Cupto

Une géneralisation des exceptions et du contrôle

Didier Rémy

Didier.Remy@inria.fr

INRIA-Rocquencourt

Cupto

Une géneralisation des exceptions et du contrôle

Travail ancien

Didier Rémy

(travail en commun avec Carl Gunter and Jon Riecke)

`Didier.Remy@inria.fr`

INRIA-Rocquencourt

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme

  (fin des années 80)

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme
- Apparition des opérateurs de contrôle en ML

(début des années 90)

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme
- Apparition des opérateurs de contrôle en ML
- Difficultés de typage...

(Solution partielle, non satisfaisante)

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme
- Apparition des opérateurs de contrôle en ML
- Difficultés de typage...
  - Sur le plan *théorique*, résultat de correction faible

    \+ le typage prémunit contre les erreurs d’exécution,

    – mais les types ne sont pas préservés à l’exécution

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme
- Apparition des opérateurs de contrôle en ML
- Difficultés de typage...
  - Sur le plan *théorique*, résultat de correction faible
  - Sur le plan *pratique*, le toplevel de SML est “bricolé” pour préserver la sécurité

    (Pour ne pas laisser son âme au diable,

    le typeur introduit une possibilité d’échec dynamique)

Pourquoi le “cupto” ?

- Prolifération des opérateurs de contrôle en Scheme
- Apparition des opérateurs de contrôle en ML
- Difficultés de typage...
  - Sur le plan *théorique*, résultat de correction faible
  - Sur le plan *pratique*, le toplevel de SML est “bricolé” pour préserver la sécurité
- Que se passe-t-il ? Comment corriger ces problèmes ?

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

- avec des règles de typage et un résultat de correct fort

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

- avec des règles de typage et un résultat de correct fort
- plus expressif que les précédents :

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

- avec des règles de typage et un résultat de correct fort
- plus expressif que les précédents :
  - il capture toutes les différentes espèces de `callcc` de scheme.

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

- avec des règles de typage et un résultat de correct fort
- plus expressif que les précédents :
  - il capture toutes les différentes espèces de `callcc` de scheme.
  - il généralise les exceptions.

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

- avec des règles de typage et un résultat de correct fort
- plus expressif que les précédents :
    - ▷ il capture toutes les différentes espèces de `callcc` de scheme.
    - ▷ il généralise les exceptions.
    - ▷ il est plus facile à manipuler en pratique.

Dans cet exposé

Nous introduisons un nouvel opérateur de contrôle `cupto`

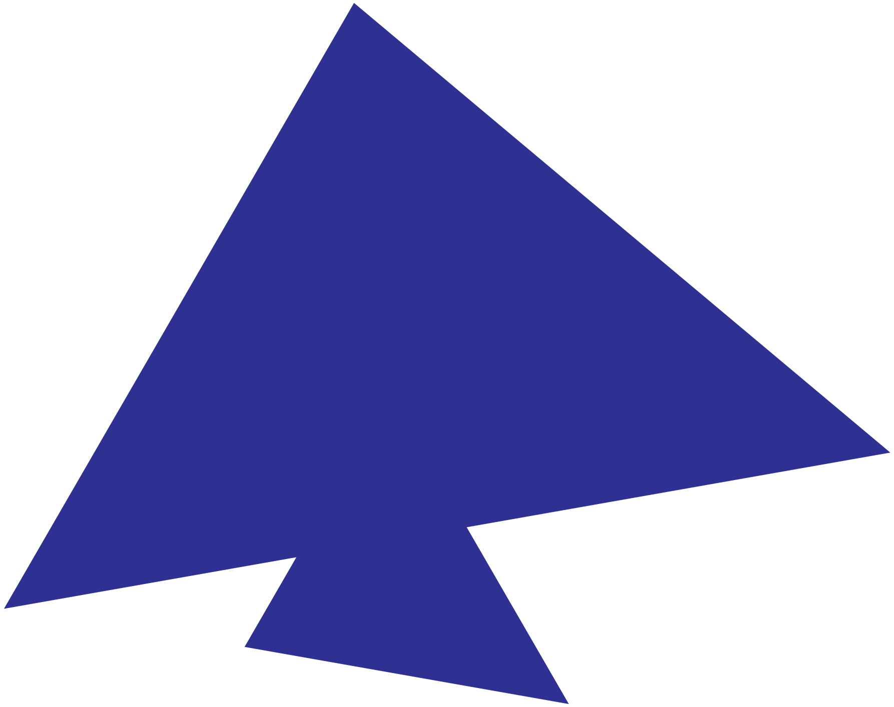
- avec des règles de typage et un résultat de correct fort
- plus expressif que les précédents :
  - il capture toutes les différentes espèces de `callcc` de scheme.
  - il généralise les exceptions.
  - il est plus facile à manipuler en pratique.

Depuis, de nouvelles constructions sont apparues

(par exemple, utiliser une hiérarchie de shift/reset)

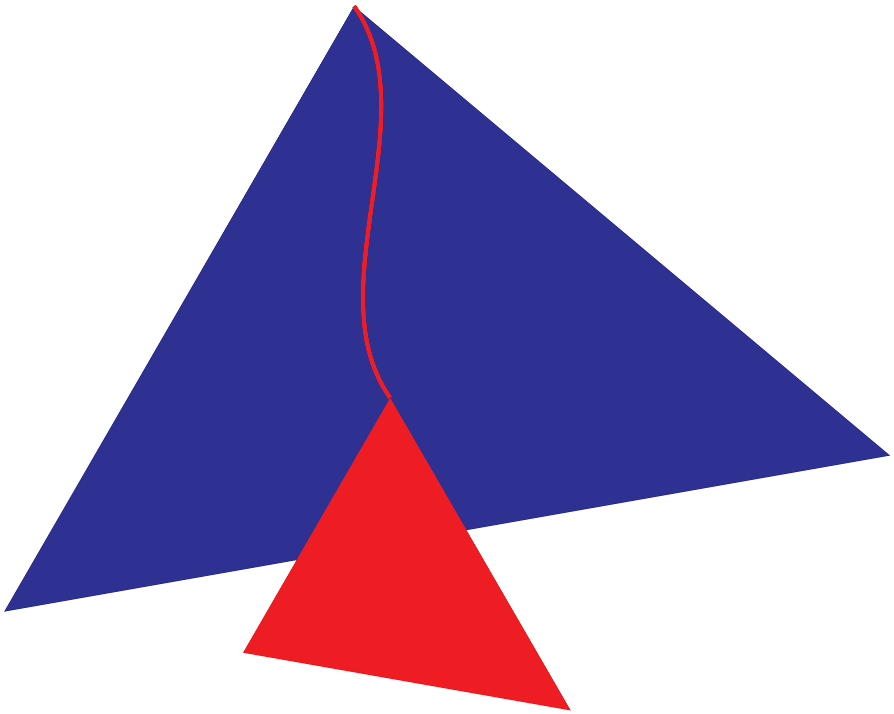
Le contrôle

Description de l’évaluation (sémantique à petits pas)



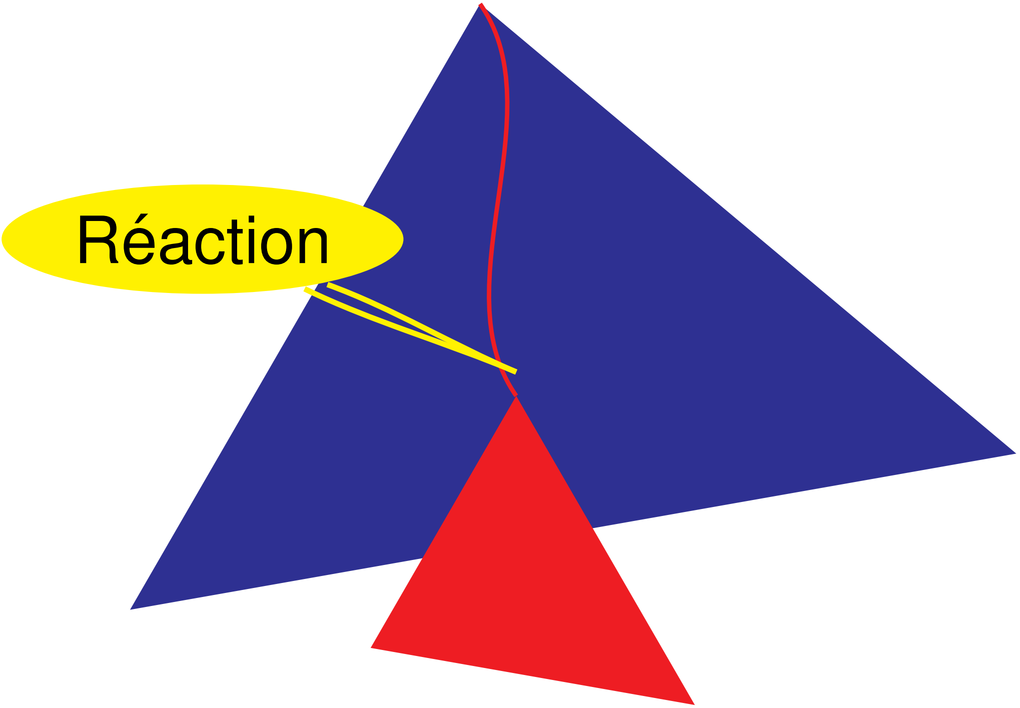
Le contrôle

Description de l’évaluation (sémantique à petits pas)



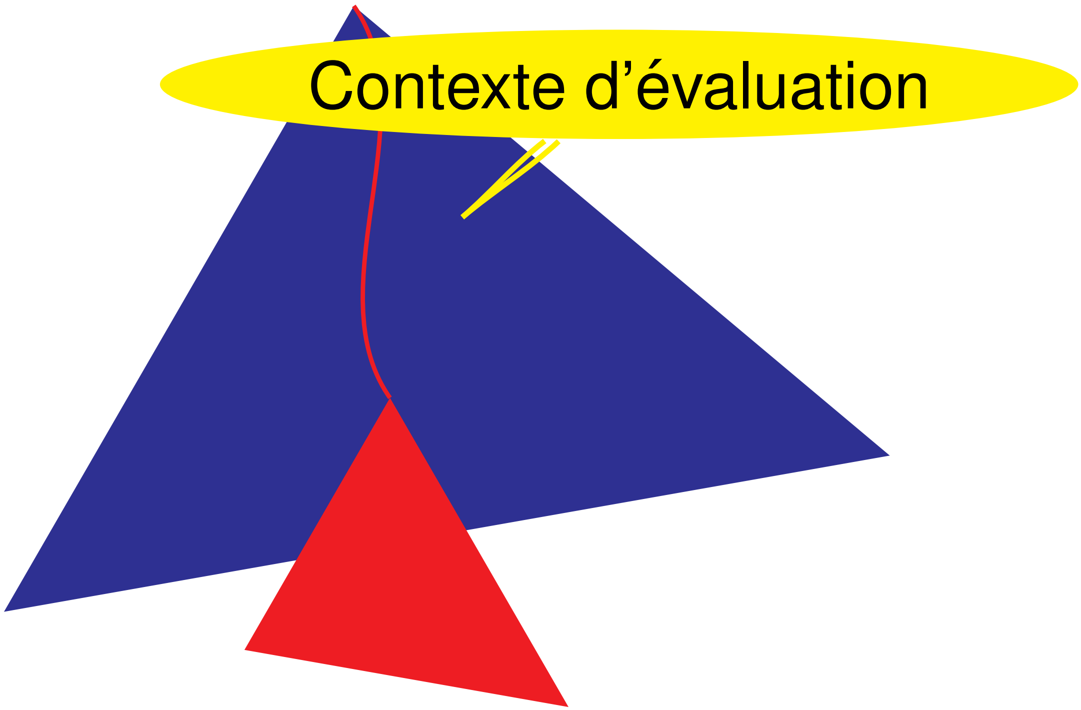
Le contrôle

Description de l’évaluation (sémantique à petits pas)



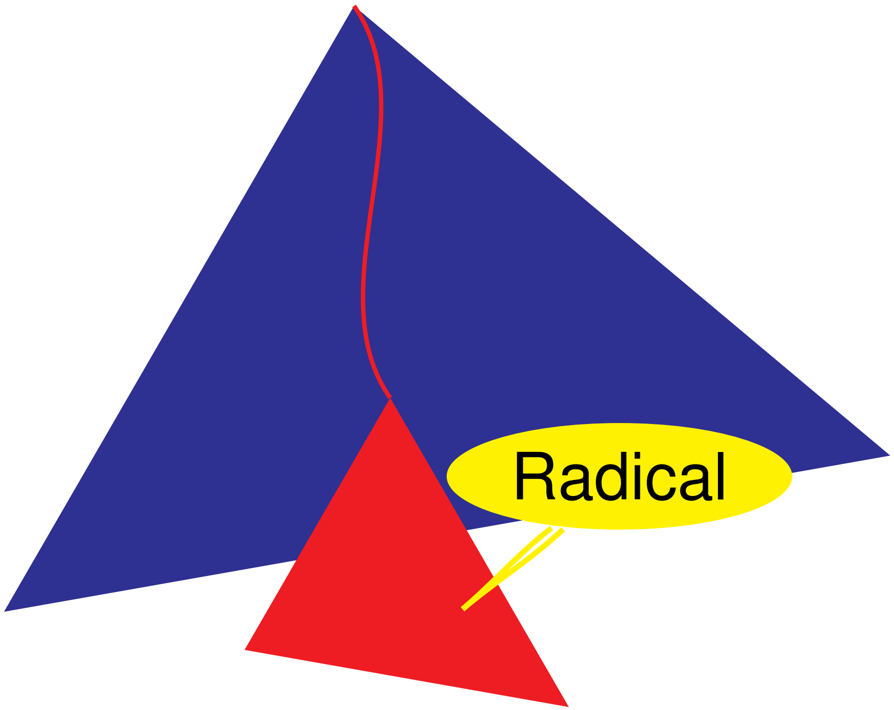
Le contrôle

Description de l’évaluation (sémantique à petits pas)



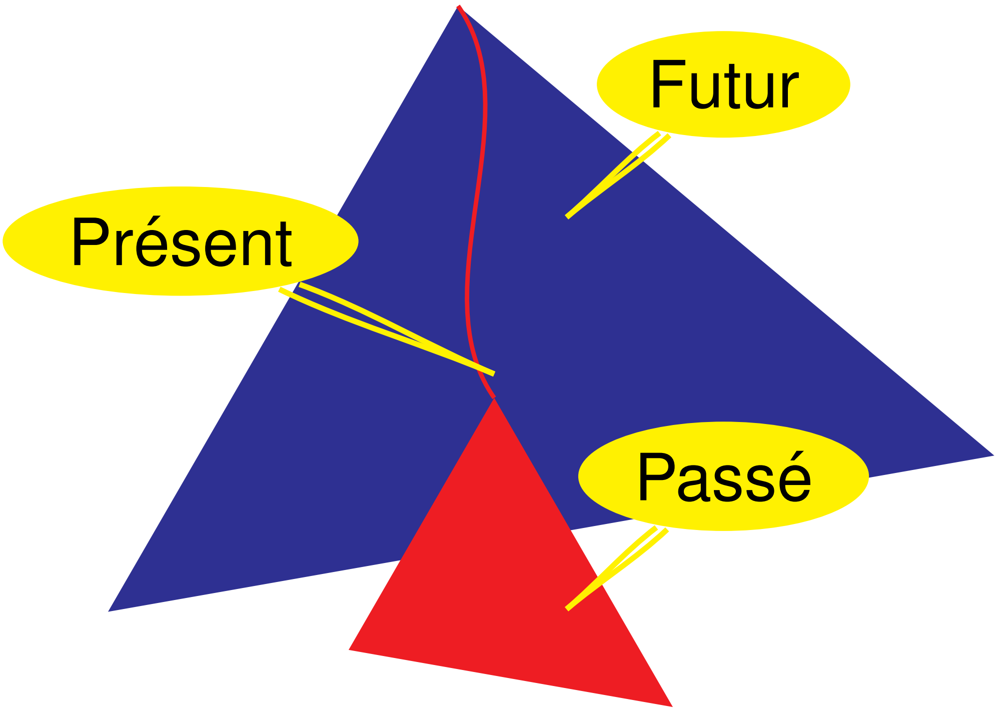
Le contrôle

Description de l’évaluation (sémantique à petits pas)



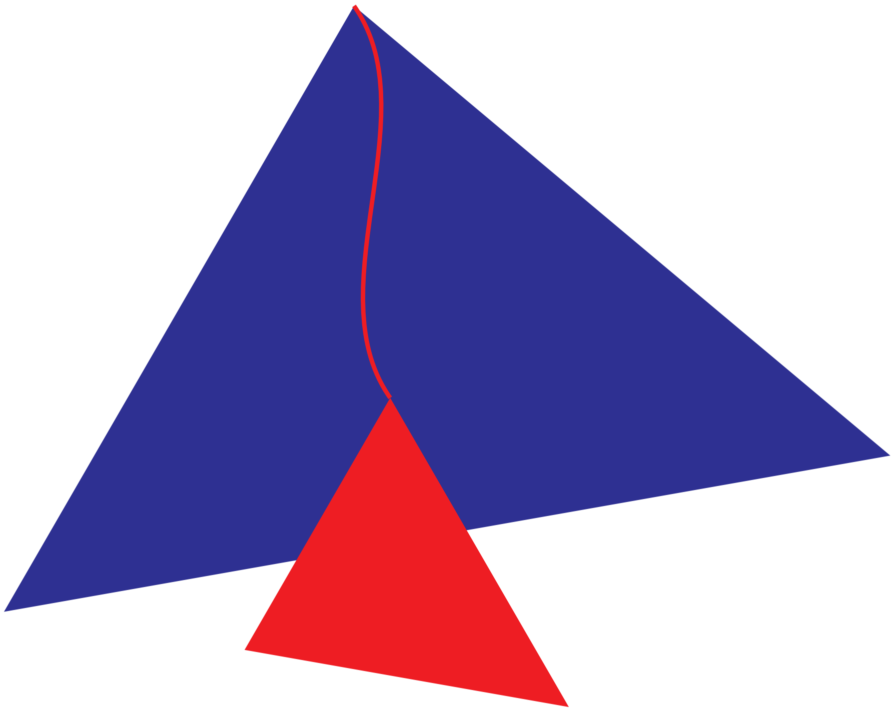
Le contrôle

Description de l’évaluation (sémantique à petits pas)



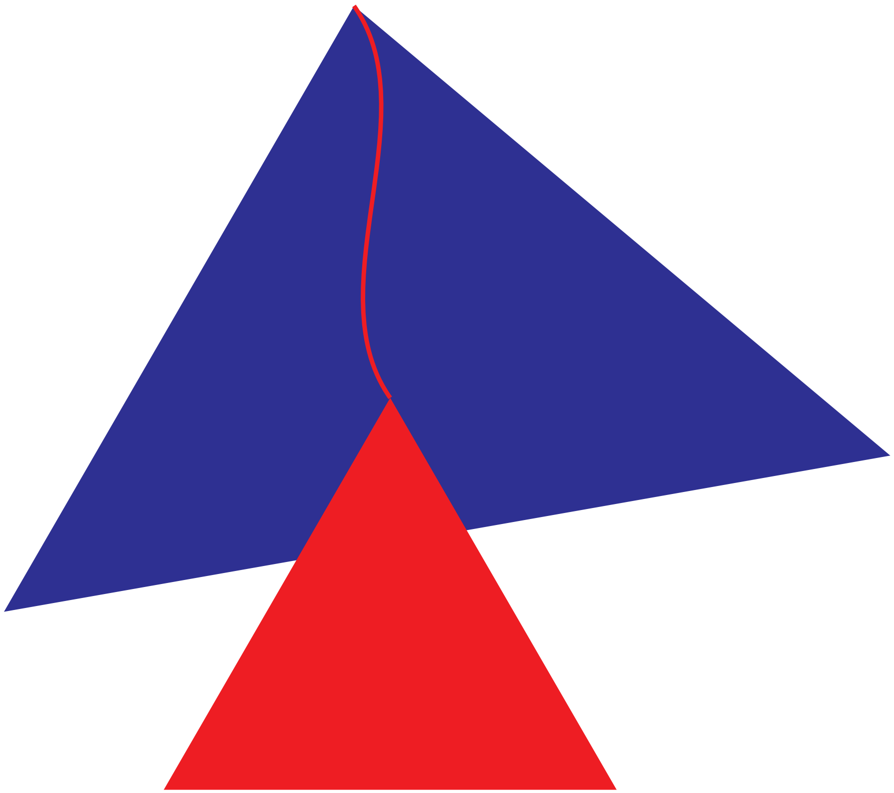
Le contrôle

Description de l’évaluation (sémantique à petits pas)



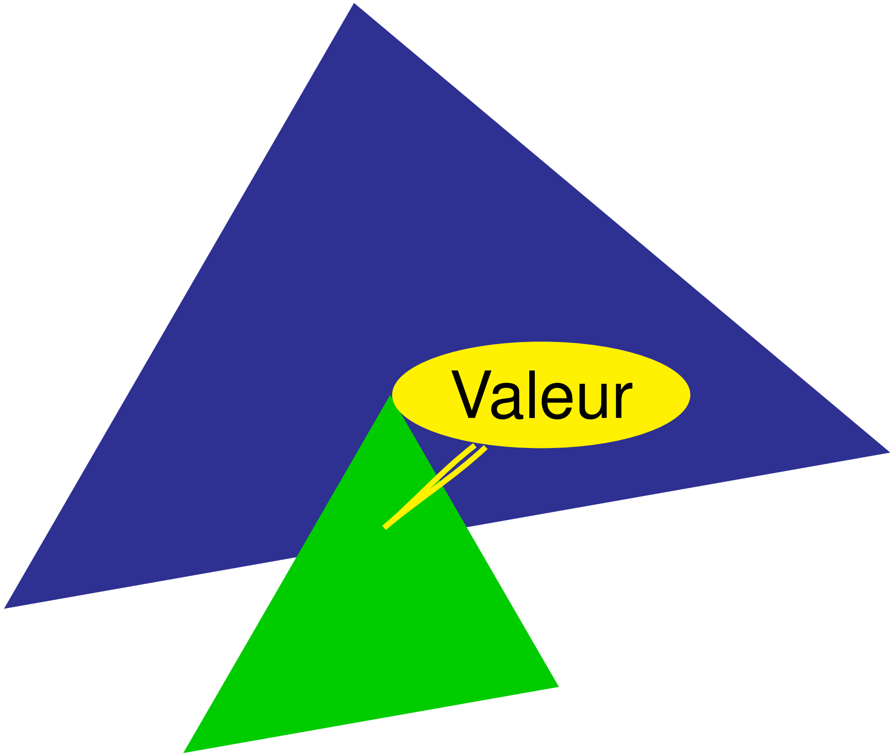
Le contrôle

Description de l’évaluation (sémantique à petits pas)



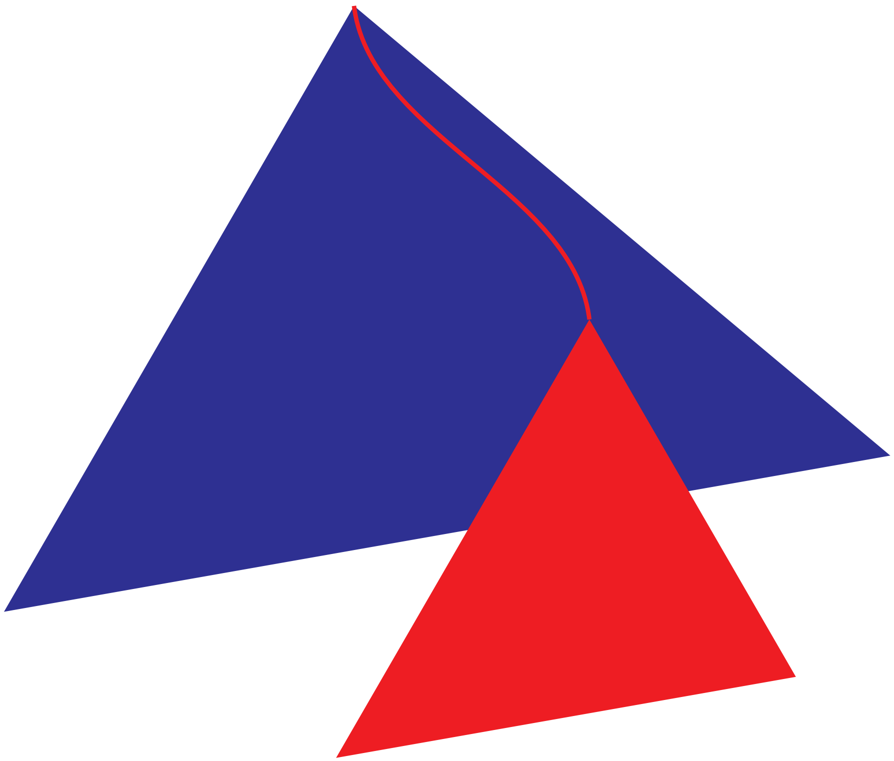
Le contrôle

Description de l’évaluation (sémantique à petits pas)



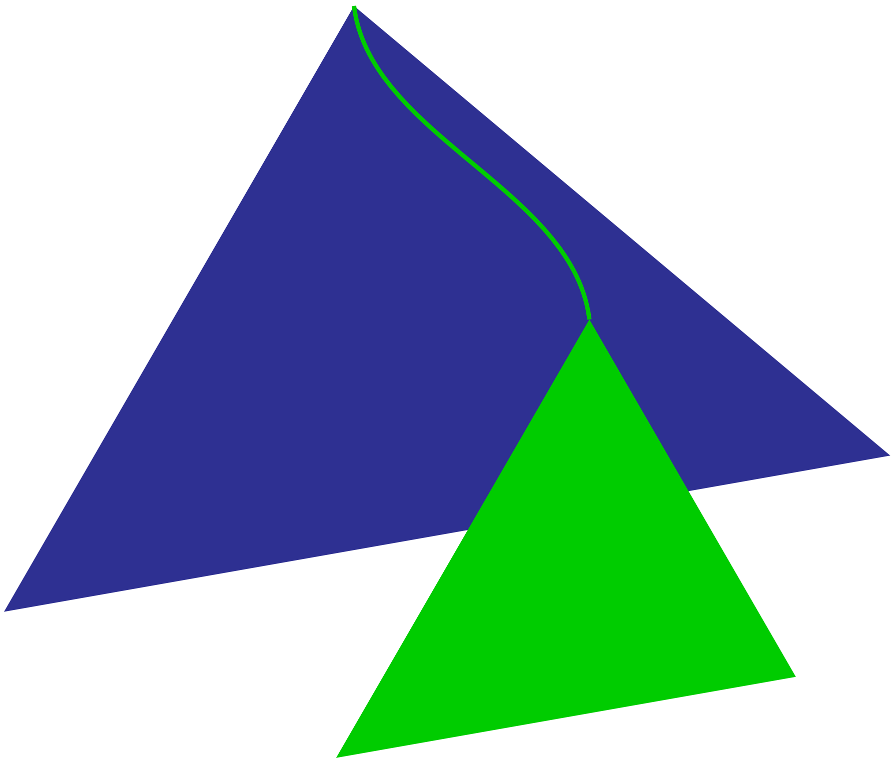
Le contrôle

Description de l'évaluation (sémantique à petits pas)



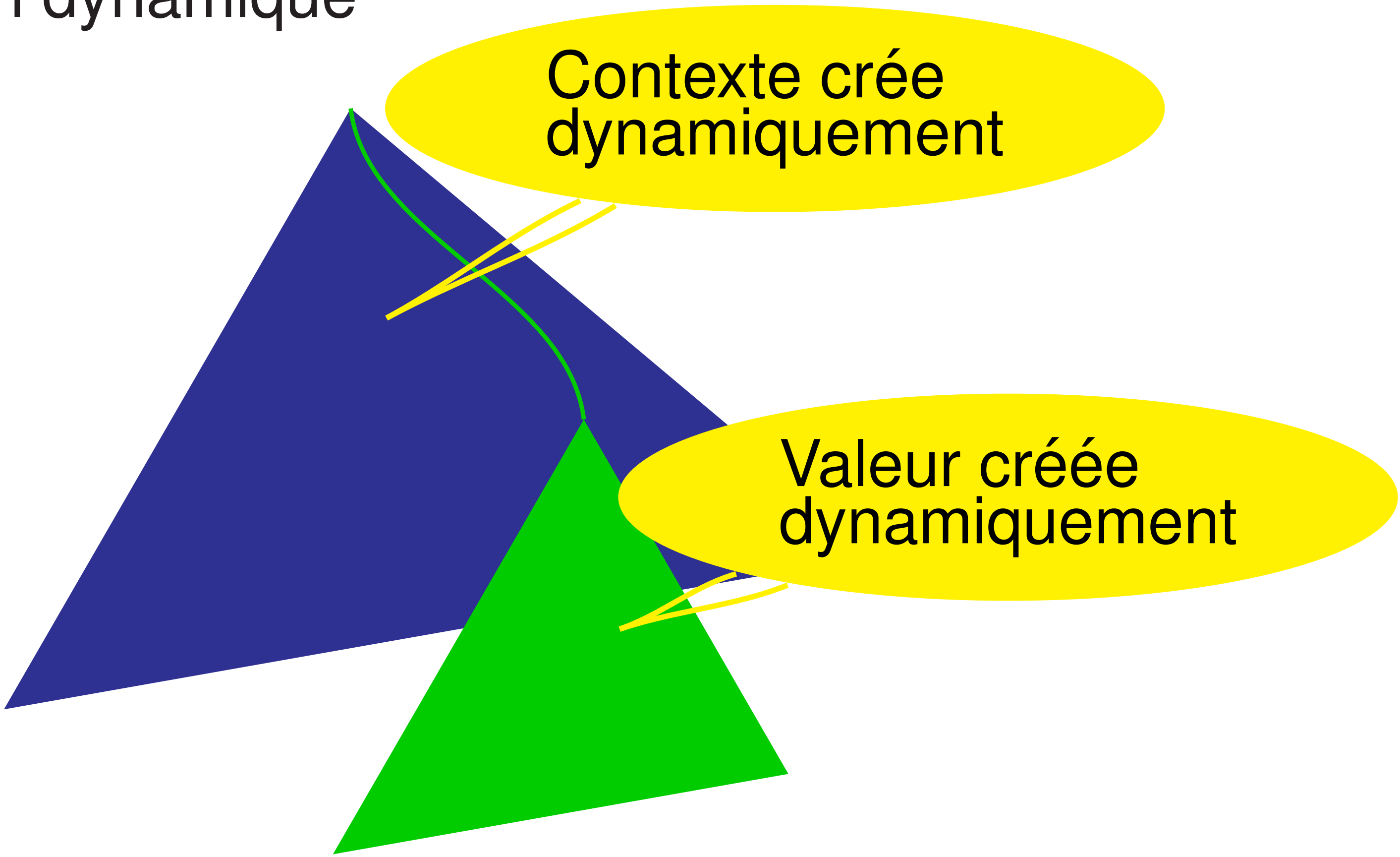
Le contrôle

Description de l’évaluation (sémantique à petits pas)



Le contrôle

Liaison dynamique

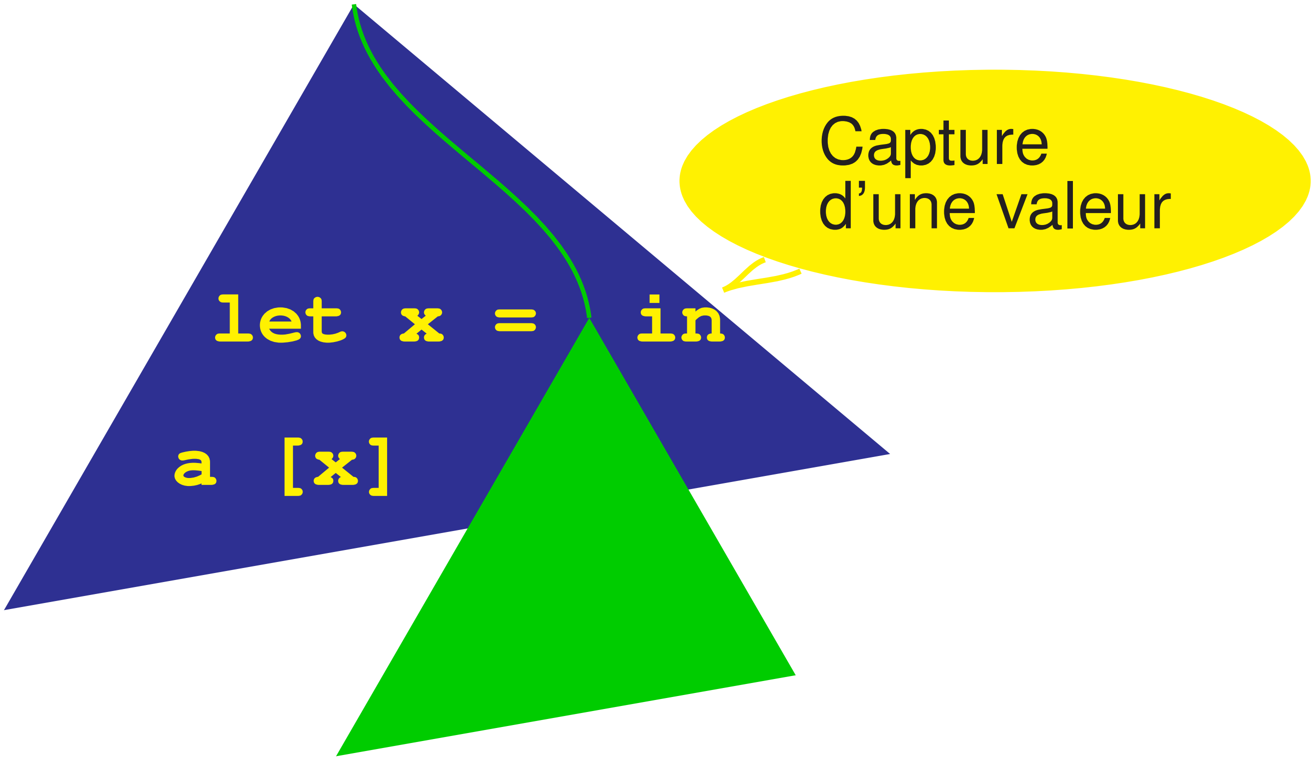


Contexte crée dynamiquement

Valeur créée dynamiquement

Le contrôle

Liaison dynamique de valeur.



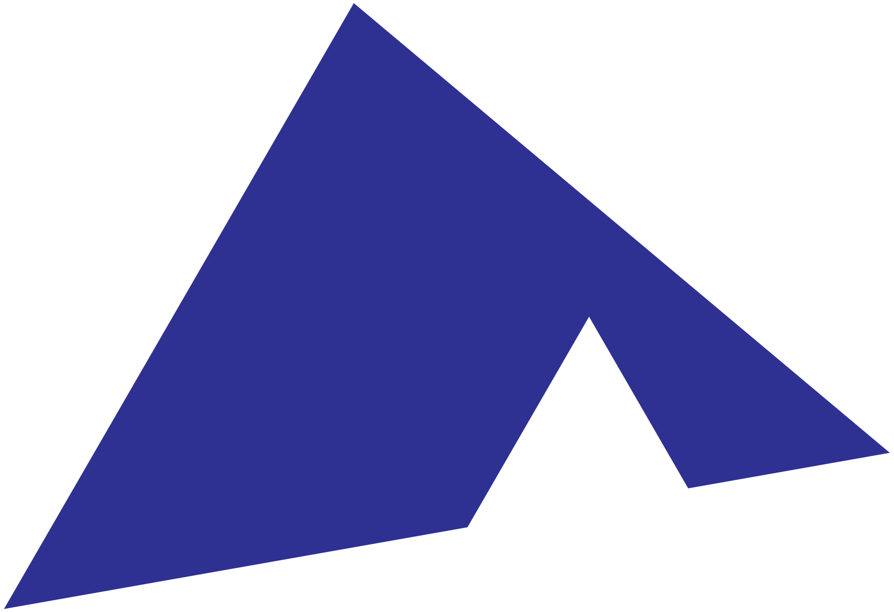
`let x =  in`

`a [x]`

Capture
d’une valeur

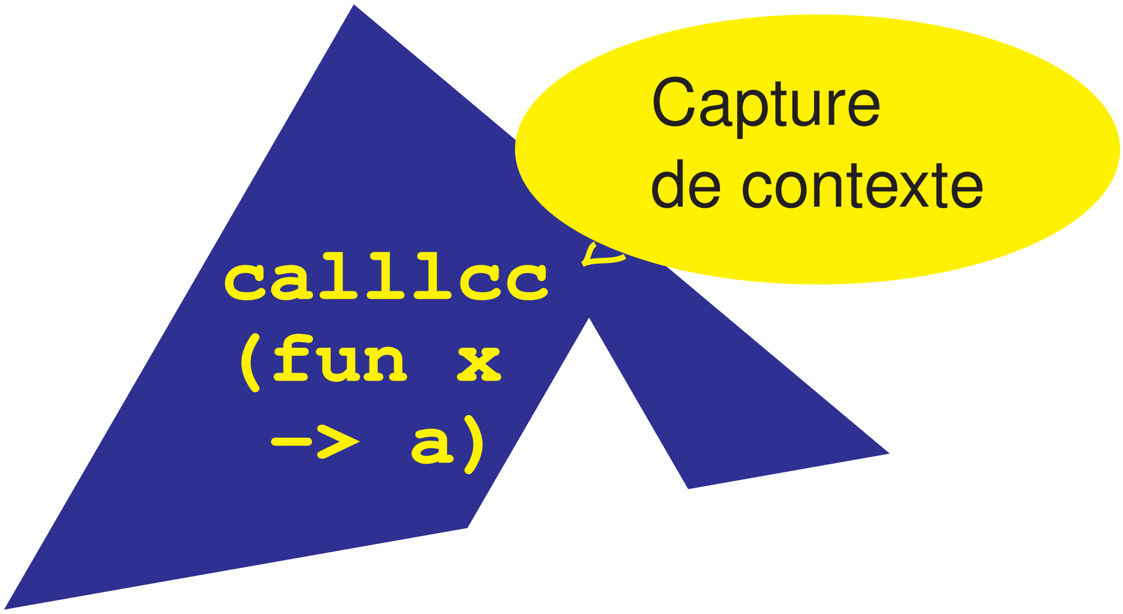
Le contrôle

Liaison dynamique de valeur. **Et de contexte ?**



Le contrôle

Liaison dynamique

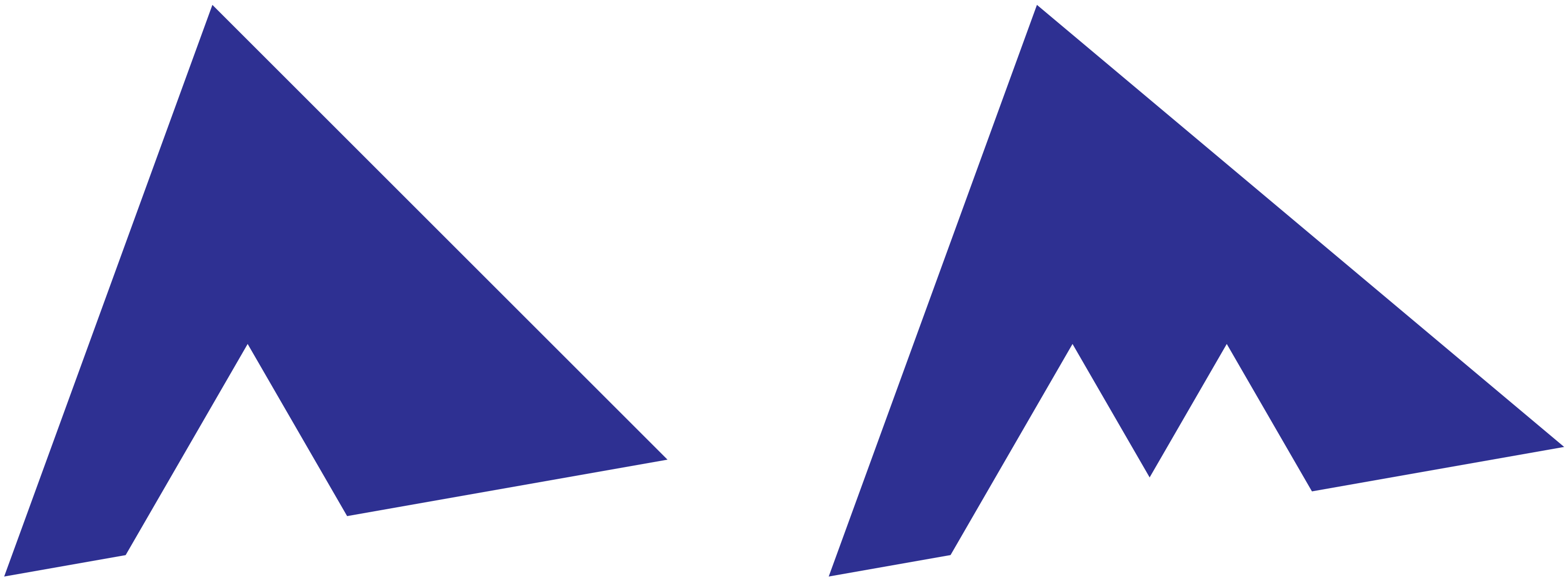


Capture
de contexte

```
calllcc
(fun x
 -> a)
```

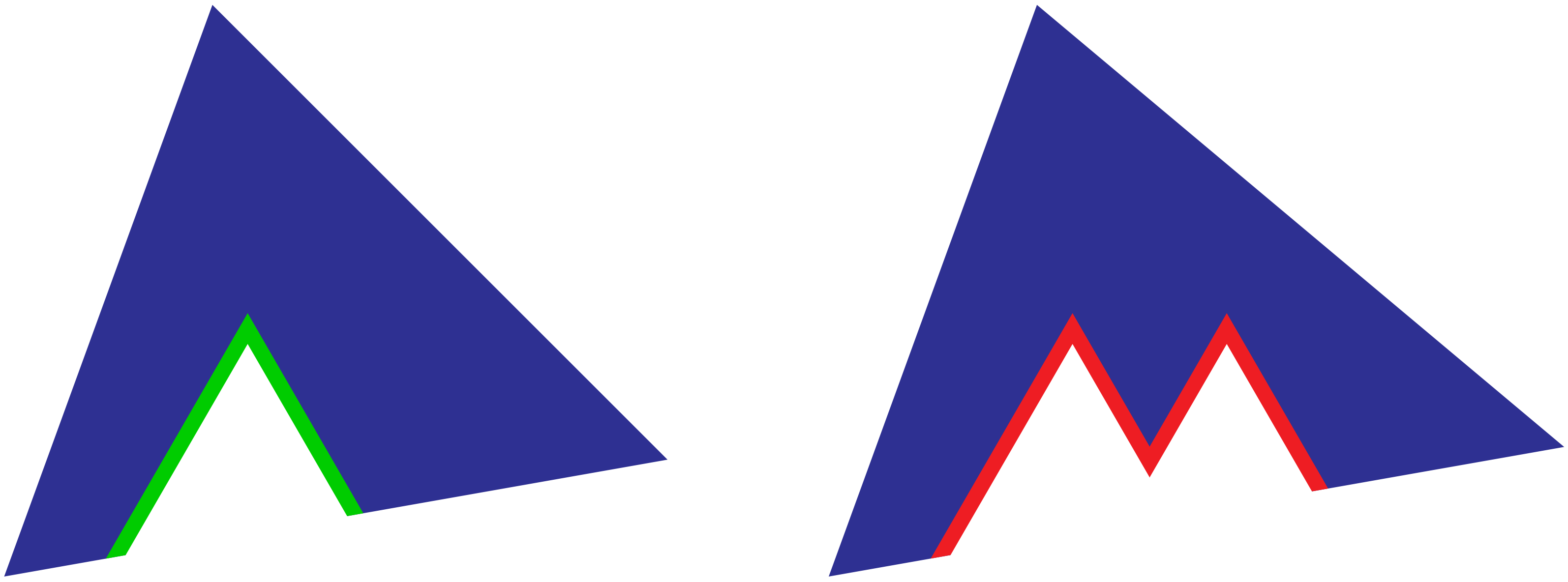
Problème de typage

Typage du contexte = Typage du trou



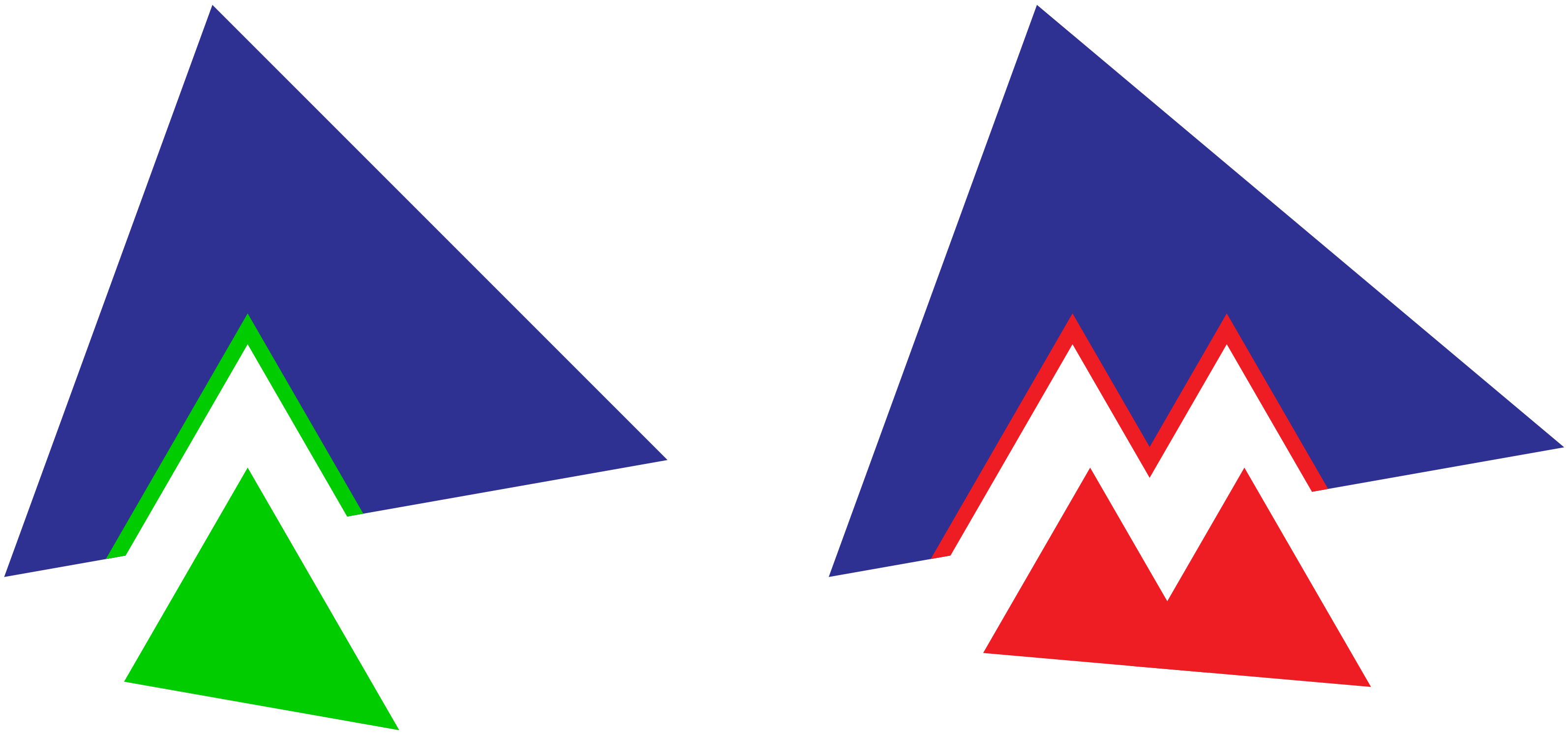
Problème de typage

Typage du contexte = Typage du trou



Problème de typage

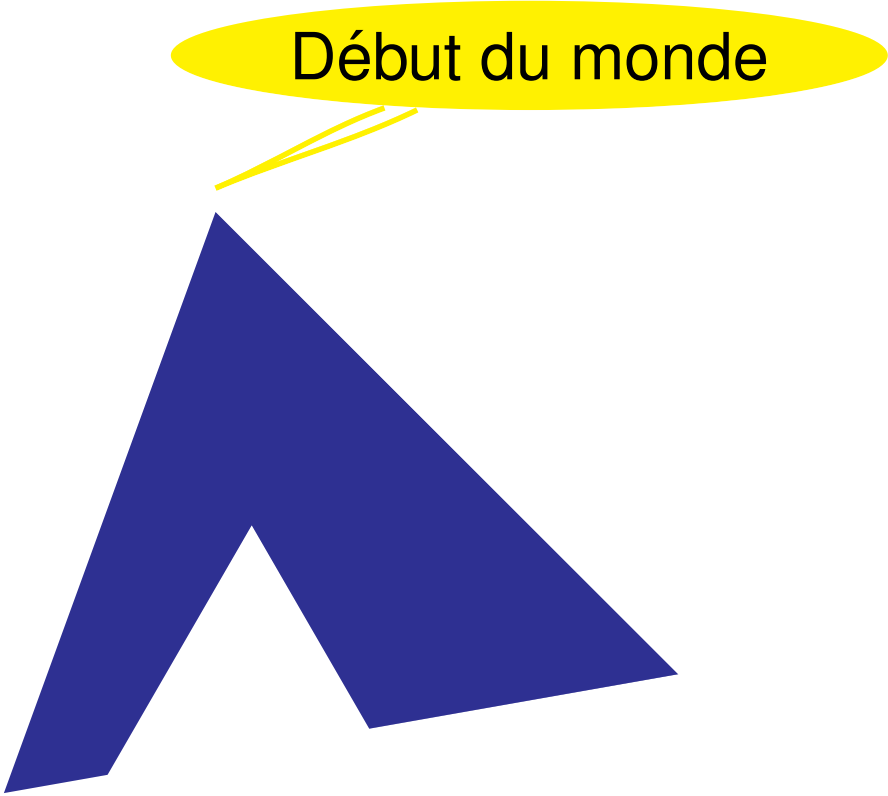
Typage du contexte = Typage du trou



Empêche de placer un terme dans un contexte dont le trou n’est pas du bon type

Problème de typage

Monde fermé (possible en théorie)



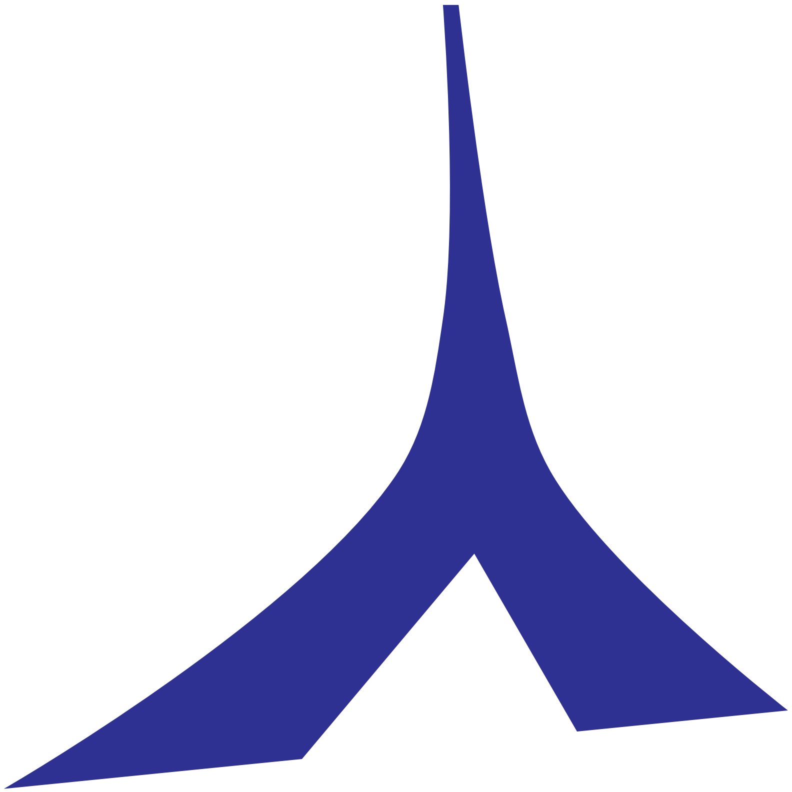
Début du monde

Tous les contextes commencent (et retournent) au big bang.

Problème de typage

Monde ouvert (nécessaire en pratique)

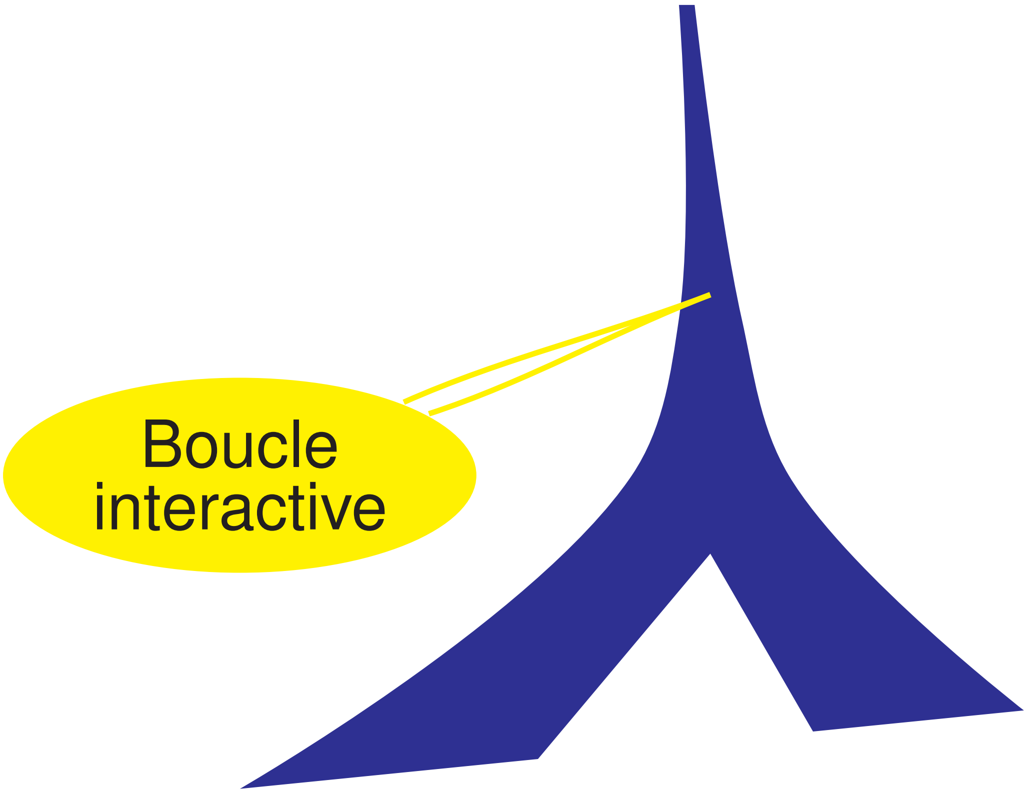
Où commence le contexte ?



Problème de typage

Monde ouvert (nécessaire en pratique)

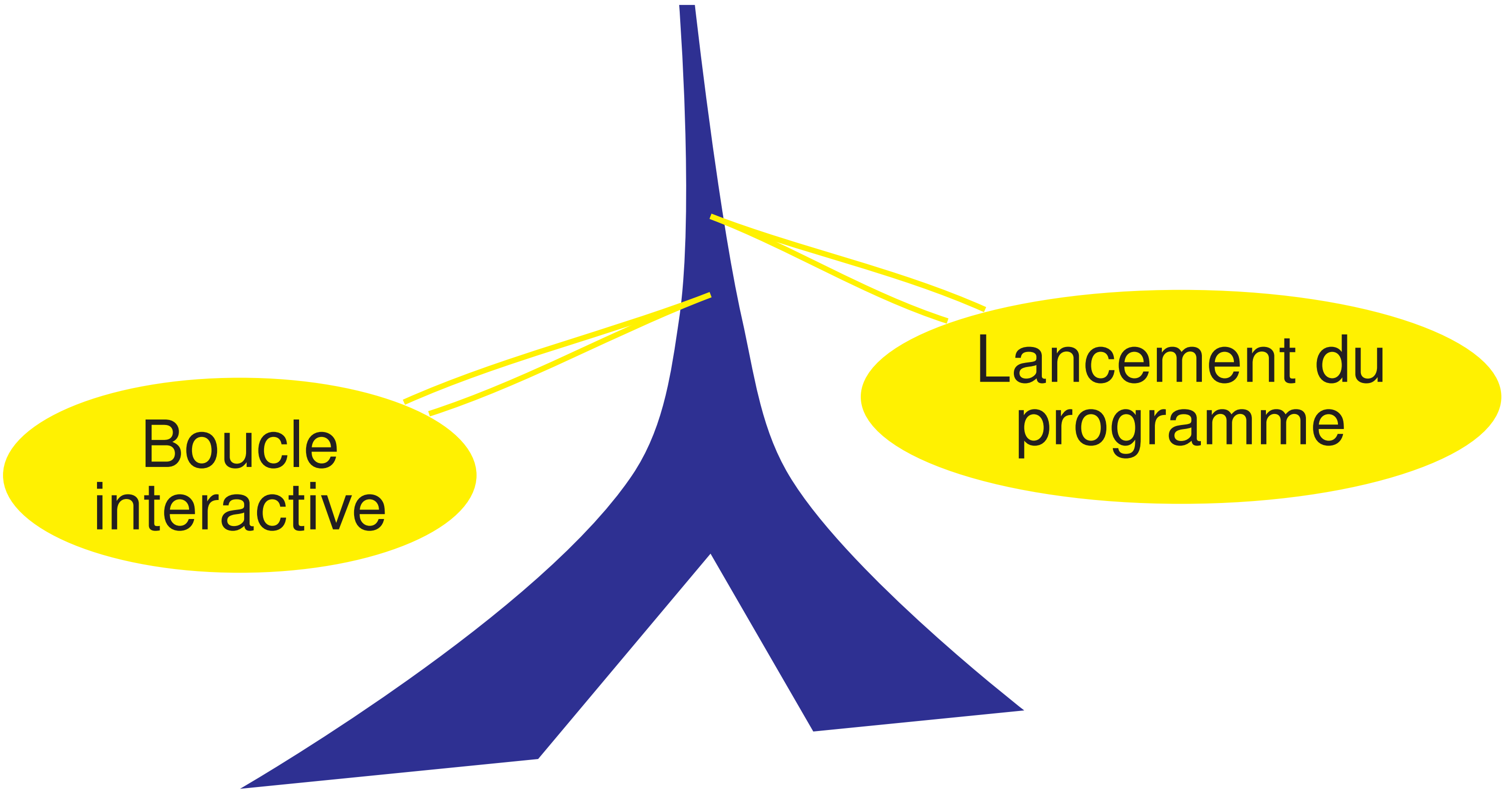
Où commence le contexte ?



Problème de typage

Monde ouvert (nécessaire en pratique)

Où commence le contexte ?



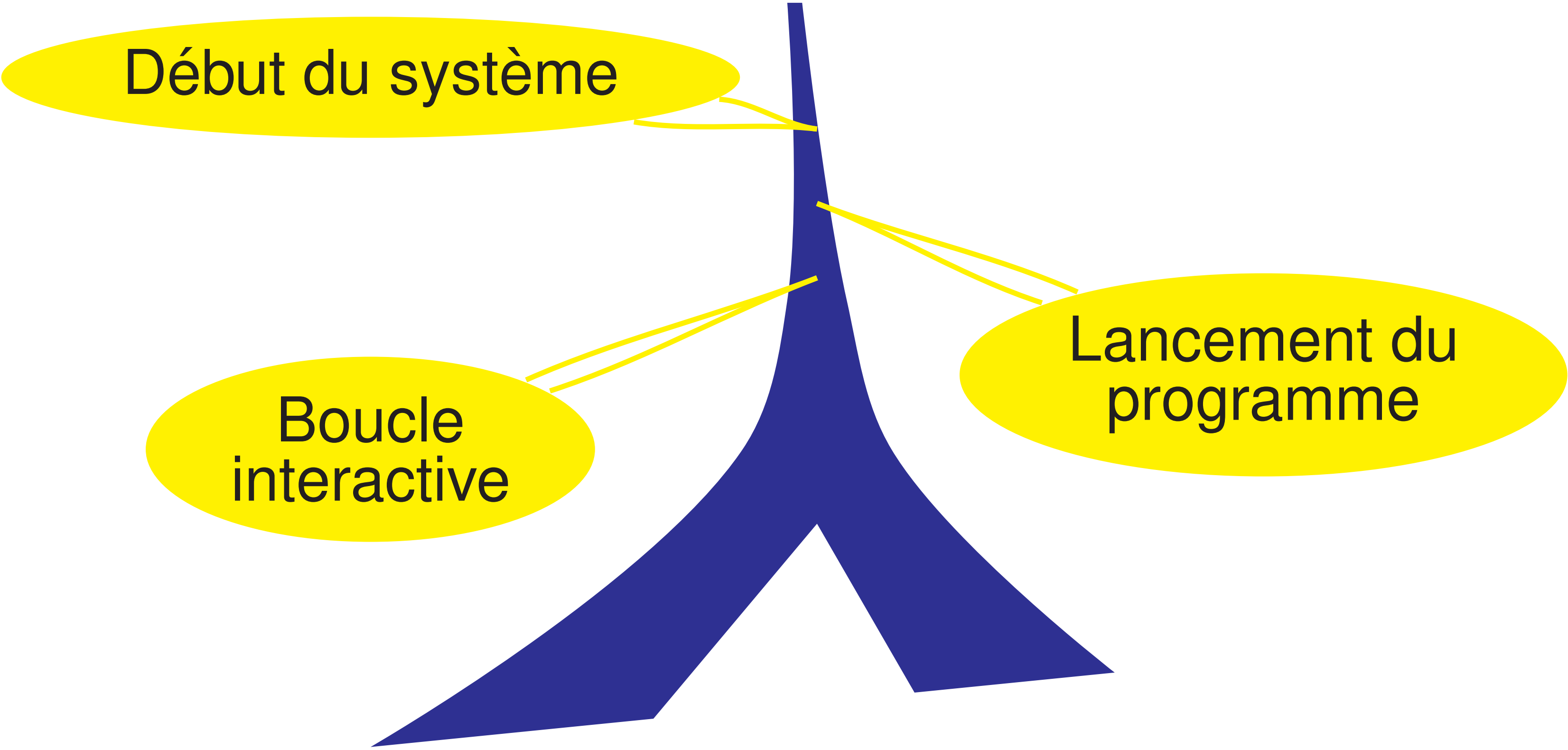
Boucle interactive

Lancement du programme

Problème de typage

Monde ouvert (nécessaire en pratique)

Où commence le contexte ?



Début du système

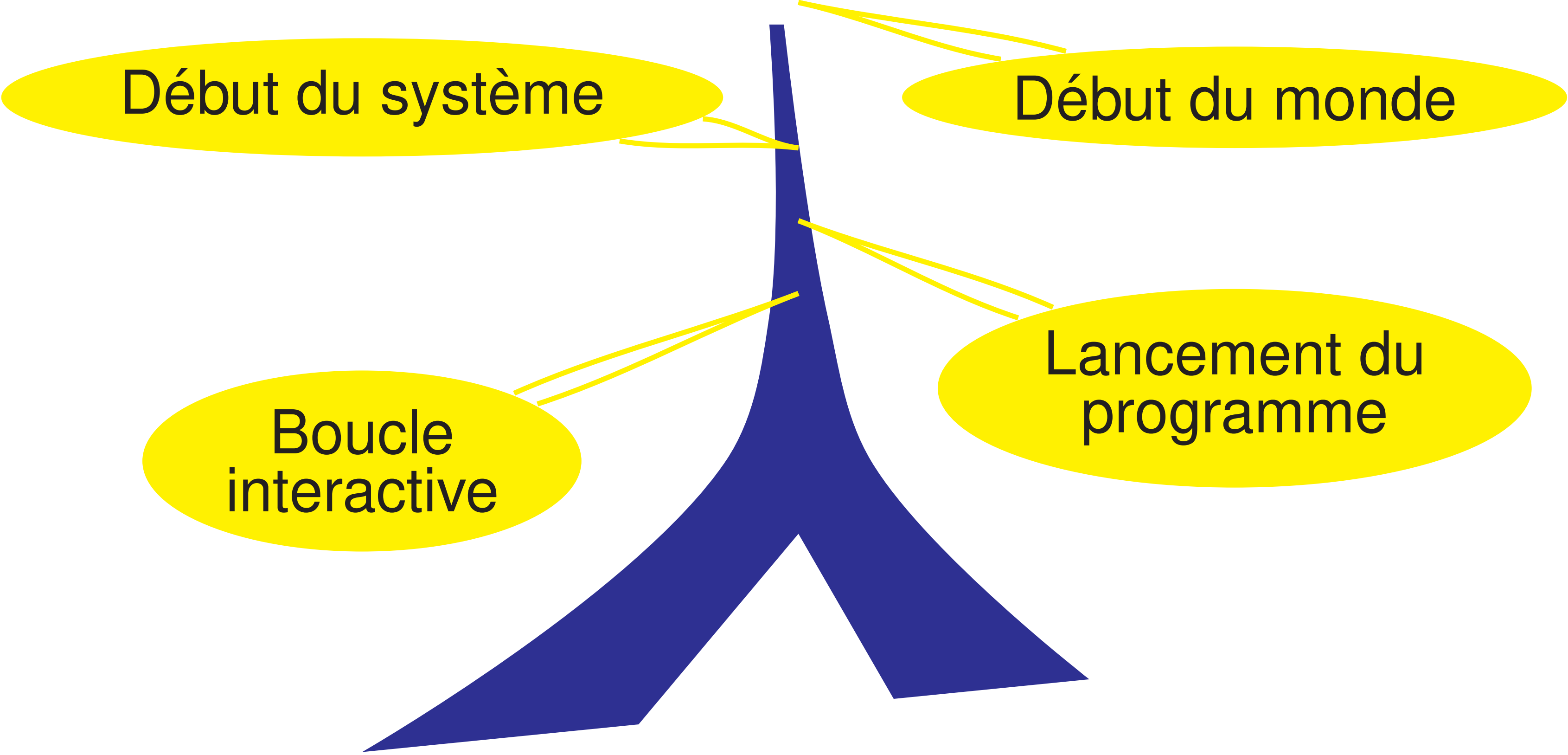
Lancement du programme

Boucle interactive

Problème de typage

Monde ouvert (nécessaire en pratique)

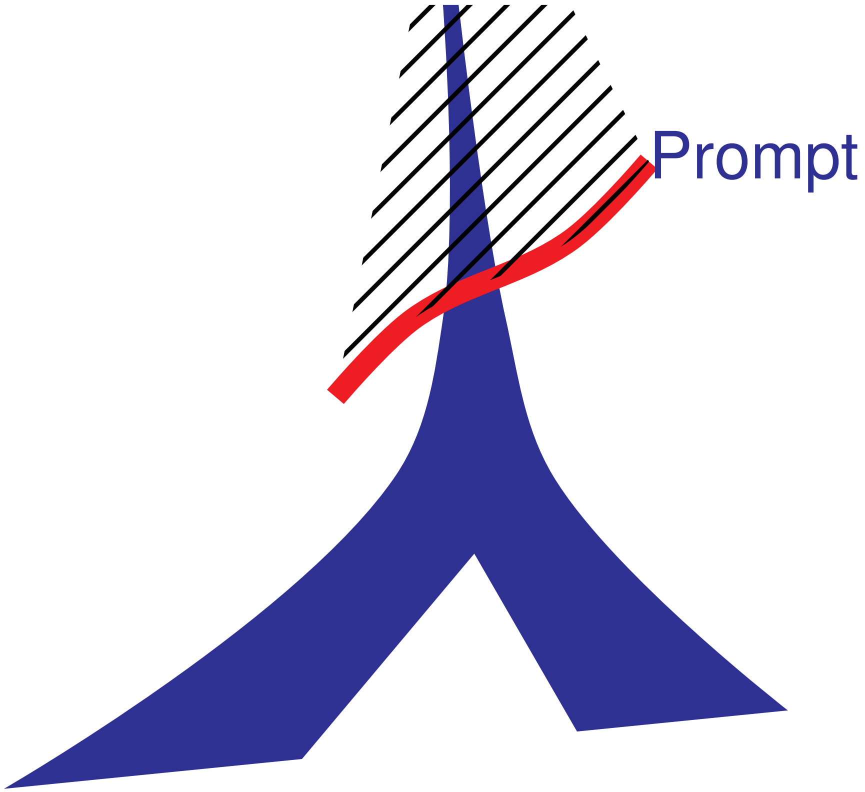
Où commence le contexte ?



Problème de typage

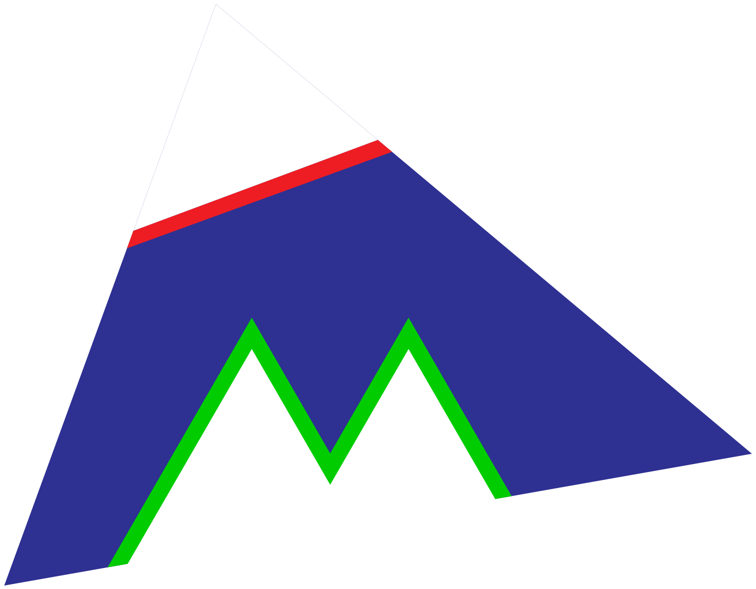
Monde ouvert (nécessaire en pratique)

Où commence le contexte ? à un prompt !



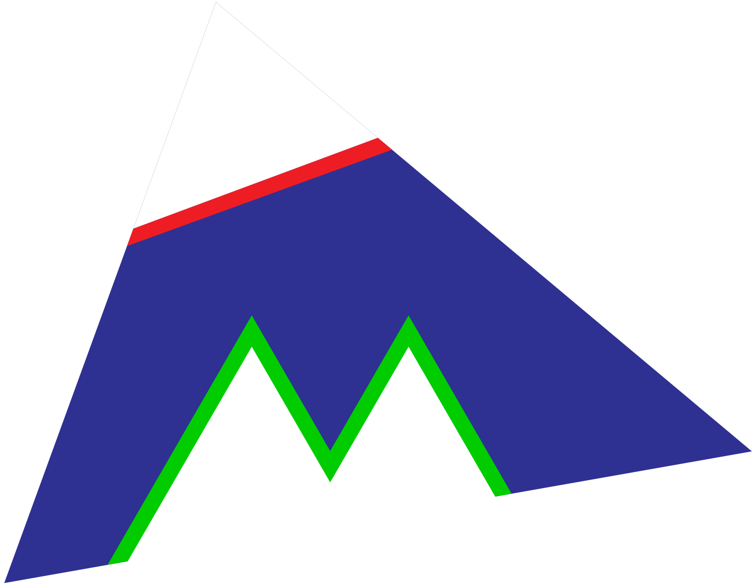
Problème de typage

Au final



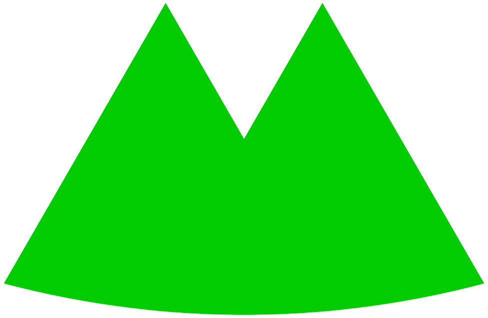
Problème de typage

Au final



Behaves as a function, that

given



produces

Une recette avec deux ingrédients

- Les continuations doivent être traîtées comme des fonctions

Une recette avec deux ingrédients

- Les continuations doivent être traîtées comme des fonctions
  - elles doivent être typées comme telles.
  - elles peuvent alors être données au programmeur.

Une recette avec deux ingrédients

- Les continuations doivent être traîtées comme des fonctions
  - elles doivent être typées comme telles.
  - elles peuvent alors être données au programmeur.
- La capture d’une continuation doit toujours être délimitée par un prompt

Une recette avec deux ingrédients

- Les continuations doivent être traîtées comme des fonctions
  - elles doivent être typées comme telles.
  - elles peuvent alors être données au programmeur.
- La capture d’une continuation doit toujours être délimitée par un prompt
  - un prompt a un type unique, connu statiquement.
  - les prompts sont posés par le programmeur, dynamiquement.
  - le programmeur crée ses prompts (de différents types).

Syntaxe

$a ::= v \mid (a_1\ a_2) \mid \texttt{let}\ x = v\ \texttt{in}\ a$	ML ...
$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Pose d’un prompt
$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Capture de la CC
$v ::= x \mid (\lambda x.\, a) \mid ()$	ML ...
$\mid p$	Prompt (valeur)
$\mid \texttt{new\_prompt}$	Création d’un prompt

Syntaxe

$a ::= v \mid (a_1\ a_2) \mid \texttt{let}\ x = v\ \texttt{in}\ a$	ML ...
$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Pose d'un prompt
$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Capture de la CC
$v ::= x \mid (\lambda x.\, a) \mid ()$	ML ...
$\mid p$	Prompt (valeur)
$\mid \texttt{new\_prompt}$	Création d'un prompt

Syntaxe

$a ::= v \mid (a_1\ a_2) \mid \texttt{let}\ x = v\ \texttt{in}\ a$	ML ...
$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Pose d'un prompt
$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Capture de la CC
$v ::= x \mid (\lambda x.\, a) \mid ()$	ML ...
$\mid p$	Prompt (valeur)
$\mid \texttt{new\_prompt}$	Création d'un prompt

Syntaxe

$a ::= v \mid (a_1\ a_2) \mid \texttt{let}\ x = v\ \texttt{in}\ a$	ML ...
$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Pose d'un prompt
$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Capture de la CC
$v ::= x \mid (\lambda x.\, a) \mid ()$	ML ...
$\mid p$	Prompt (valeur)
$\mid \texttt{new\_prompt}$	Création d'un prompt

Syntaxe

$a ::= v \mid (a_1\ a_2) \mid \texttt{let}\ x = v\ \texttt{in}\ a$	ML ...
$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Pose d’un prompt
$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Capture de la CC
$v ::= x \mid (\lambda x.\, a) \mid ()$	ML ...
$\mid p$	Prompt (valeur)
$\mid \texttt{new\_prompt}$	Création d’un prompt

Sémantique dynamique

Sémantique à réduction (à petits pas)

Sémantique dynamique

Sémantique à réduction (à petits pas)

- Contextes d'évaluation

$$
\begin{aligned}
E ::= & \ \dots \\
& \mid \texttt{set}\ E\ \texttt{in}\ a \mid \texttt{set}\ p\ \texttt{in}\ E \\
& \mid \texttt{cupto}\ E\ \texttt{as}\ x\ \texttt{in}\ a
\end{aligned}
$$

Sémantique dynamique

Sémantique à réduction (à petits pas)

- Contextes d'évaluation
- Congruence (P est l'ensemble des prompts déjà utilisés)

$$\text{If } a_1 \,/\, P_1 \longrightarrow_r a_2 \,/\, P_2 \text{ then } E[a_1] \,/\, P_1 \;\longrightarrow\; E[a_2] \,/\, P_2$$

Sémantique dynamique

Sémantique à réduction (à petits pas)

- Contextes d'évaluation
- Congruence
- Réduction β_v (locale)

$$(\texttt{new\_prompt}\ ())\ /\ P \longrightarrow_r p\ /\ \{p\} \cup P \qquad (p \notin P)$$

$$\texttt{set}\ p\ \texttt{in}\ v\ /\ P \longrightarrow_r v\ /\ P$$

Sémantique dynamique

Sémantique à réduction (à petits pas)

- Contextes d’évaluation
- Congruence
- Réduction β_v (locale)
- Plus réduction spécialisée (non locale)

$$\begin{aligned} &\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a] \,/\, P \\ &\quad \longrightarrow_r \texttt{let}\ x = \lambda y.\, E_p[y]\ \texttt{in}\ a \,/\, P \end{aligned}$$

Sémantique dynamique

Sémantique à réduction (à petits pas)

- Contextes d'évaluation
- Congruence
- Réduction β_v
- Plus réduction spécialisée (non locale)

Contexte arbitraire
ne contenant pas p

$$\texttt{set}\ {\color{red}p}\ \texttt{in}\ E_p[\texttt{cupto}\ {\color{red}p}\ \texttt{as}\ x\ \texttt{in}\ a]\ /\ P$$

$$\longrightarrow_r\ \texttt{let}\ x = \lambda y.\, E_p[y]\ \texttt{in}\ a\ /\ P$$

Sémantique statique

Sémantique statique

- Types et règles de typage *à la ML*

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto

$$\frac{A \vdash a_1 : (\tau_1\ \texttt{prompt}) \qquad A, x : (\tau_0 \rightarrow \tau_1) \vdash a_2 : \tau_1}{A \vdash \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau_0}$$

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto

$$\frac{A \vdash a_1 : (\tau_1\ \texttt{prompt}) \qquad A, x : (\tau_0 \to \tau_1) \vdash a_2 : \tau_1}{A \vdash \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau_0}$$

Intuitivement, si a_1 est évalué,

$$\texttt{set}\ (a_1 : \tau_1\ \texttt{prompt})\ \texttt{in}$$

$$(E_p[\texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ (a_2 : \tau_1)] : \tau_1)$$

se réduit en

$$\texttt{let}\ x = \lambda y.\, (E_p[y : \tau_0]) : \tau_1\ \texttt{in}\ (a_2 : \tau_1)$$

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto

$$\frac{A \vdash a_1 : (\tau_1\ \mathtt{prompt}) \qquad A, x : (\tau_0 \to \tau_1) \vdash a_2 : \tau_1}{A \vdash \mathtt{cupto}\ a_1\ \mathtt{as}\ x\ \mathtt{in}\ a_2 : \tau_0}$$

Intuitivement, si a_1 est évalué,

$$\mathtt{set}\ (a_1 : \tau_1\ \mathtt{prompt})\ \mathtt{in}$$

$$(E_p[\mathtt{cupto}\ a_1\ \mathtt{as}\ x\ \mathtt{in}\ (a_2 : \tau_1)] : \tau_1)$$

se réduit en

$$\mathtt{let}\ x = \lambda y.\,(E_p[y : \tau_0]) : \tau_1\ \mathtt{in}\ (a_2 : \tau_1)$$

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto
- Le typage des prompts : polymorphisme réservé aux valeurs (*cf.*. références)

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto
- Le typage des prompts : polymorphisme réservé aux valeurs (*cf.*. références)

$$\frac{}{A \vdash \texttt{new\_prompt} : (\texttt{unit} \rightarrow \tau\ \texttt{prompt})}$$

  - ▷ Une fonction qui crée un prompt est polymorphe.
  - ▷ Une fois crée (par une application) un prompt est monomorphe.

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto
- Le typage des prompts : polymorphisme réservé aux valeurs (*cf.*. références)

$$\frac{p : \tau \in A}{A \vdash p : (\tau\ \texttt{prompt})}$$

- ▷ Dans l'environnement de typage un prompt est monomorphe.

Sémantique statique

- Types et règles de typage *à la ML*
- Le typage du Cupto
- Le typage des prompts : polymorphisme réservé aux valeurs (*cf.*. références)

$$\frac{A \vdash a_1 : (\tau\ \texttt{prompt}) \qquad A \vdash a_2 : \tau}{A \vdash \texttt{set}\ a_1\ \texttt{in}\ a_2 : \tau}$$

  - ▷ Le prompt et sa pose sont monomorphes.
  - ▷ Une fonction qui le pose peut être polymorphe.

Correction forte

TH : Auto-réduction *La réduction préserve le typage*

TH : Progrès *Les programmes bien typés qui ne sont pas des valeurs peuvent être réduits ou bien sont des captures de continuations qui s'échappent.*

Correction forte

TH : Auto-réduction *La réduction préserve le typage*

TH : Progrès *Les programmes bien typés qui ne sont pas des valeurs peuvent être réduits ou bien sont des captures de continuations qui s’échappent.*

- Erreur dynamique possible

  (analogue à la levée d’une exception non rattrapée)

Correction forte

TH : Auto-réduction *La réduction préserve le typage*

TH : Progrès *Les programmes bien typés qui ne sont pas des valeurs peuvent être réduits ou bien sont des captures de continuations qui s'échappent.*

- Erreur dynamique possible
- Évitable par une analyse plus fine (*cf.*. analyse d'exceptions)

(ou par un catch-all au toplevel)

Correction forte

TH : Auto-réduction *La réduction préserve le typage*

TH : Progrès *Les programmes bien typés qui ne sont pas des valeurs peuvent être réduits ou bien sont des captures de continuations qui s'échappent.*

- Erreur dynamique possible
- Évitable par une analyse plus fine (*cf.*. analyse d'exceptions)
- Preuve classique et modulaire

analogue au typage des références et des exceptions

Expressivité

- Les exceptions sont encodables
  - La présence de prompt multiples et abstraits est cruciale

Expressivité

- Les exceptions sont encodables
- Callcc est encodable

Un programmme a est traduit en

let p = prompt () **in**

let callcc f = **cupto** p **as** k **in** set p **in** k (f k) **in**

let throw k v = **cupto** p **as** trash **in** set p **in** k v **in**

set p **in** a

non global

▷ Invariant : il y a toujours un unique prompt p en tête

Expressivité

- Les exceptions sont encodables
- Callcc est encodable
- Les références sont encodables

Compositionalité

- Les codages peuvent être composés.

$$SML \xrightarrow{\text{Références} \circ \text{Callcc} \circ \text{Exception}} (CoreML{+}cupto)$$

Compositionalité

- Les codages peuvent être composés.

$$SML \xrightarrow{\text{Références} \circ \text{Callcc} \circ \text{Exception}} (CoreML{+}cupto)$$

- L'ordre est important
  - Mémoire partagée : Références ○ Callcc
  - Mémoire séparée (fork en Unix) : Callcc ○ Références

Compositionalité

- Les codages peuvent être composés.

$$SML \xrightarrow{\text{Références} \circ \text{Callcc} \circ \text{Exception}} (CoreML{+}cupto)$$

- L’ordre est important
  - Mémoire partagée : Références ○ Callcc
  - Mémoire séparée (fork en Unix) : Callcc ○ Références
- Pas de garanti... C’est au programmeur de s’assurer de la non/bonne interférence.

Codage inverse

On peut coder le cupto avec le callcc plus une référence !

[*cf.*. Filinski 94. *Representing Monads*]
Module Ocaml (callcc fourni en version bytecodée)

```
exception Prompt      (* levée lorsqu'un prompt s'échappe *)
type 'a prompt
val new_prompt : unit -> 'a prompt
val set :   'a  prompt -> (unit -> 'a) -> 'a
val cupto : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
```

Typable —sans “magic_obj”— à condition d'avoir un type concret dynamiquement extensible (exn ou types variantes).

Peut-on se priver des références ?

- Peut-on coder cupto dans [core ML + callcc] ?
- Probablement non : cupto permet de coder les références, mais pas callcc : les programmes bien typés dans [core ML + callcc – rec] terminent.
- Attention ! en Scheme, le let rec sur les valeurs a une sémantique d'affectation qui permet de coder les références avec callcc (on peut arrêter le calcul avec callcc au milieu de la fabrication du point fixe et capturer la sémantique de l'affectation)

Zoologie du contrôle : plein d'espèces

- *Felleisen* a introduit toute une famille d'opérateurs de contrôle $\mathcal{F}$ codables avec le cupto (un seul prompt) :

```
let  F f = cupto p as x in set p in f  x
let  F_plus f = cupto p as x in f x                 (* Généralisation *)
```

- *Danvy* et *Filinski* ont introduit les opérateurs Shift / Reset qui sont en fait égal à

```
let  F_minus f =
  cupto p as y in  let  x u = set p in   y u in set p in f  x
```

Dans F_minus la continuation est transformée pour replacer systématiquement un prompt.

Encore d'autres espèces

- *Queinnec* et *Serpette* ont introduit les Splitters et call/pc (call with partial continuation)

  Prompts multiples, typage statique (proche du cupto) Mais une sémantique dynamique très différente (les prompts sont effacés lors de la capture).
- *Hieb* et Dybvig ont introduit le Spawn conçu pour une utilisation dans un contexte distribué. Plus proche des splitters que du cupto. Codable également...

```
let spawn F =
  let p = new_prompt() in
  let c f = cupto p as y in f y in
  set p in F c
```

Pratique du `cupto`

Descente le long d’un arbre avec des pauses interactives

(emprumté à Felleisen)

Pratique du `cupto`

Descente le long d’un arbre avec des pauses interactives

- Énoncé

```
datatype tree =
  Null | Cell of int | Pair of tree * tree
```

```
datatype answer = None | Some of int
```

Écrire deux fonctions

```
get_first : tree -> answer
```

```
get_next : unit -> answer
```

qui permettent de lire les feuilles une à une dans l’ordre de gauche à droite.

Pratique du `cupto`

Descente le long d'un arbre avec des pauses interactives

- Exemple

```
let tree = Pair (Pair (Cell 1, Null), Pair
(Cell 2, Cell 3))
# get_next ();;
None : answer
# get_first tree;;
Some 1 : answer
# get_next ();;
Some 2 : answer # get_next ();;
Some 3 : answer
# get_next ();;
None : answer
```

Pratique du `cupto`

Descente le long d'un arbre avec des pauses interactives

- Solution : écrire `start` et `suspend` pour lancer le toplevel et reprendre un toplevel en cours.

```
let walk = function
   Null -> None
 | Cell i -> suspend (Some i)
 | Pair (t1,t2) -> (walk t1; walk t2)

fun get_first t = start (fun () -> walk t)

fun get_next () = start (fnn () -> !resume

None)
```

Pratique du `cupto`

Descente le long d’un arbre avec des pauses interactives

- Solution : écrire `start` et `suspend`
- Solution avec cupto

```
let toplevel = new_prompt() : answer prompt
let resume = ref (fun (x :answer) -> x)
let start f = set toplevel f
let suspend v =
 cupto toplevel (fun k -> (resume := k); v)
```

Pratique du `cupto`

Descente le long d'un arbre avec des pauses interactives

- Solution : écrire `start` et `suspend`
- Solution avec callcc

```
let exit = ref (fun (x :answer) -> x)
let resume = ref (fun (x :answer) -> x)
let start f =
  callcc (fun toplevel ->
    (exit := throw toplevel; f();
    !exit None))
let suspend v =
  callcc (fun rest ->
    (resume := (fun x -> throw rest x);
    !exit v))
```

Pratique du `cupto`

Descente le long d’un arbre avec des pauses interactives

- Solution : écrire `start` et `suspend`
- Solution avec callcc

```
let exit = ref (fun (x :answer) -> x)
let resume = ref (fun (x :answer) -> x)
let start f =
  callcc (fun toplevel ->
    (exit := throw toplevel; f();
    !exit None))
let suspend v =
  callcc (fun rest ->
    (resume := (fun x -> throw rest x);
    !exit v))
```

Des exemples intéressants ?

- Les coroutines
    - Application pratique, utile, et intéressante du cupto (la seule ?), mais...
    - Les processus sont une meilleure solution.
- Évaluation forte (appliquée à l'évaluation partielle)
    - application intéressante mais encore théorique.

      Voir [Danvy. *Travaux multiples*]
- Le backtracking

  Les exemples ne sont pas très convainquant. Un codage directe est plus flexible et aussi efficace.

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas globaux avec un store

$$\begin{array}{lcll} S[v], E[a] & \longrightarrow & S[v], E[a'] & (\text{si } a \longrightarrow a') \\ \\ S[v], E[] & \longrightarrow & S[v], E[v] & \\ S[v], E[write\, w] & \longrightarrow & S[w], E[()] & \end{array}$$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d'arrêter l'évaluation de l'expression, d'envoyer un message à la mémoire et de relancer le calcul :

$$\begin{array}{lcl} S[v], E[read] & \longrightarrow & S[v]\ (Read(E)) \\ S[v], E[write\, w] & \longrightarrow & S[v]\ (Write(w, E)) \end{array}$$

La mémoire $S[v]$ (où v est la valeur mémorisée) interprète les messages $Read$ et $Write$ et relance le calcul

$$\begin{array}{lcl} S[v]\ (Read(E)) & \longrightarrow & S[v], E[v] \\ S[v]\ (Write(w, E)) & \longrightarrow & S[w], E[()] \end{array}$$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d'arrêter l'évaluation de l'expression, d'envoyer un message à la mémoire et de relance

`cupto , as` E `in` $Read(E)$

$$
\begin{array}{rcl}
S[v], E[read] & \longrightarrow & S[v]\ (Read(E)) \\
S[v], E[write\, w] & \longrightarrow & S[v]\ (Write(w, E))
\end{array}
$$

La mémoire $S[v]$ (où v est la valeur mémorisée) interprète les messages $Read$ et $Write$ et relance le calcul

$$
\begin{array}{rcl}
S[v]\ (Read(E)) & \longrightarrow & S[v], E[v] \\
S[v]\ (Write(w, E)) & \longrightarrow & S[w], E[()]
\end{array}
$$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d'arrêter l'évaluation de l'expression, d'envoyer un message à la mémoire et de relancer le calcul :

$\texttt{cupto}\ ,\ \texttt{as}\ E\ \texttt{in}\ Write(w, E)$

$$S[v], E[write\, w] \quad \longrightarrow \quad S[v]\ (Write(w, E))$$

La mémoire $S[v]$ (où v est la valeur mémorisée) interprète les messages $Read$ et $Write$ et relance le calcul

$$\begin{array}{lcl} S[v]\ (Read(E)) & \longrightarrow & S[v], E[v] \\ S[v]\ (Write(w, E)) & \longrightarrow & S[w], E[()] \end{array}$$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d'arrêter l'évaluation de l'expression, d'envoyer un message à la mémoire et de relancer le calcul :

$$\begin{array}{lcl} S[v], E[read] & \longrightarrow & S[v]\ (Read(E)) \\ S[v], E[write\, w] & \longrightarrow & S[v]\ (Write(w, E)) \end{array}$$

La mémoire $S[v]$ (où v est la valeur mémorisée) interprète les messages $Read$ et $Write$ et r

$S[v]$ `(set , in` $E[v]$`)`

$$\begin{array}{lcl} S[v]\ (Read(E)) & \longrightarrow & S[v], E[v] \\ S[v]\ (Write(w, E)) & \longrightarrow & S[w], E[()] \end{array}$$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d’arrêter l’évaluation, d’envoyer un message à la mémoire et relancer le calcul

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d'arrêter l'évaluation, d'envoyer un message à la mémoire et relancer le calcul
- **Multiple prompts** : La virgule (pour une cellule mémoire) est replacée par des prompts (un pour chaque cellule mémoire)

$$\# S_3 \; p_3 \; S_2 \; p_2 \; S_1 \; p_1 \; E[a]$$

Le prompt $\#$ permet d'allouer une cellule mémoire en tête.

$$\# E[ref \; v \; (\lambda r.\, a)] \longrightarrow \# S[v] \; p \; E[a \; p]$$

La seule difficulté est d'écrire le code de ref de telle façon que l'ordre d'évaluation soit correct : E doit être correctement restauré avant de poursuivre l'évaluation de $a \; p$

Codage des références

- **Principe** : copier la sémantique à réduction à petit pas avec un store
- **Une seule référence** : la virgule est un prompt qui permet d’arrêter l’évaluation, d’envoyer un message à la mémoire et relancer le calcul
- **Multiple prompts**
- **Solution alternative** : une seule cellule + un type somme extensible.

Références — Le Code

```
type 'a action =
    Write of ('a * ( unit -> 'a action))
  | Read of ('a -> 'a action)
type 'a ref = 'a action prompt;;

let (!)  r = cupto r (fun f -> Read f);;
let (:=) r x = cupto r (fun f -> Write (x,f));;

let rec store r ( x:'a) = function
    Write (y,f ) -> store p y (set r f )
  | Read f -> store r x (set r (fun () -> f x))
```

Le Code (suite)

```
let ref toplevel ( x : 'a) ( a : 'a ref −> 'b) =
  let return x = cupto toplevel (fun _ −> x) in
  let a p () = a p in
   ( cupto toplevel
     ( fun z −>
       ( set toplevel
         ( fun () −>
              let p = new_prompt() in
                store p x ( set p ( fun () −> return (z (a p))))
         )))) ();;
```

Correction du codage

- Ce sont des preuves difficiles, d'équivalence observationnelles.
- À l'époque, que des outils sémantiques.
  - Une preuve informelle par transformation successive de machine asbtraites.
  - (une dizaine d'étapes toutes simples, mais difficiles à prouver)
- Depuis, outils syntaxiques pour la sémantique (*cf.*. HOOTS en général + travaux d'Andy Pitts)

Codage des exceptions

- Pas de difficulté.
  - Attention ! Les exceptions ne sont pas codables avec callcc : elle ne garantissent pas la terminaison.

Codage des exceptions

- Pas de difficulté.
- Avantage : les patchers — variation sur les exceptions.

```
type 'a warning = 'a prompt;;
let new_warning = new_prompt;;
let warning p s =
  patch p (fun (v, l) -> (v, s : :l));;
let handle p h a =
  let (v,l) = set p (fun () -> a(), [])
  in h v l;;
```

Codage des exceptions

- Pas de difficulté.
- Avantage : les patchers — variation sur les exceptions.
- Exemple

```
let arith = new_warning()
let handler v = function
    [] -> v | [s] -> print_string (s); v
  | _ -> print_string "Warning"; v

let (/) x y =
  if y = 0
    then warning arith "division"; max_int)
  else x / y

handle arith handler (fun () -> ... )
```

Compilation du `cupto`

- Codage avec le callcc toujours possible et pas innefficace.
- Implémentation directe. La comparaison est difficile.
  - runtime sans pile : les deux sont bon marché
  - runtime traditionnel, on peut prévoir :
    - Compilation naïve : de meilleures performances pour le cupto,
    - Compilation intelligente : des performances similaires.

Résumé

- Sur le plan théorique
  - Résultat de correction forte pour le contrôle.
  - Une construction plus expressive.
- Sur le plan pratique...
  - Programmez avec cupto ! (c'est bien plus simple)
- Implémentation de `cupto` est au moins aussi efficace
- L'approche est flexible.
  - prompts, multiples, introduits par l'utilisateur.
  - Généralisations possibles à une structure hiérarchique de prompt [*cf.*. Fellinski et Danvy]

Les deux commandements

- Ne jamais utiliser les opérateurs de contrôle
  - Ils sont difficiles à manipuler.
  - Il n’y a pas toujours un gain évident (et souvent de meilleurs solutions sans)
- Si vous péchez, alors péchez avec cupto
  - manipulation du contrôle plus simple et plus sure.

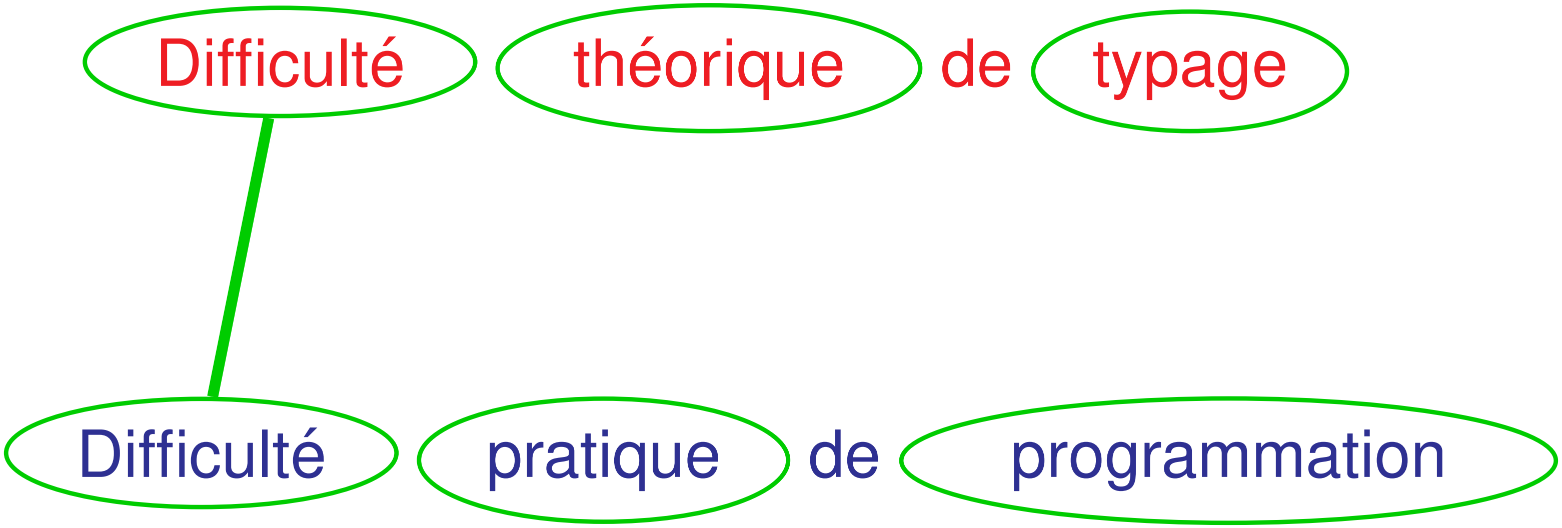
    (évite un encodage par paires de continuations)

La leçon

Difficulté théorique de typage

Difficulté pratique de programmation

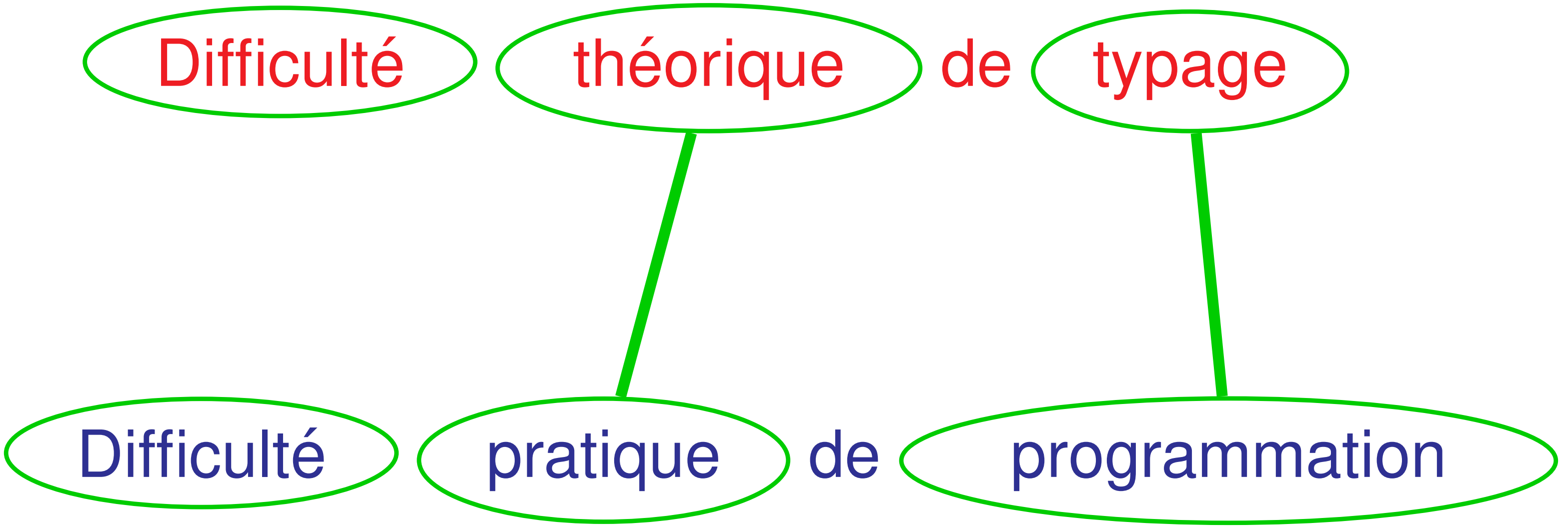
La leçon



Difficulté théorique de typage

Difficulté pratique de programmation

La leçon



Difficulté théorique de typage

Difficulté pratique de programmation

La leçon

Difficulté théorique de typage

Difficulté pratique de programmation

Nous avons

- **amélioré** les deux aspects simultanément, et
- **réduit** mais **pas éliminé** la difficulté

Bibliography du contrôle

- Autres travaux (ancients) : le μ-calcul de Parigot.

Bibliography du contrôle

- Autres travaux (ancients) : le μ-calcul de Parigot.
- Travaux plus recents (depuis 1996) sur le contrôle :
  - Andrzej Filinski *Representing Layered Monads*. POPL 99. http ://www.brics.dk/˜andrzej/papers/
  - Olivier Danvy and Zhe Yang. *An Operational Investigation of the CPS Hierarchy* ESOP 99. http ://www.brics.dk/˜danvy/

Bibliography du contrôle

- Autres travaux (ancients) : le μ-calcul de Parigot.
- Travaux plus recents (depuis 1996) sur le contrôle :
  - Andrzej Filinski *Representing Layered Monads*. POPL 99. http ://www.brics.dk/˜andrzej/papers/
  - Olivier Danvy and Zhe Yang. *An Operational Investigation of the CPS Hierarchy* ESOP 99. http ://www.brics.dk/˜danvy/
- Travail récent de Curien-Herbelin (et autres trauvaux similaires) sur les jeux et la symétrie dans le calcul entre contexte/futur et expressions/passé.