Dependent Type Systems for delimited control operators

Tristan Crolard*

LACL – Université Paris-Est

February 3, 2010

*. Joint work with Emmanuel Polonowski.

Introduction

Problem:

- How do you prove the correctness of this program from [Wadler, 1994]?

$$\textbf{let}\ g = (\textbf{reset}\ (\textbf{if}\ (\textbf{shift}\ \lambda f.f)\ \textbf{then}\ 2\ \textbf{else}\ 3))$$
$$\textbf{in}\ (g\ \mathit{True}) + (g\ \mathit{False})$$

Informally:

- *"Here f (and hence g) is bound to the function that returns 2 if passed True, and 3 if passed False, hence the value of the given term is 5."*

Formally:

- use Filinski's encoding of **shift**/**reset** with **callcc**/**throw** and a global meta-continuation.
- define a program logic for a language with control and mutable state.

Control and mutable state

We need a program logic for control and higher-order mutable state.

Have to choose between:

- "pure" imperative langage (idealized Algol-like language)
  - low-level state and control (flowcharts) [Floyd, 1967]
  - structured programming [Hoare, 1969] [Dijkstra, 1976]
  - *procedures and control: error-prone* [O'Donnell, 1982]
- "impure" functional language (ML-like references)
  - dependent type systems for higher-order mutable state [Filliâtre, 2003] [Honda et al., 2005] [Nanevski et al., 2006]
  - *but what about control?*

And what about dependent type systems for pure imperative languages?

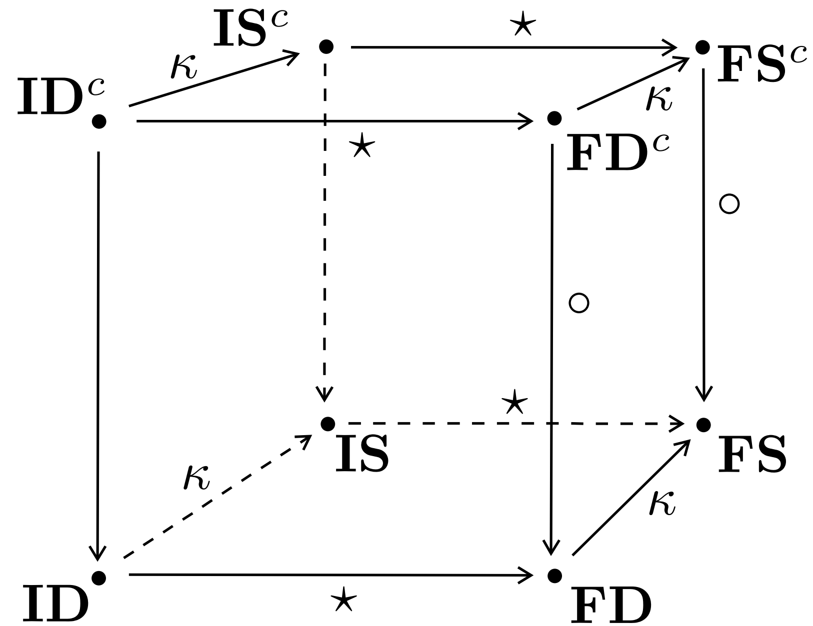
Roadmap

Define an imperative dependent (classical) type system by translation into a functional dependent (classical) type system. That is, revisit Landin's work in the light of the formulae-as-types interpretation of classical logic.

References

- [Landin, 1965a] "A correspondence between ALGOL 60 and Church's Lambda-notation"
- [Landin, 1965b] "A Generalization of Jumps and Labels"
- [Felleisen et al., 1987] "A syntactic theory of sequential control"
- [Griffin, 1990] "A formulæ-as-types notion of control"
- [Murthy, 1990] "Extracting Constructive Content from Classical proofs"
- [Leivant, 1990] [Krivine and Parigot, 1990] [Parigot, 1992] ...

Overview



I: imperative. **F**: functional.
D: dependent. **S**: simple.
_^{c}: classical.

$\star$: translation from **I** to **F**
κ : erasure function from **D** to **S**
$\circ$: $\neg\neg$-translation

Languages and type systems

ID/FD. Heyting arithmetic (**FD** is actually **M1LP** from [Leivant, 1990]);

ID^c/**FD**^{c}**.** Peano arithmetic;

F. Gödel System T (in call-by-value);

F^c**.** Gödel System T + **callcc**/**throw** (in call-by-value);

I. LOOP$^\omega$ [Crolard et al., 2009];

I^c**.** LOOP$^\omega$ + **non-local jumps**.

LOOP$^\omega$: a "pure" imperative language

An imperative counterpart to Gödel System T:

- an extension of the LOOP langage [Meyer and Ritchie, 1976] with higher-order procedures and procedural variables;
- a simple location-free semantics [Donahue, 1977];
- a pseudo-dynamic type system [Morrisett et al., 1999];
- a dependent type system [Xi, 2000].

Disclaimer: we are not yet considering program logics for "realistic" programming languages. However, we are not that far away ...

Spark ADA (Praxis/AdaCore) is a “pure” imperative language – a subset of Ada used in the industry for developping *verified* critical real-time systems:

- Arrays
- Records
- Procedures
- Packages (i.e. modules)
- Generics (soon)

But:

- No pointers
- No exceptions
- Limited side-effects

and *bounded loops* are preferred for Worst-Case Execution Time analysis.

LOOP^{ω} – syntax

$$
\begin{array}{rrcl}
(\textit{command}) & c & ::= & \{s\}_{\vec{x}} \\
 & & | & \textbf{for } y := 0 \textbf{ until } e\ \{s\}_{\vec{x}} \\
 & & | & y := e \quad | \quad \textbf{inc}(y) \quad | \quad \textbf{dec}(y) \\
 & & | & p(\vec{e}\,;\vec{y}\,) \\
 & & & \\
(\textit{sequence}) & s & ::= & \varepsilon \\
 & & | & c\,;\ s \\
 & & | & \textbf{cst } y = e;\ s \\
 & & | & \textbf{var } y := e;\ s \\
 & & & \\
(\textit{anonymous procedure}) & a & ::= & \textbf{proc } (\textbf{in } \vec{y}\,; \textbf{out } \vec{z}\,)\ \{s\}_{\vec{z}} \\
 & & & \\
(\textit{expression}) & e & ::= & y \quad | \quad w \\
(\textit{procedure}) & p & ::= & y \quad | \quad a \\
(\textit{value}) & w & ::= & * \quad | \quad \bar{q} \quad | \quad a
\end{array}
$$

LOOP$^\omega$ – remarks

- **Block annotations:** in a block $\{s\}_{\vec{x}}$, the variables $\vec{x}$ correspond to the free mutable variables occurring in the sequence s (such annotations can automatically be inferred).
- **No aliasing:** in order to avoid parameter-induced aliasing problems, we assume that all y_i are pairwise distinct in a procedure call $p(\vec{e}\,;\vec{y}\,)$.
- **No backpatching:** no free mutable variable is allowed in the body of a procedure (only its **out** parameters). The following block (which would define a fixpoint) is thus illegal:

$$\{$$
$$\quad \textbf{var}\ f\colon \textbf{proc}\ (\textbf{out}\ int) := \textbf{proc}\ (\textbf{out}\ x\colon int)\{\ \mathit{fix}(x);\ \}_x;$$
$$\quad \mathit{fix} := f;$$
$$\}_{fix}$$

LOOP$^\omega$ – remarks

- **Block annotations:** in a block $\{s\}_{\vec{x}}$, the variables $\vec{x}$ correspond to the free mutable variables occurring in the sequence s (such annotations can automatically be inferred).
- **No aliasing:** in order to avoid parameter-induced aliasing problems, we assume that all y_i are pairwise distinct in a procedure call $p(\vec{e}\,;\vec{y}\,)$.
- **No backpatching:** no free mutable variable is allowed in the body of a procedure (only its **out** parameters). The following block (which would define a fixpoint) is thus illegal:

$$\{$$
$$\quad \textbf{var}\ f\colon \textbf{proc}\ (\textbf{out}\ int) := \textbf{proc}\ (\textbf{out}\ x\colon int)\{\ \mathit{fix}(x);\ \}_x;$$
$$\quad \mathit{fix} := f;$$
$$\}_{fix}$$

LOOP^{ω} – remarks

- **Block annotations:** in a block $\{s\}_{\vec{x}}$, the variables $\vec{x}$ correspond to the free mutable variables occurring in the sequence s (such annotations can automatically be inferred).
- **No aliasing:** in order to avoid parameter-induced aliasing problems, we assume that all y_i are pairwise distinct in a procedure call $p(\vec{e}\,; \vec{y}\,)$.
- **No backpatching:** no free mutable variable is allowed in the body of a procedure (only its **out** parameters). The following block (which would define a fixpoint) is thus illegal:

$$
\begin{array}{l}
\{ \\
\quad \mathbf{var}\ f\colon \mathbf{proc}\ (\mathbf{out}\ int) := \mathbf{proc}\ (\mathbf{out}\ x\colon int)\{\ \mathit{fix}(x);\ \}_x; \\
\quad \mathit{fix} := f; \\
\}_{fix}
\end{array}
$$

$\textsc{Loop}^\omega$ – the Ackermann procedure

$$
\begin{array}{l}
\textbf{cst}\ Ack = \textbf{proc}\ (\textbf{in}\ M, N; \textbf{out}\ Z)\ \{ \\
\quad \textbf{var}\ G := \textbf{proc}\ (\textbf{in}\ Y; \textbf{out}\ P)\ \{ \\
\quad\quad P := Y; \\
\quad\quad \textbf{inc}(P); \\
\quad \}_P; \\
\quad \textbf{for}\ I := 0\ \textbf{until}\ M\ \{ \\
\quad\quad \textbf{cst}\ H = G; \\
\quad\quad G := \textbf{proc}\ (\textbf{in}\ Y; \textbf{out}\ P)\ \{ \\
\quad\quad\quad P := 2; \\
\quad\quad\quad \textbf{for}\ J := 0\ \textbf{until}\ Y\ \{ \\
\quad\quad\quad\quad H(P; P); \\
\quad\quad\quad \}_P; \\
\quad\quad \}_P; \\
\quad \}_G; \\
\quad G(N; Z); \\
\}_Z
\end{array}
$$

$\textsc{Loop}^{\omega}$ – transition semantics (1)

– a state is a pair (s, μ), where s is a sequence and μ is a store;
– a store μ maps mutable variables onto closed imperative values.

$$((\{\}_{\vec{z}}; s), \mu) \mapsto (s, \mu)$$

$$\frac{(s_1, \mu) \mapsto (s'_1, \mu')}{((\{s_1\}_{\vec{z}};\ s_2), \mu) \mapsto ((\{s'_1\}_{\vec{z}};\ s_2), \mu')}$$

$$((\mathbf{var}\ \ y := e;\ \ \varepsilon), \mu) \mapsto (\varepsilon, \mu)$$

$$\frac{e =_{\mu} w \qquad (s, (\mu, y \leftarrow w)) \mapsto (s', (\mu', y \leftarrow w'))}{((\mathbf{var}\ \ y := e;\ \ s), \mu) \mapsto ((\mathbf{var}\ \ y := w';\ \ s'), \mu')}$$

$$\frac{e =_{\mu} w}{((\mathbf{cst}\ \ y = e;\ \ s), \mu) \mapsto (s[y \leftarrow w], \mu)}$$

$\text{L}\textsc{oop}^\omega$ – transition semantics (2)

$$\frac{e =_\mu w}{((y := e;\ s), \mu) \mapsto (s, \mu[y \leftarrow w])}$$

$$\frac{\mu(y) = \bar{q}}{((\mathbf{inc}(y);\ s), \mu) \mapsto ((y := \overline{q+1};\ s), \mu)} \qquad \frac{\mu(y) = \bar{q}}{((\mathbf{dec}(y);\ s), \mu) \mapsto ((y := \overline{q \dot{-} 1};\ s), \mu)}$$

$$\frac{\vec{e} =_\mu \vec{w} \qquad p =_\mu \mathbf{proc}\ (\mathbf{in}\ \vec{y}\,;\mathbf{out}\ \vec{z}\,)\{s'\}_{\vec{z}}}{((p(\vec{e}\,;\vec{r}\,);\ s), \mu) \mapsto ((\{s'[\vec{y} \leftarrow \vec{w}][\vec{z} \hookleftarrow \vec{r}\,]\}_{\vec{r}};\ s), \mu[\vec{r} \leftarrow *\,])}$$

$$\frac{e =_\mu \bar{0}}{((\mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s\}_{\vec{z}};\ s'), \mu) \mapsto (s', \mu)}$$

$$\frac{e =_\mu \overline{q+1}}{((\mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s\}_{\vec{z}};\ s'), \mu) \mapsto ((\{\mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ \bar{q}\ \ \{s\}_{\vec{z}};\ s[y \leftarrow \bar{q}]\}_{\vec{z}};\ s'), \mu)}$$

Gödel System T (call-by-value)

(terms)

$$
\begin{array}{rcl}
t & ::= & x \\
 & | & 0 \\
 & | & S(t) \\
 & | & \mathbf{pred}(t) \\
 & | & t_1 \ \ t_2 \\
 & | & \lambda x.t \\
 & | & (t_1, \ldots, t_n) \\
 & | & \mathbf{let}\ (x_1, \ldots, x_n) = t_1 \ \mathbf{in}\ t_2 \\
 & | & \mathbf{rec}(t_1, t_2, t_3)
\end{array}
$$

(values)

$$
\begin{array}{rcl}
v & ::= & x \\
 & | & 0 \\
 & | & S(v) \\
 & | & (v_1, \ldots, v_n) \\
 & | & \lambda x.t
\end{array}
$$

(contexts)

$$
\begin{array}{rcl}
C[\;] & ::= & [\;] \\
 & | & C[\;]\ t \\
 & | & v\ C[\;] \\
 & | & S(C[\;]) \\
 & | & \mathbf{pred}(C[\;]) \\
 & | & \mathbf{rec}(C[\;], t_2, t_3) \\
 & | & \mathbf{rec}(v_1, C[\;], t_3) \\
 & | & \mathbf{rec}(v_1, v_2, C[\;]) \\
 & | & (v_1, \ldots v_{i-1}, C[\;], t_{i+1} \ldots, t_n) \\
 & | & \mathbf{let}\ (x_1, \ldots, x_n) = C[\;] \ \mathbf{in}\ t
\end{array}
$$

(evaluation rules)

$$
\begin{array}{rcl}
C[\lambda x.t\ v] & \rightsquigarrow & C[t[v/x]] \\
C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\
C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\
C[\mathbf{rec}(0, v_2, \lambda x.\lambda y.t)] & \rightsquigarrow & C[v_2] \\
C[\mathbf{rec}(S(v_1), v_2, \lambda x.\lambda y.t)] & \rightsquigarrow & C[\lambda x.\lambda y.t\ v_1\ \mathbf{rec}(v_1, v_2, \lambda x.\lambda y.t)] \\
C[\mathbf{let}\ (x_1, \ldots, x_n) = (v_1, \ldots, v_n)\ \mathbf{in}\ t] & \rightsquigarrow & C[t[v_1/x_1, \ldots, v_n/x_n]]
\end{array}
$$

Functional simple type system

$$\begin{array}{rcl} \sigma, \tau & ::= & \mathbf{nat} \\ & \mid & \mathbf{unit} \\ & \mid & \sigma \rightarrow \tau \\ & \mid & \tau_1 \times \ldots \times \tau_n \end{array}$$

$\boxed{\Gamma \vdash t : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \Gamma \vdash 0 : \mathbf{nat}$$

$$\frac{\Gamma \vdash t : \mathbf{nat}}{\Gamma \vdash \mathbf{pred}(t) : \mathbf{nat}} \qquad \frac{\Gamma \vdash t : \mathbf{nat}}{\Gamma \vdash S(t) : \mathbf{nat}}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n} \qquad \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau \qquad \Gamma \vdash u : \tau_1 \times \ldots \times \tau_n}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t\ : \sigma \rightarrow \tau} \qquad \frac{\Gamma \vdash t_1 : \sigma \rightarrow \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \tau \quad \Gamma, x : \mathbf{nat}, y : \tau \vdash t_3 : \tau}{\Gamma \vdash \mathbf{rec}(t_1, t_2, \lambda x.\lambda y.t_3) : \tau}$$

Translation from I to F

(expressions)

- $\bar{n}^\star = S^n(0)$
- $y^\star = y$
- $*^\star = ()$
- $(\mathbf{proc}\ (\mathbf{in}\ \vec{y}\,; \mathbf{out}\ \vec{z}\,)\{s\}_{\vec{z}})^\star = \lambda\vec{y}\,.(s)^\star_{\vec{z}}\,[\vec{()}/\vec{z}\,]$

(sequences)

- $(\varepsilon)^\star_{\vec{x}} = \vec{x}$
- $(\mathbf{var}\ y := e;\ s)^\star_{\vec{x}} = (s)^\star_{\vec{x}}[e^\star/y]$
- $(\mathbf{cst}\ y = e;\ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(y := e;\ s)^\star_{\vec{x}} = \mathbf{let}\ y = e^\star\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{inc}(y);\ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{succ}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\mathbf{dec}(y);\ s)^\star_{\vec{x}} = \mathbf{let}\ y = \mathbf{pred}(y)\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(p(\vec{e}\,; \vec{z}\,);\ s)^\star_{\vec{x}} = \mathbf{let}\ \vec{z}\, = p^\star\ \vec{e}^{\,\star}\ \mathbf{in}\ (s)^\star_{\vec{x}}$
- $(\{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \mathbf{let}\ \vec{z}\, = (s_1)^\star_{\vec{z}}\ \mathbf{in}\ (s_2)^\star_{\vec{x}}$
- $(\mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s_1\}_{\vec{z}};\ s_2)^\star_{\vec{x}} = \mathbf{let}\ \vec{z}\, = \mathbf{rec}(e^\star, \vec{z}\,, \lambda y.\lambda\vec{z}\,.(s_1)^\star_{\vec{z}})\ \mathbf{in}\ (s_2)^\star_{\vec{x}}$

Example: the Ackermann function

```
val Ack = fn (M , N) =>
    let val G = fn (Y) =>
            let val P = Y
                val P = succ(P)
            in P end
        val G = rec (M , G , fn I => fn (G) =>
            let val H = G
                val G = fn (Y) =>
                    let val P = 2
                        val P = rec (Y , P , fn J => fn (P) =>
                            let val P = H(P)
                            in P end)
                    in P end
            in G end)
        val Z = G(N)
    in Z end
```

Lock-step simulation

Theorem. *For any state* (s, μ)*, if* $\vec{x} = dom(\mu)$ *and* $\vec{z} \subseteq \vec{x}$ *we have:*

$$(s, \mu) \mapsto (s', \mu') \textit{ implies } (s)^{\star}_{\vec{z}}[\mu(\vec{x})^{\star}/\vec{x}] \rightsquigarrow (s')^{\star}_{\vec{z}}[\mu'(\vec{x})^{\star}/\vec{x}]$$

Proof. [Crolard et al., 2009] □

Remark.

- The updatable store is simply simulated by a meta-substitution.
- The call-by-value evaluation strategy of Loop$^{\omega}$ is built into the imperative syntax (explicit sequence and no nested procedure calls).
  $\Rightarrow$ the translated term is already in monadic normal form (all intermediate computations are named).

Imperative simple type system IS

Example:

sequence s	image of s by *
$x := * ;$	$\textbf{let } x = * \textbf{ in}$
$x := 0;$	$\textbf{let } x = 0 \textbf{ in}$
$\textbf{inc}(x);$	$\textbf{let } x = \textbf{succ}(x) \textbf{ in}$
$p(z; x);$	$\textbf{let } x = p\ z \textbf{ in}$
...	...

The functional term is typable (assuming that $p\ z$ is typable), so why should we prevent sequence s from being typable?

This remark leads to a pseudo-dynamic type system.

Pseudo-dynamic type system

$$\sigma, \tau \; ::= \; \mathbf{nat} \mid \mathbf{unit} \mid \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma})$$

$\boxed{\Gamma; \Omega \vdash e : \tau}$ (*expressions*)

$$\frac{x : \tau \in \Gamma; \Omega}{\Gamma; \Omega \vdash x : \tau}$$

$$\frac{}{\Gamma; \Omega \vdash \bar{q} : \mathbf{nat}}$$

$$\frac{}{\Gamma; \Omega \vdash * : \mathbf{unit}}$$

$$\frac{\Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \overrightarrow{\mathbf{unit}} \vdash s \rhd \vec{z} : \vec{\tau}}{\Gamma; \Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{z})\{s\}_{\vec{z}} : \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau})}$$

$\boxed{\Gamma; \Omega \vdash s \rhd \Omega}$ *(sequences)*

$$\frac{}{\Gamma; \Omega \vdash \varepsilon \rhd \Omega}$$

$$\frac{\Gamma; \Omega \vdash e\colon \tau \qquad \Gamma, y\colon \tau; \Omega \vdash s \rhd \Omega'}{\Gamma; \Omega \vdash \mathbf{cst}\ y = e;\ \ s\ \rhd \Omega'} \qquad \frac{\Gamma; \Omega \vdash e\colon \tau \qquad \Gamma; \Omega, y\colon \tau \vdash s \rhd \Omega', y\colon \tau'}{\Gamma; \Omega \vdash \mathbf{var}\ y := e;\ \ s\ \rhd \Omega'}$$

$$\frac{\Gamma; \vec{x} : \vec{\sigma} \vdash s \rhd \vec{x} : \vec{\tau} \qquad \Gamma; \Omega, \vec{x} : \vec{\tau} \vdash s' \rhd \Omega', \vec{x} : \vec{\tau}\,'}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \{s\}_{\vec{x}};\ \ s' \rhd \Omega', \vec{x} : \vec{\tau}\,'}$$

$$\frac{\Gamma; \Omega, y\colon \mathbf{nat} \vdash s \rhd \Omega', y\colon \tau}{\Gamma; \Omega, y\colon \mathbf{nat} \vdash \mathbf{inc}(y);\ \ s \rhd \Omega', y\colon \tau} \qquad \frac{\Gamma; \Omega, y\colon \mathbf{nat} \vdash s \rhd \Omega', y\colon \tau}{\Gamma; \Omega, y\colon \mathbf{nat} \vdash \mathbf{dec}(y);\ \ s \rhd \Omega', y\colon \tau}$$

$$\frac{\Gamma; \Omega, y\colon \sigma \vdash e\colon \tau \qquad \Gamma; \Omega, y\colon \tau \vdash s \rhd \Omega', y\colon \tau'}{\Gamma; \Omega, y\colon \sigma \vdash y := e;\ \ s \rhd \Omega', y\colon \tau'}$$

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash e\colon \mathbf{nat} \quad \Gamma, y\colon \mathbf{nat}; \vec{x} : \vec{\sigma} \vdash s \rhd \vec{x} : \vec{\sigma} \quad \Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash s' \rhd \Omega', \vec{x} : \vec{\sigma}\,'}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \mathbf{for}\ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}};\ \ s' \rhd \Omega', \vec{x} : \vec{\sigma}\,'}$$

$$\frac{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p\colon \mathbf{proc}\ (\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}\,) \quad \Gamma; \Omega, \vec{r} : \vec{\omega} \vdash \vec{e} : \vec{\tau} \quad \Gamma; \Omega, \vec{r} : \vec{\sigma} \vdash s \rhd \Omega', \vec{r} : \vec{\sigma}\,'}{\Gamma; \Omega, \vec{r} : \vec{\omega} \vdash p(\vec{e}\,; \vec{r}\,);\ \ s \rhd \Omega', \vec{r} : \vec{\sigma}\,'}$$

Properties of IS and FS

Theorem. (*type preservation*). *For any state* (s, μ)*, if* $\vec{z} \vdash (s, \mu) \rhd \Omega$ *in* **IS** *and* $(s, \mu) \mapsto (s', \mu')$ *then* $\vec{z} \vdash (s', \mu') \rhd \Omega$ *in* **IS**.

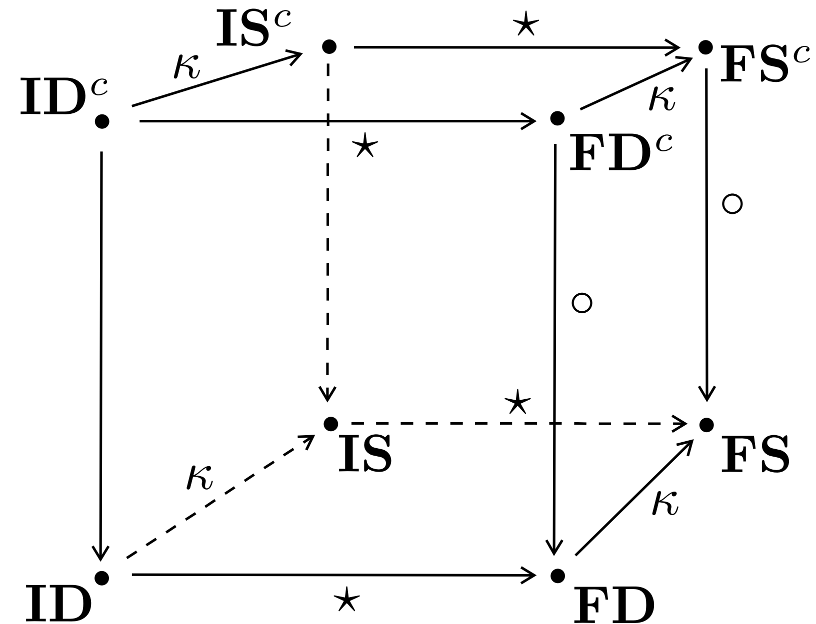
Proposition. (*progress*). *For any state* (s, μ)*, if* $\vec{z} \vdash (s, \mu) \rhd \Omega$ *in* **IS** *then either* $s = \varepsilon$ *and no more evaluation step can occur, or there is a unique state* (s', μ') *such that* $(s, \mu) \mapsto (s', \mu')$.

Proposition. (*termination*). *For any state* (s, μ)*, if* $\vec{z} \vdash (s, \mu) \rhd \Omega$ *in* **IS** *then the evaluation of* (s, μ) *terminates.*

Theorem. (*translation* * *preserves types*). *For any environments* Γ *and* Ω*, any expression* e*, any sequence* s *we have:*

- $\Gamma; \Omega \vdash e : \tau$ *in* **IS** *implies* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$ *in* **FS**.
- $\Gamma; \Omega \vdash s \rhd \Omega'$ *in* **IS** *implies* $\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}} : \vec{\sigma}^\star$ *in* **FS** *for any* $\vec{z} : \vec{\sigma} \subset \Omega'$.

Overview



I: imperative. **F**: functional.
D: dependent. **S**: simple.
$_^c$: classical.

$\star$: translation from **I** to **F**
κ : erasure function from **D** to **S**
$\circ$: $\neg\neg$-translation

Dependent functional type system (1)

Dependent type system (*à la* Leivant-Krivine-Parigot) parameterized by an equational system $\mathcal{E}$ (containing definitions for $\mathbf{s}, \mathbf{p}, +, \times$).

$$\begin{array}{lcll} \tau & ::= & \mathbf{nat}(n) & (\textit{special cases}) \\ & \mid & (n = m) & \\ & \mid & \forall \vec{\imath}\,(\tau_1 \Rightarrow \tau_2) & \forall i(\mathbf{nat}(i) \Rightarrow \tau) \\ & \mid & \exists \vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_n) & \exists i(\mathbf{nat}(i) \wedge \tau) \end{array}$$

$\boxed{\Gamma \vdash t : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \Gamma \vdash 0 : \mathbf{nat}(0)$$

$$\frac{\Gamma \vdash t : \mathbf{nat}(n)}{\Gamma \vdash S(t) : \mathbf{nat}(\mathbf{s}(n))} \qquad \frac{\Gamma \vdash t : \mathbf{nat}(n)}{\Gamma \vdash \mathbf{pred}(t) : \mathbf{nat}(\mathbf{p}(n))}$$

Dependent functional type system (2)

$$\frac{\Gamma \vdash t_1 \colon \forall \vec{\imath}\,(\sigma \Rightarrow \tau) \quad \Gamma \vdash t_2 \colon \sigma[\vec{n}/\vec{\imath}\,]}{\Gamma \vdash t_1\ t_2 : \tau[\vec{n}/\vec{\imath}\,]} \qquad \frac{\Gamma, x \colon \tau \vdash t \colon \sigma}{\Gamma \vdash \lambda x.t\ : \forall \vec{\imath}\,(\tau \Rightarrow \sigma)} \quad \vec{\imath} \notin \mathcal{FV}(\Gamma, \tau)$$

$$\frac{\Gamma \vdash t_1 \colon \tau_1[\vec{m}/\vec{\imath}\,] \quad \ldots \quad \Gamma \vdash t_k \colon \tau_k[\vec{m}/\vec{\imath}\,]}{\Gamma \vdash (t_1, \ldots, t_k) \colon \exists \vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_k)}$$

$$\frac{\Gamma, x_1 \colon \tau_1, \ldots, x_k \colon \tau_k \vdash t \colon \tau \qquad \Gamma \vdash u \colon \exists \vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_k)}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_k) = u\ \mathbf{in}\ t \colon \tau} \quad \vec{\imath} \notin \mathcal{FV}(\Gamma, \tau)$$

$$\frac{\Gamma \vdash t_1 \colon \mathbf{nat}(n) \quad \Gamma \vdash t_2 \colon \tau[\mathbf{0}/i] \quad \Gamma, x \colon \mathbf{nat}(i), y \colon \tau \vdash t_3 \colon \tau[\mathbf{s}(i)/i]}{\Gamma \vdash \mathbf{rec}(t_1, t_2, \lambda x.\lambda y.t_3) : \tau[n/i]} \quad i \notin \mathcal{FV}(\Gamma)$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma \vdash () \colon (n = m)} \qquad \frac{\Gamma \vdash t \colon \tau[n/i] \quad \Gamma \vdash u \colon (n = m)}{\Gamma \vdash t \colon \tau[m/i]}$$

Example: the addition function

From the defining equations for the addition:

$$\begin{array}{lcl} (1) \;\; x + 0 & = & x \\ (2) \;\; x + \mathbf{s}(i) & = & \mathbf{s}(x+i) \end{array}$$

You can derive the totality of $+$ in **FD**:

$$\dfrac{\dfrac{[y\colon \mathbf{nat}(m)] \qquad \dfrac{[x\colon \mathbf{nat}(n)]}{x\colon \mathbf{nat}(n+0)}\text{by (1)} \qquad \dfrac{\dfrac{[z\colon \mathbf{nat}(n+u)]}{S(z)\colon \mathbf{nat}(\mathbf{s}(n+u))}}{S(z)\colon \mathbf{nat}(n+\mathbf{s}(u))}\text{by (2)}}{\dfrac{\mathbf{rec}(y,x,\lambda i.\lambda z.S(z))\colon \mathbf{nat}(n+m)}{\lambda y.\mathbf{rec}(y,x,\lambda i.\lambda z.S(z))\colon \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n+m))}}}{\lambda x.\lambda y.\mathbf{rec}(y,x,\lambda i.\lambda z.S(z))\colon \forall n(\mathbf{nat}(n) \Rightarrow \forall m(\mathbf{nat}(m) \Rightarrow \mathbf{nat}(n+m)))}$$

The Ackermann function

Similarly, from the defining equations:

$$\begin{array}{lll} (a1) & \mathbf{a}(0,n) & = \mathbf{s}(n) \\ (a2) & \mathbf{a}(\mathbf{s}(z),0) & = \mathbf{s}(\mathbf{s}(0)) \\ (a3) & \mathbf{a}(z,\mathbf{a}(\mathbf{s}(z),u)) & = \mathbf{a}(\mathbf{s}(z),\mathbf{s}(u)) \end{array}$$

You can derive the totality of **a** in **FD**:

$$\begin{array}{rcl} ack & : & \forall m(\mathbf{nat}(m) \Rightarrow \forall n(\mathbf{nat}(n) \Rightarrow \mathbf{nat}(\mathbf{a}(m,n)))) \\ \text{where } ack & = & \lambda x.\mathbf{rec}(x, \lambda y.S(y), \lambda i.\lambda f.\lambda y.\mathbf{rec}(y, S(S(0)), \lambda j.\lambda k.(f\ k))) \end{array}$$

From [Leivant, 1990]:

Subject reduction: *If* $\Gamma \vdash t\colon \sigma$ *in* **FD** *and* $t \rightsquigarrow t'$ *then* $\Gamma \vdash t'\colon \sigma$.

Representation theorem: *Given an equational system* $\mathcal{E}$ *and an* n*-ary function symbol* f, *if* $\vdash_{\mathcal{E}} t\colon \forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \mathbf{nat}(f(\vec{n}))$ *is derivable in* **FD** *then* t *represents* f (that is: $t\ \bar{q} \rightsquigarrow^{\star} \overline{f(q)}$ for any numeral q).

Imperative dependent type system

$$\sigma,\tau \ ::= \ \mathbf{nat}(n) \mid \mathbf{proc}\ (\{\vec{\imath}\}\mathbf{in}\ \vec{\tau};\{\vec{\jmath}\}\mathbf{out}\ \vec{\sigma}) \mid n = m$$

$\boxed{\Gamma;\Omega \vdash e\colon \tau}$ *(expressions)*

$$\frac{x\colon \tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x\colon \tau}$$

$$\frac{}{\Gamma;\Omega \vdash \bar{q}\colon \mathbf{nat}(\mathbf{s}^q(\mathbf{0}))}$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma;\Omega \vdash * : n = m}$$

$$\frac{\Gamma;\Omega \vdash t\colon \tau[n/i] \qquad \Gamma;\Omega \vdash e\colon n = m}{\Gamma;\Omega \vdash t\colon \tau[m/i]}$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y} : \vec{\sigma}; \vec{z} : \vec{\top} \vdash s \rhd \{\vec{\jmath}\}\vec{z} : \vec{\tau}}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z})\{s\}_{\vec{z}} : \mathbf{proc}\ (\{\vec{\imath}\}\mathbf{in}\ \vec{\sigma};\{\vec{\jmath}\}\mathbf{out}\ \vec{\tau})}$$

$\boxed{\Gamma;\Omega \vdash s \rhd \{\vec{\jmath}\,\}\Omega'}$ *(sequences)*

$$\frac{}{\Gamma;\Omega[\vec{n}/\vec{\jmath}\,] \vdash \varepsilon \rhd \{\vec{\jmath}\,\}\Omega} \qquad \frac{\Gamma;\Omega \vdash s \rhd \{\vec{\jmath}\,\}\Omega[n/i] \qquad \Gamma;\Omega \vdash e\colon n = m}{\Gamma;\Omega \vdash s \rhd \{\vec{\jmath}\,\}\Omega[m/i]}$$

$$\frac{\Gamma;\Omega, y\colon \mathbf{nat}(\mathbf{s}(n)) \vdash s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau}{\Gamma;\Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y); s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau} \qquad \frac{\Gamma;\Omega, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau}{\Gamma;\Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y); s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau}$$

$$\frac{\Gamma;\Omega, y\colon \sigma \vdash e\colon \tau \qquad \Gamma;\Omega, y\colon \tau \vdash s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau'}{\Gamma;\Omega, y\colon \sigma \vdash y := e; s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau'} \qquad \frac{\Gamma;\vec{x}:\vec{\tau} \vdash s \rhd \{\vec{\jmath}\,\}\vec{x}:\vec{\sigma} \qquad \Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash s' \rhd \{\vec{\kappa}\}\Omega',\vec{x}:\vec{\sigma}\,'}{\Gamma;\Omega,\vec{x}:\vec{\tau} \vdash \{s\}_{\vec{x}}; s' \rhd \{\vec{\kappa}\,\}\Omega',\vec{x}:\vec{\sigma}\,'}$$

$$\frac{\Gamma;\Omega \vdash e\colon \tau \qquad \Gamma, y\colon \tau;\Omega \vdash s \rhd \{\vec{\jmath}\,\}\Omega'}{\Gamma;\Omega \vdash \mathbf{cst}\ \ y = e;\ \ s\ \rhd \{\vec{\jmath}\,\}\Omega'} \qquad \frac{\Gamma;\Omega \vdash e\colon \tau \qquad \Gamma;\Omega, y\colon \tau \vdash s \rhd \{\vec{\jmath}\,\}\Omega', y\colon \tau'}{\Gamma;\Omega \vdash \mathbf{var}\ \ y := e;\ \ s\ \rhd \{\vec{\jmath}\,\}\Omega'}$$

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\sigma}\,[\mathbf{0}/i] \vdash e\colon \mathbf{nat}(n) \quad \Gamma, y\colon \mathbf{nat}(i);\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\sigma}\,[\mathbf{s}(i)/i] \quad \Gamma;\Omega,\vec{x}:\vec{\sigma}\,[n/i] \vdash s' \rhd \{\vec{\jmath}\,\}\Omega',\vec{x}:\vec{\sigma}\,'}{\Gamma;\Omega,\vec{x}:\vec{\sigma}\,[\mathbf{0}/i] \vdash \mathbf{for}\ \ y := 0\ \ \mathbf{until}\ \ e\ \ \{s\}_{\vec{x}}; s' \rhd \{\vec{\jmath}\,\}\Omega',\vec{x}:\vec{\sigma}\,'}$$

$$\frac{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p\colon \mathbf{proc}\ (\{\vec{\imath}\,\}\mathbf{in}\ \vec{\tau}; \{\vec{\jmath}\,\}\mathbf{out}\ \vec{\sigma}) \quad \Gamma;\Omega,\vec{r}:\vec{\omega} \vdash \vec{e}:\vec{\tau}\,[\vec{u}/\vec{\imath}\,] \quad \Gamma;\Omega,\vec{r}:\vec{\sigma}\,[\vec{u}/\vec{\imath}\,] \vdash s \rhd \{\vec{\kappa}\,\}\Omega',\vec{r}:\vec{\sigma}\,'}{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p(\vec{e}\,;\vec{r}\,); s \rhd \{\vec{\kappa}\,\}\Omega',\vec{r}:\vec{\sigma}\,'}$$

*where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$ in (PROC) and $i \notin \mathcal{FV}(\Gamma)$ in (FOR) and $\vec{\jmath} \notin \mathcal{FV}(\Gamma,\Omega)$ and $\vec{\jmath} \setminus \vec{\kappa} \notin \mathcal{FV}(\Omega',\vec{\sigma}\,')$ in (BLOCK) and (CALL)

Typing the Ackermann procedure in ID

```
cst Ack = proc (in M, N; out Z) {

    var G := proc (in Y; out P) {
        P := Y;
        inc(P);
    }_P;

    for I := 0 until M {
        cst H = G;

        G := proc (in Y; out P) {
            P := 2;
            for J := 0 until Y {
                H(P; P);
            }_P;
        }_P;

    }_G;

    G(N; Z);
}_Z
```

$- (M\colon \mathbf{nat}(m), N\colon \mathbf{nat}(n))[Z\colon \top]$

$\quad - (Y\colon \mathbf{nat}(y))[P\colon \top]$

$\quad\quad [P\colon \mathbf{nat}(y)]$

$\quad\quad [P\colon \mathbf{nat}(\mathbf{s}(y))]$

$\quad [G\colon \mathbf{proc}\ (\{y\}\mathbf{in}\ \mathbf{nat}(y); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(0, y)))]$ by $(a1)$

$\quad - (I\colon \mathbf{nat}(i))[G\colon \mathbf{proc}\ (\{y\}\mathbf{in}\ \mathbf{nat}(y); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(i, y)))]$

$\quad\quad (H\colon \mathbf{proc}\ (\{y\}\mathbf{in}\ \mathbf{nat}(y); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(i, y))))$

$\quad\quad - (Y\colon \mathbf{nat}(y))[P\colon \top]$

$\quad\quad\quad [P\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(i), 0))]$ by $(a2)$

$\quad\quad\quad - (J\colon \mathbf{nat}(j))[P\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(i), j))]$

$\quad\quad\quad\quad [P\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(i), \mathbf{s}(j)))]$ by $(a3)$

$\quad\quad\quad [P\colon \mathbf{nat}(\mathbf{a}(\mathbf{s}(i), y)))]$

$\quad\quad [G\colon \mathbf{proc}\ (\{y\}\mathbf{in}\ \mathbf{nat}(y); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(\mathbf{s}(i), y))))]$

$\quad [G\colon \mathbf{proc}\ (\{y\}\mathbf{in}\ \mathbf{nat}(y); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(m, y)))]$

$\quad [Z\colon \mathbf{a}(m, n)]$

$(Ack\colon \mathbf{proc}\ (\{m, n\}\mathbf{in}\ \mathbf{nat}(m), \mathbf{nat}(n); \mathbf{out}\ \mathbf{nat}(\mathbf{a}(m, n))))$

Translation from ID to FD

Translation of dependent types

- $(t = u)^\star = (t = u)$
- $(\mathbf{nat}(u))^\star = \mathbf{nat}(u)$
- $(\mathbf{proc}\ (\{\vec{\imath}\,\}\mathbf{in}\ \vec{\tau}\,; \{\vec{\jmath}\,\}\mathbf{out}\ \vec{\sigma}\,))^\star = \forall \vec{\imath}\,(\vec{\tau}^{\,\star} \Rightarrow \exists \vec{\jmath}\,(\vec{\sigma}^{\,\star}))$

Theorem. (*translation* $\star$ *preserves types*). *For any environments* Γ *and* Ω, *any expression* e, *any sequence* s *we have:*

– $\Gamma; \Omega \vdash e : \tau$ *in* **ID** *implies* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$ *in* **FD**.

– $\Gamma; \Omega \vdash s \rhd \Omega'$ *in* **ID** *implies* $\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{z}} : \vec{\sigma}^{\,\star}$ *in* **FD** *for any* $\vec{z} : \vec{\sigma} \subset \Omega'$.

Corollary. (*representation theorem for* **ID**). *Given an equational system* $\mathcal{E}$ *and an* n*-ary function symbol* f, *if* $\vdash p$: $\mathbf{proc}(\{\vec{n}\,\}\mathbf{in}\ \mathbf{nat}(\vec{n}); \mathbf{out}\ \mathbf{nat}(f(\vec{n}\,)))$ *is derivable in* **ID** *then* p *represents* f.

Erasure functions for FD/ID

Erasure for FD types:

- $\kappa(n = m) = \mathbf{unit}$
- $\kappa(\mathbf{nat}(n)) = \mathbf{nat}$
- $\kappa(\forall\vec{\imath}\,(\sigma \Rightarrow \tau)) = \kappa\sigma \rightarrow \kappa\tau$
- $\kappa(\exists\vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_n) = \kappa\tau_1 \times \ldots \times \kappa\tau_n$

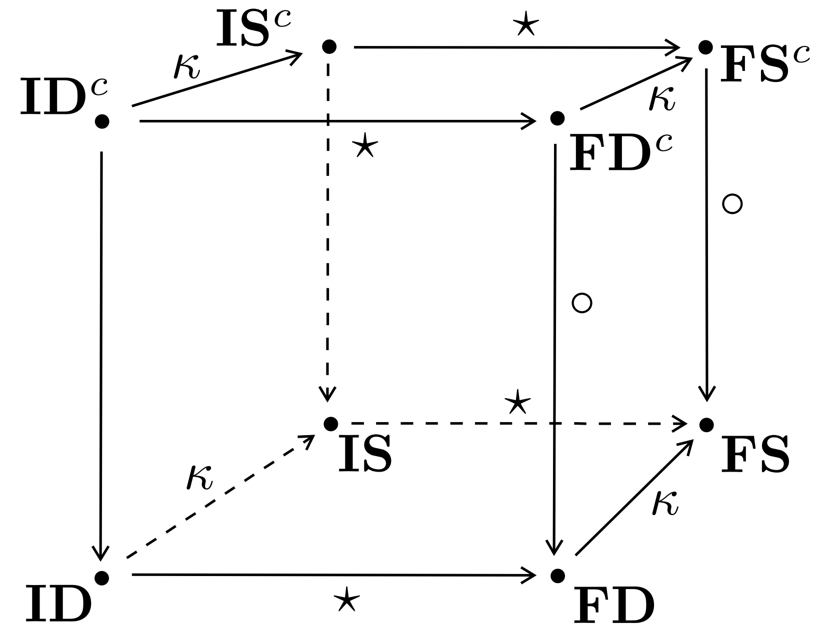
Proposition. *If* $\Gamma \vdash t\colon \sigma$ *is derivable in* **FD** *then* $\kappa\Gamma \vdash t\colon \kappa\sigma$ *is derivable in* **FS**.

Erasure for ID types:

- $\kappa(n = m) = \mathbf{unit}$
- $\kappa(\mathbf{nat}(n)) = \mathbf{nat}$
- $\kappa(\mathbf{proc}\ (\{\vec{\imath}\,\}\mathbf{in}\ \vec{\tau}; \{\vec{\jmath}\,\}\mathbf{out}\ \vec{\sigma}\,)) = \mathbf{proc}\ (\mathbf{in}\ \kappa\vec{\tau}; \mathbf{out}\ \kappa\vec{\sigma}\,)$

Proposition. *If* $\Gamma; \Omega \vdash s \rhd \Omega'$ *is derivable in* **ID** *then* $\kappa\Gamma; \kappa\Omega \vdash s \rhd \kappa\Omega'$ *is derivable in* **IS**.

Overview



I: imperative. **F**: functional.
D: dependent. **S**: simple.
$_^c$: classical.

$\star$: translation from **I** to **F**
κ : erasure function from **D** to **S**
$\circ$: $\neg\neg$-translation

System $\mathbf{FD^c}$

Given a propositional constant "absurd" written $\bot$, we define the negation $\neg\varphi$ as an abbreviation for $\varphi \Rightarrow \bot$.

We extend **FD** with two constants **callcc** and **throw** with the following types:

$$
\begin{aligned}
\textbf{callcc} &: (\neg\varphi \Rightarrow \varphi) \Rightarrow \varphi \\
\textbf{throw} &: (\neg\varphi \wedge \varphi) \Rightarrow \psi
\end{aligned}
$$

The semantics of **callcc** and **throw** is given by a CPS-transform.

The continuation monad

Computational meta-language with monad ∇:

$$\frac{\Gamma \vdash u : \varphi}{\Gamma \vdash \mathbf{val}\ u : \nabla\varphi}\ (\mathit{unit}) \qquad \frac{\Gamma \vdash u : \nabla\varphi \qquad \Gamma, x : \varphi \vdash t : \nabla\psi}{\Gamma \vdash \mathbf{let\ val}\ x = u\ \mathbf{in}\ t : \nabla\psi}\ (\mathit{bind})$$

The continuation monad ∇ is then defined as $\nabla\varphi = \neg_o\neg_o\varphi$ together with the following two abbreviations:

$$\begin{aligned} \mathbf{val}\ u &= \lambda z.(z\ u) \\ \mathbf{let\ val}\ x = u\ \mathbf{in}\ t &= \lambda z.(u\ \lambda x.(t\ z)) \end{aligned}$$

Moreover, control operators *callcc* and *throw* are definable:

$$\begin{aligned} \mathit{callcc} &: (\neg_o\varphi \Rightarrow \nabla\varphi) \Rightarrow \nabla\varphi \\ &= \lambda h.\lambda k.(h\ k\ k) \end{aligned}$$

$$\begin{aligned} \mathit{throw} &: (\neg_o\varphi \wedge \varphi) \Rightarrow \nabla\psi \\ &= \lambda(k, a).\lambda k'.(k\ a) \end{aligned}$$

¬¬-translation

Translation of dependent types from $\mathbf{FD}^c$ to $\mathbf{FD}$ (where $\nabla\varphi = \neg_o\neg_o\varphi$ and o is a fixed proposition constant):

$$\begin{array}{rcl} \mathbf{nat}(n)^\circ & = & \mathbf{nat}(n) \\ (n = m)^\circ & = & (n = m) \\ (\exists\vec{n}\,(\varphi_1 \wedge \ldots \wedge \varphi_n))^\circ & = & \exists\vec{n}\,(\varphi_1^\circ \wedge \ldots \wedge \varphi_n^\circ) \\ (\forall\vec{n}\,(\varphi \Rightarrow \psi))^\circ & = & \forall\vec{n}\,(\varphi^\circ \Rightarrow \nabla\psi^\circ) \\ (\neg\varphi)^\circ & = & \neg_o\varphi^\circ \\ \bot^\circ & = & o \end{array}$$

[Murthy, 1990] noticed that this translation acutally corresponds to Kuroda's ¬¬-translation [Kuroda, 1951].

CPS-translation

(values)

$$
\begin{array}{rcl}
()^\bullet & = & () \\
x^\bullet & = & x \\
0^\bullet & = & 0 \\
S(v)^\bullet & = & S(v^\bullet) \\
(\lambda x.u)^\bullet & = & (\lambda x.u^\circ) \\
(v_1, \ldots, v_k)^\bullet & = & (v_1^\bullet, \ldots, v_k^\bullet) \\
(\mathbf{callcc})^\bullet & = & \mathit{callcc} \\
(\mathbf{throw})^\bullet & = & \mathit{throw}
\end{array}
$$

(monadic normal forms)

$$
\begin{array}{rcl}
(v)^\circ & = & \mathbf{val}\ (v^\bullet) \\
(v_1\ v_2)^\circ & = & (v_1^\bullet\ v_2^\bullet) \\
(\mathbf{let}\ (x_1, \ldots, x_n) = t\ \mathbf{in}\ u)^\circ & = & \mathbf{let\ val}\ (x_1, \ldots, x_n) = t^\bullet\ \mathbf{in}\ u^\circ \\
\mathbf{rec}(v_1, v_2, \lambda x.\lambda y.t)^\circ & = & \mathbf{rec}(v_1^\bullet, v_2^\circ, \lambda x.\lambda r.\mathbf{let\ val}\ y = r\ \mathbf{in}\ t^\circ) \\
\mathbf{pred}(v)^\circ & = & \mathbf{val\ pred}(v^\bullet)
\end{array}
$$

Non-local jumps

We extend **I** with *void*, $\neg$ and two procedural constants:

$$\begin{array}{lcl}\mathbf{callcc} & : & \mathbf{proc}\ (\mathbf{in}\ \mathbf{proc}\ (\mathbf{in}\ \neg\vec{\sigma}\,;\mathbf{out}\ \vec{\sigma}\,);\mathbf{out}\ \vec{\sigma}\,) \\ \mathbf{throw} & : & \mathbf{proc}\ (\mathbf{in}\ \neg\vec{\sigma}\,,\vec{\sigma}\,;\mathbf{out}\ \vec{\tau}\,)\end{array}$$

And we introduce the following two abbreviations:

$$\begin{array}{rcl} k{:}\,\{s\}_{\vec{z}} & = & \mathbf{cst}\ \vec{z}\,'=\vec{z}\,;\ \mathbf{callcc}(\mathbf{proc}(\mathbf{in}\ k;\mathbf{out}\ \vec{z}\,)\{\vec{z}\,:=\vec{z}\,';\ s\}_{\vec{z}};\vec{z}\,) \\ \mathbf{jump}(k,\vec{e}\,)_{\vec{z}} & = & \mathbf{throw}(k,\vec{e}\,;\vec{z}\,)\end{array}$$

Remark.

- jump to *the end* of the block with label k (i.e. exit the block)
- but labels are first-class citizens (so jump is more general than exit)

Filinski's encoding of shift/reset

The semantics of **shift**/**reset** [Danvy and Filinski, 1989] is given by a double cps-transform (required to obtain genuine cps-terms).

- The first cps-transform corresponds to a parametrized continuation monad [Atkey, 2008] :
$$M(\alpha, \beta, \gamma) \;=\; (\gamma \to \beta) \to \alpha$$
- The second cps-transform corresponds the usual continuation monad (where o is a fixed output type):
$$\nabla\sigma \;=\; (\sigma \to o) \to o$$

Composing both transformations gives the following parametrized monad:

$$\begin{aligned}
(\gamma \to \nabla\beta) \to \nabla\alpha &\cong ((\gamma \times (\beta \to o)) \to o) \to ((\alpha \to o) \to o) \\
&\cong (\alpha \to o) \to (((\gamma \times (\beta \to o)) \to o) \to o) \\
&= (\alpha \to o) \to \nabla(\gamma \times (\beta \to o))
\end{aligned}$$

which corresponds to a state-passing style transform (where the state is a continuation) followed by the usual cps-transform.

Back to direct style

Encoding from [Filinski, 1994] (with a fixed anwer type *ans*).

$(*\ reset : (unit \rightarrow ans) \rightarrow ans\ *)$
fun *reset t* = **callcc** (**fn** *k* ⇒
  let val *m* = !*mk* **in**
  mk := (**fn** *r* ⇒ (*mk* := *m*; *k r*)); **throw** !*mk* (*t* ())
  end)

$(*\ shift : ((\text{'}a \rightarrow ans) \rightarrow ans) \rightarrow \text{'}a\ *)$
fun *shift h* = **callcc** (**fn** *k* ⇒
  throw !*mk* (*h* (**fn** *v* ⇒ *reset* (**fn** () ⇒ **throw** *k v*))))

where: **type** 'a K = 'a → *void*
val callcc: ('a K → 'a) → 'a
val throw: 'a K → 'a → 'b

and **val mk**: (*ans* → *void*) *ref*

Reset in state-passing style

Same encoding but in state-passing style and with *polymorphic answer type*.

```
val reset: ('a K → 'c * 'c K) * 'd K → 'a * 'd K =
   fn (p, mk') ⇒
       let
          val (r, mk) = ((), mk')
          val (r', mk') = (r, mk)
          val (r, mk) =
               callcc (fn k ⇒
                          let
                             val (r, mk) = (r', mk')
                             val m = mk
                             val mk = fn r ⇒
                                          let val z = throw k (r, m)
                                          in z end
                             val (y, mk) = p(mk)
                             val (r, mk) = throw mk y
                          in (r, mk) end)
       in (r, mk) end
```

Shift in state passing style

```
val shift: (('a * 'b K → 'c * 'b K) * 'd K → 'e * 'e K) * 'd K → 'a * 'c K =
  fn (p, mk') ⇒
     let
        val (r, mk) = ((), mk')
        val (r', mk') = (r, mk)
        val (r, mk) =
            callcc (fn k ⇒
                      let
                         val (r, mk) = (r', mk')
                         val q = fn (v, mk) ⇒
                                   let val (r, mk) =
                                          reset (fn mk ⇒
                                                   let val (z, mk) = throw k (v, mk)
                                                   in (z, mk) end, mk)
                                   in (r, mk) end
                         val (y, mk) = p(q, mk)
                         val (r, mk) = throw mk y
                      in (r, mk) end)
     in (r, mk) end
```

Inperative encoding of shift/reset

The functional types:

$$\begin{array}{rcl} \mathit{reset} & : & (\neg\alpha \Rightarrow \gamma \wedge \neg\gamma) \wedge \neg\delta \Rightarrow \alpha \wedge \neg\delta \\ \mathit{shift} & : & ((\alpha \wedge \neg\beta \Rightarrow \gamma \wedge \neg\beta) \wedge \neg\delta \Rightarrow \varepsilon \wedge \neg\varepsilon) \wedge \neg\delta \Rightarrow \alpha \wedge \neg\gamma \end{array}$$

are translated into:

$$\begin{array}{rcl} \mathit{reset} & : & \mathbf{proc}(\mathbf{in}\ \mathbf{proc}(\mathbf{in}\ \neg\alpha; \mathbf{out}\ \beta, \neg\beta), \neg\gamma; \mathbf{out}\ \alpha, \neg\gamma) \\ \mathit{shift} & : & \mathbf{proc}(\mathbf{in}\ \mathbf{proc}(\mathbf{in}\ \mathbf{proc}(\mathbf{in}\ \alpha, \neg\beta; \mathbf{out}\ \gamma, \neg\beta), \neg\delta; \mathbf{out}\ \epsilon, \neg\epsilon), \neg\delta; \mathbf{out}\ \alpha, \neg\gamma) \end{array}$$

Abbreviations for global mutable variables:

$$\begin{array}{rcl} \mathbf{proc}(\mathbf{in}\ \vec{x}; \mathbf{out}\ \vec{y})_{\vec{z}}\{s\}_{\vec{y},\vec{z}} & = & \mathbf{proc}(\mathbf{in}\ \vec{x}, \vec{z}\,'; \mathbf{out}\ \vec{y}, \vec{z})\{\vec{z} := \vec{z}\,'; s\}_{\vec{y},\vec{z}} \\ p(\vec{e}; \vec{y})_{\vec{z}} & = & p(\vec{e}, \vec{z}; \vec{y}, \vec{z}) \end{array}$$

Imperative encoding of shift/reset

$$
\begin{array}{lll}
\textbf{cst}\ \textit{reset} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{\\
& & \quad k{:}\ \{\\
& & \quad\quad \textbf{cst}\ m\ =\ mk;\\
& & \quad\quad mk\ :=\ \textbf{proc}(\textbf{in}\ r; \textbf{out}\ z)\{\textbf{jump}\ (k, r, m)_z;\}_z;\\
& & \quad\quad \textbf{var}\ y;\ \ p(;y)_{mk};\ \textbf{jump}\ (mk, y)_{r,mk};\\
& & \quad \}_{r,mk};\\
& & \}_{r,mk};\\
\textbf{cst}\ \textit{shift} & = & \textbf{proc}(\textbf{in}\ p; \textbf{out}\ r)_{mk}\{\\
& & \quad k{:}\ \{\\
& & \quad\quad \textbf{cst}\ q\ =\ \textbf{proc}\ q(\textbf{in}\ v; \textbf{out}\ r)_{mk}\{\\
& & \quad\quad\quad \textit{reset}\ (\textbf{proc}(\textbf{out}\ z)_{mk}\{\textbf{jump}\ (k, v, mk)_{z,mk};\}_{z,mk}; r);\\
& & \quad\quad \}_{r,mk};\\
& & \quad\quad \textbf{var}\ y;\ \ p(q;y)_{mk};\ \textbf{jump}\ (mk, y)_{r,mk};\\
& & \quad \}_{r,mk};\\
& & \}_{r,mk}
\end{array}
$$

Example

Problem:

- How do you prove the correctness of this program from [Wadler, 1994]?

$$\begin{array}{l}\mathbf{let}\ g = (\mathbf{reset}\ (\mathbf{if}\ (\mathbf{shift}\ \lambda f.f)\ \mathbf{then}\ 2\ \mathbf{else}\ 3)) \\ \mathbf{in}\ (g\ \mathit{True}) + (g\ \mathit{False})\end{array}$$

Informally:

- "*Here f (and hence g) is bound to the function that returns 2 if passed True, and 3 if passed False, hence the value of the given term is 5.*"

Formally:

- Translate into an imperative program, with **shift**/**reset** defined from **callcc**/**throw** and a global meta-continuation in state passing style.
- Derive the expected specification in $\mathbf{ID}^c$.

Example

Given the following sequence s:

$$
\begin{array}{l}
\mathbf{cst}\ q = \mathbf{proc}(;\ \mathbf{out}\ r)_{mk}\,\{ \\
\quad \mathbf{cst}\ p = \mathbf{proc}(\mathbf{in}\ f;\ \mathbf{out}\ h)_{mk}\,\{\ h := f;\} \\
\quad \mathbf{var}\ b;\ \mathit{shift}(p;\ b)_{mk}; \\
\quad r := 3; \\
\quad \mathbf{for}\ i := 0\ \mathbf{until}\ b\,\{ \\
\qquad r := 2; \\
\quad \}; \\
\}; \\
\mathbf{var}\ \ g;\ \ \mathit{reset}(q;\ g)_{mk}; \\
\mathbf{var}\ \ x;\ g(0;\ x)_{mk}; \\
\mathbf{var}\ \ y;\ \ g(1;\ y)_{mk}; \\
\mathit{add}(x,\ y;\ z)_{mk};
\end{array}
$$

We can derive $z{:}\top \vdash s \triangleright z{:}\,\mathbf{nat}(f_{32}(0) + f_{32}(1))$ in $\mathbf{ID}^c$ where f_{32} is defined by:

$$
\begin{array}{l}
f_{32}(0) = 3 \\
f_{32}(S(i)) = 2
\end{array}
$$

Work in progress

Embedding a Floyd-Hoare logic in **ID** is straightforward:

$$\Gamma; \Omega \vdash \{\sigma\} s \{\tau\} \rhd \Omega'$$

becomes (where *assert* is some fixed global variable):

$$\Gamma; \Omega, \mathit{assert}\colon \sigma \vdash s \rhd \mathit{assert}\colon \tau, \Omega'$$

The consequence rule:

$$\frac{\Gamma, \Omega \vdash \varphi' \Rightarrow \varphi \qquad \Gamma; \Omega \vdash \{\varphi\} s \{\psi\} \rhd \Omega' \qquad \Gamma, \Omega \vdash \psi \Rightarrow \psi'}{\Gamma; \Omega \vdash \{\varphi'\} s \{\psi'\} \rhd \Omega'}$$

is derivable if $\varphi' \Rightarrow \varphi$ and $\psi \Rightarrow \psi'$ are negative (without computational content). But what about classical logic ($\mathbf{ID}^c$)?

Possible solution: rely on the modality from [Thielecke, 2008].

Conclusion and future work

- Type systems and translation formalized with Ott multi-prover front-end and Twelf backend as executables specifications. Need to check the meta-theory (certainly with Coq).
- Define a direct transition semantics for LOOP$^{\omega}$+jumps
  - stack based abstract machine
  - implements imperative coroutines, generators...
- Consider extensions to:
  - data structures: arrays/lists (easy), trees (imperative syntax ?)
  - second order arithmetic (modules and genericity)

More details

- "A program logic for higher-order procedural variables and non-local jumps" (with E. Polonowski). 2009, pp 1-44 (submitted).
- "Extending the Loop Language with Higher-Order Procedural Variables" (with E. Polonowski and P. Valarcher). ACM Transactions on Computational Logic. Special Issue on Implicit Computational Complexity. Volume 10, Number 4, 2009, pp 1-37.
- "A Formally Specified Program Logic for Higher-Order Procedural Variables and non-local Jumps." LACL Technical Report 10 (2009).
- "A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables." (with E. Polonowski). LACL Technical Report 03 (2009).
- "LoopW – Technical Reference (v0.3)." E. Polonowski. LACL Technical Report 08 (2009).

Available from:

`http://lacl.u-pec.fr/crolard`

Bibliography

[Atkey, 2008] Atkey, R. (2008). Parameterised notions of computation. *Journal of Functional Programming*.

[Crolard et al., 2009] Crolard, T., Polonowski, E., and Valarcher, P. (2009). Extending the loop language with higher-order procedural variables. *Special issue of ACM TOCL on Implicit Computational Complexity*, 10(4):1–37.

[Danvy and Filinski, 1989] Danvy, O. and Filinski, A. (1989). A functional abstraction of typed contexts. Technical report, Copenhagen University.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A discipline of programming*. Prentice Hall.

[Donahue, 1977] Donahue, J. E. (1977). Locations considered unnecessary. *Acta Inf.*, 8:221–242.

[Felleisen et al., 1987] Felleisen, M., Friedman, D. P., Kohlbecker, E., and Duba, B. F. (1987). A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237.

[Filinski, 1994] Filinski, A. (1994). Representing monads. In *Conference Record of the Twenty-First Annual Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon.

[Filliâtre, 2003] Filliâtre, J.-C. (2003). Verification of non-functional programs using interpretations in type theory. *J. Funct. Program*, 13(4):709–745.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1.

[Griffin, 1990] Griffin, T. G. (1990). A formulæ-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Langages*, pages 47–58.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Honda et al., 2005] Honda, K., Yoshida, N., and Berger, M. (2005). An observationally complete program logic for imperative higher-order functions. *Symposium on Logic in Computer Science, LICS*, 5:270–279.

[Krivine and Parigot, 1990] Krivine, J.-L. and Parigot, M. (1990). Programming with proofs. *J. Inf. Process. Cybern. EIK*, 26(3):149–167.

[Kuroda, 1951] Kuroda, S. (1951). Intuitionistische untersuchungen der formalistischen logik. *Nagoya Math. J*, 2:35–47.

[Landin, 1965a] Landin, P. J. (1965a). A correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101.

[Landin, 1965b] Landin, P. J. (1965b). A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research.

[Leivant, 1990] Leivant, D. (1990). Contracting proofs to programs. In *Logic and Computer Science*, pages 279–327. Academic Press.

[Meyer and Ritchie, 1976] Meyer, A. R. and Ritchie, D. M. (1976). The complexity of loop programs. In *Proc. ACM Nat. Meeting*.

[Morrisett et al., 1999] Morrisett, G., Walker, D., Crary, K., and Glew, N. (1999). From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568.

[Murthy, 1990] Murthy, C. R. (1990). *Extracting Constructive Content from Classical proofs*. PhD thesis, Cornell University, Department of Computer Science.

[Nanevski et al., 2006] Nanevski, A., Morrisett, G., and Birkedal, L. (2006). Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73. ACM New York, NY, USA.

[O'Donnell, 1982] O'Donnell, M. J. (1982). A critique of the foundations of hoare style programming logics. *Commun. ACM*, 25(12):927–935.

[Parigot, 1992] Parigot, M. (1992). $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Logic Prog. and Autom. Reasoning*, volume 624 of *LNCS*, pages 190–201.

[Thielecke, 2008] Thielecke, H. (2008). Control effects as a modality. *Journal of Functional Programming*, 19:17–26.

[Wadler, 1994] Wadler, P. (1994). Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55.

[Xi, 2000] Xi, H. (2000). Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara.