Certification de programmes impératifs d'ordre supérieur avec mécanismes de contrôle

Tristan Crolard

LACL – Université Paris-Est

31 mars 2010

A Critique of the Foundations of Hoare Style Programming Logics

« **Abstract.**
[...] Several popular rules in the Hoare language are, in fact, not sound. These rules have been accepted because they have not been subjected to sufficiently strong standards of correctness.
[...] Convenient and elegant rules for reasoning about certain programming constructs will probably require a more flexible notation than Hoare's ».

[O'Donnell, 1982]

A Critique of the Foundations of Hoare Style Programming Logics

« **Conclusion.**
[...] The problem seems to be that partial correctness reasoning in the Hoare language is very natural for programs with only conditionals and loops for control structures, but not for programs with defined functions and/or **Gotos**.
Goto commands destroy the Hoare style analysis of programs by structural induction [...]. **Goto** commands are handled very naturally in the Floyd style of reasoning. »

[O'Donnell, 1982]

« 4.2.3 Inference for goto's and Labels

We now describe the extension of specification logic to encompass goto statements and labels. The basic idea behind this extension was first presented in [Clint and Hoare, 1972]. To avoid a full-fledged exposition of continuation semantics, which is beyond the scope of this book, our description will be somewhat informal. [...] »

[Reynolds, 1981]

The Craft of Programming

« 4.2.3 Inference for goto's and Labels

We now describe the extension of specification logic to encompass goto statements and labels. The basic idea behind this extension was first presented in [Clint and Hoare, 1972]. To avoid a full-fledged exposition of continuation semantics, which is beyond the scope of this book, our description will be somewhat informal. [...] »

[Reynolds, 1981]

« Although we have not formalized the treatment of goto's and labels in separation logic (or Hoare logic), it is essentially straightforward (except for jumps out of blocks or procedure bodies). »

[Reynolds, 2008]

Mécanismes de contrôle

- *locaux*
  - **goto**
- *non-locaux* **sans copie de pile**
  - sortie « brutale » de bloc/boucle/fonction : **exit**/**break**/**return**
  - sauts non-locaux : **setjmp**/**longjmp** (C)
  - exceptions
- *non-locaux* **avec copie de pile**
  - générateurs (Icon, ... , C#)
  - **getcontext**/**setcontext** (POSIX/C)
  - coroutines (Simula-67, Modula-2, ... , Lua)
- *non-locaux* **hors discipline de pile**
  - continuations : **callcc**/**throw** (Scheme, SML/NJ)
  - continuations délimitées : **shift**/**reset** (Scala)

Mécanismes de contrôle

- *locaux*
  - **goto**
- *non-locaux* **sans copie de pile**
  - sortie « brutale » de bloc/boucle/fonction : **exit**/**break**/**return**
  - sauts non-locaux : **setjmp**/**longjmp** (C)
  - exceptions
- *non-locaux* **avec copie de pile**
  - générateurs (Icon, ... , C#)
  - **getcontext**/**setcontext** (POSIX/C)
  - coroutines (Simula-67, Modula-2, ... , Lua)
- *non-locaux* **hors discipline de pile**
  - continuations : **callcc**/**throw** (Scheme, SML/NJ)
  - continuations délimitées : **shift**/**reset** (Scala)

Mécanismes de contrôle

- *locaux*
  - **goto**
- *non-locaux* **sans copie de pile**
  - sortie « brutale » de bloc/boucle/fonction : **exit**/**break**/**return**
  - sauts non-locaux : **setjmp**/**longjmp** (C)
  - exceptions
- *non-locaux* **avec copie de pile**
  - générateurs (Icon, ... , C#)
  - **getcontext**/**setcontext** (POSIX/C)
  - coroutines (Simula-67, Modula-2, ... , Lua)
- *non-locaux* **hors discipline de pile**
  - continuations : **callcc**/**throw** (Scheme, SML/NJ)
  - continuations délimitées : **shift**/**reset** (Scala)

Mécanismes de contrôle

- *locaux*
  - **goto**
- *non-locaux* **sans copie de pile**
  - sortie « brutale » de bloc/boucle/fonction : **exit**/**break**/**return**
  - sauts non-locaux : **setjmp**/**longjmp** (C)
  - exceptions
- *non-locaux* **avec copie de pile**
  - générateurs (Icon, ... , C#)
  - **getcontext**/**setcontext** (POSIX/C)
  - coroutines (Simula-67, Modula-2, ... , Lua)
- *non-locaux* **hors discipline de pile**
  - continuations : **callcc**/**throw** (Scheme, SML/NJ)
  - continuations délimitées : **shift**/**reset** (Scala)

Du « goto » au « callcc »

L'ancêtre de **callcc** est l'opérateur **J** de [Landin, 1965b] dont le rôle était de permettre une *traduction fonctionnelle en style directe* (sans continuations explicites) du **goto** de Algol 60 [Landin, 1965a].

Références

- [Landin, 1965a] "A correspondence between ALGOL 60 and Church's Lambda-notation"
- [Landin, 1965b] "A Generalization of Jumps and Labels"

Du « goto » au « callcc »

L'ancêtre de **callcc** est l'opérateur **J** de [Landin, 1965b] dont le rôle était de permettre une *traduction fonctionnelle en style directe* (sans continuations explicites) du **goto** de Algol 60 [Landin, 1965a].

Références

- [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."

Du « goto » au « callcc »

L'ancêtre de **callcc** est l'opérateur **J** de [Landin, 1965b] dont le rôle était de permettre une *traduction fonctionnelle en style directe* (sans continuations explicites) du **goto** de Algol 60 [Landin, 1965a].

Le **callcc** donne un contenu calculatoire au raisonnement par l'absurde.

Références

- [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"

Correspondance entre preuves et programmes

Isomorphisme de Curry-Howard :

formule	$\longleftrightarrow$	type/spécification
preuve	$\longleftrightarrow$	programme
normalisation	$\longleftrightarrow$	évaluation

Applications :

- Contenu calculatoire des preuves
- Implantation des assistants de preuve
- Conception de systèmes de types statiques
- Cadre formel pour des « logiques de programmes »

Correspondance entre preuves et programmes

Isomorphisme de Curry-Howard :

formule	$\longleftrightarrow$	type/spécification
preuve	$\longleftrightarrow$	programme
normalisation	$\longleftrightarrow$	évaluation

Applications :

- Contenu calculatoire des preuves
- Implantation des assistants de preuve
- Conception de systèmes de types statiques
- Cadre formel pour des « logiques de programmes »

Correspondance entre preuves et programmes

Isomorphisme de Curry-Howard :

formule	$\longleftrightarrow$	type/spécification
preuve	$\longleftrightarrow$	programme
normalisation	$\longleftrightarrow$	évaluation

Applications :

- Contenu calculatoire des preuves
- Implantation des assistants de preuve
- Conception de systèmes de types statiques
- Cadre formel pour des « logiques de programmes »

Correspondance entre preuves et programmes

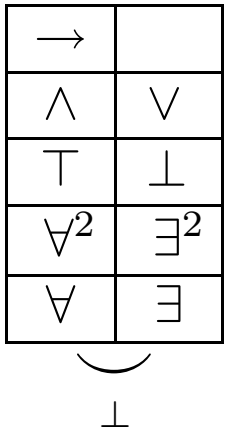
Isomorphisme de Curry-Howard :

formule	$\longleftrightarrow$	type/spécification
preuve	$\longleftrightarrow$	programme
normalisation	$\longleftrightarrow$	évaluation

Applications :

- Contenu calculatoire des preuves
- Implantation des assistants de preuve
- Conception de systèmes de types statiques
- Cadre formel pour des « logiques de programmes »

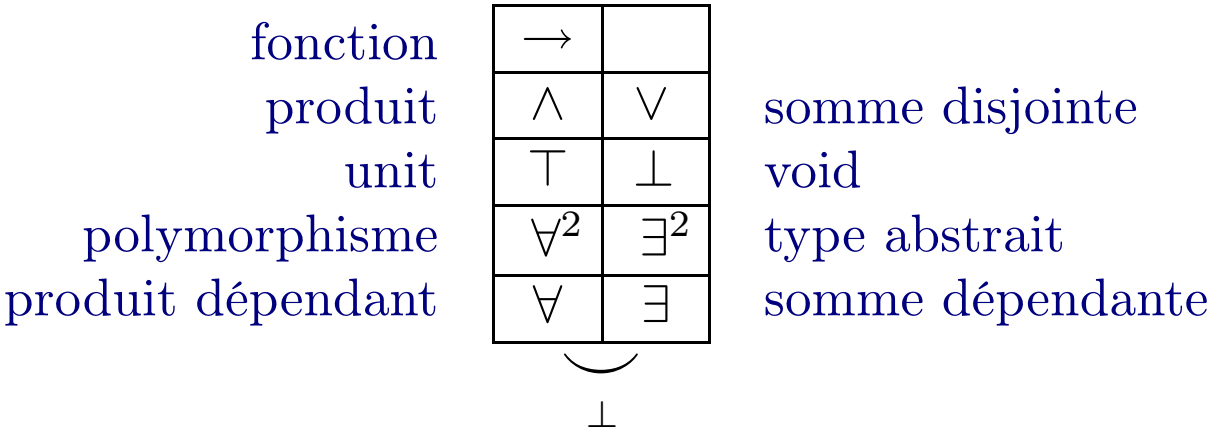
Correspondance entre formules et types



Où $\_^\perp$ représente :

- la dualité et la négation en logique classique

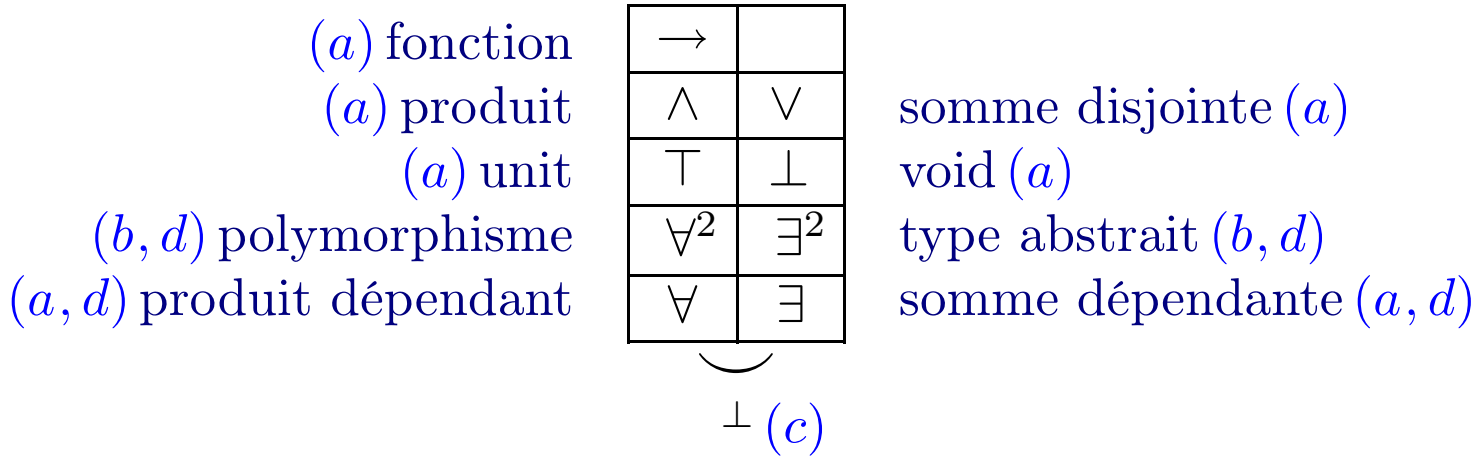
Correspondance entre formules et types



Où $\_^{\perp}$ représente :

- la dualité et la négation en logique classique
  la négation permet de typer les continuations de première classe

Correspondance entre formules et types



Où $\_^{\perp}$ représente :

- la dualité et la négation en logique classique
  la négation permet de typer les continuations de première classe

Références :

a) [Curry and Feys, 1958] [Howard, 1969]
b) [Girard, 1972] [Reynolds, 1974] [Mitchell and Plotkin, 1985]
c) [Griffin, 1990] [Murthy, 1990]
d) [Leivant, 1990] [Krivine and Parigot, 1990] [Parigot, 1992] ...

Objectif

Revisiter le travail de Landin à la lumière de l'interprétation calculatoire de la logique classique.

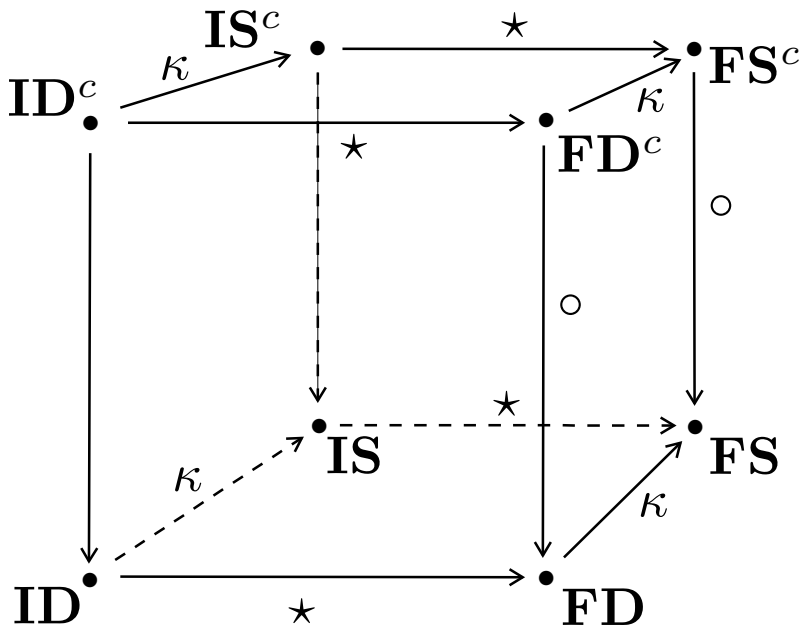
Définir :

- un système de types dépendants classique pour un langage impératif (avec variables mutables, commandes, séquences, boucles...) par traduction dans un système de types dépendants classique fonctionnel.

et obtenir :

- une logique de programmes *en style direct* pour un langage impératif d'ordre supérieur avec mécanismes de contrôle.

Vue d'ensemble



I: imperatif. **F**: fonctionnel.
D: dépendant. **S**: simple.
$\_^c$: classique.

$\star$: traduction de **I** vers **F**
κ : effacement de **D** vers **S**
$\circ$: $\neg\neg$-traduction

Langages et systèmes de types

ID/FD. Arithmétique de Heyting (en fait **M1LP** [Leivant, 1990]);

ID^c/FD^c. Arithmétique de Peano;

F. Système T de Gödel (en appel par valeur);

F^c. Système T de Gödel + **callcc**/**throw** (en appel par valeur);

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

I^c. Loop$^\omega$ + **sauts non-locaux**.

Loop$^{\omega}$: un langage impératif « pur »

Une version impérative du Système T de Gödel :

- une extension du langage Loop [Meyer and Ritchie, 1976] avec des procédures d'ordre supérieur et des variables procédurales ;
- une sémantique simple sans adresses mémoire [Donahue, 1977];
- un système de types pseudo-dynamique [Morrisett et al., 1999];
- un système de types dépendants [Xi, 2000].

$\textsc{Loop}^{\omega}$ – syntaxe

$$
\begin{array}{rrcl}
(\textit{commande}) & c & ::= & \{s\}_{\vec{x}} \\
 & & | & \textbf{for } y := 0 \textbf{ until } e \ \{s\}_{\vec{x}} \\
 & & | & p(\vec{e}\,;\vec{y}) \quad | \quad \textbf{inc}(y) \quad | \quad \textbf{dec}(y) \\
 & & | & \vec{y} := e \\
 & & & \\
(\textit{séquence}) & s & ::= & \varepsilon \\
 & & | & c\,;\ s \\
 & & | & \textbf{cst } y = e;\ s \\
 & & | & \textbf{var } y := e;\ s \\
 & & & \\
(\textit{procédure anonyme}) & a & ::= & \textbf{proc } (\textbf{in } \vec{y}\,;\textbf{out } \vec{z}\,)\ \{s\}_{\vec{z}} \\
 & & & \\
(\textit{expression}) & e & ::= & y \quad | \quad a \quad | \quad \bar{q} \quad | \quad * \quad | \quad (e_1, ..., e_n) \\
(\textit{procédure}) & p & ::= & y \quad | \quad a
\end{array}
$$

Loop$^\omega$ – remarques

- **Annotations de blocs :** pour un bloc $\{s\}_{\vec{x}}$, les variables $\vec{x}$ correspondent aux variables mutables libres qui apparaissent dans la séquence s (elles peuvent être inférées automatiquement).
- **Pas d'alias :** dans un appel de procédure $p(\vec{e}\,;\,\vec{y}\,)$ les y_i doivent être deux à deux disjoints.
- **Pas de « backpatching » :** les variables mutables globales ne sont pas autorisées dans le corps d'une procédure. Le bloc suivant (qui définit un point fixe) est donc illégal :

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

Loop$^\omega$ – remarques

- **Annotations de blocs :** pour un bloc $\{s\}_{\vec{x}}$, les variables $\vec{x}$ correspondent aux variables mutables libres qui apparaissent dans la séquence s (elles peuvent être inférées automatiquement).
- **Pas d'alias :** dans un appel de procédure $p(\vec{e}\,;\,\vec{y}\,)$ les y_i doivent être deux à deux disjoints.
- **Pas de « backpatching » :** les variables mutables globales ne sont pas autorisées dans le corps d'une procédure. Le bloc suivant (qui définit un point fixe) est donc illégal :

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

Loop$^\omega$ – remarques

- **Annotations de blocs :** pour un bloc $\{s\}_{\vec{x}}$, les variables $\vec{x}$ correspondent aux variables mutables libres qui apparaissent dans la séquence s (elles peuvent être inférées automatiquement).
- **Pas d'alias :** dans un appel de procédure $p(\vec{e}\,;\,\vec{y}\,)$ les y_i doivent être deux à deux disjoints.
- **Pas de « backpatching » :** les variables mutables globales ne sont pas autorisées dans le corps d'une procédure. Le bloc suivant (qui définit un point fixe) est donc illégal :

$\{$
  var f: **proc** (**out** int) := **proc** (**out** x: int)$\{$ $fix(x)$; $\}_x$;
  $fix := f$;
$\}_{fix}$

Loop$^\omega$ – variables mutables globales

- **Pas d'effets de bord :** les variables globales mutables sont simulées par un passage d'état explicite (*state passing style*), sous forme de paramètres supplémentaires passés à la fois en **in** et **out**.

  Pour alléger la syntaxe, on introduit les abréviations suivantes :

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

LOOP$^\omega$ – fonction d'Ackermann

```
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
```

Loop$^\omega$ – sémantique transitionnelle (1)

- un état est une paire (s, μ), où s est une séquence et μ une mémoire ;
- la mémoire μ associe une expression close à chaque variable mutable.

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

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

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

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

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

Loop$^\omega$ – sémantique transitionnelle (2)

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

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

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

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

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

Système T de Gödel (en appel par valeur)

(termes)

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

(valeurs)

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

(contextes)

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

(règles d'évaluation)

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

Traduction de I vers F

- $\bar{n}^\star = S^n(0)$ *(expressions)*
- $y^\star = y$
- $(e_1, ..., e_n)^\star = (e_1^\star, ..., e_n^\star)$
- $(\mathbf{proc}\ (\mathbf{in}\ \vec{y}; \mathbf{out}\ \vec{z})\{s\}_{\vec{z}})^\star = \lambda\vec{y}.(s)^\star_{\vec{z}}\ [()/\vec{z}]$

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

Exemple : la fonction d'Ackermann

```
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
```

Simulation pas-à-pas

Théorème. *Pour tout état* (s, μ)*, ce diagramme commute* (*où* $\vec{x} = dom(\mu)$) *:*

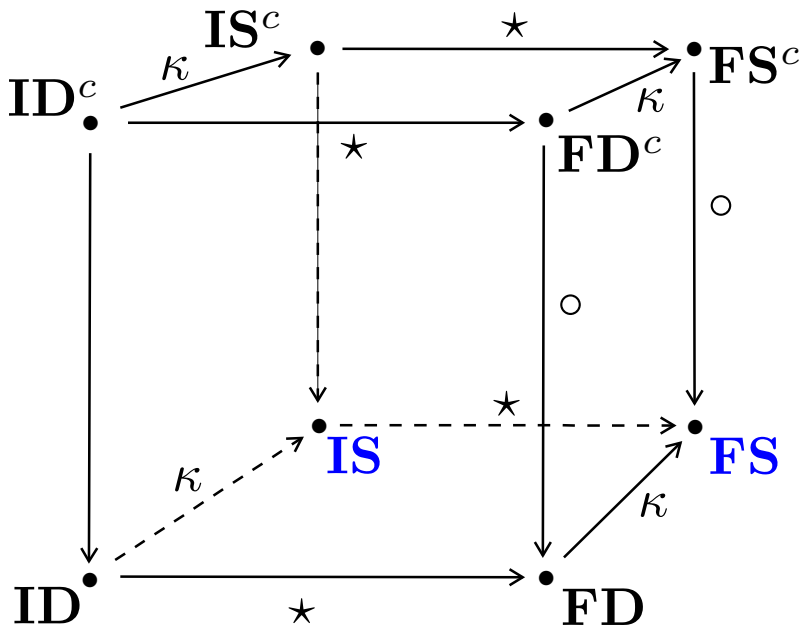
$$\begin{array}{ccc} (s,\mu) & \mapsto & (s',\mu') \\ \downarrow\star & & \downarrow\star \\ (s)^{\star}_{\vec{x}}[\mu(\vec{x})^{\star}/\vec{x}] & \rightsquigarrow & (s')^{\star}_{\vec{x}}[\mu'(\vec{x})^{\star}/\vec{x}] \end{array}$$

Démonstration. [Crolard et al., 2009] □

Remarque.

- La mémoire mutable est simulée simplement par la meta-substitution.
- La stratégie d'évaluation en appel par valeur est implicite dans la syntaxe impérative de Loop$^{\omega}$ (séquence explicite et imbrication des appels de procédure impossible).

Systèmes de types simples



I: imperatif. **F**: fonctionnel.
D: dépendant. **S**: simple.
$\_^c$: classique.

$\star$: traduction de **I** vers **F**
κ : effacement de **D** vers **S**
$\circ$: $\neg\neg$-traduction

Système de types fonctionnel simple FS

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

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

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

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

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

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

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

Système de types impératif simple IS

Exemple :

séquence s	image de s par *
$x := a;$	$\mathbf{let}\ x = a^\star\ \mathbf{in}$
$x := b;$	$\mathbf{let}\ x = b^\star\ \mathbf{in}$
$p(c; x);$	$\mathbf{let}\ x = p^\star\ c^\star\ \mathbf{in}$
$\mathbf{inc}(x);$	$\mathbf{let}\ x = \mathbf{succ}(x)\ \mathbf{in}$
...	...

Le terme fonctionnel est typable si $a^\star$, $b^\star$ le sont et $p^\star\ c^\star$ est de type nat. Pourquoi la séquence impérative s ne serait-elle pas typable ?

Cette remarque suggère un système de types impératif pseudo-dynamique.

Système de type pseudo-dynamique

$$\sigma, \tau \ ::= \ \mathbf{nat} \mid \mathbf{unit} \mid \mathbf{proc}\ (\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau}) \mid (\tau_1, ..., \tau_n)$$

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

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

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

$$\frac{\Gamma; \Omega \vdash e_1 : \tau_1 \quad ... \quad \Gamma; \Omega \vdash e_n : \tau_n}{\Gamma; \Omega \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$$

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

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

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

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

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

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

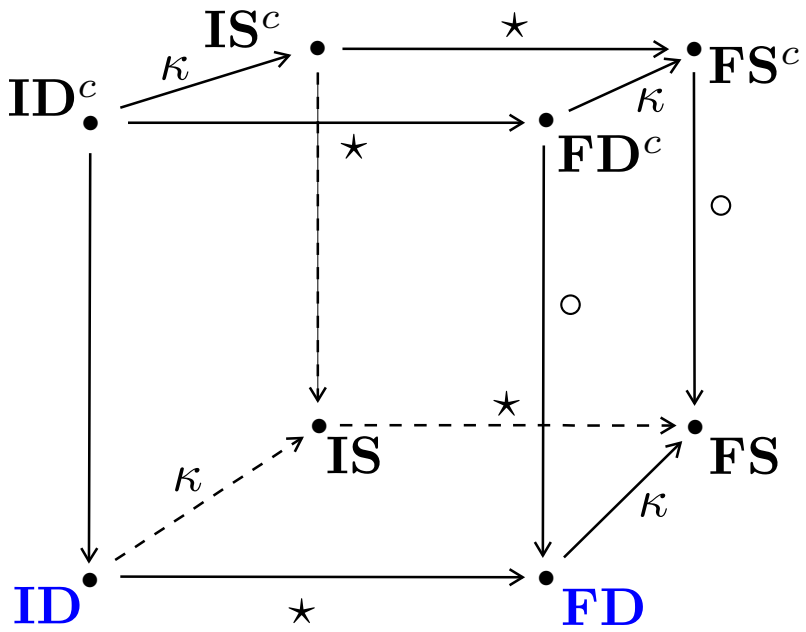
$$\frac{\Gamma;\Omega, y:\sigma \vdash e:\tau}{\Gamma;\Omega, y:\sigma \vdash y := e \rhd y:\tau} \qquad \frac{\Gamma;\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\tau}}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}} \rhd \vec{x}:\vec{\tau}}$$

$$\frac{}{\Gamma;\Omega, y:\mathbf{nat} \vdash \mathbf{inc}(y) \rhd y:\mathbf{nat}} \qquad \frac{}{\Gamma;\Omega, y:\mathbf{nat} \vdash \mathbf{dec}(y) \rhd y:\mathbf{nat}}$$

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

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

Systèmes de types dépendants



I: imperatif. **F**: fonctionnel.
D: dépendant. **S**: simple.
$\_^c$: classique.

$\star$: traduction de **I** vers **F**
κ : effacement de **D** vers **S**
$\circ$: $\neg\neg$-traduction

Système de types dépendants fonctionnel (1)

Système de types dépendants fonctionnel (*à la* Leivant-Krivine) paramétré par un système équationnel $\mathcal{E}$ (contenant les définitions de $\mathbf{p}, +, \times$).

$$
\begin{array}{rcll}
\tau & ::= & \mathbf{nat}(n) & (\textit{cas particuliers}) \\
 & | & (n = m) & \\
 & | & \forall \vec{\imath}\,(\tau_1 \Rightarrow \tau_2) & \forall i(\mathbf{nat}(i) \Rightarrow \tau) \\
 & | & \exists \vec{\imath}\,(\tau_1 \wedge \ldots \wedge \tau_n) & \exists i(\mathbf{nat}(i) \wedge \tau)
\end{array}
$$

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

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

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

Système de types dépendants fonctionnel (2)

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

$$\frac{\Gamma \vdash t_1 : \tau_1[\vec{n}/\vec{i}] \quad \ldots \quad \Gamma \vdash t_k : \tau_k[\vec{n}/\vec{i}]}{\Gamma \vdash (t_1, \ldots, t_k) : \exists \vec{i}\,(\tau_1 \wedge \ldots \wedge \tau_k)}$$

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

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

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

La fonction d'Ackermann

À partir des équations :

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

on peut dériver la totalité de **a** dans **FD**:

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

Propriétés importantes de FD

Tirées de [Leivant, 1990] :

Préservation du typage : *Si* $\Gamma \vdash t : \sigma$ *dans* **FD** *et* $t \rightsquigarrow t'$ *alors* $\Gamma \vdash t' : \sigma$.

Théorème de représentation : *Etant donné un système équationnel* $\mathcal{E}$ *et un symbole de fonction n-aire* f, *si* $\vdash_{\mathcal{E}} t$: $\forall \vec{n}.\mathbf{nat}(\vec{n}) \Rightarrow \mathbf{nat}(f(\vec{n}))$ *est dérivable dans* **FD** *alors* t *représente* f (c'est-à-dire: $t\ \bar{q} \rightsquigarrow^{\star} \overline{f(q)}$ pour tout entier q).

Système de types dépendants impératif (1)

$$\sigma, \tau \ ::= \ \mathbf{nat}(n) \mid n = m \mid \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \mid \exists \vec{\jmath}\,(\tau_1, \ldots, \tau_n)$$

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

$$\frac{\Gamma; \Omega \vdash e : \tau[n/i] \qquad \Gamma; \Omega \vdash e' : n = m}{\Gamma; \Omega \vdash e : \tau[m/i]} \qquad \frac{\vdash_{\mathcal{E}} n = m}{\Gamma; \Omega \vdash * : n = m}$$

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

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

$$\frac{\Gamma; \Omega \vdash e_1 : \tau_1[\vec{n}/\vec{\jmath}] \quad \ldots \quad \Gamma; \Omega \vdash e_n : \tau_n[\vec{n}/\vec{\jmath}]}{\Gamma; \Omega \vdash (e_1, \ldots, e_n) : \exists \vec{\jmath}\,(\tau_1, \ldots, \tau_n)}$$

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

Système de types dépendants impératif (1)

$$\sigma, \tau \ ::= \ \mathbf{nat}(n) \mid n = m \mid \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\tau}; \mathbf{out}\ \vec{\sigma}) \mid \exists \vec{\jmath}\,(\tau_1, \ldots, \tau_n)$$

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

$$\frac{\Gamma; \Omega \vdash e : \tau[n/i] \qquad \Gamma; \Omega \vdash e' : n = m}{\Gamma; \Omega \vdash e : \tau[m/i]} \qquad \frac{\vdash_{\mathcal{E}} n = m}{\Gamma; \Omega \vdash * : n = m}$$

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

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

$$\frac{\Gamma; \Omega \vdash e_1 : \tau_1[\vec{n}/\vec{\jmath}] \quad \ldots \quad \Gamma; \Omega \vdash e_n : \tau_n[\vec{n}/\vec{\jmath}]}{\Gamma; \Omega \vdash (e_1, \ldots, e_n) : \exists \vec{\jmath}\,(\tau_1, \ldots, \tau_n)}$$

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

Système de types dépendants impératif (2)

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

$$\frac{\Gamma; \Omega \vdash s \rhd \Omega'[n/i] \qquad \Gamma; \Omega \vdash e : n = m}{\Gamma; \Omega \vdash s \rhd \Omega'[m/i]}$$

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

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

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

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

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash e \rhd \exists \vec{\jmath}\,.\, \vec{x} : \vec{\tau} \qquad \Gamma; \Omega, \vec{x} : \vec{\tau} \vdash s \rhd \Omega'}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \vec{x} := e;\ \ s\ \rhd \Omega'} \qquad \vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega, \Omega')$$

Système de types dépendants impératif (2)

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

$$\frac{\Gamma; \Omega \vdash s \rhd \Omega'[n/i] \qquad \Gamma; \Omega \vdash e : n = m}{\Gamma; \Omega \vdash s \rhd \Omega'[m/i]}$$

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

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

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

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

$$\frac{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash e \rhd \exists \vec{\jmath}\,.\, \vec{x} : \vec{\tau} \qquad \Gamma; \Omega, \vec{x} : \vec{\tau} \vdash s \rhd \Omega'}{\Gamma; \Omega, \vec{x} : \vec{\sigma} \vdash \vec{x} := e;\ \ s\ \rhd \Omega'} \qquad \vec{\jmath} \notin \mathcal{FV}(\Gamma, \Omega, \Omega')$$

Système de types dépendants impératif (3)

$\boxed{\Gamma; \Omega \vdash c \rhd \Omega'}$ *(commandes)*

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

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

$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y) \rhd y\colon \mathbf{nat}(\mathbf{s}(n))}$$

$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y) \rhd y\colon \mathbf{nat}(\mathbf{p}(n))}$$

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

$$\frac{\Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash p\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau}) \quad \Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash \vec{e}\colon \vec{\sigma}[\vec{n}/\vec{\imath}]}{\Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash p(\vec{e}; \vec{r}) \rhd \vec{r}\colon \vec{\tau}[\vec{n}/\vec{\imath}]}$$

Système de types dépendants impératif (3)

$\boxed{\Gamma; \Omega \vdash c \rhd \Omega'}$ *(commandes)*

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

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

$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y) \rhd y\colon \mathbf{nat}(\mathbf{s}(n))}$$

$$\frac{}{\Gamma; \Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y) \rhd y\colon \mathbf{nat}(\mathbf{p}(n))}$$

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

$$\frac{\Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash p\colon \mathbf{proc}\ \forall \vec{\imath}\,(\mathbf{in}\ \vec{\sigma}; \mathbf{out}\ \vec{\tau}) \quad \Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash \vec{e}\colon \vec{\sigma}[\vec{n}/\vec{\imath}]}{\Gamma; \Omega, \vec{r}\colon \vec{\omega} \vdash p(\vec{e}; \vec{r}) \rhd \vec{r}\colon \vec{\tau}[\vec{n}/\vec{\imath}]}$$

Typage de la procédure *Ack* dans ID

$$
\begin{array}{l|l}
\textbf{cst}\ Ack = \textbf{proc}\ (\textbf{in}\ M, N; \textbf{out}\ Z)\ \{ & -(M\colon \textbf{nat}(m), N\colon \textbf{nat}(n))[Z\colon \top] \\
 & \quad | \\
\quad \textbf{var}\ G := \textbf{proc}\ (\textbf{in}\ Y; \textbf{out}\ P)\ \{ & \quad | \quad -(Y\colon \textbf{nat}(y))[P\colon \top] \\
\quad\quad P := Y; & \quad | \quad\ | \quad [P\colon \textbf{nat}(y)] \\
\quad\quad \textbf{inc}(P); & \quad | \quad\ | \quad [P\colon \textbf{nat}(\textbf{s}(y))] \\
\quad \}_P; & \quad | \quad [G\colon \textbf{proc}\ \forall y(\textbf{in nat}(y); \textbf{out nat}(\textbf{a}(0, y)))] \qquad \text{by } (a1) \\
 & \quad | \\
\quad \textbf{for}\ I := 0\ \textbf{until}\ M\ \{ & \quad | \quad -(I\colon \textbf{nat}(i))[G\colon \textbf{proc}\ \forall y(\textbf{in nat}(y); \textbf{out nat}(\textbf{a}(i, y)))] \\
\quad\quad \textbf{cst}\ H = G; & \quad | \quad\ | \quad (H\colon \textbf{proc}\ \forall y(\textbf{in nat}(y); \textbf{out nat}(\textbf{a}(i, y)))) \\
 & \quad | \quad | \\
\quad\quad G := \textbf{proc}\ (\textbf{in}\ Y; \textbf{out}\ P)\ \{ & \quad | \quad\ | \quad -(Y\colon \textbf{nat}(y))[P\colon \top] \\
\quad\quad\quad P := 2; & \quad | \quad\ | \quad\ | \quad [P\colon \textbf{nat}(\textbf{a}(\textbf{s}(i), 0))] \qquad \text{by } (a2) \\
\quad\quad\quad \textbf{for}\ J := 0\ \textbf{until}\ Y\ \{ & \quad | \quad\ | \quad\ | \quad -(J\colon \textbf{nat}(j))[P\colon \textbf{nat}(\textbf{a}(\textbf{s}(i), j))] \\
\quad\quad\quad\quad H(P; P); & \quad | \quad\ | \quad\ | \quad\ | \quad [P\colon \textbf{nat}(\textbf{a}(\textbf{s}(i), \textbf{s}(j)))] \qquad \text{by } (a3) \\
\quad\quad\quad \}_P; & \quad | \quad\ | \quad\ | \quad [P\colon \textbf{nat}(\textbf{a}(\textbf{s}(i), y)))] \\
\quad\quad \}_P; & \quad | \quad\ | \quad [G\colon \textbf{proc}\ \forall y(\textbf{in nat}(y); \textbf{out nat}(\textbf{a}(\textbf{s}(i), y))))] \\
 & \quad | \quad | \\
\quad \}_G; & \quad | \quad [G\colon \textbf{proc}\ \forall y(\textbf{in nat}(y); \textbf{out nat}(\textbf{a}(m, y)))] \\
 & \quad | \\
\quad G(N; Z); & \quad | \quad [Z\colon \textbf{a}(m, n)] \\
\}_Z & (Ack\colon \textbf{proc}\ \forall m, n(\textbf{in nat}(m), \textbf{nat}(n); \textbf{out nat}(\textbf{a}(m, n))))
\end{array}
$$

Traduction de ID vers FD

Traduction des types dépendants

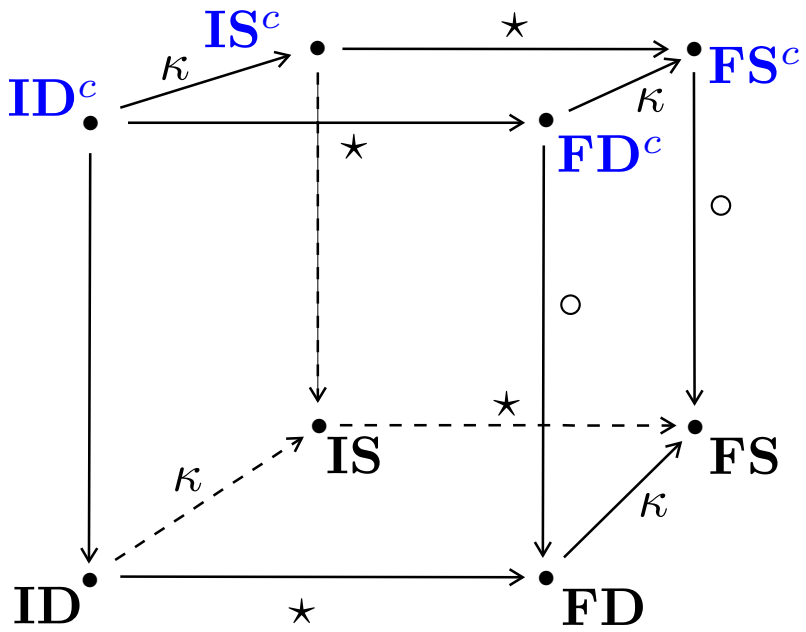
- $(t = u)^\star = (t = u)$
- $(\mathbf{nat}(u))^\star = \mathbf{nat}(u)$
- $(\mathbf{proc}\ \forall\vec{\imath}\,(\mathbf{in}\ \vec{\tau}\,;\mathbf{out}\ \vec{\sigma}\,))^\star = \forall\vec{\imath}\,(\vec{\tau}^\star \Rightarrow \vec{\sigma}^\star)$
- $(\exists\vec{\jmath}\,(\tau_1, ..., \tau_n))^\star = \exists\vec{\jmath}\,(\tau_1^\star \wedge ... \wedge \tau_n^\star)$

Théorème. (*la traduction* * *préserve le typage*). *Pour tous environnements* Γ *et* Ω, *toute expression* e, *toute séquence* s, *on a :*

- $\Gamma; \Omega \vdash e : \tau$ *dans* **ID** *implique* $\Gamma^\star, \Omega^\star \vdash e^\star : \tau^\star$ *in* **FD**.
- $\Gamma; \Omega \vdash s \rhd \vec{x} : \vec{\sigma}$ *dans* **ID** *implique* $\Gamma^\star, \Omega^\star \vdash (s)^\star_{\vec{x}} : \vec{\sigma}^\star$ *dans* **FD**.

Corollaire. (*théorème de représentation pour* **ID**). *Étant donné un système équationnel* $\mathcal{E}$ *et un symbole de fonction* n*-aire* f, *si on peut dériver dans* **ID** $\vdash p : \mathbf{proc}\,\forall\vec{n}\,(\mathbf{in}\ \mathbf{nat}(\vec{n})\,;\mathbf{out}\ \mathbf{nat}(f(\vec{n})))$ *alors* p *représente* f.

Mécanismes de contrôle



I: impératif. **F**: fonctionnel.
D: dépendant. **S**: simple.
$\_^c$: classique.

$\star$: traduction de **I** vers **F**
κ : effacement de **D** vers **S**
$\circ$: $\neg\neg$-traduction

Le système $\mathbf{FD^c}$

On se donne une constante propositionnelle « absurde » notée $\bot$, et on définit la négation $\neg\varphi$ comme un abréviation pour $\varphi \Rightarrow \bot$.

On étend ensuite **FD** avec deux constantes **callcc** et **throw** typées ainsi :

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

La sémantique de **callcc** et **throw** est donnée par une traduction CPS (et une $\neg\neg$-traduction au niveau des types) dans [Crolard and Polonowski, 2010].

Remarque. Il est aussi possible d'en donner une sémantique contextuelle, ou sous forme de machine abstraite.

Sauts non-locaux

On étend **I** avec *void*, $\neg$ et deux constantes procédurales :

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

On peut alors définir des macros impératives qui se traduisent par * ainsi :

$$\begin{array}{rcl} k\colon\{s\}^\star_{\vec{z}} & = & \textbf{callcc}\ \lambda k.\{s\}^\star_{\vec{z}} \\ \textbf{jump}(k,\vec{e}\,)^\star & = & \textbf{throw}\ (k,\vec{e}^{\,\star}) \end{array}$$

Remarque.

- **jump** saute à la fin du bloc ayant le label k (permet de simuler **exit**)
- mais les labels sont des citoyens de première classe et **jump** est donc plus général qu'une instruction **exit**.

Continuations délimitées

Les types fonctionnels de [Danvy and Filinski, 1989] vus comme des formules classiques (où $\alpha/\delta \to \gamma/\beta$ est interprété par $\alpha \wedge \neg\beta \Rightarrow \gamma \wedge \neg\delta$) :

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

Ces types proviennent de l'encodage de *shift*/*reset* [Filinski, 1994] à partir de **callcc**/**throw** et d'une variable globale mutable (pour la meta-continuation).

Les versions impératives de *shift*/*reset* ont les types suivants :

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

Encodage impératif de shift/reset

$$
\begin{array}{l}
\textbf{cst } \mathit{reset} \;=\; \textbf{proc}(\textbf{in } p; \textbf{out } r)_{mk}\{ \\
\qquad k\colon\{ \\
\qquad\qquad \textbf{cst } m \;=\; mk; \\
\qquad\qquad mk \;:=\; \textbf{proc}(\textbf{in } r; \textbf{out } z)\{\textbf{jump } (k, r, m)_z;\}_z; \\
\qquad\qquad \textbf{var } y;\; p(; y)_{mk};\; \textbf{jump } (mk, y)_{r,mk}; \\
\qquad \}_{r,mk}; \\
\}_{r,mk};
\end{array}
$$

$$
\begin{array}{l}
\textbf{cst } \mathit{shift} \;=\; \textbf{proc}(\textbf{in } p; \textbf{out } r)_{mk}\{ \\
\qquad k\colon\{ \\
\qquad\qquad \textbf{cst } q \;=\; \textbf{proc } q(\textbf{in } v; \textbf{out } r)_{mk}\{ \\
\qquad\qquad\qquad \mathit{reset}\ (\textbf{proc}(\textbf{out } z)_{mk}\{\textbf{jump } (k, v, mk)_{z,mk};\}_{z,mk}; r)_{mk}; \\
\qquad\qquad \}_{r,mk}; \\
\qquad\qquad \textbf{var } y;\; p(q; y)_{mk};\; \textbf{jump } (mk, y)_{r,mk}; \\
\qquad \}_{r,mk}; \\
\}_{r,mk}
\end{array}
$$

Exemple

Problème :

- Comment prouver la correction de l'exemple suivant [Wadler, 1994] ?

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

Informellement :

- "*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.*"

Formellement :

- Traduire l'expression en un programme impératif.
- Dériver la spécification attendue dans $\mathbf{ID}^c$.

Logique de Floyd-Hoare

Plonger une logique de Floyd-Hoare dans **ID** se fait directement, le jugement :

$$\Gamma; \Omega \vdash \{\varphi\} s \{\psi\}$$

devient (où *assert* est une variable globale fixée) :

$$\Gamma; \Omega, \mathit{assert}\colon \varphi \vdash s \rhd \Omega, \mathit{assert}\colon \psi$$

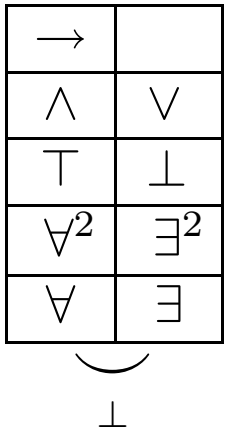
La règle de conséquence :

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

est dérivable si $\varphi' \Rightarrow \varphi$ et $\psi \Rightarrow \psi'$ sont négatives (sans contenu calculatoire).

Le cas classique ($\mathbf{ID}^c$) est plus délicat : il faut utiliser une modalité pour isoler le raisonnement classique ayant un contenu calculatoire [Thielecke, 2008].

Correspondance entre formules et types

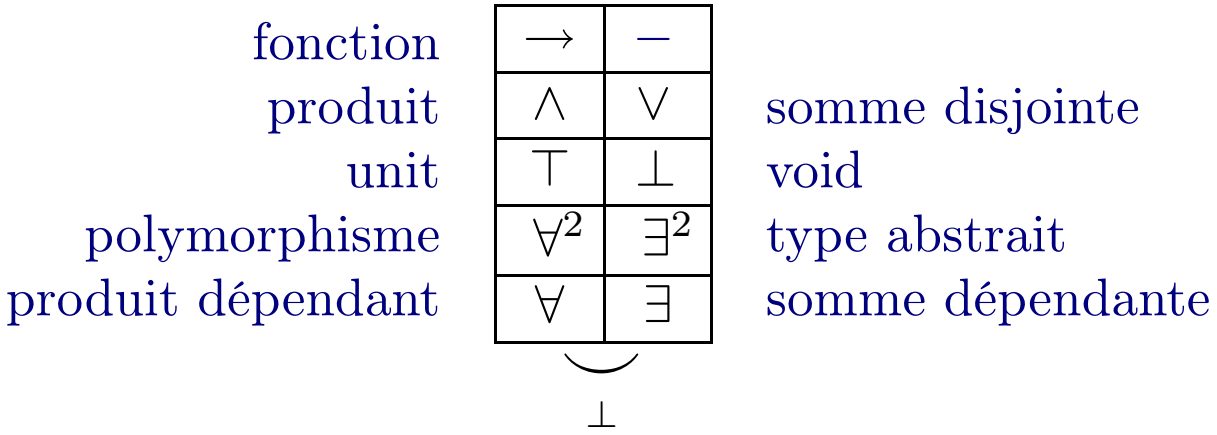


Où $\_^{\bot}$ représente :

- la dualité et la négation en logique classique

- la dualité en logique intuitionniste
  (symmétrie syntaxique et dualité sémantique)

Correspondance entre formules et types

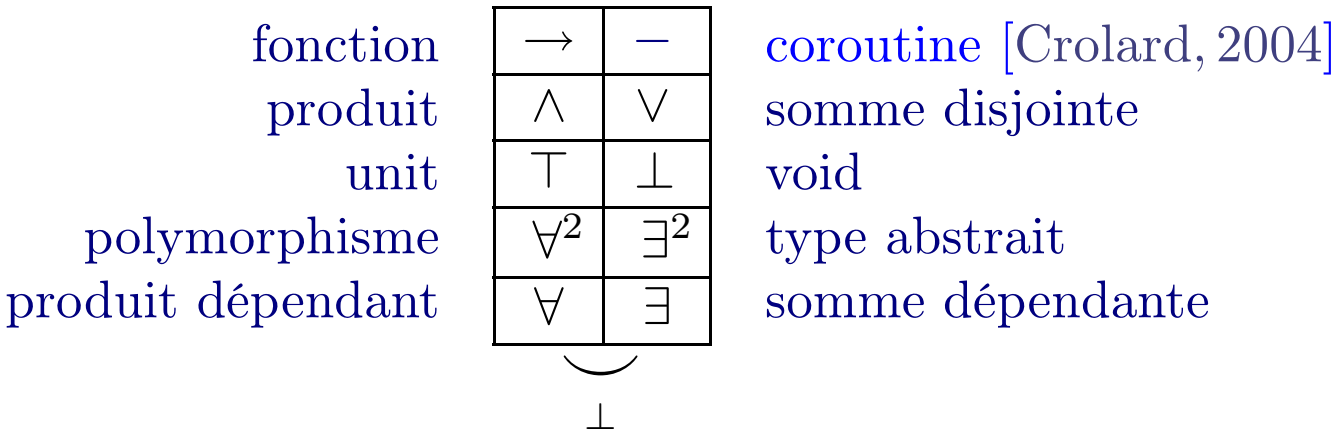


Où $\_^{\perp}$ représente :

- la dualité et la négation en logique classique
  la négation permet de typer les continuations de première classe.

- la dualité en logique intuitionniste
  (symmétrie syntaxique et dualité sémantique)

Correspondance entre formules et types



Où $\_^{\perp}$ représente :

- la dualité et la négation en logique classique
  la négation permet de typer les continuations de première classe.
  (la soustraction est alors définissable par $A - B \equiv A \wedge \neg B$)
- la dualité en logique intuitionniste
  (symmétrie syntaxique et dualité sémantique)
  la soustraction permet de typer des coroutines de première classe.

Conclusion et perspectives

- Définir une sémantique transitionnelle directe de LOOP$^{\omega}$+*labels/jumps*
  - sous forme d'une machine à environnement ;
  - considérer les *coroutines impératives*, les générateurs...
- Considérer les extensions suivantes :
  - structures de données : listes/tableaux, types inductifs (arbres)
  - arithmétique du second ordre (modules et généricité)
- Étudier le plongement de la logique de Hoare (et la *complétude*).
- Les langages, les systèmes de types et la traduction ont été formalisés sous forme de spécifications exécutables dans Isabelle/HOL et Twelf. *Projet* : vérifier la meta-théorie (certainement avec Coq).
- Vérifier des exemples plus conséquents (comme *same-fringe*).

Conclusion et perspectives

- Définir une sémantique transitionnelle directe de LOOP$^{\omega}$+*labels/jumps*
  - sous forme d'une machine à environnement ;
  - considérer les *coroutines impératives*, les générateurs...
- Considérer les extensions suivantes :
  - structures de données : listes/tableaux, types inductifs (arbres)
  - arithmétique du second ordre (modules et généricité)
- Étudier le plongement de la logique de Hoare (et la *complétude*).
- Les langages, les systèmes de types et la traduction ont été formalisés sous forme de spécifications exécutables dans Isabelle/HOL et Twelf. *Projet* : vérifier la meta-théorie (certainement avec Coq).
- Vérifier des exemples plus conséquents (comme *same-fringe*).

Conclusion et perspectives

- Définir une sémantique transitionnelle directe de LOOP$^{\omega}$+*labels/jumps*
  - sous forme d'une machine à environnement ;
  - considérer les *coroutines impératives*, les générateurs...
- Considérer les extensions suivantes :
  - structures de données : listes/tableaux, types inductifs (arbres)
  - arithmétique du second ordre (modules et généricité)
- Étudier le plongement de la logique de Hoare (et la *complétude*).
- Les langages, les systèmes de types et la traduction ont été formalisés sous forme de spécifications exécutables dans Isabelle/HOL et Twelf. *Projet* : vérifier la meta-théorie (certainement avec Coq).
- Vérifier des exemples plus conséquents (comme *same-fringe*).

Conclusion et perspectives

- Définir une sémantique transitionnelle directe de LOOP$^{\omega}$+*labels/jumps*
  - sous forme d'une machine à environnement ;
  - considérer les *coroutines impératives*, les générateurs...
- Considérer les extensions suivantes :
  - structures de données : listes/tableaux, types inductifs (arbres)
  - arithmétique du second ordre (modules et généricité)
- Étudier le plongement de la logique de Hoare (et la *complétude*).
- Les langages, les systèmes de types et la traduction ont été formalisés sous forme de spécifications exécutables dans Isabelle/HOL et Twelf. *Projet* : vérifier la meta-théorie (certainement avec Coq).
- Vérifier des exemples plus conséquents (comme *same-fringe*).

Conclusion et perspectives

- Définir une sémantique transitionnelle directe de LOOP$^{\omega}$+*labels/jumps*
  - sous forme d'une machine à environnement ;
  - considérer les *coroutines impératives*, les générateurs...
- Considérer les extensions suivantes :
  - structures de données : listes/tableaux, types inductifs (arbres)
  - arithmétique du second ordre (modules et généricité)
- Étudier le plongement de la logique de Hoare (et la *complétude*).
- Les langages, les systèmes de types et la traduction ont été formalisés sous forme de spécifications exécutables dans Isabelle/HOL et Twelf. *Projet* : vérifier la meta-théorie (certainement avec Coq).
- Vérifier des exemples plus conséquents (comme *same-fringe*).

La logique soustractive

Sémantique étudiée dans [Rauszer, 1974] :

- conservative sur la logique intuitionniste dans le cadre propositionnel ;
- conservative sur la logique à domaine constant (CDL) au premier ordre (modèle de Kripke à domaine constant [Grzegorczyk, 1964]).
- CDL [Görnemann, 1971] correspond à la logique intuitionniste étendue par le schéma DIS :

$$\forall x(A(x) \vee B) \vdash \forall x.A(x) \vee B \qquad \text{où } x \notin \mathcal{FV}(B)$$

- L'arithmétique soustractive est conservative sur HA (formulé comme **FD** et en se limitant aux formules relativisées).
- Il n'existe pas de calcul des séquents *conventionnel* admettant l'élimination des coupures pour CDL [Lopez-Escobar, 1983].

La logique soustractive

Sémantique étudiée dans [Rauszer, 1974] :

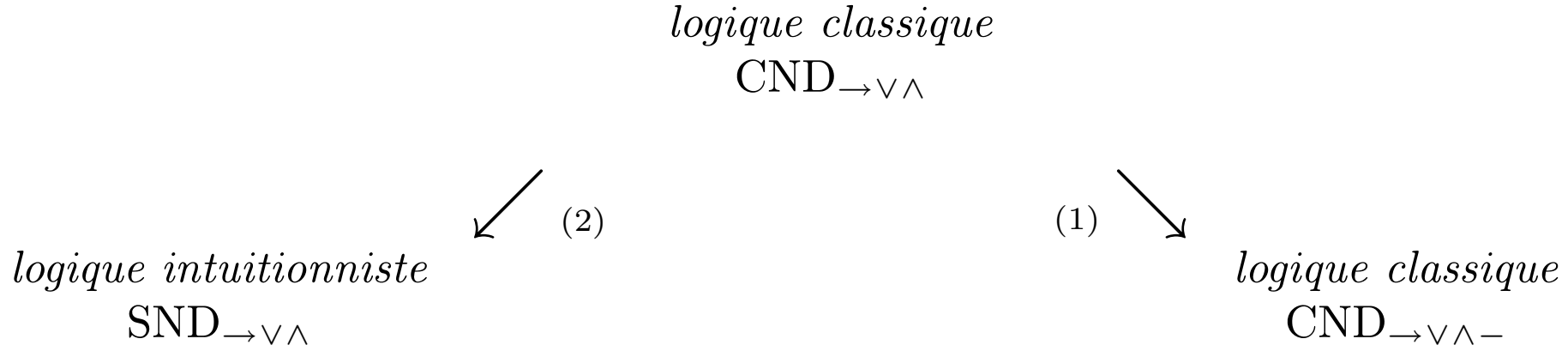
- conservative sur la logique intuitionniste dans le cadre propositionnel ;
- conservative sur la logique à domaine constant (CDL) au premier ordre (modèle de Kripke à domaine constant [Grzegorczyk, 1964]).
- CDL [Görnemann, 1971] correspond à la logique intuitionniste étendue par le schéma DIS :

$$\forall x(A(x) \vee B) \vdash \forall x.A(x) \vee B \qquad \text{où } x \notin \mathcal{FV}(B)$$

- L'arithmétique soustractive est conservative sur HA (formulé comme **FD** et en se limitant aux formules relativisées).
- Il n'existe pas de calcul des séquents *conventionnel* admettant l'élimination des coupures pour CDL [Lopez-Escobar, 1983].

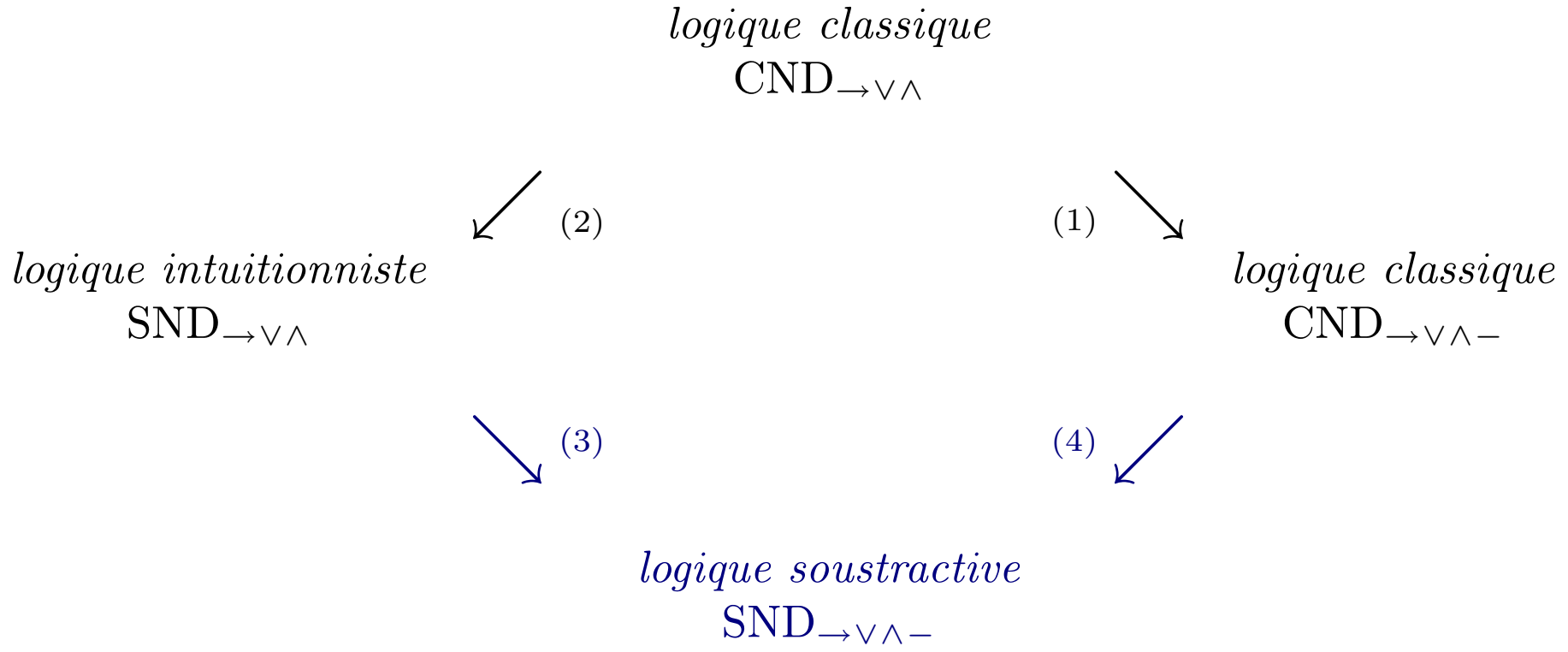
$\Longrightarrow$ Notion de séquent à multiples conclusions et liens de dépendance. Théorème d'élimination des coupures et interprétation calculatoire en terme de coroutines dans [Crolard, 2004].

Vue d'ensemble



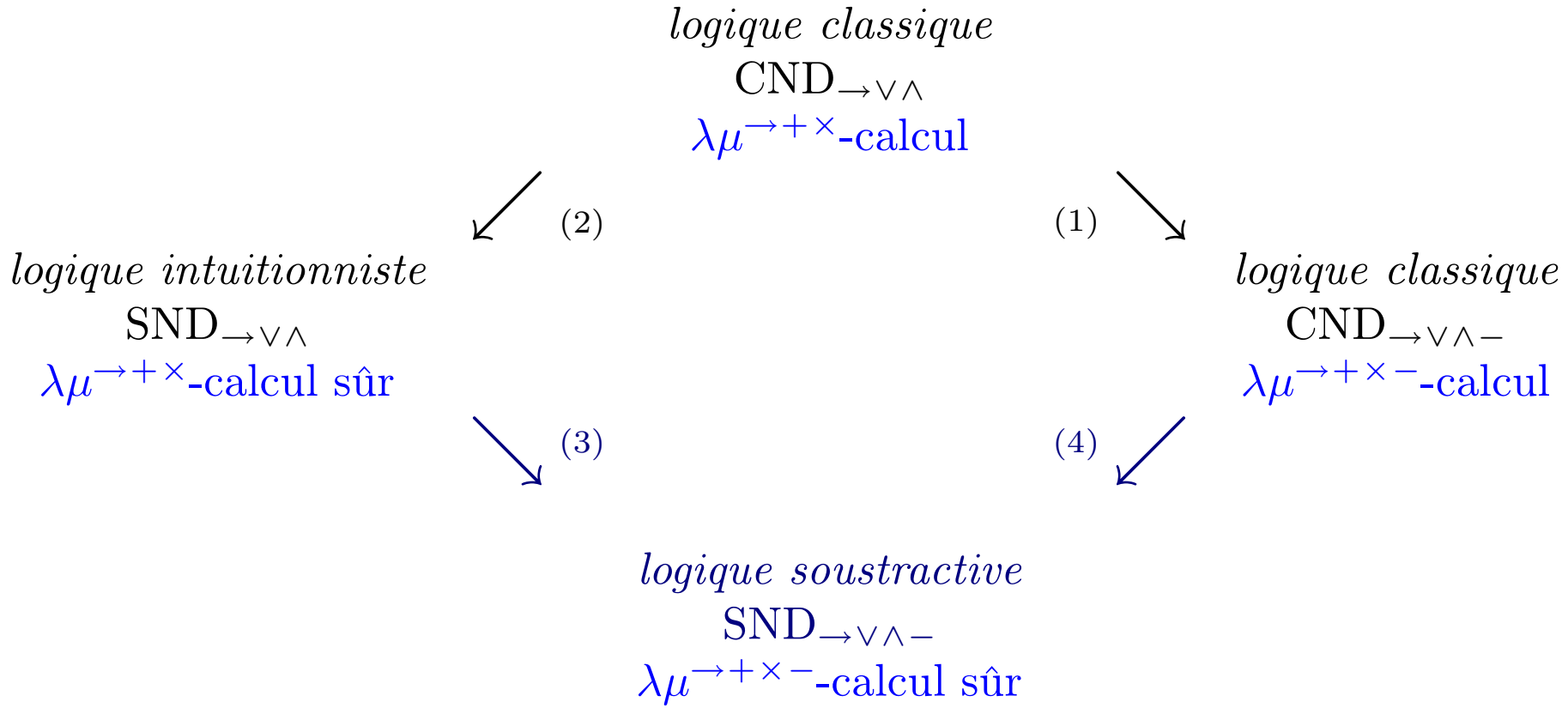
(1) Définition de la soustraction en logique classique ($A - B \equiv A \wedge \neg B$).
(2) Restriction de $\mathrm{CND}_{\to\vee\wedge}$ à la logique intuitionniste (resp. CDL).

Vue d'ensemble



(1) Définition de la soustraction en logique classique ($A - B \equiv A \wedge \neg B$).
(2) Restriction de $\mathrm{CND}_{\to\vee\wedge}$ à la logique intuitionniste (resp. CDL).
(3) Extension de $\mathrm{SND}_{\to\vee\wedge}$ avec des règles contraintes pour la soustraction.
(4) Restriction duale de $\mathrm{CND}_{\to\vee\wedge-}$ à la logique soustractive.

Vue d'ensemble



(1) Définition de la soustraction en logique classique ($A - B \equiv A \wedge \neg B$).
(2) Restriction de $\mathrm{CND}_{\to\vee\wedge}$ à la logique intuitionniste (resp. CDL).
(3) Extension de $\mathrm{SND}_{\to\vee\wedge}$ avec des règles contraintes pour la soustraction.
(4) Restriction duale de $\mathrm{CND}_{\to\vee\wedge-}$ à la logique soustractive.

Interprétation calculatoire

Dans le $\lambda\mu$-calcul :

- les labels sont des citoyens de seconde classe, mais les continuations de première classe sont définissables :

$$\textbf{callcc}\ t = \textbf{catch}\ \alpha\ (t\ \lambda x.\textbf{throw}\ \alpha\ x)$$

Dans le $\lambda\mu$-calcul sûr :

- **callcc** n'est pas définissable : des règles de visibilité imposent qu'à chaque fil d'exécution corresponde un environnement propre (exactement comme dans les langages à pile, où la pile contient l'environnement).
- une coroutine de première classe de type $A - B$ contient alors (en plus de la continuation qui accepte du B) un état local de type A qui sert à construire l'environnement lors de l'invocation de la coroutine.

Machine de Krivine pour le λ-calcul

Syntaxe :

$$
\begin{array}{lrcl}
\textbf{(term)} & t & ::= & x \mid \lambda x.t \mid (t_1\ t_2) \\
\textbf{(env)} & \mathcal{E} & \in & (\textbf{var} \to \textbf{clos}) \\
\textbf{(clos)} & c & ::= & \langle t, \mathcal{E} \rangle \\
\textbf{(stack)} & \mathcal{S} & ::= & () \mid c::\mathcal{S} \\
\textbf{(state)} & \sigma & ::= & [t, \mathcal{E}, \mathcal{S}]
\end{array}
$$

Relation de transition $\to$ entre états :

$$
\begin{array}{lcll}
[x, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}', \mathcal{S}] & \text{où } \langle t, \mathcal{E}' \rangle = \mathcal{E}_\lambda(x) \\
[\lambda x.t, \mathcal{E}, c::\mathcal{S}] & \to & [t, (\{\mathcal{E}_\lambda, x = c\}, \mathcal{E}_\mu), \mathcal{S}] & \\
[(t\ u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, \langle u, \mathcal{E} \rangle::\mathcal{S}] &
\end{array}
$$

Machine de De Groote pour le $\lambda\mu$-calcul

Syntaxe :

$$
\begin{array}{lrcl}
\textbf{(term)} & t & ::= & x \mid \lambda x.t \mid (t_1\ t_2) \mid \textbf{catch}\ \alpha\ t \mid \textbf{throw}\ \alpha\ t \\
\textbf{(env)} & \mathcal{E} & \in & (\textbf{var} \to \textbf{clos}) \times (\mu\textbf{-var} \to \textbf{stack}) \\
\textbf{(clos)} & c & ::= & \langle t, \mathcal{E} \rangle \\
\textbf{(stack)} & \mathcal{S} & ::= & () \mid c :: \mathcal{S} \\
\textbf{(state)} & \sigma & ::= & [t, \mathcal{E}, \mathcal{S}]
\end{array}
$$

Relation de transition $\to$ entre états :

$$
\begin{array}{lcll}
[x, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}', \mathcal{S}] & \text{où } \langle t, \mathcal{E}' \rangle = \mathcal{E}_\lambda(x) \\
[\lambda x.t, \mathcal{E}, c :: \mathcal{S}] & \to & [t, (\{\mathcal{E}_\lambda, x = c\}, \mathcal{E}_\mu), \mathcal{S}] & \\
[(t\ u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, \langle u, \mathcal{E} \rangle :: \mathcal{S}] & \\
[\textbf{catch}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}_\lambda, \mathcal{E}'_\mu), \mathcal{S}] & \text{où } \mathcal{E}'_\mu = \{\mathcal{E}_\mu, \alpha = \mathcal{S}\} \\
[\textbf{throw}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}_\lambda, \mathcal{E}_\mu), \mathcal{S}'] & \text{où } \mathcal{S}' = \mathcal{E}_\mu(\alpha)
\end{array}
$$

Machine de De Groote pour le $\lambda\mu$-calcul sûr

Syntaxe :

$$
\begin{array}{lrcl}
\textbf{(term)} & t & ::= & x \mid \lambda x.t \mid (t_1\ t_2) \mid \textbf{get-context}\ \alpha\ t \mid \textbf{set-context}\ \alpha\ t \\
\textbf{(env)} & \mathcal{E} & \in & (\textbf{var} \to \textbf{clos}) \times (\mu\textbf{-var} \to ((\textbf{var} \to \textbf{clos}) \times \textbf{stack})) \\
\textbf{(clos)} & c & ::= & \langle t, \mathcal{E} \rangle \\
\textbf{(stack)} & \mathcal{S} & ::= & () \mid c :: \mathcal{S} \\
\textbf{(state)} & \sigma & ::= & [t, \mathcal{E}, \mathcal{S}]
\end{array}
$$

Relation de transition $\to$ entre états :

$$
\begin{array}{lcll}
[x, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}', \mathcal{S}] & \text{où } \langle t, \mathcal{E}' \rangle = \mathcal{E}_\lambda(x) \\
[\lambda x.t, \mathcal{E}, c :: \mathcal{S}] & \to & [t, (\{\mathcal{E}_\lambda, x = c\}, \mathcal{E}_\mu), \mathcal{S}] & \\
[(t\ u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, \langle u, \mathcal{E} \rangle :: \mathcal{S}] & \\
[\textbf{get-context}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}_\lambda, \mathcal{E}'_\mu), \mathcal{S}] & \text{où } \mathcal{E}'_\mu = \{\mathcal{E}_\mu, \alpha = (\mathcal{E}_\lambda, \mathcal{S})\} \\
[\textbf{set-context}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}'_\lambda, \mathcal{E}_\mu), \mathcal{S}'] & \text{où } (\mathcal{E}'_\lambda, \mathcal{S}') = \mathcal{E}_\mu(\alpha)
\end{array}
$$

Machine de De Groote pour le $\lambda\mu$-calcul sûr

Syntaxe :

$$
\begin{array}{lrcl}
\textbf{(term)} & t & ::= & x \mid \lambda x.t \mid (t_1\ t_2) \mid \textbf{get-context}\ \alpha\ t \mid \textbf{set-context}\ \alpha\ t \\
\textbf{(env)} & \mathcal{E} & \in & (\textbf{var} \to \textbf{clos}) \times (\mu\textbf{-var} \to ((\textbf{var} \to \textbf{clos}) \times \textbf{stack})) \\
\textbf{(clos)} & c & ::= & \langle t, \mathcal{E} \rangle \\
\textbf{(stack)} & \mathcal{S} & ::= & () \mid c :: \mathcal{S} \\
\textbf{(state)} & \sigma & ::= & [t, \mathcal{E}, \mathcal{S}]
\end{array}
$$

Relation de transition $\to$ entre états :

$$
\begin{array}{lcll}
[x, \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}', \mathcal{S}] & \text{où } \langle t, \mathcal{E}' \rangle = \mathcal{E}_\lambda(x) \\
[\lambda x.t, \mathcal{E}, c :: \mathcal{S}] & \to & [t, (\{\mathcal{E}_\lambda, x = c\}, \mathcal{E}_\mu), \mathcal{S}] & \\
[(t\ u), \mathcal{E}, \mathcal{S}] & \to & [t, \mathcal{E}, \langle u, \mathcal{E} \rangle :: \mathcal{S}] & \\
[\textbf{get-context}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}_\lambda, \mathcal{E}'_\mu), \mathcal{S}] & \text{où } \mathcal{E}'_\mu = \{\mathcal{E}_\mu, \alpha = (\mathcal{E}_\lambda, \mathcal{S})\} \\
[\textbf{set-context}\ \alpha\ t, \mathcal{E}, \mathcal{S}] & \to & [t, (\mathcal{E}'_\lambda, \mathcal{E}_\mu), \mathcal{S}'] & \text{où } (\mathcal{E}'_\lambda, \mathcal{S}') = \mathcal{E}_\mu(\alpha)
\end{array}
$$

Bibliography

[Bellin, 2005] Bellin, G. (2005). A term assignment for dual intuitionistic logic. In *Proceedings of the LICS'05-IMLA'05 Workshop*.

[Clint and Hoare, 1972] Clint, M. and Hoare, C. A. R. (1972). Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224.

[Crolard, 2004] Crolard, T. (2004). A Formulæ-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14(4):529–570.

[Crolard and Polonowski, 2010] Crolard, T. and Polonowski, E. (2010). A program logic for higher-order procedural variables and non-local jumps. Submitted.

[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.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic*. North-Holland.

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

[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.

[Girard, 1972] Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Thèse de doctorat d'état, Université Paris VII.

[Görnemann, 1971] Görnemann, S. (1971). A logic stronger than intuitionism. *The Journal of Symbolic Logic*, 36:249–261.

[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.

[Grzegorczyk, 1964] Grzegorczyk, A. (1964). A philosophically plausible formal interpretation of intuitionistic logic. *Nederl. Akad. Wet., Proc., Ser. A*, 67:596–601.

[Howard, 1969] Howard, W. A. (1969). The formulæ-as-types notion of constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculs and Formalism*, pages 479–490. Academic Press.

[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.

[Lopez-Escobar, 1983] Lopez-Escobar, E. G. K. (1983). A second paper "on the interpolation theorem for the logic of constant domains". *The Journal of Symbolic Logic*, 48(3):595–599.

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

[Mitchell and Plotkin, 1985] Mitchell, J. C. and Plotkin, G. D. (1985). Abstract types have existential type. In *12th Annual ACM symposium on Principles of Programming Languages*.

[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.

[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.

[Rauszer, 1974] Rauszer, C. (1974). Semi-boolean algebras and their applications to intuitionistic logic with dual operations. In *Fundamenta Mathematicae*, volume 83, pages 219–249.

[Rauszer, 1980] Rauszer, C. (1980). An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathematicae*, volume 167. Institut Mathématique de l'Académie Polonaise des Sciences.

[Reynolds, 1974] Reynolds, J. C. (1974). Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer.

[Reynolds, 1981] Reynolds, J. C. (1981). *The Craft of Programming*. Prentice Hall PTR Upper Saddle River, NJ, USA.

[Reynolds, 2008] Reynolds, J. C. (2008). An introduction to separation logic. In *Lecture Notes for the FIRST PhD Fall School on Logics and Semantics of State*.

[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.