types</th>
<th>Source Definition</th>
<th>Store types</th>
<th>Type Definition</th>
<th>Stores</th>
<th>Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A ::= X \mid A \to B$</td>
<td>$F ::= A \mid A^\perp$</td>
<td>$\Upsilon ::= Y \mid \emptyset \mid \Upsilon, F \mid \Upsilon; \Upsilon'$</td>
<td>$\tau ::= \delta \mid [] \mid \tau[t] \mid \tau; \tau'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

“Appended to a store of type $\Upsilon'$, the store $\tau$ is of type $\Upsilon$."

\[
\begin{align*}
\Gamma \vdash [] : \emptyset & \to_\tau \emptyset \\
\Gamma \vdash [t] : \Upsilon_0 & \to_\tau T \\
\Gamma \vdash \tau : \Upsilon_0 & \to_\tau \Upsilon \\
\Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) & \to_\tau \Upsilon' \\
\Gamma \vdash \tau; \tau' : \Upsilon_0 & \to_\tau \Upsilon; \Upsilon'
\end{align*}
\]

**Remark**

- **type of a store** = **list of source types**
- **how these types are translated** = $\to_\tau$ = parameter of the target
In this paper, we only use **lists** to represent stores:

**Source types**  
\[ A ::= X \mid A \to B \]  
\[ F ::= A \mid A^\perp \]

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

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

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

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

\[ \begin{align*}
\Gamma \vdash [] : \emptyset & \Rightarrow_\tau \emptyset \\
\Gamma \vdash [t] : \Upsilon_0 & \Rightarrow_\tau T \\
\Gamma \vdash \tau : \Upsilon_0 & \Rightarrow_\tau \Upsilon \\
\Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) & \Rightarrow_\tau \Upsilon' \\
\Gamma \vdash \tau; \tau' : \Upsilon_0 & \Rightarrow_\tau \Upsilon; \Upsilon'
\end{align*} \]

**Remark**

- **type of a store** = *list of source types*  
- **how these types are translated** = \( \Rightarrow_\tau = \textit{parameter} \) of the target
Explicit witnesses of list inclusions:

1. **Base case**
   \[
   \Gamma \vdash \varepsilon : \emptyset <: \emptyset^{(\varepsilon)}
   \]

2. **Local identity**
   \[
   \Gamma \vdash \sigma : \Upsilon' <: \Upsilon^{(\prec:)}
   \]

3. **Strict extension**
   \[
   \Gamma \vdash \sigma : \Upsilon' <: \Upsilon^{(\prec:)}
   \]

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

Remark: this corresponds to the function
\[
\begin{align*}
\bullet_0 & \mapsto \to_1 \\
\bullet_1 & \mapsto \to_2 \\
\bullet_2 & \mapsto \to_4
\end{align*}
\]
Coercions

Explicit witnesses of list inclusions:

1. Base case

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

2. Local identity

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

3. Strict extension

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

Example:

\[ \vdash \Uparrow (\Uparrow \epsilon) : T_0, T, U, T_1 <: T, U \]

Remark: this corresponds to the function

\[ \cdot 0 \mapsto \rightarrow 1 \cdot 1 \mapsto \rightarrow 2 \cdot 2 \mapsto \rightarrow 4 \]
Coercions

\[ \vdash \sigma : \Upsilon' <: \Upsilon \]

**Explicit witnesses of list inclusions:**

1. **Base case**
   \[
   \Gamma \vdash \varepsilon : \emptyset <: \emptyset \quad (\varepsilon)
   \]

2. **Local identity**
   \[
   \begin{align*}
   \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \\
   \Gamma \vdash \sigma'^+ : (\Upsilon', F) <: (\Upsilon, F)
   \end{align*}
   \quad (<:+)
   \]

3. **Strict extension**
   \[
   \begin{align*}
   \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \\
   \Gamma \vdash \Upsilon \uparrow \sigma : (\Upsilon', F) <: \Upsilon
   \end{align*}
   \quad (<:\uparrow)
   \]

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

Explicit witnesses of list inclusions:

1. Base case

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

2. Local identity

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

3. Strict extension

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

Example:

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

Remark: this corresponds to the function

- \(0 \mapsto 1\)
- \(1 \mapsto 2\)
- \(2 \mapsto 4\)
Coercions

\[ \vdash \sigma : \Upsilon' <: \Upsilon \]

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
In broad lines

System $F_\gamma$ extended with stores and coercions

1Actually, false advertizing, the situation is more involved.
System $F_Y$

Syntax:

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

Types

$T ::= X \mid T \to U \mid \gamma' <: \gamma \to T \mid \gamma \triangleright_{\tau} \gamma' \to T \mid \forall Y. T$

Terms

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

| split $\tau$ at $n$ along $\sigma: \gamma' <: \gamma$ 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**

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

**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 $t$

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

![Diagram](image)
System $F_\Upsilon$

**Syntax:**

<table>
<thead>
<tr>
<th>Store type $\Upsilon$  +  Stores $\tau$  +  Coercions $\sigma$ +</th>
</tr>
</thead>
</table>

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 $t$ |

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

```
\begin{tikzpicture}
  \node (Y) at (0,0) {$\Upsilon$};
  \node (0) at (-1,0) {$0$};
  \node (n) at (1,0) {$n$};
  \node (expected) at (0,-1) {$\text{expected}$};
  \node (tau) at (-2,-2) {$\tau$};
  \node (delta0) at (-1,-2) {$\delta_0 : Y_0$};
  \node (x) at (-0.5,-2) {$x$};
  \node (delta1) at (0,-2) {$\delta_1 : Y_1$};
  \node (split) at (1,-2) {$\text{split}$};
  \draw [->] (Y) -- (0);
  \draw [->] (Y) -- (n);
  \draw [->] (expected) -- (tau);
  \draw [->] (tau) -- (delta0);
  \draw [->] (tau) -- (x);
  \draw [->] (tau) -- (delta1);
  \draw [->] (split) -- (delta0);
  \draw [->] (split) -- (x);
  \draw [->] (split) -- (delta1);
\end{tikzpicture}
```
System $F_\gamma$

Syntax: 

| Types | $T ::= X \mid T \to U \mid \gamma' <: \gamma \to T \mid \gamma >_\tau \gamma' \to T \mid \forall Y.T$ |
| Terms | $t ::= k \mid x \mid \lambda x.t \mid t u \mid \lambda s.t \mid t \sigma \mid \lambda \delta.t \mid t \tau \mid \lambda Y.t \mid t \gamma \mid \text{split } \tau \text{ at } n \text{ along } \sigma : \gamma' <: \gamma \text{ as } (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1) \text{ in } t$ |

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.
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:

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 $F_\Upsilon$:

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

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

\[\text{Parameter depending on the translation}\]

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.
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:

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.

$F_\Upsilon$ 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

From a logical viewpoint:

CEPS \cong \text{Kripke forcing interleaved with a negative translation}

Connection between \textbf{forcing and environment} already known:

\begin{center}
\begin{tikzcd}
\text{Presheaves} & \text{Forcing} \\
& \text{State monad}
\end{tikzcd}
\end{center}
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?
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?
Conclusion

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?

$\cdot \triangleright_t A$ is a function : store type $\mapsto$ 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?

   $\cdot \triangleright_t A$ is a function : store type $\mapsto$ type

   \[
   \square \cal F \triangleq \forall \Upsilon \mapsto \forall \Upsilon' <: \forall \Upsilon. \Upsilon' \mapsto (\cal F \Upsilon') \mapsto \bot
   \]

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