Algorithmes, Types, Preuves

Martin Strecker

INU Champollion

Année 2021/2022

Plan

Plan

Théorie des langages - pourquoi ?

Permet de répondre aux questions ...

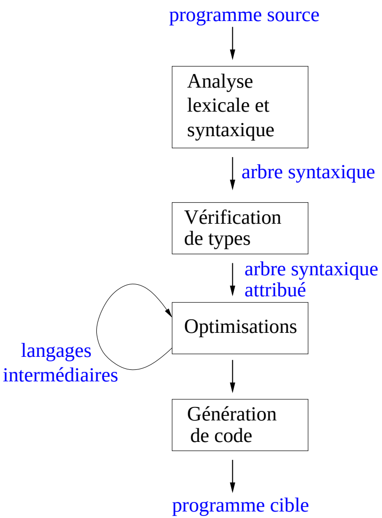
... d'un utilisateur d'un langage de programmation :

- pourquoi un tel problème de compilation / exécution ?
- comment rendre un programme plus sûr ?
- comment rendre un programme plus efficace ?

... d'un développeur d'un langage de programmation :

- quel langage pour quels besoins d'une clientèle (DSL : *domain specific languages*)
- quels mécansimes pour des programmes plus fiables ?
- quelles optimisations pour des programmes plus efficaces ?

Processus de compilation



Analyse syntaxique

Prérequis pour tout traitement ultérieur : correction syntaxique.
Exemples :

- $(3 + x) - (/ * 8)$
  - syntaxiquement mal formé
- $(3 + x) - (y * 8)$
  - syntaxiquement bien formé
  - signification : si $x = 20$ et $y = 2$, alors le résultat est 7
  - Comment le définir mieux ?

En langue naturelle, la situation est moins nette.
Exemple : Time flies like arrows

Typage - c'est quoi ?

On associe un type à des unités élémentaires :

- des constantes :
  `3` est de type `int`, `2.5` est de type `float`
- à des variables :
  `int n; float x;`
- à des fonctions
  `int fac (int n) { ... }`

... et peut ainsi déterminer le type d'expressions complexes :
`n + fac(3)` a le type `int`

Typage - pourquoi ?

Pour des raisons de

- allocation de mémoire : le type d'une valeur détermine sa taille en mémoire
  Ex. : valeur de type `int` : 2 octets, de type `float` : 4 octets (varie selon l'architecture)
- prévention de fautes :
  - involontaires : addition d'un `int` et un pointeur
  - volontaires (brèches de sécurité)
- documentation
  - pour le programmeur
  - pour le compilateur (génération de code plus efficace)

Typage - comment ?

Différentes combinaisons de ...

Dynamique vs. statique

- *Dynamique :* le type d'une expression n'est connu que lors de l'exécution
- *Statique :* le type est déjà connu au temps de la compilation

Fort vs. faible

- *Fort :* Une discipline stricte est imposée
- *Faible :* Des déviations de la discipline de typage sont tolérées

Unique vs. multiple

- *Unique :* une expression a un seul type
- *Multiple :* une expression peut avoir plusieurs types (possiblement hiérarchisés)

Typage - Lisp

Lisp (LISt Processor)

- langage fonctionnel
- développé ≈ 1960

Typage dynamique ; typage multiple
typage fort (erreurs de typage détectés pendant exécution) :

```
(defun foo (a)
  (cond ((<= a 3) (+ a 1))
        (t (+  (car a) 1))))
```

- Appel `(foo 3)` donne `4`
- Appel `(foo 4)` : erreur (`car` : tête de liste)
- Appel `(foo '(2 3))` donne `3`

Typage - Caml (1)

ML/Caml

- Famille de langages fonctionnels
- développés ≈ 1980-1990
- *Typage unique :* chaque terme de Caml a un seul type (plus précisément : un seul type “principal” (généricité !))

```
# 2 + 3;;
- : int = 5
# 2.0 +. 3.5 ;;
- : float = 5.5
# 2 + 3.5 ;;    pas de conversions automatiques!
Characters 4-7:
  2 + 3.5 ;;
      ^^^
This expression has type float
  but is here used with type int
# 2 + int_of_float 3.5 ;;
- : int = 5
```

Typage - Caml (2)

- *Typage statique :*
  erreurs de typage détectées avant début de l'évaluation

```
# 3 / 0;;
Exception: Division_by_zero.
# (3 / 0) + [2] ;;
Characters 10-13:
  (3 / 0) + [2] ;;
            ^^^
This expression has type 'a list
  but is here used with type int
```

- *Typage fort*, mais ...
- pas de déclarations explicites : *inférence de types*

Préservation de typage :

Lors de l'évaluation, une expression "garde son type"

Conséquence : pas d'erreur de type lors de l'exécution

Typage - C (1)

C

- Langage impératif, développé $\approx$ 1970
- *Typage statique* (pendant la compilation)
- *Types multiples* :
  - Une expression peut adopter des types différents, selon contexte
  - Le compilateur insère des *conversions / casts*
    $\leadsto$ résultat souvent difficile à prédire

```
printf("%d", (2+4)/5); (* résultat: 1 *)
printf("%f", (2+4)/5); (* parfois: -0.045894 *)
printf("%f", (2.+4)/5); (* résultat: 1.2000 *)
printf("%f", (2+4)/5.); (* résultat: 1.2000 *)
```

Typage - C (2)

Langage au typage *faible* et *bizarre*

- pas de type booléen
  Quelle est la valeur imprimée par :

```
x = 0.5;
if (x = 2.5) printf ("true\n");
else  printf ("false\n");
```

Typage - C (3)

- confusion entre tableaux et types de pointeur
- conversions arbitraires entre caractères, entiers et pointeurs

```
int n;
int * p;
p = (int *) malloc (sizeof(int) * 2);
p[0] = 12345;
p[1] = 67899;
n = (int) p;
n = n + 4;
p = (int *) n;
printf ("%c\n", (char)*p);
```

Valeur imprimée : `;`

Java

- Langage orienté objet (classes, interfaces)
- développé $\approx$ 2000
- Typage *statique fort*
  ($\rightsquigarrow$ pas d'erreur de type lors de l'exécution)
- Sous-typage
  - classes $\leftrightarrow$ classes
  - interfaces $\leftrightarrow$ interfaces

  ... et réalisation
  - classes $\leftrightarrow$ interfaces

  $\rightsquigarrow$ typage multiple (assez complexe ...)

$\Longrightarrow$ Syntaxiquement pareil à C, Java a un typage considérablement plus “sûr” que C $\Longleftarrow$

Plan

Définitions de valeurs et fonctions (1)

En Caml, une définition associe une valeur à un nom.

```
# let x = 3 ;;
val x : int = 3
```

Le nom peut être utilisé plus tard :

```
# let y = x + 2 ;;
val y : int = 5
```

La valeur peut être une fonction (valeur *d'ordre supérieur*) :

```
# let m2 = fun a -> 2 * a;;
val m2 : int -> int = <fun>
```

Définitions de valeurs et fonctions (2)

Il y a une multitude de manières de définir la fonction f à deux paramètres a, b qui calcule $2 * a + b + 3$:

```
# let f = fun a -> fun b -> 2 * a + b + 3 ;;
val f : int -> int -> int = <fun>
```

```
# let f = fun a b -> 2 * a + b + 3 ;;
# let f a = fun b -> 2 * a + b + 3 ;;
# let f a = function b -> 2 * a + b + 3 ;;
# let f a b = 2 * a + b + 3 ;;
```

... et encore d'autres.

Nous n'adoptons que la première

Définitions de valeurs et fonctions (3)

Le nom à définir (*definiendum*) ne peut pas apparaître dans le terme qui le définit (*definiens*) :

```
# let sum = fun n -> if (n = 0) then 0
                     else n + sum (n - 1) ;;
Unbound value sum
```

. . . sauf dans le cas d'une définition *récursive* :

```
# let rec sum = fun n -> if (n = 0) then 0
                         else n + sum (n - 1) ;;
val sum : int -> int = <fun>
```

Définitions de valeurs et fonctions (4)

Le combinateur `fix`

```
# let rec fix f = f (fix f) ;;
val fix : ('a -> 'a) -> 'a = <fun>
```

déplie son argument indéfiniement :

```
fix f = f (fix f) = f (f (fix f)) ...
```

plus de détails en TD et TP !

Définitions de valeurs et fonctions (5)

`fix` permet de convertir toute fonction récursive en fonction (syntaxiquement !) non-récursive :

```
# let sum2 = fix (fun sum -> fun n ->
      if (n = 0) then 0 else n + sum (n - 1)) ;;
```

équivalent à :

```
# let rec sum = fun n ->
      if (n = 0) then 0 else n + sum (n - 1) ;;
```

Conclusion :
On peut supposer que toute définition a la forme

```
# let d = e
```

où `e` ne contient pas `d` (mais possiblement `fix`)

Vérification vs. Inférence de types (1)

Pour déterminer si un programme est bien typé, on peut

- vérifier si les types du programme sont cohérents
  Préalable : Existence d'annotations de type
- inférer des annotations cohérentes
  (pourvu qu'elles existent)

à préciser :

- *annotation* de type
- *cohérence* de types

Vérification vs. Inférence de types (2)

Caml

permet la *vérification* de programmes annotés :

```
# let f = fun (x : int) -> x + 2 ;;
val f : int -> int = <fun>
# let g = fun (x : bool) -> x + 2 ;;
This expression has type bool
but is here used with type int
```

permet l'*inférence* de types

```
# let f = fun x -> x + 2 ;;
val f : int -> int = <fun>
```

Vérif. de types : Fragment fonctionnel pur (1)

Types *T* du langage de programmation :

- *Types de base :* `bool`, `int`, `...`
- *Types fonctionnels* de la forme `T -> T'`, où `T`, `T'` sont des types

Conventions syntaxiques : `->` associe à droite :
`int -> int -> int` est équivalent à
`int -> (int -> int)`, non pas à
`(int -> int) -> int`

Vérif. de types : Fragment fonctionnel pur (2)

Expressions e du langage de programmation :

- *Constantes :* `2`, `true`, `...`
- *Variables :* `x`, `f`, `...`
- *Abstractions* de la forme `fun` x : T `->` e, où
  x est une variable, T est un type
  e est une expression
- *Applications* de la forme $(e\ e')$, où
  e, e' sont des expressions

Vérif. de types : Fragment fonctionnel pur (3)

Sont réductibles à ce format :

- Abstractions à plusieurs paramètres :

  ```
  # let plus = fun(a : int) (b : int) -> a + b ;;
  ```

  abbrévie :

  ```
  # let plus = fun(a:int)-> fun(b:int)-> a + b ;;
  ```

- Applications à plusieurs arguments :

  ```
  # (plus 2 3) ;;
  ```

  abbrévie :

  ```
  # ((plus 2) 3) ;;
  ```

  Donc : l'application associe à gauche :
  $f\ e_1\ e_2$ est équivalent à $((f\ e_1)\ e_2)$, non pas à $(f\ (e_1\ e_2))$

- Opérateurs infixe / mixfixe : Écrire en préfixe !

Vérif. de types : Fragment fonctionnel pur (4)

Un environnement associe un type à une variable.
Nous représentons l'environnement par une liste d'association : $[(v_1, T_1); \ldots; (v_n, T_n)]$.
L'environnement est construit / modifié

- lors d'une définition :

```
# let f = fun (x: int) -> x + 2 ;;
val f : int -> int = <fun>
# let a = 3 ;;
val a : int = 3
```

  Environnement : `[(a, int); (f, int -> int)]`
- pendant la vérification de types *(... à voir)*

Vérif. de types : Fragment fonctionnel pur (4)

Règles de typage de la forme $Env \vdash e : T$

Constantes : Toute constante a son type “naturel” :

$$\frac{n \in \mathbb{Z}}{Env \vdash n : int} \qquad \frac{b \in \{true, false\}}{Env \vdash b : bool}$$

Variables :

$$\frac{tp(x, Env) = T}{Env \vdash x : T}$$

où $tp(x, Env) = T$ si $(x,\ T)$ est la décl. la plus à gauche dans Env

Vérif. de types : Fragment fonctionnel pur (5)

Abstraction :

$$\frac{(x, A) :: \mathit{Env} \vdash e \ : \ B}{\mathit{Env} \vdash \texttt{fun}(x : A) \text{ -> } e \ : \ A\text{->}B}$$

Application :

$$\frac{\mathit{Env} \vdash f \ : \ A\text{->}B \qquad \mathit{Env} \vdash a \ : \ A}{\mathit{Env} \vdash (f \ a) \ : \ B}$$

Vérif. de types : Fragment fonctionnel pur (6)

Exercices : Vérifier les types des expressions suivantes dans l'environnement `Env = [(a, int); (f, int -> int)]`

- `(f a)`
- `(f 3)`
- `(f true)`
- `fun (x : int) -> (f x)`
- `fun (x : int) -> (f a)`
- `fun (x : int) -> f`
- `(fun (x : int) -> f) a`
- `fun (x : int) -> fun (a : bool) -> (f x)`
- `fun (x : int) -> fun (a : bool) -> (f a)`

Vérif. de types : Paires (1)

Première extension du langage :

```
# (1, 2) ;;
- : int * int = (1, 2)
# (1, true) ;;
- : int * bool = (1, true)
```

Types *T* du langage étendu :

- *Types de base :* `bool`, `int`, `...` (comme avant)
- *Types fonctionnels :* `T -> T'` (comme avant)
- *Types de paires :* `T * T'`

Vérif. de types : Paires (2)

Expressions e du langage étendu :

- Constantes, variables, abstractions, applications : comme avant
- *Paires* de la forme (e, e'), où e et e' sont des expressions

Développer règle de typage

Pair

$$\frac{\ldots}{Env \vdash (e, e') \ : \ T * T'}$$

Vérifier le type de : `(false, fun (x: int) -> x * x)`

Curryfication (1)

On peut écrire une fonction à *n* paramètres :

- avec un seul paramètre de type *n*-uplet :

```
# let fp = fun (a, b) -> a + b + 2 ;;
val fp : int * int -> int = <fun>
```

- *curryfié*, en itérant *n* déclarations `fun` :

```
# let fc = fun a -> fun b -> a + b + 2 ;;
val fc : int -> int -> int = <fun>
```

Curryfication (2)

- La fonction non-curryfiée s'applique à un seul argument (de type *n*-uplet) :

```
# fp (2, 3) ;;
- : int = 7
# fp 2 3 ;;
This function is applied to too many arguments
```

- La fonction curryfiée s'applique à *n* arguments :

```
# fc 2 3 ;;
- : int = 7
# fc (2, 3) ;;
This expression has type int * int
but is here used with type int
```

Curryfication (3)

- La fonction curryfiée permet une application partielle :

```
# List.map (fc 2) [1; 2; 3] ;;
- : int list = [5; 6; 7]
```

- *Conclusion :* Préférer la version curryfiée !

Interlude : Notes historiques (1)

Alan Turing (1912-1954)

John von Neumann (1903-1957)

Interlude : Notes historiques (2)

- Alan Turing : machine de Turing
  On computable numbers (1936)
  Premier modèle *mathématique* d'un langage de programmation impératif
- John von Neumann
  Architecture d'un ordinateur mélangeant données et contrôle

à partir de 1941 : réalisation des premiers ordinateurs ayant la puissance d'une machine de Turing (*"Turing-complet"*)

Interlude : Notes historiques (3)

Alonzo Church (1903-1995)

Haskell Curry (1900-1982)

Interlude : Notes historiques (4)

Précurseurs des langages de programmation fonctionnels :

- Alonzo Church : lambda-calcul
  An unsolvable problem of elementary number theory (1936)
  $\lambda x.(f\ x)$ s'écrit `fun x -> (f x)`
- Haskell Curry : *Logique combinatoire (thèse, 1930)*
  version du lambda-calcul sans variables
- *Théorème :* La machine de Turing et le lambda-calcul ont une puissance de calcul équivalente

Plan

Rappel : Types inductifs (1)

... aussi appelés *types de données* ou *types d'utilisateur*

Exemple : Type des arbres binaires

```
type bintree =
    Leaf of int
  | Node of int * bintree * bintree
```

Terminologie :

- `Leaf` et `Node` sont les *constructeurs* de `bintree`
- `Leaf` est un constructeur *de base*
- `Node` est un constructeur à deux arguments *inductifs*

Instance d'un `bintree` :

```
# Node (1, Leaf 2, Leaf 3) ;;
- : bintree = Node (1, Leaf 2, Leaf 3)
```

Rappel : Types inductifs (2)

Définition de fonctions par *récursion structurelle* :

```
# let rec sum_bintree = fun bt ->
  match bt with
      Leaf n -> n
    | Node (n, bt1, bt2) ->
      n + sum_bintree bt1 + sum_bintree bt2 ;;
  val sum_bintree : bintree -> int = <fun>
```

- une *clause* par constructeur
- (normalement) un appel *récursif* par argument *inductif*

Définir la fonction `bintree2list` qui construit la liste des éléments d'un `bintree` (ordre préfixe).

Rappel : Types inductifs (3)

Variante syntaxique : En Caml,

```
fun x -> match x with ...
```

peut être abbrévié par

```
function ...
```

Définition alternative de `sum_bintree` :

```
# let rec sum_bintree = function
    Leaf n -> n
  | Node (n, bt1, bt2) ->
      n + sum_bintree bt1 + sum_bintree bt2 ;;
```

Vérif. de types : Types inductifs (1)

Deuxième extension du langage.

Types T du langage étendu :

- Types de base, types de fonctions et de paires : comme avant
- *Types inductifs*, définis par le schéma

```
type T =
     C_1 of T_1
   | ...
   | C_n of T_n
```

Vérif. de types : Types inductifs (2)

Expressions *e* du langage étendu :

- Constantes, variables, ... (comme avant)
- *Constructeurs* `C (e_1, ... e_n)`
- *Schémas de filtrage* (un schéma par type inductif)

```
match e with
    C_1(x_1_1 .. x_1_n1) -> e_1
  | ...
  | C_n(x_n_1 .. x_n_nn) -> e_n
```

Vérif. de types : Types inductifs (3)

Comment traduire :

- des motifs imbriqués ?
- des “jokers” ?

Exemple :

```
match e with
    Leaf n -> e1
  | Node(n1, Node(n2, Leaf l1, Leaf l2), n3) -> e2
  | _ -> e3
```

Comment représenter `if` e `then` e_t `else` e_e
avec le schéma `match ... with` ?

Vérif. de types : Types inductifs (4)

Chaque définition de type

```
type T =
     C_1 of T_1
   | ...
   | C_n of T_n
```

augmente l'environnement actuel avec les déclarations
`(C_1, T_1 -> T), ... (C_n, T_n -> T)`
Conditions de bonne formation de la définition de `T` ?
Règle de typage pour constructeurs :
$\rightsquigarrow$ se réduit à la règle de typage pour variables

Vérif. de types : Types inductifs (5)

Typer les expressions :

- `Leaf 3`
- `Leaf true`
- `fun (x: int) -> fun (y : int) -> Node(x + y, Leaf x, Leaf y)`

Spécificités de Caml :

- Distinction lexicale entre constructeurs (en majuscule) et variables (en minuscule)
- Constructeurs fonctionnels : toujours avec argument : `List.map Leaf [1; 2; 3]` est rejeté.
  comment le réécrire ?

Vérif. de types : Types inductifs (6)

Soit le type des `bintree` “intérieurs” :

```
type ibintree =
    ILeaf
  | INode of int * ibintree * ibintree
```

Typer les expressions :

- `ILeaf true`
- `INode(5, ILeaf, ILeaf)`

Vérif. de types : Types inductifs (7)

Typage du filtrage (fragments) :

$$\frac{\begin{array}{c} Env \vdash e \ : \ T \\ [(x_1, T_1), \ldots (x_n, T_n)]@Env \vdash e' \ : \ T' \end{array}}{Env \vdash \text{match } e \text{ with } \ \ldots | C(x_1 \ldots x_n) -> e' \ : \ T'}$$

En plus, assurer les conditions :

- T type inductif avec constructeur `C of T_1 *...T_n`
- pour tous les constructeurs, on a le même type résultat T'
- filtrage exhaustif (tous les constructeurs sont traités)

Vérif. de types : Types inductifs (8)

Vérifier

```
match (Node(3, Leaf 1, Leaf 2)) with
  Leaf x -> x + 1
| Node (x, l1, l2) -> x + 2
```

Vérifier la fonction

```
let s_bt = fix (
   fun (sum_bintree: bintree -> int) ->
    fun (bt: bintree) ->
    match bt with
    Leaf n -> n
  | Node (n, bt1, bt2) ->
      n + sum_bintree bt1 + sum_bintree bt2)
```

Plan

Plan

2. Unification
   - Motivation et terminologie
   - Unification syntaxique
   - Unification modulo théories

But de l'unification

Étant donnés deux termes t_1, t_2.

Un problème d'unification a la forme $t_1 \stackrel{?}{=} t_2$.

Le but de l'unification est de trouver une substitution σ telle que t_1, t_2 deviennent égaux, c. à. d., $t_1\sigma = t_2\sigma$.

à préciser :

- notion de "terme"
- notion d'égalité

Applications de l'unification (1)

Langages de programmation : Inférence de types

Question typique :
Est-ce que la fonction `List.rev : 'a list -> 'a list` est applicable à `[2; 3] : int list` ?
se réduit au problème d'unification `'a list` $\stackrel{?}{=}$ `int list`
Réponse : Unificateur `['a` $\leftarrow$ `int]`
Donc : `List.rev [2; 3] : int list`
Ici :

- "Termes" : termes de types, par exemple : `(int * bool)`, `int list`, ...
- "Égalité" : égalité structurelle

Applications de l'unification (2)

Langages de programmation : Filtrage

Question typique : Étant donné le code :

```
match Node(3, Leaf 1, Leaf 2) with
    Leaf x -> x
  | Node (y, nd, Leaf lf) -> lf
```

Quelle valeur est renvoyée ?
se réduit aux problèmes d'unification

- `Leaf x` $\stackrel{?}{=}$ `Node(3, Leaf 1, Leaf 2)`
  $\rightsquigarrow$ échec
- `Node(y,nd,Leaf lf)` $\stackrel{?}{=}$ `Node(3,Leaf 1,Leaf 2)`
  $\rightsquigarrow$ unificateur [y $\leftarrow$ 3, nd $\leftarrow$ Leaf 1, lf $\leftarrow$ 2]

Valeur renvoyée : 2

Applications de l'unification (3)

Preuves : unifier hypothèses et conclusion

Question typique : Dans le calcul des séquents, est-ce que la conclusion est une conséquence des hypothèses ?

$$\underbrace{P(a),\ P(f\ a)}_{\text{Hypothèses}} \vdash \underbrace{P(f\ x)}_{\text{Conclusion}}$$

se réduit aux problèmes d'unification

- $P(a) \stackrel{?}{=} P(f\ x) \rightsquigarrow$ échec
- $P(f\ a) \stackrel{?}{=} P(f\ x) \rightsquigarrow$ unificateur $[x \leftarrow a]$

Voir règle de preuve `assumption` $\rightsquigarrow$ traitée plus tard

Applications de l'unification (4)

Preuves : Règle de résolution (1)

Base :

- Formules en forme normale conjonctive :
  $(A \vee B) \wedge (C \vee D) \wedge (\neg C \vee B)$
- Écriture comme ensemble de clauses : $\{\{A, B\}, \{C, D\}, \{\neg C, B\}\}$

Résolution des clauses $\{C, D\}$ et $\{\neg C, B\}$:

$$\frac{\{C, D\} \quad \{\neg C, B\}}{\{D, B\}}$$

Applications de l'unification (5)

Preuves : Règle de résolution (2)

En général :

$$\frac{\{F_1, \ldots F_n, F\} \quad \{\neg G, G_1 \ldots G_m\}}{\{F_1, \ldots F_n, G_1 \ldots G_m\}\sigma}$$

si F et G sont unifiables avec unificateur σ

Exemple :

$$\cfrac{\cfrac{\{P(a)\} \quad \{\neg P(x), Q(f\ x)\}}{\{Q(f\ a)\}}\ \sigma = [x \leftarrow a] \qquad \{\neg Q(y)\}}{\{\}}\ \sigma = [y \leftarrow (f\ a)]$$

La résolution est un mécanisme essentiel du langage Prolog

Applications de l'unification (6)

Preuves : Application d'une équation lors d'une simplification

Question typique : Étant donné la règle de réécriture

$$length(x@y) = length(x) + length(y)$$

montrer que $length(\ell_1 @ \ell_2) = length(\ell_2 @ \ell_1)$

Où peut-on appliquer la règle de réécriture ?

1. Application de la règle avec $\sigma_1 = [x \leftarrow \ell_1, y \leftarrow \ell_2]$ au sous-terme $length(\ell_1 @ \ell_2)$ donne :
   $length(\ell_1) + length(\ell_2) = length(\ell_2 @ \ell_1)$
2. Application de la règle avec $\sigma_2 = [x \leftarrow \ell_2, y \leftarrow \ell_1]$ au sous-terme $length(\ell_2 @ \ell_1)$ donne :
   $length(\ell_1) + length(\ell_2) = length(\ell_2) + length(\ell_1)$

voir la partie réécriture

Plan

2. Unification
   - Motivation et terminologie
   - Unification syntaxique
   - Unification modulo théories

Unification syntaxique (1)

On s'intéresse à savoir quelle substitution satisfait une équation *syntaxiquement*

- $X + 2 \stackrel{?}{=} 2 + X$ peut être unifié syntaxiquement : $\sigma = [X \leftarrow 2]$

sans s'intéresser à d'éventuelles significations des symboles des fonctions :

- $3 * X + 2 \stackrel{?}{=} 2 + X$ ne peut pas être unifié syntaxiquement
- ...mais il y a une solution sous l'interprétation habituelle de $+$ et $*$: $\sigma = [X \leftarrow 0]$
- $\rightsquigarrow$ voir “unification modulo théories”

On préférera désormais des symboles neutres :
$p(X, 2) \stackrel{?}{=} p(2, X)$ et $p(m(3, X), 2) \stackrel{?}{=} p(2, X)$

Unification syntaxique (2)

La substitution doit

- respecter la structure des termes
- non pas produire une égalité “superficielle”

Exemple : unifier syntaxiquement $2 + c * 5 \stackrel{?}{=} X * 5$

- Tentative d’unification avec $\sigma = [X \leftarrow 2 + c]$
- Superficiellement : "$2 + c * 5$" égale ("$2 + c$" suivi de "$*5$")
- Structurellement : $2 + (c * 5) \neq (2 + c) * 5$

$\rightsquigarrow 2 + c * 5 \stackrel{?}{=} X * 5$ n’est pas unifiable

Termes

Les termes t du langage ont la forme suivante :

$$\begin{array}{lll} t & ::= & V & \text{(variables)} \\ & | & f(t, \ldots, t) & \text{(application)} \end{array}$$

Conventions :

- Noms des *variables* en majuscules : $X, Y, Z, \ldots$
- Les fonctions sans arguments sont perçues comme des *constantes*
  et s'écrivent typiquement sans arguments : $c() = c$

Substitution (1)

Une substitution σ est une fonction qui associe un terme à chaque variable.
Notation : $\sigma = [X_1 \leftarrow t_1; \ldots; X_n \leftarrow t_n]$
avec les X_i mutuellement différents

Extension à la structure des termes
(notation postfixe : $t\sigma$ au lieu de $\sigma(t)$)
Définition par récursion structurelle :

- Variables : $V\sigma = \sigma(V)$ où :
  $\sigma(V) = t$ si $\sigma = [X_1 \leftarrow t_1; \ldots V \leftarrow t; \ldots X_n \leftarrow t_n]$
  $\sigma(V) = V$ si σ ne contient pas $V \leftarrow t$
- Application : $f(t_1, \ldots, t_n)\sigma = f((t_1\sigma), \ldots, (t_n\sigma))$

Substitution (2)

Exemple : $(f(X, g(Y)))\sigma$, où $\sigma = [X \leftarrow h(Y); Y \leftarrow 42]$

$$
\begin{aligned}
&(f(X, g(Y)))\sigma, \\
&\quad = f(X\sigma, (g(Y))\sigma) \\
&\qquad = f(X\sigma, g(Y\sigma)) \\
&\qquad\quad = f(h(Y), g(42))
\end{aligned}
$$

à noter : substitution *parallèle*, pas séquentielle :
$(f(X, g(Y)))\sigma \neq f(h(42), g(42))$

Substitution (3)

La substitution σ est plus générale que σ' s'il existe un σ_i avec $\sigma' = \sigma_i \circ \sigma$

Exemple : $\sigma = [X \leftarrow g(Y)]$ est plus générale que $\sigma' = [X \leftarrow g(5), Z \leftarrow 3]$. Ici, $\sigma_i = [Y \leftarrow 5, Z \leftarrow 3]$.

Donc : $(f(X,Z))\sigma' = f(g(5),3) = f(g(Y),Z)\sigma_i = (f(X,Z))\sigma\sigma_i$

Unificateur

Un unificateur des termes t_1, t_2 est une substitution σ telle que $t_1\sigma = t_2\sigma$.
On appelle l'unificateur le plus général (*most general unifier,* mgu) de t_1, t_2 l'unificateur qui est plus général que tout autre unificateur de t_1, t_2.
Exemple : Pour $g(X, f(Y)) \stackrel{?}{=} g(X, f(f(Z)))$

- le *mgu* est $[Y \leftarrow f(Z)]$
- Des unificateurs moins généraux sont : $[Y \leftarrow f(4); Z \leftarrow 4]$ et $[Y \leftarrow f(Z), X \leftarrow 7]$

Variables libres

L'ensemble des variables libres d'un terme t (notation : $fv(t)$) est l'ensemble des variables qui apparaissent dans t.

Exemples :

- $fv(f(X, g(Y)) = \{X, Y\}$
- $fv(f(h(X, X), c)) = \{X\}$

Écrire une fonction en Caml qui calcule *fv*

Algorithme (1)

L'algorithme simplifie des paires (E, S), où

- E est un multi-ensemble d'équations à résoudre
- S est un ensemble de solutions

Les *règles de simplification* ont la forme $(E, S) \Longrightarrow (E', S')$
Un ensemble de solutions S a la forme $\{X_1 = s_1 \dots X_n = s_n\}$ où

- tous les X_i sont différents
- aucun des X_i n'apparaît dans l'un des s_j

Idée : Étant donnés t_1, t_2 à unifier :

- L'algorithme simplifie $(\{t_1 \stackrel{?}{=} t_2\}, \{\})$ jusqu'à $(\{\}, \{X_1 = s_1 \dots X_n = s_n\})$ ou termine avec un échec
- En cas de non-échec, le *mgu* est $[X_1 \leftarrow s_1 \dots X_n \leftarrow s_n]$

Algorithme (2)

Règles de simplification :

- *Delete* : $(\{t \stackrel{?}{=} t\} \cup E, S) \Longrightarrow (E, S)$
- *Decompose* :
  $(\{f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n)\} \cup E, S) \Longrightarrow (\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\} \cup E, S)$
- *Clash* : $(\{f(s_1, \ldots, s_n) \stackrel{?}{=} g(t_1, \ldots, t_m)\} \cup E, S) \Longrightarrow$ *fail*
  si f et g sont des symboles de fonction différents
- *Eliminate* : $(\{X \stackrel{?}{=} t\} \cup E, S) \Longrightarrow (E[X \leftarrow t]; S[X \leftarrow t] \cup \{X = t\})$
  si $t \neq X$ et $X \notin fv(t)$
- *Check* : $(\{X \stackrel{?}{=} t\} \cup E, S) \Longrightarrow$ *fail*
  si $t \neq X$ et $X \in fv(t)$
- *Orient* : $(\{t \stackrel{?}{=} X\} \cup E, S) \Longrightarrow (\{X \stackrel{?}{=} t\} \cup E, S)$
  si t n'est pas une variable

Algorithme (3)

Observations :

- Les règles sont (presque) mutuellement exclusives
- Question : Quelle règle faut-il modifier pour avoir une simplification déterministe ? Comment ?
- Question : Pourquoi est-ce que ceci ne modifie pas le résultat de l'algorithme ?
- Conclure : On peut appliquer les règles dans n'importe quel ordre

Algorithme (4)

Exemple :

$$
\begin{array}{lll}
 & (\{p(X,2)\stackrel{?}{=}p(2,X)\};\{\}) & \\
\Longrightarrow & (\{X\stackrel{?}{=}2,\ \ 2\stackrel{?}{=}X\};\{\}) & \text{(Decompose)} \\
\Longrightarrow & (\{2\stackrel{?}{=}2\};\{X=2\}) & \text{(Eliminate)} \\
\Longrightarrow & (\{\};\{X=2\}) & \text{(Delete)}
\end{array}
$$

Donc : $\sigma = [X \leftarrow 2]$

Faire l'exemple $p(m(3,X),2)\stackrel{?}{=}p(2,X)$

Algorithme (5)

Exemple :

$$
\begin{array}{lll}
 & (\{f(X,Y)\stackrel{?}{=}f(Y,g(X))\};\{\}) & \\
\Longrightarrow & (\{X\stackrel{?}{=}Y,\ Y\stackrel{?}{=}g(X)\};\{\}) & \text{(Decompose)} \\
\Longrightarrow & (\{X\stackrel{?}{=}Y,\ Y\stackrel{?}{=}g(X)\};\{\}) & \text{(Delete)} \\
\Longrightarrow & (\{Y\stackrel{?}{=}g(Y)\};\{X\stackrel{?}{=}Y\}) & \text{(Eliminate)} \\
\Longrightarrow & \mathit{fail} & \text{(Check)}
\end{array}
$$

Terminaison

Argument semi-formel : Après chaque application de règle $(E, S) \Longrightarrow (E', S')$

- le nombre de variables dans E' est inférieur au nombre de variables dans E, *ou bien*
- le nombre des variables dans E et E' est égal, mais la taille des termes décroît, *ou bien*
- le nombre des vars et la taille des termes restent égaux, mais le nombre d'équations décroît

Vérifier !

Argument formel : Combinaison d'un ordre lexicographique et multi-ensemble $\rightsquigarrow$ traité plus tard.

Correction et Complétude (1)

Questions à se poser : Étant donnés deux termes t_1, t_2 :

- Correction : Est-ce que l'algorithme est correct, c.à.d. est-ce qu'il fournit un σ tel que $t_1\sigma = t_2\sigma$?
- Complétude : Est-ce que l'algorithme est complet, c.à.d. est-ce qu'il fournit un unificateur si t_1, t_2 sont unifiables ?
- Non-blocage : Est-ce que l'algorithme termine ou bien avec *fail*, ou bien avec un résultat de la forme $(\{\}, S)$?

- Prouver la propriété de non-blocage
- Démontrer : Enlever l'une des règles de simplification peut entraîner une situation de blocage.
  Exemple : Sans la règle *Delete*, il est impossible de simplifier $(\{X \stackrel{?}{=} X\}, S)$

Correction et Complétude (2)

Théorème : Pour deux termes t_1, t_2, l'algorithme d'unification est

- correct
- complet : il calcule le *mgu* de t_1, t_2

Preuve : Par induction sur la longueur de la dérivation

$$(E_0, S_0) \Longrightarrow (E_1, S_1) \Longrightarrow \ldots \Longrightarrow (E_n, S_n)$$

ou

$$(E_0, S_0) \Longrightarrow (E_1, S_1) \Longrightarrow \ldots \Longrightarrow \mathit{fail}$$

Correction et Complétude (3)

en utilisant le Lemme :

- Si $(E, S) \Longrightarrow (E', S')$, alors l'ensemble des unificateurs de (E, S) est égal à l'ensemble des unificateurs de (E', S')
- Si $(E, S) \Longrightarrow \mathit{fail}$, alors (E, S) n'a pas d'unificateurs

Compléter la preuve en

- précisant la notion d'"ensemble des unificateurs de (E, S)"
- démontrant la propriété pour chaque règle

Plan

Différentes notions d'égalité

Unification syntaxique :

- prend en compte uniquement la structure des termes

Exemple : Syntaxiquement, on ne peut pas unifier :

- $(F\ 5) = 5$ (F variable fonctionnelle !)
- $X \oplus 2 = Y \oplus 3$

Unification d'ordre supérieur :

- prend en compte la sémantique d'exécution des fonctions

Unification "modulo" :

- prend en compte la "signification" de certains opérateurs (*par exemple :* associativité et commutativité de $\oplus$)

Unification d'ordre supérieur (1)

Exemple : Trouver la fonction F telle que $(F\ 5) = 5$
Deux réponses possibles :

- *Imitation :* La fonction constante 5 :
  F `= fun y -> 5`
- *Projection :* La fonction qui renvoie son argument :
  F `= fun y -> y`

Tester le résultat de

- `(fun y -> 5) 5`
- `(fun y -> y) 5`

Unification d’ordre supérieur (2)

Observations :

- Deux solutions *indépendantes* (l’une n’est pas plus générale que l’autre)
- *Indécidabilité :*
  - Il n’est pas décidable si deux fonctions Caml ont le même comportement calculatoire
  - $\rightsquigarrow$ indécidabilité de l’unification d’ordre supérieur
- Il existe un algorithme qui *énumère* les solutions

Unification modulo ACU (1)

Exemple : trouver la solution pour $X \oplus 2 = Y \oplus 3$ si $\oplus$ est un opérateur commutatif quelconque.

- Solution : $[X \leftarrow 3; Y \leftarrow 2]$, parce que $3 \oplus 2 = 2 \oplus 3$
- Non-solution : $[X \leftarrow 1; Y \leftarrow 0]$, parce que $\oplus$ n'est pas $+$!

Unification modulo ACU (2)

Une axiomatisation est un ensemble de propriétés
But : Fixer les propriétés d'un ou plusieurs opérateurs
Théorie : ensemble de *modèles* qui satisfont une axiomatisation.
Axiomatisation d'un opérateur $\oplus$ qui est associatif et commutatif et a une unité e (ACU) :

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$$

$$X \oplus Y = Y \oplus X$$

$$X \oplus e = X$$

Exemple : Dans la théorie ACU, $e \oplus X = X$ est valide

Unification modulo ACU (3)

Les termes des problèmes d'unification seront construits à partir de variables, la constante e et l'application binaire de $\oplus$.
On n'utilise pas d'autres symboles de fonctions.
Exemple : Résoudre le problème

$$X \oplus (X \oplus Y) \stackrel{?}{=} Z \oplus (Z \oplus Z)$$

Quelques solutions possibles (où U_1, U_2, U_3 sont de nouvelles variables) :

1. $\sigma_1 = [X \leftarrow U_1, Y \leftarrow U_1, Z \leftarrow U_1]$
2. $\sigma_2 = [X \leftarrow e, Y \leftarrow U_2 \oplus (U_2 \oplus U_2), Z \leftarrow U_2]$
3. $\sigma_3 = [X \leftarrow U_3 \oplus (U_3 \oplus U_3), y \leftarrow e, Z \leftarrow (U_3 \oplus U_3)]$

Unification modulo ACU (4)

Plus systématiquement : Pour déterminer les unificateurs de

$$X \oplus (X \oplus Y) \stackrel{?}{=} Z \oplus (Z \oplus Z)$$

calculer les solutions entières positives de $2n_x + n_y = 3n_z$:
$(1, 1, 1), (0, 3, 1), (3, 0, 2)$
Tout unificateur est une *combinaison linéaire* de ces solutions.
Le *mgu* a donc la forme :

$$\begin{array}{rcl} \sigma & = & [X \leftarrow X\sigma_1 \oplus X\sigma_2 \oplus X\sigma_3, \\ & & Y \leftarrow Y\sigma_1 \oplus Y\sigma_2 \oplus Y\sigma_3, \\ & & Z \leftarrow Z\sigma_1 \oplus Z\sigma_2 \oplus Z\sigma_3] \end{array}$$

Unification : Classification

Unification syntaxique :

- Le problème d'unification est décidable
- L'algorithme produit un *mgu* unique

Unification d'ordre supérieur :

- Le problème d'unification n'est pas décidable
- L'algorithme énumère les unificateurs les plus généraux

Unification modulo ACU :

- Le problème d'unification est décidable
- L'algorithme fait appel à des algorithmes de programmation linéaire.
- Il produit un *mgu* unique

Plan

Plan

Polymorphisme (1)

Un type est dit polymorphe s'il admet de multiples *instances de types*.
Exemple : Listes polymorphes

```
# [1; 2; 3] ;;
- : int list = [1; 2; 3]
# [true; false; true] ;;
- : bool list = [true; false; true]
# [[1]; [2]; [3]] ;;
- : int list list = [[1]; [2]; [3]]
```

Les éléments doivent pourtant avoir un type *uniforme* :

```
# [1; true; [3]] ;;
  [1; true; [3]] ;;
This expression has type bool
but is here used with type int
```

Polymorphisme (2)

Une fonction polymorphe n'a pas besoin de connaître les instances de types de ses arguments :

```
# let rec long = function
      [] -> 0
    | x :: xs -> 1 + long xs;;
    val long : 'a list -> int = <fun>
```

Polymorphisme (3)

Un type polymorphe est comme une fonction sur des types :

- Un type polymorphe a un ou plusieurs paramètres de type
- ... dénotés par des variables de types
  notation en Caml : `'a, 'b, ...`
  exemple `'a list`
- Une instance d'un type polymorphe a des arguments de type, possiblement imbriqués
  exemple :
  `[[(1, true)]; [(2, false); (3, false)]]`
  a le type `(int * bool) list list`

Contrairement aux fonctions, notation postfixe :
`int list` au lieu de `list (int)`

Polymorphisme (4)

Types du langage avec polymorphisme :

T	::=	VT	(Variables de type : `'a, 'b, ...`)
	|	`int` | `...`	(types de base)
	|	(T ...T) FT	(application de fonctions de type)
	|	T `->` T	(types fonctionnels)
	|	T $\star$ T	(types de paires)

Exemples d'applications de fonctions de type :

- `int list`
- `int list list` ($\equiv$ `(int list) list`)
- `(int, bool) p2_bintree`

Bonne formation de types :

Chaque fonction de type a une arité fixe

Exemple : Arbres binaires

Définir les types polymorphes

- `p1_bintree` avec des `Leaf` et `Node` d'un seul type `'a`
- `p2_bintree` avec des `Leaf` d'un type `'a` et `Node` d'un type `'b`.

Définir sur `p2_bintree`

- une fonction `nmb_nds` qui compte tous les nœuds (internes et externes)
- une fonction `nds_extern` qui renvoie la liste des feuilles
- une fonction `nds_intern` qui renvoie la liste des nœuds internes

Exemple : Le type `option`

Le type (prédéfini) `option` :

```
type 'a option = None | Some of 'a
```

Utile pour modéliser la présence / absence d'information :

- `None` : aucune information disponible
- `Some x` : l'information `x` est disponible

Le type `option` et les listes d'association (1)

Une liste d'association associe une *clé* et une *valeur*.
Exemple : clé : nom de la personne ; *valeur :* son age

```
[("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)]
```

En cas de non-unicité de la clé : on prend la valeur la plus à gauche.

Le type `option` et les listes d'association (2)

Trouver la valeur associée à une clé :

Première solution : Utilisation de la fonction prédéfinie

```
List.assoc : 'a -> ('a * 'b) list -> 'b
```

```
# List.assoc "Anne"
  [("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)];;
- : int = 22
# List.assoc "Julie"
  [("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)];;
Exception: Not_found.
```

Traitement de l'exception :

```
try
  List.assoc "Julie"
   [("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)]
with Not_found -> 0 ;;
- : int = 0
```

Le type `option` et les listes d'association (3)

Deuxième solution : Utilisation d'un type `option`:

```
assoc_opt : 'a -> ('a * 'b) list -> 'b option
```

A définir en TP !

```
# assoc_opt "Anne"
  [("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)];;
- : int option = Some 22
# assoc_opt "Julie"
  [("Mathieu", 20); ("Anne", 22); ("Nicolas", 22)];;
- : int option = None
```

Traitement du cas indéfini :

```
match assoc_opt "Julie" ... with
| None -> 0
| Some n -> n
```

Manipulation de types polymorphes

Une substitution $\sigma = [\alpha_1 \leftarrow S_1, \ldots, \alpha_n \leftarrow S_n]$, appliquée à un type T, remplace en parallèle toute occurrence des variables de type α_i dans T par S_i.

Notation : $T\sigma$

Exemple : $(\texttt{'a list})[\texttt{'a} \leftarrow \textit{bool}] = \texttt{bool list}$

Un type T_i est une instance de T s'il existe σ tel que $T_i = T\sigma$

Exemple :
`(int, bool) p2_bintree` et `('a, int) p2_bintree` sont des instances de `('a, 'b) p2_bintree`

Vers la règle de typage

Observations :

- Si une application polymorphe est bien typée :
  `assoc: 'a -> ('a * 'b) list -> 'b;`
  `c: 'a; lst: ('a * 'b) list` $\vdash$ `assoc c lst: 'b`
- … alors aussi toute instance :
  `assoc: string -> (string * int) list -> int;`
  `c:string; lst:(string*int)list` $\vdash$ `assoc c lst: int`
  pour une substitution de types
  $\sigma = [\texttt{'a} \leftarrow \texttt{string}, \texttt{'b} \leftarrow \texttt{int}]$
- … mais pas pour un remplacement non-homogène des variables :
  `assoc: string -> (string * int) list -> int;`
  `c:string; lst:(string*bool)list` $\not\vdash$ `assoc c lst: int`

Polymorphisme : Règle de typage

$$\frac{Env \vdash e \ : \ T}{Env\sigma \vdash e\sigma : T\sigma}$$

où :

- $Env\sigma$:
  Si $Env = [(x_1 : A_1); \ldots (x_n : A_n)]$,
  alors $Env\sigma = [(x_1 : A_1\sigma); \ldots (x_n : A_n\sigma)]$
- $e\sigma$: Substituter avec σ dans chaque type de e
  Exemple : $(fun(x : {}'\mathtt{a}) -> x)[{}'\mathtt{a} \leftarrow bool] = (fun(x : bool) -> x)$

Typage : Exemple (1)

Pour typer : `fun f -> fun x -> (f x)`
posons : `fun (f: 'f) -> fun (x: 'a) -> (f x)`
et faisons la dérivation :

$$
\dfrac{\dfrac{\dfrac{f :' f; x :' a \vdash f :' f}{f : ('a \rightarrow' b); x :' a \vdash f : ('a \rightarrow' b)}\ {\color{red}\sigma} \qquad f : ('a \rightarrow' b); x :' a \vdash x :' a}{f : ('a \rightarrow' b); x :' a \vdash (f\ x) :' b}}{\vdash \texttt{fun}(f : 'a \rightarrow' b) \rightarrow \texttt{fun}(x :' a) \rightarrow (f\ x) : ('a \rightarrow' b) \rightarrow ('a \rightarrow' b)}
$$

avec : $\sigma = ['f \leftarrow' a \rightarrow' b]$

Typage : Exemple (2)

- Soit `nds_intern: ('a, 'b) p2_bintree -> 'b list`
- Soit `Node (1, Leaf true, Leaf false): (bool, int) p2_bintree`
- Alors : `nds_intern (Node (1, Leaf true, Leaf false)) : int list`

ici, $\sigma = [{}'\mathtt{a} \leftarrow \mathtt{bool}, {}'\mathtt{b} \leftarrow \mathtt{int}]$

Typage : Exemple (3)

- Soit `f: ('a*int*'c) -> ('a*'b*'c)`
- Soit `x : (bool * 'b * 'c)`
- Alors `f x : (bool * int * 'c)`

ici, $\sigma = [\texttt{'a} \leftarrow \texttt{bool}, \texttt{'b} \leftarrow \texttt{int}]$

Typage : Exemple (4)

1. Typez `fun (f: 'a -> int) -> ((f 3) + (f 4))`
   En Caml :

```
# fun (f: 'a -> int) ->  ((f 3) + (f 4)) ;;
- : (int -> int) -> int = <fun>
```

2. Typez `fun (f: 'a -> int) -> ((f 3) + (f true))`
   En Caml :

```
# fun (f: 'a -> int) ->  ((f 3) + (f true)) ;;
Characters 35-39:
  fun (f: 'a -> int) ->  ((f 3) + (f true)) ;;
                                     ^^^^
Error: This expression has type bool
    but an expression was expected of type int
```

Plan

Motivation (1)

Terme annoté :

```
# fun (f : int -> int) -> f (f 3) ;;
- : (int -> int) -> int = <fun>
```

En Caml, il suffit d'écrire :

```
# fun f -> f (f 3) ;;
- : (int -> int) -> int = <fun>
```

Inférence de types :

+ Termes plus succints, plus commodes à écrire
+ Même niveau de “sûreté” par typage
– Perte de l'aspect “documentation” du typage

Motivation (2)

Est-ce que l'inférence reconstruit toujours l'annotation de type ?
Terme annoté :

```
# fun (f: int -> int) -> fun (x: int) -> f (f x);;
- : (int -> int) -> int -> int = <fun>
```

Terme non annoté :

```
# fun f -> fun x -> f (f x) ;;
- : ('a -> 'a) -> 'a -> 'a = <fun>
```

Le type inféré peut être plus général que le type annoté.
Définition : T est *plus général* que T' s'il existe substitution σ telle que $T' = T\sigma$

Motivation (3)

Exigences : Étant donné une expression *e* et un environnement *Env*, un algorithme d'inférence de types doit

- calculer le type le plus général T tel que $Env \vdash e : T$ (s'il existe)
- indiquer que *e* n'est pas typable dans *Env* (si un tel T n'existe pas)

Terminologie : type le plus général = type principal d'une expression

Exemples :

- `(fun x -> x + 1) true` n'est pas typable
- `fun x -> x(x)` n'est pas typable

Unification de types (1)

But : trouver une substitution σ telle que deux types T_1, T_2 deviennent égaux, c. à. d., $T_1\sigma = T_2\sigma$. Un tel σ s'appelle un *unificateur* de T_1 et T_2.

Idée :

- Décomposer les types tant que leurs constructeurs sont égaux
- *fail* si les constructeurs sont différents
- Mémoriser la substitution si l'un des types est une variable de type

Note : On peut directement utiliser l'algorithme d'unification pour expressions

Unification de types (2)

Exemples :

1. $\text{Unif}(\texttt{'a -> bool, int -> 'b})$
   $\quad \text{Unif}(\texttt{'a, int}) = [\texttt{'a} \leftarrow \texttt{int}]$
   $\quad \text{Unif}(\texttt{bool, 'b}) = [\texttt{'b} \leftarrow \texttt{bool}]$
   $= [\texttt{'a} \leftarrow \texttt{int}, \texttt{'b} \leftarrow \texttt{bool}]$

   Vérification de la solution :
   $\texttt{'a -> bool}[\texttt{'a} \leftarrow \texttt{int}, \texttt{'b} \leftarrow \texttt{bool}] =$
   $\texttt{int -> 'b}[\texttt{'a} \leftarrow \texttt{int}, \texttt{'b} \leftarrow \texttt{bool}] = \texttt{int -> bool}$

2. $\text{Unif}(\texttt{int -> (int -> bool), (int * int) -> bool})$
   $\quad \text{Unif}(\texttt{int, (int * int)}) = \text{fail}$
   $= \text{fail}$

Algorithme d'inférence (1)

Idée :

- Construction et résolution d'un système de contraintes d'égalité de types
- Synthèse de types à partir de sous-expressions

Cas le plus complexe : Application d'une fonction $(f\ a)$:

- Supposons que a a le type A, f a le type F
- Ajouter la contrainte $F = A \rightarrow B$
  (et la résoudre $\rightsquigarrow$ unification)
- Alors $(f\ a)\ :\ B$

Algorithme d'inférence (2)

Au préalable : Annoter toute variable non annotée avec une nouvelle variable de type :

```
fun f -> fun (x: int) -> (f x)
⇝ fun (f: 'a) -> fun (x: int) -> (f x)
```

Algorithme *PT* ("Principal Type") :

- Entrées : un environnement ; une expression
- Sorties :
  - une instance de l'environnement et de l'expression et son type l'instance est la plus générale ("principal") qui soit typable :
    $PT(Env, e) = (Env', e', T')$ tel que :
    $Env' \vdash e' : T'$ et $Env' = Env\sigma$ et $e' = e\sigma$
  - ou : échec si une telle instance n'existe pas

Exemple :

```
PT([], fun (f: 'a) -> fun (x: int) -> (f x)) =
  ( [],
    fun (f: int -> 'b) -> fun (x: int) -> (f x),
    (int -> 'b) -> int -> 'b)
```

Algorithme d'inférence (3)

L'algorithme *PT* ("Principal Type") défini par récursion sur la structure des expressions :

- *Constantes :* $PT(Env, c) = (Env, c, T)$
  si T est le type "naturel" de c
- *Variables :* $PT(Env, x) = (Env, x, T)$
  si $tp(x, Env) = T$
  (sinon erreur : variable non déclarée)
- *Abstraction :* si $PT((x : A) :: Env, e) = ((x : A') :: Env', e', B)$
  alors
  $PT(Env, \texttt{fun}\ (x : A) \to e) = (Env', \texttt{fun}\ (x : A') \to e', A' \to B)$

Algorithme d'inférence (4)

- *Application :*
  si $PT(Env, f) = (Env_f, f', F)$
  et $PT(Env, a) = (Env_a, a', A)$
  et B est une nouvelle variable de type
  et $E = (Env_f \stackrel{?}{=} Env_a)$
  - si $\sigma = Unif(E \cup (F \stackrel{?}{=} (A \rightarrow B)))$
    alors $PT(Env,\ f\ a) = (Env_f\sigma,\ (f\ a)\sigma,\ B\sigma)$
  - si échec de l'unification, alors $PT(Env,\ f\ a) = fail$

Problème d'unification généré par deux environnements :
Si $Env_f = [x_1 : T_1; \ldots; x_n : T_n]$
et $Env_a = [x_1 : T'_1; \ldots; x_n : T'_n]$
alors $(Env_f \stackrel{?}{=} Env_a) = \{T_1 \stackrel{?}{=} T'_1, \ldots, T_n \stackrel{?}{=} T'_n\}$

Exemple pour le cas *application*

```
PT([f: 'a -> int], f true)
   PT([f: 'a -> int], f)
   = ([f: 'a -> int], f, 'a -> int)
   PT([f: 'a -> int], true)
   = ([f:'a -> int], true, bool)
```

Unif('a -> int, bool -> 'b)

$= ['a \leftarrow \texttt{bool}, 'b \leftarrow \texttt{int}]$

= ([f: bool -> int], f true, int)

Si possible, inférez le type de :

```
PT([f: 'a -> int], f (f true))
```

Algorithme d'inférence : Exemples (1)

Exemple : dans `Env = [(plus, int -> int -> int)]` :

```
PT(Env, fun (x: 'x) -> plus x 2)
   PT((x: 'x)::Env, ((plus x) 2))
      PT((x: 'x)::Env, (plus x))
         PT((x: 'x)::Env, plus)
         = ((x: 'x)::Env, plus, int -> (int -> int))
         PT((x: 'x)::Env, x) = ((x: 'x)::Env, x, 'x)
         Unif(int -> (int -> int), 'x -> 'y)
         = ['x ← int,'y ← (int-> int)]
      = ((x: int) :: Env, plus x, int -> int)
      PT((x: int)::Env, 2) = ((x: int)::Env, int)
      Unif(int -> int, int -> 'z) = ['z ← int]
   = ((x: int)::Env, plus x 2, int)
= (Env, fun (x: int) -> plus x 2, int -> int)
```

Algorithme d'inférence : Exemples (2)

Exemple : dans `Env = [(not, bool -> bool)]` :

`PT(Env, not 42)`
  `PT(Env, not)` = `(Env, bool -> bool)`
  `PT(Env, 42)` = `(Env, int)`
  Unif(`bool -> bool, int -> 'a`) = fail
= fail

Plan

Similarité entre abstraction et `let`

En Caml, on peut *en principe* se passer d'un `let` :

```
# let x = 5 in x + 2 ;;
- : int = 7
# (fun x -> x + 2) 5 ;;
- : int = 7
```

En général, transformer

`let x = e in e'`

en :

`(fun x -> e') e`

Différence entre abstraction et let (1)

La transformation échoue en présence de polymorphisme :

Acceptable :

```
# let f = (fun x -> x) in f f ;;
- : '_a -> '_a = <fun>
```

Inacceptable :

```
# (fun f -> f f) (fun x -> x) ;;
Characters 12-13:
  (fun f -> f f) (fun x -> x) ;;
              ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

Différence entre abstraction et `let` (2)

Analyse échec application :

- Déjà le sous-terme `fun f -> f f` est rejeté.
  Faites l'inférence de type et expliqué le message d'erreur !
- Problème (informel) : impossible de doner un type cohérent à `f` : `'a` ou `'a -> 'b` ou `('a -> 'b) -> ('a -> 'b)` ou ... ?
- ... parce qu'un argument de `fun f -> f f` doit avoir l'un des types `'a`, `'a -> 'b`, ...

Analyse réussite `let` :

- Admissible d'avoir des instances de type différentes :
  let `f = (`**fun** `x -> x)` **in** $\texttt{f}^{('a->'a)->('a->'a)}\ \texttt{f}^{('a->'a)}$

Idée de l'inférence avec `let` (1)

Notation :

- *Variable déclarée :* toute variable x introduite par un **`fun`** `x -> e`
- *Variable définie :* toute variable x introduite par un **`let`** `x = e` **`in`** `e'`

L'environnement contient désormais déclarations et définitions

Exemple :

```
PT([], fun x:int -> let f= (fun y -> y) in (f f) x)
```

$\leadsto$

```
PT([x : int; f : 'a -> 'a = (fun y -> y)], (f f) x)
```

Idée de l'inférence avec `let` (2)

Inférence de type d'une variable *déclarée* (comme avant) :
$PT(Env, x) = (Env, x, T)$
si $tp(x, Env) = T$
$\rightsquigarrow$ Variable a un type uniforme dans un environnement

Inférence de type d'une variable *définie* :
$PT(Env, x) = (Env, x, T['a_1, 'b_1, 'c_1, \dots])$
si $tp(x, Env) = T['a, 'b, 'c, \dots]$
et $T['a_1, 'b_1, 'c_1, \dots]$ est une variante de $T['a, 'b, 'c, \dots]$
avec nouvelles variables de type $'a_1, 'b_1, 'c_1, \dots$
$\rightsquigarrow$ Variable a des types multiples dans un environnement

Idée de l'inférence avec `let` (3)

Exemple (suite) : Dans l'environnement

`Env = [x : int; f : 'a -> 'a = (fun y -> y)]`

```
PT(Env, (f f) x)
    PT(Env, f f)
        PT(Env, f) = (Env, f, 'a1 -> 'a1)
        PT(Env, f) = (Env, f, 'a2 -> 'a2)
        Unif('a1 -> 'a1, ('a2 -> 'a2) -> 'b)
        = ['a1 ← ('a2-> 'a2),'b ← ('a2-> 'a2)]
    = Env, f f, ('a2 -> 'a2)
    PT(Env, x) = (Env, x, int)
    Unif('a2 -> 'a2, int -> 'c) = ['a2 ← int,'c ← int]
= (Env, (f f) x, int)
```

Plan

Plan

4 Induction

- Induction structurelle
- Théorie des points fixes
- Induction sur des règles

Rappel : Induction sur les nombres naturels (1)

Principe : à montrer : $\forall n.P(n)$

Démarche :

- Montrer $P(0)$
- Montrer $P(n) \longrightarrow P(n+1)$

Exemple : à montrer : $\forall n.\Sigma_{i=0}^{n} i = \frac{n(n+1)}{2}$

Démarche :

- Montrer $\Sigma_{i=0}^{0} i = \frac{0(0+1)}{2}$
- Montrer

$$\Sigma_{i=0}^{n} i = \frac{n(n+1)}{2} \longrightarrow \Sigma_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

Compléter la preuve

Rappel : Induction sur les nombres naturels (2)

Ingrédients d'une preuve par *induction* : Fonctions *récursives* :

- $\Sigma_{i=0}^{0} i = 0$
- $\Sigma_{i=0}^{n+1} i = (\Sigma_{i=0}^{n} i) + (n+1)$

Donner une définition de $\Sigma_{i=0}^{n} i$ en Caml

Faire la preuve de :

$$\forall n. \Sigma_{i=0}^{n} (2i+1) = (n+1)^2$$

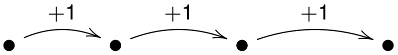
Donner une définition d'une fonction Caml `sigf f n` qui calcule

$$\Sigma_{i=0}^{n} f(i)$$

Les nombres naturels comme ensemble inductif

L'ensemble des entiers naturels *Nat* est le plus petit ensemble qui respecte les conditions suivantes :

1. *Zéro :* $0 \in Nat$
2. *Successeur :* Si $n \in Nat$, alors $(Sn) \in Nat$



représentation décimale	0	1	2	3
représentation unaire	0	$S0$	$SS0$	$SSS0$

Types inductifs, fonctions récursives, preuves (1)

Les éléments d'un type inductif sont générés par application successive des constructeurs.

Exemple : Listes :

- Constructeurs :
  - Liste vide : `[] : 'a list`
  - "Cons" : `(::) : 'a -> 'a list -> 'a list`
- Éléments (par exemple `int list`) :
  - `[]`
  - `0::[], 1::[], 2::[], ...`
  - `0:: 0:: [], 0:: 1:: [], 1:: 0:: [], ...`

Types inductifs, fonctions récursives, preuves (2)

Une fonction primitif-récursive sur un type de données

- réduit son argument à des arguments plus simples
- ... inverse le processus de construction des éléments
- ... ce qui assure la terminaison

Exemple :

```
let rec length = fun xs ->
   match xs with
      [] -> 0
    | x :: xs' -> 1 + length xs'
```

Trace :

```
length 3 :: 4 :: []    -->      2
     length 4 :: []    -->    1
          length []    -->  0
```

Types inductifs, fonctions récursives, preuves (3)

Une preuve par induction structurelle

- permet d'établir la vérité d'une proposition sur *tous les éléments* d'un type inductif
- en reconstituant le processus de construction des éléments

Exemple : Montrer que `length xs` $\geq$ `0` pour toute liste `xs`.
Preuve :

- `length [] = 0` $\geq$ `0`
- `length (0::[]) = 1 + length []` $\geq$ `0`, parce que `length []` $\geq$ `0`
- `length (2::0::[]) = 1 + length (0::[])` $\geq$ `0`, parce que `length (0::[])` $\geq$ `0`

Induction sur des listes

Schéma d'induction structurelle sur les listes

Pour montrer $P(xs)$ pour toute liste xs, montrer

- $P([])$
- $P(xs') \longrightarrow P(x :: xs')$ pour tout élément x et liste xs'

Application : Montrer que `length xs` $\geq$ `0` pour toute liste `xs`.

- `length []` $\geq$ `0`
  (est vrai, utilisant la définition de `length`)
- `length xs'` $\geq$ `0` $\longrightarrow$ `length (x :: xs')` $\geq$ `0`
  se réduit à
  `length xs'` $\geq$ `0` $\longrightarrow$ `1 + length xs'` $\geq$ `0`
  (utilisant la définition de `length`)
  (est vrai, par raisonnement arithmétique)

Application : Fusion (1)

La fonction `listmap` applique une fonction à tous les éléments d'une liste.

Exemple :

```
# listmap (fun x -> x + 2) [1; 2; 3] ;;
- : int list = [3; 4; 5]
# listmap (fun x -> [x]) [1; 2; 3] ;;
- : int list list = [[1]; [2]; [3]]
```

Définir la fonction `listmap`.
(*NB :* La fonction est prédéfinie comme `List.map` dans la librairie Caml).

Application : Fusion (2)

Fusion de deux `listmap` :
Deux applications consécutives de `listmap` peuvent être réalisées par une seule :

```
# listmap (fun x -> x + 4)
          (listmap (fun x -> 2 * x) [1; 2; 3]) ;;
- : int list = [6; 8; 10]
# (listmap (fun x -> 2 * x + 4) [1; 2; 3]) ;;
- : int list = [6; 8; 10]
```

Définir la fonction `comp` qui compose deux fonctions.

```
# comp (fun x -> x + 4) (fun x -> 2 * x) 3 ;;
- : int = 10
# comp (fun x -> [x]) (fun x -> 2 * x) 3 ;;
- : int list = [6]
```

Application : Fusion (3)

Fusion, écrite avec `comp` :

```
# listmap (fun x -> x + 4)
          (listmap (fun x -> 2 * x) [1; 2; 3]) ;;
- : int list = [6; 8; 10]
# listmap (comp (fun x -> x + 4)
          (fun x -> 2 * x)) [1; 2; 3] ;;
- : int list = [6; 8; 10]
```

Démontrer pour toutes fonctions `f, g` et toute liste `xs` :

```
listmap f (listmap g xs) = listmap (comp f g) xs
```

Application : Fusion (4)

Comparaison :

	# parcours liste	cellules mémoire
listmap f (listmap g xs)	2	2 * (length xs)
listmap (comp f g) xs	1	length xs

Conséquences :

- Réduction du temps d'exécution
- Réduction de la consommation de mémoire
  $\rightsquigarrow$ incidence sur temps d'exécution : moins d'activations du ramasse-miette (*garbage collector*)

Applications :

- Optimisation de compilateurs
- Optimisation de moteurs de requêtes (bases de données)

Les nombres naturels comme type inductif

Les nombres naturels se définissent comme type inductif avec les constructeurs 0 et S (successeur, $\equiv +1$)
Exemple : La représentation de 3 est $S(S(S(0)))$
Réinterprétation du schéma d'induction :
Pour montrer $\forall n.P(n)$

- montrer $P(0)$
- montrer $P(n) \longrightarrow P(S(n))$

Définir l'addition et la multiplication par récursion primitive sur 0 et S.
$\rightsquigarrow$ arithmétique de Peano
Montrer par induction : l'addition est commutative

The Italian Connection

Francesco Maurolico (1494-1575)

Première preuve par induction

Giuseppe Peano (1858-1932)

Axiomatisation des nombres naturels

Autres types inductifs (1)

Pour tout type inductif T, on peut dériver un schéma d'induction de manière mécanique :
Pour montrer $\forall t : T.P(t)$

- Pour tout constructeur de base C_b `of` $T_1 * \cdots * T_n$, montrer $\forall x_1 \ldots x_n.P(C_b(x_1, \ldots, x_n))$
- Pour tout constructeur inductif C_i `of` $\underbrace{T * \ldots T}_{k\times} * T_1 * \ldots T_m$, montrer

  $\forall t_1 \ldots t_k, x_1 \ldots x_m.P(t_1) \wedge \cdots \wedge P(t_k) \longrightarrow P(C_i(t_1, \ldots, t_k, \ x_1, \ldots, x_m))$

Autres types inductifs (2)

Arbres binaires

```
type 'a bintree =
    Leaf of 'a
  | Node of 'a  * 'a bintree * 'a bintree
```

Schéma d'induction :

Pour montrer $\forall t : \texttt{'a bintree}. P(t)$

- montrer $\forall \ell . P(\texttt{Leaf}(\ell))$
- montrer $\forall t_1, t_2, n. P(t_1) \wedge P(t_2) \longrightarrow P(\texttt{Node}(n, t_1, t_2))$

Autres types inductifs (3)

Définir la fonction `swap` qui échange partout les sous-arbres gauches et droits :

```
# swap (Node(1, Leaf 2, Node(3, Leaf 4, Leaf 5)));;
- : Node (1, Node (3, Leaf 5, Leaf 4), Leaf 2)
```

Montrer par induction : $\forall t.\mathtt{swap}(\mathtt{swap}(t)) = t$

- montrer $\forall \ell.\mathtt{swap}(\mathtt{swap}(\mathtt{Leaf}\ \ell)) = \mathtt{Leaf}\ \ell$
- montrer $\forall n.\mathtt{swap}(\mathtt{swap}(t_1)) = t_1 \wedge \mathtt{swap}(\mathtt{swap}(t_2)) = t_2 \longrightarrow \mathtt{swap}(\mathtt{swap}(\mathtt{Node}(n, t_1, t_2))) = (\mathtt{Node}(n, t_1, t_2))$

Compléter la preuve !

Plan

Motivation et démarche

But : Généraliser les définitions inductives (nombres naturels ; listes)
Résultat : Méthode générale d'obtention de principes d'induction
Applications :

- Large classe de types inductifs
- Règles définies inductivement
  - exemple : règles de typage
  - exemple : sémantique des langages de programmation
- Ordres fondés
  $\rightsquigarrow$ raisonnement sur la terminaison de programmes

Exemple : $\mathbb{N}$ comme type inductif (1)

Définition (fictive) des nombres naturels $\mathbb{N}$ comme type inductif :

```
type nat =
     0
   | S of nat
```

Ici : `S` est la fonction *successeur* : `S(`n`)` = $n + 1$
Exemple : $S(S(0)) = 2$; en général : $S^n(0) = n$

Définition inductive par des règles :

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{n+1 \in \mathbb{N}}$$

Lecture : $\mathbb{N}$ est le plus petit ensemble A tel que :

- $0 \in A$
- si $n \in A$, alors $S(n) \in A$

Exemple : $\mathbb{N}$ comme type inductif (2)

Définissons la fonction génératrice de $\mathbb{N}$:

$$f_{nat}(A) = \{m.\ m = 0 \vee \exists n.n \in A \wedge m = S(n)\}$$

Approximation de $\mathbb{N}$ par :

$A_0 = \{\}$

$A_1 = f_{nat}(A_0) = \{0\}$

$A_2 = f_{nat}(A_1) = \{0, 1\}$

...

$\mathbb{N} = \bigcup_n f_{nat}^n(\{\})$

$\mathbb{N}$ comme point fixe

Définition : un point fixe d'une application f est un x tel que $f(x) = x$
Constats :

- $\mathbb{N}$ est un point fixe de f_{nat} : $f_{nat}(\mathbb{N}) = \mathbb{N}$
- Il y a d'autres points fixes, par exemple $\mathbb{Z}$
  vérifiez : $\forall z \in \mathbb{Z}.\ z = 0 \vee \exists n. n \in \mathbb{Z} \wedge z = S(n)$
  Mais : $\mathbb{N} \subset \mathbb{Z}$
- Pour tout $A \subset \mathbb{N}$: $A \subset f_{nat}(A)$

Conséquence : $\mathbb{N}$ est le plus petit point fixe de f_{nat}
("le plus petit ensemble tel que ...")

Théorème de Knaster-Tarski

Une fonction sur des ensembles est monotone si $A \subseteq B$ implique $f(A) \subseteq f(B)$.

P est appelé pre-point fixe de f si $f(P) \subseteq P$

Notation : on écrit $lfp(f)$ (“least fixed point”) le plus petit point fixe de f.

Théorème de Knaster-Tarski : Soit f une fonction monotone. Alors $lfp(f) = \bigcap\{P | f(P) \subseteq P\}$.
(preuve en TD)

A retenir : Deux caractérisations possibles de $\mathbb{N}$:

- Comme limite d’une approximation : $\mathbb{N} = \bigcup_n f_{nat}^n(\{\})$
- Comme intersection de tous les pre-points fixes : $\mathbb{N} = \bigcap\{P | f_{nat}(P) \subseteq P\}$.

Knaster-Tarski et induction (1)

Dérivons le principe d'induction de la caractérisation par pre-points :
Raisonnement générique :

- Soit $\bigcap\{P | f_{nat}(P) \subseteq P\} = \mathbb{N}$.
  donc : $\forall n.(n \in \bigcap\{P | f_{nat}(P) \subseteq P\}) = (n \in \mathbb{N})$
- *Rappel :* $n \in \bigcap A = (\forall B.B \in A \longrightarrow n \in B)$
  Donc : $\forall n.(\forall P.(f_{nat}(P) \subseteq P) \longrightarrow n \in P) = (n \in \mathbb{N})$
- Surtout : $\forall P.(f_{nat}(P) \subseteq P) \longrightarrow \forall n \in \mathbb{N}.\ n \in P$
- Donc, pour un P fixe : $(f_{nat}(P) \subseteq P)$ est une condition suffisante pour prouver $\forall n \in \mathbb{N}.\ n \in P$

Notes :

- Raisonnement “générique” parce que indépendant de $\mathbb{N}$ et de la définition de f_{nat}
- … donc applicable à d'autres définitions inductives

Knaster-Tarski et induction (2)

Raisonnement specifique - principe d'induction pour $\mathbb{N}$:

- Transformons $f_{nat}(P) \subseteq P$
- *Rappel :* $f_{nat}(P) = \{m.\ m = 0 \vee \exists n.\ n \in P \wedge m = S(n)\}$
- Expansion de f_{nat} :
  $\{m.\ m = 0 \vee \exists n.\ n \in P \wedge m = S(n)\} \subseteq P$
- Après des transformations (vérifiez !) :
  $0 \in P \wedge (\forall n.\ n \in P \longrightarrow S(n) \in P)$
  (Version “ensembliste”)
- Version prédicative pour prédicat unaire $P(n)$ au lieu de $n \in P$:
  $P(0) \wedge (\forall n.\ P(n) \longrightarrow P(S(n)))$
  (Schéma d'induction habituel sur $\mathbb{N}$)

Dérivation du principe d'induction d'un type inductif (1)

Pour tout type `T` défini inductivement :

```
type T = Cb of T_1 * ... T_n
       | ...
       | Ci of T * ... T * T_1 * ... T_m
```

construire fonction génératrice

$$
\begin{array}{lll}
f_T(A) = & \{a. & \\
 & & (\exists t_1 \in T_1 \dots t_n \in T_n.\ a = Cb(t_1 \dots t_n)) \\
 & \vee & \dots \\
 & \vee & (\exists a_1 \in A \dots a_k \in A, t_1 \in T_1 \dots t_m \in T_m. \\
 & & a = Ci(a_1 \dots a_m, t_1 \dots t_n)) \\
 & \} &
\end{array}
$$

Dérivation du principe d'induction d'un type inductif (2)

- Dériver $\forall P.(f_T(P) \subseteq P) \longrightarrow \forall t \in T.t \in P$ (raisonnement générique)
- Transformer $(f_T(P) \subseteq P)$
- Obtenir principe d'induction pour type T (voir plus haut)

Important : la fonction f_T doit être monotone : précondition de Knaster-Tarski.
(voir exercice sur définitions non positives)

Plan

Définitions inductives de relations (1)

Vu jusqu'à maintenant :

- Définition inductive de types
- Type $\approx$ ensemble de valeurs
- Dérivation d'un principe d'induction sur ces types

Ici : Extension du même principe à

- des ensembles
- des relations (comme ensembles de n-uplets)
- ... avec principe d'induction sur les règles

Définitions inductives de relations (2)

Définition inductive des nombres pairs : Les nombres pairs sont le plus petit ensemble tel que

- 0 est un nombre pair
- si n est un nombre pair, aussi $n+2$ est un nombre pair

Écriture comme ensemble :

$$\frac{}{0 \in Pair}$$

$$\frac{n \in Pair}{(n+2) \in Pair}$$

Écriture comme prédicat :

$$\frac{}{Pair(0)}$$

$$\frac{Pair(n)}{Pair(n+2)}$$

Chaînage en avant

Une règle correspond à une implication universellement quantifiée :
Ex. : $\forall n. n \in Pair \longrightarrow (n+2) \in Pair$

Chaînage en avant :
Application de la règle dans le sens de l'implication.
Dérivez :

- $Pair(6)$
- $\forall n. Pair(n) \longrightarrow Pair(n+4)$

Règles inductives et induction (1)

Un ensemble de règles ne se résume pas à un ensemble de faits et d'implications.
Au sens logique, les trois formules suivantes sont cohérentes (ont un modèle) :

- $Pair(0)$
- $\forall n.\ Pair(n) \longrightarrow Pair(n+2)$
- $Pair(5)$

Donc :
$\neg Pair(5)$ ne peut pas être une conséquence des deux premières.
Que manque ? Principe d'induction sur les règles !

Règles inductives et induction (2)

En général sans principe d'induction, impossible de dériver

- des énoncés sur tous les éléments de la relation inductive : $\forall n.\ Pair(n) \longrightarrow n \mod 2 = 0$
- des faits négatifs : $\neg Pair(5)$
  NB. : $\neg Pair(5) \ \equiv\ \forall n. Pair(n) \longrightarrow n \neq 5$

Le principe d'induction exprime la minimalité de la relation définie par les règles.
La dérivation du principe d'induction pour les règles inductives se fait de la même manière que pour les types inductifs.

Principe d'induction pour la relation Pair :
$\forall P.\ (P(0) \wedge (\forall n. P(n) \longrightarrow P(n+2))) \longrightarrow$
$\forall n. Pair(n) \longrightarrow P(n)$
Comparez avec induction sur $\mathbb{N}$!

Règles inductives et induction (3)

Exemple : Preuve détaillée de $\forall n.\ Pair(n) \longrightarrow n \mod 2 = 0$

L'énoncé est une instance de la conclusion du principe d'induction pour $P(n) = n \mod 2 = 0$.

Il suffit donc de montrer la prémisse pour cette instance :

- $0 \mod 2 = 0$
- $\forall n.n \mod 2 = 0 \longrightarrow (n+2) \mod 2 = 0$

Fermeture réflexive-transitive (1)

Définition inductive de la fermeture réflexive-transitive R^* d'une relation R :
R^* est la plus petite relation définie par :

- $R^*(x, x)$
- si $R(x, y)$ et $R^*(y, z)$, alors $R^*(x, z)$

Exercice : montrer que si $R(a, b)$ et $R(b, c)$, alors $R^*(a, c)$

Principe d'induction associé : pour tout prédicat P :

- si $\forall x. P(x, x)$
- et $\forall x,\ y,\ z.\ R(x, y) \longrightarrow P(y, z) \longrightarrow P(x, z)$
- alors $\forall x,\ z.\ R^*(x, z) \longrightarrow P(x, z)$

Fermeture réflexive-transitive (2)

Preuve par induction sur des relations :
Exemple 1 : Soit $R(a, b)$ et $R(b, c)$ et $\neg R(x, y)$ pour tous les autres $x, y \in \{a, b, c\}$. Montrer $\neg R^*(c, a)$.
Preuve par induction :

- Réécrire $\neg R^*(b, a)$ comme : $\forall x\, z. R^*(x, z) \longrightarrow \neg(x = c \wedge z = a)$
- Appliquer le principe d'induction avec $P(x, z) := \neg(x = c \wedge z = a)$
- Montrer $\forall x. \neg(x = c \wedge x = a)$
  évident pour constantes $a \neq c$
- Montrer
  $\forall x,\ y,\ z.\ R(x, y) \longrightarrow \neg(y = c \wedge z = a) \longrightarrow \neg(x = c \wedge z = a)$

compléter la preuve

Fermeture réflexive-transitive (2)

Preuve par induction sur des relations :
Exemple 1 : Soit $R(a, b)$ et $R(b, c)$ et $\neg R(x, y)$ pour tous les autres $x, y \in \{a, b, c\}$. Montrer $\neg R^*(c, a)$.
Preuve par induction :

- Réécrire $\neg R^*(b, a)$ comme : $\forall x\, z. R^*(x, z) \longrightarrow \neg(x = c \wedge z = a)$
- Appliquer le principe d'induction avec $P(x, z) := \neg(x = c \wedge z = a)$
- Montrer $\forall x. \neg(x = c \wedge x = a)$
  évident pour constantes $a \neq c$
- Montrer
  $\forall x,\ y,\ z.\ R(x, y) \longrightarrow \neg(y = c \wedge z = a) \longrightarrow \neg(x = c \wedge z = a)$

$\forall x,\ y,\ z.\ R(x, y)\ (x = c \wedge z = a) \longrightarrow (y = c \wedge z = a)$
$\forall x,\ y,\ z.\ R(c, y) \longrightarrow (y = c)$
vrai parce que $\neg \exists y. R(c, y)$.
Faire la preuve de : $\neg R^*(b, a)$.

Fermeture réflexive-transitive (3)

Exemple 2 : montrer que R^* est transitive :
$\forall a, b, c.\ R^*(a, b) \longrightarrow (R^*(b, c) \longrightarrow R^*(a, c))$
Preuve par induction : fixer a, b, c, instancier
$P(x, z) := (R^*(z, c) \longrightarrow R^*(x, c))$
compléter la preuve

Plan

Plan

Analyses statiques - pourquoi ?

Détection d'erreurs ou dysfonctionnements avant l'exécution :

- Variables non initialisées $\leadsto$ Definite Assignment
- Code mort / inatteignable

Optimisations lors de la compilation :

- Élimination de code mort $\leadsto$ code plus compact
- Élimination / substitution de constantes
  $\leadsto$ calcul lors de la compilation vs. pendant exécution du code
- Réduction de la taille de la pile / du nombre de registres
  $\leadsto$ moins de consommation de mémoire, code plus efficace

$\Longrightarrow$ besoins similaires à la motivation de systèmes de types

Présentation suivante inspiréé de :
Nipkow / Klein : *Concrete Semantics*

Analyses précises / approximatives (1)

Des analyses ont pour but d'avertir de situtations indésirables.
Classification des résultats d'une analyse :

- *vrai positif :* l'analyse signale un problème, et le problème existe définitivement
- *faux positif :* l'analyse signale un problème, mais le problème n'existe pas
- *vrai négatif :* l'analyse signale l'absence d'un problème, et il n'existe effectivement pas de problème
- *faux négatif :* l'analyse signale l'absence d'un problème, mais il y a un problème

Analyses précises / approximatives (2)

Exemple : Analyse de division par zéro
Soit $n \in [0..6]$. Supposons une analyse qui affiche des divisions par zéro en rouge et les divisions non-critiques en vert

```
x = 1 / n;     // vrai positif, problème pour n = 0
if (n > 3)
   x = 2 / n;  // faux positif, ici: n ∈ [4 .. 6]
else
   x = 3 / n;   // faux négatif, ici: n ∈ [0 .. 3]
x = 4 / (n + 2); // vrai négatif: n+2 ∈ [2 .. 8]
```

Analyses précises / approximatives (3)

Une analyse est parfaitement précise si elle partitionne l'ensemble des situations en vrai positifs et vrai négatifs. Elle ne fait pas d'erreurs (faux positifs / faux négatifs).

Une analyse approximative admet :

- de faux négatifs : critique pour la sûreté d'un logiciel (erreur non détectée), *ou*
- de faux positifs : embêtants pour un reviewer (fausses alarmes), mais pas critiques

Analyses précises / approximatives (4)

Est-ce qu'une analyse parfaitement précise est possible ?
Dans la plupart des cas : non. Elle permettrait de résoudre le problème d'arrêt (indécidable) pour n'importe quel programme `P`.

Exemple : Supposons une analyse parfaitement précise de division par zéro :

- `P; x = 1/0;` signifie : `P` s'arrête et la division produit une erreur.
- `P; x = 1/0;` signifie : `P` ne s'arrête pas ; la division n'est pas atteignable et ne produit pas d'erreur.

Conséquence :

- On n'arrivera pas à concevoir une analyse parfaitement précise
- … mais au moins une analyse *correcte* : sans faux négatifs
- … et de prouver la correction

Analyses étudiées

Nous étudions en détail deux analyses :

- *Definite Assignment :*
  - Vérifie que toute variable est initialisée avant d'être lue.
  - Propage l'information du début vers la fin (*forward analysis*).
- *Variables vivantes :*
  - Vérifie que toute écriture d'une variable sera suivie d'une lecture.
  - Propage l'information de la fin vers le début (*backward analysis*).

Plan

Motivation

But : Vérification que toute lecture d'une variable est précédée d'au moins une lecture, pour n'importe quel chemin d'exécution.
Utilisation : Fait partie de la suite de vérifications de Java
Exemple : Partie d'un programme :

```
if (cond) {
    x = 3; }
else {
    y = 3; }
y = x;
```

```
> javac Main.java
Main.java:22:
error: variable x might not have been initialized
       y = x;
           ^
```

Expressions et instructions de $\mathcal{L}_e$ (1)

Expressions

E	::=	n	(Constantes entières $n \in \mathbb{Z}$)
	|	b	(Constantes booléennes $b \in \mathit{Bool}$)
	|	v	(Variables $v \in \mathcal{V}$)
	|	(E + E) | (E - E) | ...	
	|	(E == E) | (E < E) | ...	
	|	! E | (E && E) | ...	

Instructions

C	::=	`Skip`	("ne fait rien")
	|	$v = $ E	(Affectation)
	|	C ; C	(Séquence)
	|	`if` E `then` C `else` C	
	|	`while` E `do` C	

Idée

Proméner un ensemble sur le programme, dès le début vers la fin et :

- Rajouter toute variable qui est affectée (à gauche d'une affectation).
- Vérifier que toute variable lue (dans une expression) est dans l'ensemble.

Exemple (réussite de l'analyse) :

```
{ }
x = 3;

y = x + 2;

x = x + y;
```

Idée

Proméner un ensemble sur le programme, dès le début vers la fin et :

- Rajouter toute variable qui est affectée (à gauche d'une affectation).
- Vérifier que toute variable lue (dans une expression) est dans l'ensemble.

Exemple (réussite de l'analyse) :

```
x = 3;
{ x }
y = x + 2;

x = x + y;
```

Idée

Proméner un ensemble sur le programme, dès le début vers la fin et :

- Rajouter toute variable qui est affectée (à gauche d'une affectation).
- Vérifier que toute variable lue (dans une expression) est dans l'ensemble.

Exemple (réussite de l'analyse) :

```
x = 3;

y = x + 2;
{ x, y }
x = x + y;
```

Idée

Proméner un ensemble sur le programme, dès le début vers la fin et :

- Rajouter toute variable qui est affectée (à gauche d'une affectation).
- Vérifier que toute variable lue (dans une expression) est dans l'ensemble.

Exemple (réussite de l'analyse) :

```
x = 3;

y = x + 2;

x = x + y;
{ x, y }
```

Idée

Proméner un ensemble sur le programme, dès le début vers la fin et :

- Rajouter toute variable qui est affectée (à gauche d'une affectation).
- Vérifier que toute variable lue (dans une expression) est dans l'ensemble.

Exemple (échec de l'analyse parce que $z \notin \{x, y\}$) :

```
x = 3;

y = x + 2;
{ x, y }
x = x + z;
```

Conditionnels

```
{ }
if e {
    x = 2;
    y = 3;
    { x, y }
}
else {
    x = 3;
    z = 4;
    { x, z }
}
{ x }
```

- A la fin d'un `if`, une variable est uniquement considérée affectée si elle l'est pour toute branche.
- $\rightsquigarrow$ intersection des résultats des deux branches
- Pas d'analyse fine des conditions de branchement.

Boucles

```
{ i, n }
while (i <= n) {
    { i, n }
    r = r * i;
    // r non initialisé!
    i = i + 1;
}
{ i, n }
```

- Variables initialisées avant = variables initialisées après la boucle
  (possibilité que le corps de la boucle n'est jamais exécutée)
- Il faut vérifier les variables à l'intérieur de la boucle

Formalisation

Prédicat *Def* :

- Forme : $Def(A, c, A')$ où :
  - A, A' sont des ensembles de variables
  - c est une instruction
- Sémantique :
  Si toute variable $x \in A$ est définie avant exécution de c, alors il n'y a pas d'accès à une variable non-initialisée lors de l'exécution de c
  et toute $y \in A'$ est définie après exécution de c

Exemples :

- $Def(\{x\}, \mathrm{y = x + 2}, \{x, y\})$
- Il n'y a pas de A tel que $Def(\{\}, \mathrm{y = x + 2}, A)$

Prérequis : Fonction $fv(e)$ calcule l'ensemble des variables libres d'une expression

Exemple : $fv((x + y) - (2 * x)) = \{x, y\}$

Règles

$$\frac{}{Def(A, \texttt{Skip}, A)}$$

$$\frac{fv(e) \subseteq A}{Def(A, x = e, \{x\} \cup A)}$$

$$\frac{Def(A_1, c_1, A_2) \qquad Def(A_2, c_2, A_3)}{Def(A_1, (c_1; c_2), A_3)}$$

$$\frac{fv(e) \subseteq A \qquad Def(A, c_1, A_1) \qquad Def(A, c_2, A_2)}{Def(A, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, A_1 \cap A_2)}$$

$$\frac{fv(e) \subseteq A \qquad Def(A, c, A')}{Def(A, \texttt{while}\ e\ \texttt{do}\ c, A)}$$

Application (1)

Soit P le programme :

```
x = 2;
y = x + 2;
if (y < 3) {
  z = x + y;
}
else {
  // variante: remplacez par z = ...
  x = x - y;
}
r = z * 3;
```

Lancez l'analyse avec $Def(\{\}, P, ?B)$

Application (2)

Soit P le programme :

```
x = 2;
while (x < y) {
  z = x * y + 3;
}
r = z + 5;
```

Lancez l'analyse avec $Def(\{y\}, P, ?B)$

Que diriez vous du programme suivant ?

```
x = 3;
if (x < x) { y = y + 1} else { y = x };
y = y + 1;
```

Correctement initialisé ?

Implantation

Résumé - analyse : on calcule $\mathit{Def}(A, c, ?B)$ par récursion sur c, où A est donné (typiquement : $A = \{\}$) et on essaye de trouver $?B$.

Analyse par propagation en avant : On convertit prédicat $\mathit{Def}(A, c, B)$ en fonction récursive partielle `defassign(A, c)`.
Résultats possibles :

- `defassign(A, c)` renvoie `B` tel que $\mathit{Def}(A, c, B)$: Succès de l'analyse
- `defassign(A, c)` ne renvoie pas de `B`
(alternatives : exception ou résultat `option` avec Some ou `None`)
Dans ce cas, il n'existe pas de B avec $\mathit{Def}(A, c, B)$

Plan

Idée de l'énoncé de correction

Notion de correction (première version) :

- Si l'analyse réussit : $Def(A, c, A')$
  alors le programme c s'exécute sans erreur
- Si l'analyse échoue
  le programme peut produire une erreur ou non
  (l'analyse n'est qu'approximative)

Pour cela : Sémantique qui trace si une variable est affectée ou non

Notion d'état partiel (1)

Etat traçant la situation d'affectation à des variables :

```
type state = (var * value) list
type value =  ...
           | IntV of int (* entiers etc ... *)
           | UndefV      (* valeur indéfinie *)
```

Exemple : Soit σ l'état :

```
[(x, IntV 3); (y, UndefV); (z, UndefV)]
```

Donner l'état après la séquence d'instructions

```
y = 2;
x = 4;
```

Notion d'état partiel (2)

Notion d'un état en situation d'erreur : `state option` avec :

- *Some*(σ) : état bien défini σ
- *None* : état en situtation d'erreur

Exemple : Soit σ l'état :

`[(x, IntV 3); (y, UndefV); (z, UndefV)]`

On démarre en état *Some*(σ)

Examiner la situation après exécution de :

Programme 1 :

```
y = x + 1;
z = y + 2;
```

Notion d'état partiel (2)

Notion d'un état en situation d'erreur : `state option` **avec :**

- *Some*(σ) : état bien défini σ
- *None* : état en situtation d'erreur

Exemple : Soit σ l'état :

`[(x, IntV 3); (y, UndefV); (z, UndefV)]`

On démarre en état *Some*(σ)

Examiner la situation après exécution de :

Programme 1 :

```
y = x + 1;
z = y + 2;
```

État sans erreur :

`Some [(x, IntV 3); (y, IntV 4); (z, IntV 6)]`

Notion d'état partiel (2)

Notion d'un état en situation d'erreur : `state option` avec :

- *Some*(σ) : état bien défini σ
- *None* : état en situtation d'erreur

Exemple : Soit σ l'état :

`[(x, IntV 3); (y, UndefV); (z, UndefV)]`

On démarre en état *Some*(σ)

Examiner la situation après exécution de :

Programme 1 :

```
y = x + 1;
z = y + 2;
```

Programme 2 :

```
z = y + 1;
y = x + 2;
```

État sans erreur :

`Some [(x, IntV 3); (y, IntV 4); (z, IntV 6)]`

Notion d'état partiel (2)

Notion d'un état en situation d'erreur : `state option` avec :

- *Some*(σ) : état bien défini σ
- *None* : état en situtation d'erreur

Exemple : Soit σ l'état :

```
[(x, IntV 3); (y, UndefV); (z, UndefV)]
```

On démarre en état *Some*(σ)

Examiner la situation après exécution de :

Programme 1 :

```
y = x + 1;
z = y + 2;
```

État sans erreur :

```
Some [(x, IntV 3); (y,
IntV 4); (z, IntV 6)]
```

Programme 2 :

```
z = y + 1;
y = x + 2;
```

État avec erreur : `None`

Adaptation des règles :

Modifier la définition de la fonction $\mathcal{A}(a, \sigma)$

Affectation

$$\frac{\mathcal{A}(a, \sigma) = v \neq \text{UndefV}}{\langle x := a, Some(\sigma)\rangle \rightarrow_t Some(\sigma.x \leftarrow v)}$$

$$\frac{\mathcal{A}(a, \sigma) = \text{UndefV}}{\langle x := a, \sigma\rangle \rightarrow_t None}$$

Définir les autres règles

Notion de correction, formellement

Soit l'ensemble des variables définies :
$dom(\sigma) = \{x | \sigma(x) = v \neq UndefV\}$

Correction de l'analyse de Definite Assignment :
Soit $Def(A, c, A')$.

- Informellement :
  Si on démarre c dans un état σ où les variables A sont définies,
  et si le programme termine,
  alors dans un état sans erreur où les variables A' sont définies.
- Formellement :
  - Si $\langle c, Some(\sigma)\rangle \rightarrow_t \sigma'$
  - et $A \subseteq dom(\sigma)$
  - alors $\exists t. \sigma' = Some(t) \wedge A' \subseteq dom(t)$

Preuve Par induction sur les règles de la sémantique

Plan

Motivation

But : Détecter des affectations superflues parce que la valeur affectée n'est jamais lue dans le programme.

Utilisation :

- Pour détecter des erreurs dans un programme écrit manuellement.
- Pour optimiser un programme (souvent généré automatiquement)

Exemples :

Affectation inutile :

```
x = x + 3;
y = 2 * x + z;
return (x);
```

Variable écrasée avant lecture :

```
y = 5;          (1)
y = x + 3;      (2)
x = x + y;
```

Affectation (2) écrase (1)

Idée

Définition : Variable v est vivante dans une position P, si

- il y a un chemin d'exécution commençant avec P sur lequel v est lue
- sans avoir été redéfinie précédemment

Démarche : Proméner l'ensemble des variables vivantes sur le programme, dès la fin vers le début :

- Enlever la variable qui est redéfinie
  (elle n'a plus d'utilité à ce point) ;
- Rajouter toutes les variables dans des expressions
  (toute variable lue est "utile" et donc vivante).

Dans la suite, définition implicite d'un fonction de variables vivantes $vv(c, A)$ pour une instruction c et un ensemble A de variables vivantes après c.

Affectation

```
x = y + 3;

y = 2 * x + z;
{ x }           // x est utilisé dans return
return (x);
```

Affectation

```
x = y + 3;
{ x, z }     // x et z utilisés dans expression
y = 2 * x + z;
{ x }        // x est utilisé dans return
return (x);
```

Affectation

```
{ y, z }     // x inutile parce que redéfini
x = y + 3;
{ x, z }     // x et z utilisés dans expression
y = 2 * x + z;
{ x }        // x est utilisé dans return
return (x);
```

Affectation

```
{ y, z }     // x inutile parce que redéfini
x = y + 3;
{ x, z }     // x et z utilisés dans expression
y = 2 * x + z;
{ x }        // x est utilisé dans return
return (x);
```

En général : $vv(x = e; A) = fv(e) \cup (A - \{x\})$

$fv(e) \cup (A - \{x\})$

```
x = e;
```

A

Conditionnels

```
{ x, y, z }
if (x < 3) {
{ x, y }
z = x + y;
}
else {
{ x, z }
y = x + z;
}
{ y }
return (y);
```

En général :

$fv(e) \cup A1 \cup A2$

```
if (e) {
    A1
    c1
    A
}
else {
    A2
    c2
    A
}
A
```

Boucles

```
{ x, y }
while (x < 3) {
    { x }
    y = 2* x;
    { x, y }
    x = x + y;
}
{ y }
return(y);
```

En général :

$fv(e) \;\cup\; A1 \;\cup\; A$

```
while (e) {
    A1
    c
    A
}
A
```

Plan

Idée

Une affectation à une variable qui est morte dans la suite peut être supprimée.

Programme original :

```
{ y, z }
x = y + 3;
{ x, z }
y = 2 * x + z;
{ x }
return (x);
```

… optimisé :

```
{ y }
x = y + 3;


{ x }
return (x);
```

A noter :

- { y, z } devient { y }
- Définition plus précise des VV ? $\rightsquigarrow$ plus tard

Détails

- *Affectation :* Dans `A′ x = e; A`,
  on remplace `x = e` par `Skip` si $x \notin A$.
- *Séquence :* On optimise avec les VV “au point actuel” et non avec les VV “à la fin” :
  `x = 1; {x} y = x + 1; {y} return (y);`
  Impossible de supprimer `x = 1;` parce que `x` est encore utilisé dans `x + 1` même si `x` est mort avant `return`
- *Conditionnel :* on optimise chaque branche avec les VV en vigeur à la fin du conditionnel.

Détails

Boucle : On ne peut pas optimiser avec les VV à la fin de la boucle, mais avec les VV au début.

Programme original :

```
{ x, y }
while (x < 3) {
   { x }
   y = 2* x;
   { x, y }
   x = x + y;
}
{ y }
return(y);
```

Optimisation incorrecte :

`y = 2 * x; x = x + y; {y}`

avec résultat :

```
{ x, y }
while (x < 3) {
   { x }
   y = 2* x;
}
{ y }
return(y);
```

Optimisation correcte :

`y = 2 * x; x = x + y; {x, y}`

Donc : aucune optimisation possible

Plan

Règle d'affectation améliorée

Retour à un exemple précédent :

```
{ y, z }
x = y + 3;
{ x, z }          (1)
y = 2 * x + z;
{ x }             (2)
return (x);
```

Si `y` est une variable morte au point (2), alors `z` devrait être morte au point (1).

Règle d'affectation améliorée

Retour à un exemple précédent :

```
{ y, z }
x = y + 3;
{ x, z }              (1)
y = 2 * x + z;
{ x }                 (2)
return (x);
```

Si `y` est une variable morte au point (2), alors
`z` devrait être morte au point (1).

Nouvelle règle :

```
B
x = e;
A
```

avec :

$$B = \begin{cases} fv(e) \cup (A - \{x\}) & \text{si } x \in A \\ A & \text{sinon} \end{cases}$$

Règle d'affectation améliorée

Analyse avec nouvelle règle :

```
{ y }
x = y + 3;
{ x }              (1)
y = 2 * x + z;
{ x }              (2)
return (x);
```

Si y est une variable morte au point (2), alors
z devrait être morte au point (1).

Nouvelle règle :

```
B
x = e;
A
```

avec :

$$B = \begin{cases} fv(e) \cup (A - \{x\}) & \text{si } x \in A \\ A & \text{sinon} \end{cases}$$

Problème de la nouvelle règle

Analyse du programme suivant avec la nouvelle règle :

```
x = 2; y = 1; z = 0;
{ x, y }
while (0 < x) {
  { y }
  x = y;
  { x }
  y = z;
  { x }
}
{ x }
return (x);
```

- Exécutez ce programme.
- Exécutez le programme après optimisation : suppression de `z = 0;`
- Même résultat ?

Analyse de la boucle (1)

Origine du problème : L'effet de `z` sur `x` ne se manifeste qu'après plusieurs itérations. On déplie la boucle une fois pour le bon résultat :

```
x = 2; y = 1; z = 0;
{ x, y, z }
if (0 < x) {
  { y, z }
  x = y;
  { x, z }
  y = z;
  { x, y }
} else Skip;
{ x, y }
while (0 < x) { ... }
{ x }
return (x);
```

Analyse de la boucle (2)

Une vue plus déclarative sur le problème : une boucle

B `while (e) { c }` A

doit satisfaire :

- $fv(e) \subseteq B$
- $A \subseteq B$
  (pour le cas : e est faux)
- $vv(c, B) \subseteq B$ si vv est la fonction des variables vivantes
  (pour le cas : e est vrai et on déplie la boucle une fois)

Autrement dit : on cherche le plus petit B qui satisfait :
$fv(e) \cup A \cup vv(c, B) \subseteq B$
Autrement dit : le plus petit (pré-)point fixe de la fct. f avec
$f(B) = fv(e) \cup A \cup vv(c, B)$
On définit donc : $vv(\mathtt{while(e)\{c\}}, A) = lfp(f)$

Calcul du point fixe

On cherche $B = \mathit{lfp}(f)$.

Constat :

- f est monotone
- il y a un nombre fini de variables $\leadsto$ pas de chaîne ascendante infinie

Donc : il existe k tel que $\mathit{lfp}(f) = f^k(\{\})$

Exemple : Dans notre cas :

- $f^1(\{\}) = \{x\}$
- $f^2(\{\}) = \{x, y\}$
- $f^3(\{\}) = \{x, y, z\}$
- $f^4(\{\}) = f^3(\{\}) = \{x, y, z\} = \mathit{lfp}(f)$

Plan

Plan

6 Systèmes de réduction
- Stratégies de réduction
- Réduction de systèmes équationnels

Motivation et terminologie (1)

Évaluation stricte : Pour évaluer une application $f\ a_1 \ldots a_n$

1. évaluer les arguments $a_1 \ldots a_n$ pour obtenir des valeurs $v_1 \ldots v_n$
2. affecter $v_1 \ldots v_n$ aux paramètres formels de f
3. évaluer le corps de f

... le modèle d'exécution de la plupart des langages de programmation

Exemple :

```
# let f = fun x y z -> x * y + z ;;
val f : int -> int -> int -> int = <fun>
```

$$
\begin{array}{ll}
 & \texttt{f (2 + 3) (2 * 3) 12} \\
\longrightarrow_s & \texttt{f 5 6 12} \\
\longrightarrow_s & \texttt{5 * 6 + 12} \\
\longrightarrow_s & \texttt{42}
\end{array}
$$

Motivation et terminologie (2)

Désavantage de l'évaluation stricte : certains calculs sont éventuellement inutiles :

$$
\begin{array}{ll}
 & \texttt{f 0 (42 * 42) (2 + 3)} \\
\longrightarrow_s & \texttt{f 0 1764 5} \\
\longrightarrow_s & \texttt{0 * 1764 + 5} \\
\longrightarrow_s & \texttt{5}
\end{array}
$$

Évaluation paresseuse : évaluation d'une expression uniquement si (et quand) nécessaire

$$
\begin{array}{ll}
 & \texttt{f 0 (42 * 42) (2 + 3)} \\
\longrightarrow_p & \texttt{0 * (42 * 42) + (2 + 3)} \\
\longrightarrow_p & \texttt{2 + 3} \\
\longrightarrow_p & \texttt{5}
\end{array}
$$

. . . le modèle d'exécution de quelques langages fonctionnels (Haskell, Miranda)

Motivation et terminologie (3)

Situations similaires : conditionnels ou `match` :

```
# let g = fun b x y -> if b then 2 * x else 3 * y;;
val g : bool -> int -> int -> int = <fun>
```

. . . lors de l'évaluation de `g true (3 * 3) (4 * 4)`
Désavantage potentiel (!) de l'évaluation paresseuse : Évaluations multiples.

```
# let h = fun x -> x * x + x ;;
val h : int -> int = <fun>
```

Comparer les deux stratégies sur `h (2 * 2)`

Règles d'évaluation

Format d'une règle typique :

$$\frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'}$$

Lecture : Si on peut réduire N vers N', alors on peut aussi réduire l'application $M\ N$ vers $M\ N'$

Utilisation : de bas en haut :

$\quad$ `(fun x -> x + 2)` $\underline{\texttt{((fun y -> 3 * y) 4)}}$

$\longrightarrow$ `(fun x -> x + 2) 12`

...

parce que `((fun y -> 3 * y) 4)` $\longrightarrow$ `12`

Évaluation stricte : Valeurs (1)

On appelle une valeur toute expression qui ne peut plus être réduite *à la tête* :

- constantes : `3, true, 2.5, []`
- variables : `x, b`
- abstractions : `fun x -> e`, où `e` est une expression quelconque (même réductible)
- application d'un constructeur à des expressions qui sont toutes des valeurs ; paire de valeurs :
  `3 ::[], (3, true)`

Évaluation stricte : Valeurs (2)

Exemples : sont des valeurs :

- `Leaf 3`
- `(fun x -> x + 2)`
- `(fun x -> ((fun y -> 3 * y) x))`
  NB : `((fun y -> 3 * y) x)` est réductible !

ne sont pas de valeurs :

- `(fun x -> x + 2) 3`

Évaluation stricte : Règles (1)

Fragment fonctionnel pur :

- *Réduction de l'argument :*

$$\frac{N \longrightarrow_s N'}{M\ N \longrightarrow_s M\ N'}$$

- *Réduction de la fonction :*

$$\frac{M \longrightarrow_s M'}{M\ v \longrightarrow_s M'\ v}$$

  où `v` est une valeur

- *Appel de fonction :* si `v` est une valeur :
  - Direct : `(fun x -> e1) v` $\longrightarrow_s$ `e1[x` $\leftarrow$ `v]`
  - Expansion de définition : `f v` $\longrightarrow_s$ `e1[x` $\leftarrow$ `v]`
    si `f` est définie par `(fun x -> e1)`

Évaluation stricte : Règles (2)

Exemple de réduction :

```
     (fun x -> ((fun y -> 3 * y) x))
      ((fun z -> 2 + z) 5)
→s   (fun x -> ((fun y -> 3 * y) x)) 7
→s   (fun y -> 3 * y) 7
```

$\longrightarrow_s$ 3 * 7 $\longrightarrow_s$ 21

Non-exemple de réduction :

```
     (fun x -> ((fun y -> 3 * y) x))
      ((fun z -> 2 + z) 5)
```

(pb. : réduction sous `fun`*)*

```
→s   (fun x -> 3 * x) ((fun z -> 2 + z) 5)
```

(pb. : réduction de non-valeur)

```
→s   3 * ((fun z -> 2 + z) 5)
```

Évaluation stricte : Règles (3)

`if` : évaluation “semi-paresseuse” :

- *Réduction de la condition :*

$$\frac{M \longrightarrow_s M'}{\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \longrightarrow_s \texttt{if } M' \texttt{ then } N_1 \texttt{ else } N_2}$$

- *Sélection de la branche :*
  - $\texttt{if true then } N_1 \texttt{ else } N_2 \longrightarrow_s N_1$
  - $\texttt{if false then } N_1 \texttt{ else } N_2 \longrightarrow_s N_2$

`match... with` : Pareil

Évaluation stricte en Caml

L'ordre d'évaluation n'est pas observable en Caml *sauf*

- pour des programmes qui ne terminent pas :

```
# let rec nontermin () : int = nontermin ();;
val nontermin : unit -> int = <fun>
# (fun x -> 42) 5;;
- : int = 42
# (fun x -> 42) (nontermin ());;
Interrupted.    (* ne termine pas *)
```

- pour des programmes avec effet de bord :

```
# (fun x y -> (print_string "bar"; x + y))
     (print_string "foo"; 3)
       (print_string "baz"; 5);;
bazfoobar- : int = 8
```

Évaluation paresseuse : Règles (1)

Fragment fonctionnel pur :

- *Réduction de la fonction :* (N un terme arbitraire)

$$\frac{M \longrightarrow_s M'}{M\ N \longrightarrow_s M'\ N}$$

- *Appel de fonction :* (N un terme arbitraire)
  - Direct : `(fun x -> e1) N` $\longrightarrow_s$ `e1[x` $\leftarrow$ `N]`
  - Expansion de définition : `f N` $\longrightarrow_s$ `e1[x` $\leftarrow$ `N]` si `f` est définie par `(fun x -> e1)`
- *Réduction de l'argument :*

$$\frac{N \longrightarrow_s N'}{f\ N \longrightarrow_s f\ N'}$$

(si `f` est irréductible par $\longrightarrow_s$ et `f` n'est pas une abstraction)

Évaluation paresseuse : Règles (2)

Règles pour `if` et `match` : Comme pour la réduction stricte

Exemple de réduction :

$$
\begin{array}{ll}
& \underline{\texttt{(fun x -> ((fun y -> 3 * y) x))}} \\
& \underline{\texttt{((fun z -> 2 + z) 5)}} \\
\longrightarrow_s & \underline{\texttt{((fun y -> 3 * y) ((fun z -> 2 + z) 5))}} \\
\longrightarrow_s & \texttt{3 * } \underline{\texttt{((fun z -> 2 + z) 5)}} \\
\longrightarrow_s & 3 * (2 + 5) \longrightarrow_s \ldots
\end{array}
$$

Comparer la réduction de :

- `(fun x y -> 3 * x) 5 ((fun z -> z + 3) 4)`
- `(fun x -> x * x) ((fun z -> 4 + z) 38)`

par évaluation stricte / paresseuse

Résumé préliminaire

Nous avons vu :

- différentes stratégies d'évaluation d'un programme
- . . . qui ont des conséquences sur l'efficacité

Quelques remarques supplémentaires :

- Nous avons caché certaines subtilités (notamment : *renommage de variables*)
- Si les deux stratégies terminent, le résultat est le même ($\rightsquigarrow$ *confluence*), hors effets de bord
- Si l'évaluation stricte termine, alors aussi l'évaluation paresseuse
- . . . mais l'inverse n'est pas vrai. Donnez un exemple

Details dans l'introduction au lambda-calcul (niveau Master)
Maintenant : quelques applications pratiques

Structures infinies en Haskell (1)

Haskell (en honneur de H. Curry) est un langage fonctionnel paresseux.
Il existent :

- des séquences finies traditionnelles :

```
Hugs> [1 .. 4]
[1,2,3,4]
```

  (syntaxe inexistante en Caml)

- des séquences infinies :

```
Hugs> [1 ..]
[1,2,3,4,5,6,7,8,9,10,11,12,.....
2008,2009,{Interrupted!}
```

Structures infinies en Haskell (2)

La fonction `take` prend *n* éléments d'une séquence.
Définition :

```
take n [] = []
take n (x:xs) =
   if n == 0 then [] else x:(take (n-1) xs)
```

Note : (:) en Haskell est (: :) en Caml
Utilisation :

```
Hugs> take 3 [1 .. 5]
[1,2,3]
Hugs> take 7 [1 ..]
[1,2,3,4,5,6,7]
```

Note :

- calculer `[1 ..]` ne termine pas
- . . . mais l'évaluation est paresseuse !

Structures infinies en Haskell (3)

Pareil : Fonctions `map`, sélection, ...
Exemple : sélection des multiples de n :

```
select_multiples n xs =
    [x | x <- xs, x `rem` n == 0]
```

Application :

```
Hugs> select_multiples 3 [1 .. 33]
[3,6,9,12,15,18,21,24,27,30,33]
Hugs> take 5 (select_multiples 7 [1 .. ])
[7,14,21,28,35]
```

Structures infinies en Haskell (4)

Le crible d'Ératosthène génère la séquence des nombres premiers

Principe :

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .

Implantation en Haskell :

```
sieve(p:rest) = p:sieve[r|r<-rest, r `rem` p /= 0]
primes = sieve [2..]
```

Application :

```
Hugs> take 12 primes
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Structures infinies en Caml (1)

Type des séquences infinies :

```
type 'a seq =
    Nil
  | Cons of 'a * (unit -> 'a seq)
```

Ici, `unit` et le type dont le seul élément est `()`.
La définition `'a seq` est presque celle des listes traditionnelles (*hypothétique*) :

```
type 'a list =
    []
  | (::) of 'a * 'a list
```

…sauf que `(unit -> ...)` retarde l'évaluation

Structures infinies en Caml (2)

Génération d'une séquence infinie :

```
# let rec fromq k = Cons (k, fun() -> fromq(k+1));;
val fromq : int -> int seq = <fun>
```

`fromq 1` correspond à `[1..]` en Haskell, mais produit uniquement le début de la séquence :

```
# fromq 1 ;;
- : int seq = Cons (1, <fun>)
```

Structures infinies en Caml (3)

Sélection d'éléments :

```
# let rec takeq n = function
    Nil -> []
  | Cons(x, xq) ->
    if n = 0 then [] else x::(takeq (n-1) (xq()));;
  val takeq : int -> 'a seq -> 'a list = <fun>
# takeq 5 (fromq 3) ;;
- : int list = [3; 4; 5; 6; 7]
```

Tracer l'exécution de cet appel

Stratégies de recherche (1)

Des problèmes de recherche se posent dans plusieurs domaines :

- *Logistique :*
  - Trouver le chemin le plus court entre A et B
  - Trouver le chemin le plus large entre A et B (permettant un débit maximal)
- *Jeux :* Trouver des mouvements qui permettent de gagner
- *Résolution de contraintes :* par exemple : charger un avion avec un nombre de paquets
- *Preuves :* Appliquer des règles de manière à prouver une proposition

Stratégies de recherche (2)

Démarche :

- Partir d'une solution partielle
  Ex. : chemin incomplet entre A et B
- Appliquer des opérateur qui étendent la solution partielle
  Ex. : ajouter un tronçon au chemin
- Jusqu'à aboutir à une solution complète

Représentation comme arbre de recherche où

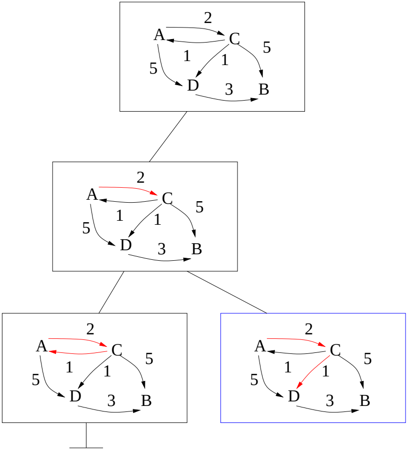
- les noeuds sont les solutions (partielles ou complètes)
- les arcs sont l'application des opérateurs

Il existe différentes manières de construire cet arbre ...

Stratégies de recherche (3)

Recherche en profondeur (*depth first*)

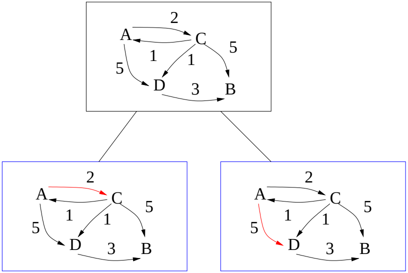
- Un noeud est actif
- Explorer récursivement les fils du noeud actif, de gauche à droite
- Arrêter la descente en cas d'échec



Stratégies de recherche (4)

Recherche en largeur (*breadth first*)

- Tous les noeuds d'un même niveau sont actifs
- Explorer récursivement tous les fils du niveau suivant
- Enlever les noeuds qui ne contribuent pas à une solution



Stratégies de recherche (5)

L'implantation de `depthfirst` et `breadthfirst` s'appuie sur

- une fonction-paramètre `next` qui calcule tous les succésseurs d'un noeud
  Exemple : extension d'un chemin par un tronçon
- une fonction-paramètre `sol` qui teste si un noeud est une solution
  Exemple : vérifier que le chemin va de A à B

Stratégies de recherche (6)

Implantation recherche en profondeur

```
let depthfirst next sol x =
  let rec dfs = function
      [] -> Nil
    | y :: ys ->
        if sol y
        then Cons(y, fun () -> dfs (next y @ ys))
        else dfs (next y @ ys)
  in dfs [x]
```

`x` est la racine de l'arbre de recherche.

Stratégies de recherche (7)

Implantation recherche en largeur

```
let breadthfirst next sol x =
  let rec bfs = function
      [] -> Nil
    | y :: ys ->
        if sol y
        then Cons(y, fun () -> bfs (ys @ next y))
        else bfs (ys @ next y)
  in bfs [x]
```

Plan

Motivation et problématique (1)

But : Faire des preuves équationnelles

Exemple : Étant donné un ensemble d'équations :

- *foldr_filter :* `foldr f (filter p xs) a = foldr (fun x r -> if (p x) then (f x r) else r) xs a`
- *filter_map :* `filter p (map g xs) = foldr (fun x r -> if (p (g x)) then (g x)::r else r) xs []`
- *foldr_map :* `foldr f (map g xs) a = foldr (comp f g) xs a`

Comment montrer :

```
foldr f (filter p (map g xs)) a =
  foldr (comp
           (fun x r -> if p x then f x r else r)
           g) xs a
```

Motivation et problématique (2)

Éléments clés :

- *Orienter les équations*, par exemple
  `foldr f (map g xs) a` $\longrightarrow$
  `foldr (comp f g) xs a`
- *Appliquer les équations* uniquement dans le sens $\longrightarrow$

Les équations orientées sont appelées *règles de réécriture*.

Questions à se poser :

- *Confluence :* Est-ce qu'on peut appliquer les règles dans n'importe quel ordre ?
- *Complétude :* Est-ce que l'orientation des équations n'entraîne pas une perte d'information ?
- *Terminaison :* Est-ce que le processus de réécriture s'arrête ?

Motivation et problématique (3)

Exemple (suite) :
Application de *filter_map* à

```
foldr f (filter p (map g xs)) a = foldr (comp..) xs a
```

produit :

```
foldr f (foldr (...) xs []) a = foldr (comp ..) xs a
```

(*improvable* avec les équations connues)
Cependant : Application de *foldr_filter*

```
foldr (...) (map g xs) a = foldr (comp..) xs a
```

suivi de *foldr_map*

```
foldr (comp..) xs a = foldr (comp..) xs a
```

…vrai par reflexivité

Termes

Les termes sont composés de

- Variables $x, y, z \dots$
- Constantes $a, b, c, \dots, f, g, h \dots$
- Applications : $(f\ a)$

Un sous-terme de la forme `fun -> ...` est traité comme une constante.

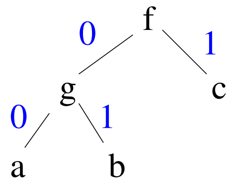
Positions

Positions : Liste de nombres désignant le sous-terme

Notation : sous-terme de t à la position p : $t|_p$

Dans le terme $f\ (g\ a\ b)\ c$

- c est à la position [1]
  donc : $f\ (g\ a\ b)\ c|_{[1]} \ = \ c$
- b est à la position [0; 1]
- $f\ (g\ a\ b)\ c$ est à la position []



Remplacement d'un terme s à la position p dans un terme t

Notation : $t[p \leftarrow s]$

Exemple : $(f\ (g\ a\ b)\ c)\ [[0] \leftarrow (h\ a)] \ = \ f\ (h\ a)\ c$

Implanter la fonction

- `pos` qui calcule le sous-terme d'un terme à une position
- $t[p \leftarrow s]$

Règles

Une règle a la forme $l \longrightarrow r$, où

- l et r sont des termes
- l n'est pas une variable
  (un système avec une règle $x \longrightarrow t$ ne terminerait jamais)
- $fv(r) \subseteq fv(l)$
  (on ne génère pas de variables)
  Exemple : $f\ x \longrightarrow g\ y$ n'est pas valide

Application d'une règle (1)

Application d'une règle à la racine (position `[ ]`) :

Ingrédients :

- Règle de réécriture $l \longrightarrow r$
- Terme t à réécrire

Procédure :

- Trouver une substitution σ telle que $l\sigma = t$
- Le résultat est $r\sigma$

Exemples : Règle : $R \equiv (f\ x\ b \longrightarrow g\ x)$

- Réécrire $(f\ (g\ a)\ b)$
    - Substitution : $\sigma = [x \leftarrow (g\ a)]$
    - Résultat : $g\ (g\ a)$

  On écrit : $(f\ (g\ a)\ b) \longrightarrow_R (g\ (g\ a))$
- Réécriture de $(f\ (g\ a)\ c)$ n'est pas possible

Application d'une règle (2)

Application d'une règle à la position p

Ingrédients :

- Règle de réécriture $l \longrightarrow r$
- Terme t à réécrire
- Position p

Procédure :

- Trouver une substitution σ telle que $l\sigma = t|_p$
- Le résultat est $t[p \leftarrow r\sigma]$

Exemple : Règle : $g\ x \longrightarrow h\ x\ x$

- Réécrire $(f\ (g\ a)\ b)$ à la position $[0]$
  - Substitution : $\sigma = [x \leftarrow a]$
  - Résultat : $(f\ (h\ a\ a)\ b)$
- Réécriture à la position $[1]$ n'est pas possible

Application d'une règle (3)

Attention : La substitution lors de la réécriture s'applique uniquement à la règle et non pas au terme à réécrire

Exemples : Étant donné la règle $g\ x\ a \longrightarrow f\ x$

- Réécrire $h\ (g\ a\ x)\ x$
  Constat : la règle n'est pas applicable
- Réécrire $h\ (g\ b\ a)\ x$
  Le résultat est $h\ (f\ b)\ x$
  et non pas $h\ (f\ b)\ b$

Relations d'équivalence et de réduction (1)

Définitions : Un ensemble de règles $\mathcal{R} = \{l_1 \longrightarrow r_1, \ldots, l_n \longrightarrow r_n\}$ engendre les relations suivantes sur les termes :

- $s \longrightarrow_{\mathcal{R}} t$ si $s \longrightarrow t$ à l'aide d'un $l_i \longrightarrow r_i \in \mathcal{R}$
- $\stackrel{+}{\longrightarrow}_{\mathcal{R}}$ est la fermeture transitive de $\longrightarrow_{\mathcal{R}}$
- $\stackrel{*}{\longrightarrow}_{\mathcal{R}}$ est la fermeture reflexive et transitive de $\longrightarrow_{\mathcal{R}}$
- $\stackrel{*}{\longleftrightarrow}_{\mathcal{R}}$ est la fermeture reflexive, transitive et symmétrique de $\longrightarrow_{\mathcal{R}}$
- Un terme s est *réductible* s'il existe un t tel que $s \longrightarrow_{\mathcal{R}} t$
- Un terme t est une *forme normale* de $\mathcal{R}$ s'il est irréductible pour $\mathcal{R}$.

On omet l'indice $\mathcal{R}$ si l'ensemble de règles est sous-entendu.

Relations d'équivalence et de réduction (2)

Exemples : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $f\ (f\ a) \longrightarrow f\ b$ et $f\ (f\ a) \longrightarrow f\ a$
- $f\ (f\ a) \xrightarrow{+} b$, mais non pas $f\ (f\ a) \xrightarrow{+} f\ (f\ a)$
- $f\ (f\ a) \xrightarrow{*} f\ (f\ a)$ et $f\ (f\ a) \xrightarrow{*} b$
- $f\ a \overset{*}{\longleftrightarrow} f\ b$,
  mais ni $f\ a \xrightarrow{*} f\ b$ ni $f\ b \xrightarrow{*} f\ a$

$$\begin{tikzcd} & f\ (f\ a) \arrow[dl] \arrow[dr] & \\ f\ a & & f\ b \end{tikzcd}$$

Church-Rosser et Confluence (1)

Sous quelles conditions peut-on remplacer un raisonnement équationnel par un raisonnement par réécriture ?

$$\begin{tikzcd} s \arrow[rr, leftrightarrow, "*"] \arrow[dr, dashed, "*"'] & & t \arrow[dl, dashed, "*"] \\ & u & \end{tikzcd}$$

Déf. : Deux termes s et t sont joignables (notation : $s \downarrow t$) s'il existe u tel que $s \xrightarrow{*} u$ et $t \xrightarrow{*} u$.

Déf. : Une relation $\longrightarrow$ a la propriété de Church-Rosser si $\forall s, t.\ s \xleftrightarrow{*} t \Longrightarrow s \downarrow t$

Note historique :

- Alonzo Church (1903-1995)
- John Barkley Rosser (1907-1989)

sont parmi les fondateurs du Lambda-calcul et ont contribué à la théorie des relations de réduction

Church-Rosser et Confluence (2)

Déf. : Une relation $\longrightarrow$ est confluente si

$$\forall s, t_1, t_2 . s \xrightarrow{*} t_1 \wedge s \xrightarrow{*} t_2 \Longrightarrow t_1 \downarrow t_2$$

Déf. : Une relation $\longrightarrow$ est localement confluente si

$$\forall s, t_1, t_2 . s \longrightarrow t_1 \wedge s \longrightarrow t_2 \Longrightarrow t_1 \downarrow t_2$$

$$\begin{tikzcd} & s \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t_1 \arrow[dr, dashed, "*"'] & & t_2 \arrow[dl, dashed, "*"] \\ & u & \end{tikzcd}$$

Confluence

$$\begin{tikzcd} & s \arrow[dl] \arrow[dr] & \\ t_1 \arrow[dr, dashed, "*"'] & & t_2 \arrow[dl, dashed, "*"] \\ & u & \end{tikzcd}$$

Confluence locale

Church-Rosser et Confluence (3)

Théorème (CR –Confluence) : $\longrightarrow$ a la propriété de Church-Rosser si et seulement si $\longrightarrow$ est confluent.
Preuve : voir TD
Fait : “Confluence locale” n’implique pas “confluence”

$$u \longleftarrow s \rightleftarrows t \longrightarrow v$$

Lemme (Newman) : Si $\longrightarrow$ termine, alors $\longrightarrow$ est confluent si et seulement si $\longrightarrow$ est localement confluent.
Preuve : voir TD

Church-Rosser et Confluence (4)

Conséquences : Si $\longrightarrow$ termine,

1. vérifier que $\longrightarrow$ est localement confluent (voir procédure de Knuth-Bendix ...)
2. donc (par le Lemme de Newman) : $\longrightarrow$ est confluent
3. donc (par le théorème CR – confluence) : $\longrightarrow$ a la propriété de Church-Rosser
4. en particulier : pour déterminer si $s \overset{*}{\longleftrightarrow} t$:
    1. réduire $s \overset{*}{\longrightarrow} s'$, où s' est irréductible (la réduction termine !)
    2. réduire $t \overset{*}{\longrightarrow} t'$, où t' est irréductible (de même !)
    3. Si $s' = t'$, alors $s \overset{*}{\longleftrightarrow} t$
    4. Si $s' \neq t'$, alors non $s \downarrow t$, donc non $s \overset{*}{\longleftrightarrow} t$

Paires critiques (1)

Pour une relation de réduction bien fondée, comment peut-on s'assurer de la confluence locale ?

1. **Analyse des paires critiques :** Analyse systématique des points de divergence
2. **Complétion :** Ajout de nouvelles règles pour faire converger les paires critiques $\rightsquigarrow$ procédure de Knuth-Bendix

Exemple : $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

$$\begin{tikzcd} & f\ (f\ a) \arrow[dl, "*"'] \arrow[dr, "*"] & \\ b & & f\ b \end{tikzcd}$$

Paire critique : $(b, f\ b)$

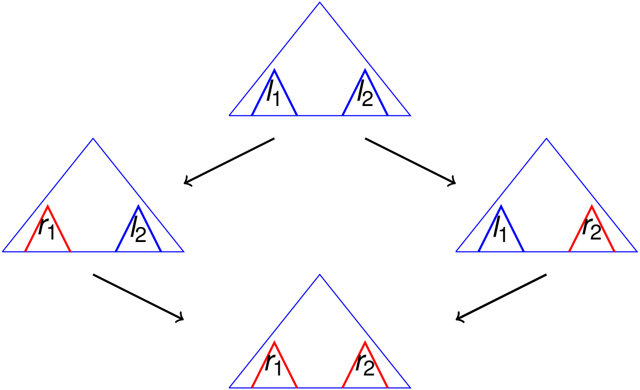
$$\begin{tikzcd} & f\ (f\ a) \arrow[dl, "*"'] \arrow[dr, "*"] & \\ b & & f\ b \arrow[ll] \end{tikzcd}$$

Nouvelle règle : $f\ b \longrightarrow b$

Paires critiques (2)

Cas 1 : Réduction de sous-arbres distincts : jamais de paire critique

Règles $\{l_1 \longrightarrow r_1, l_2 \longrightarrow r_2\}$



Paires critiques (3)

Réduction de sous-arbres distincts

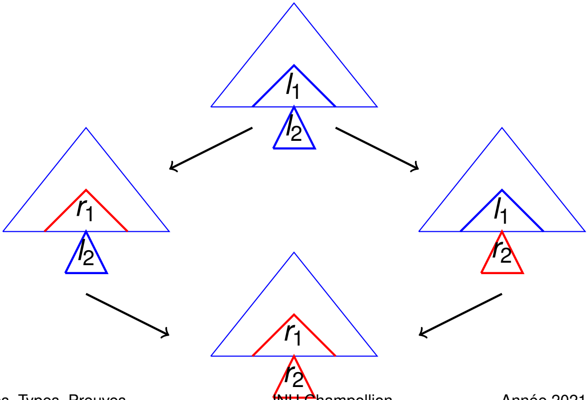
Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $g\ \underline{(f\ (f\ b))}\ (f\ a) \longrightarrow g\ (f\ b)\ \underline{(f\ a)} \longrightarrow g\ (f\ b)\ b$
- $g\ (f\ (f\ b))\ \underline{(f\ a)} \longrightarrow g\ \underline{(f\ (f\ b))}\ b \longrightarrow g\ (f\ b)\ b$

Paires critiques (4)

Cas 2 : Sous-arbres qui se chevauchent (position de variable) :
Jamais de paire critique

- Une variable se trouve à la position de $l_2\sigma_2$ dans l_1
- Le sous-terme $l_2\sigma_2$ n'est pas altéré par la règle $l_1 \longrightarrow r_1$
- …mais $l_2\sigma_2$ peut être dupliqué ou supprimé



Paires critiques (5)

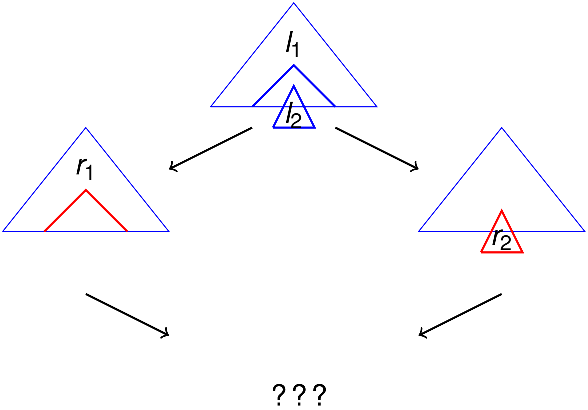
Réduction de sous-arbres distincts

Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $\underline{(f\ (f\ (f\ a)))} \longrightarrow (f\ \underline{(f\ a)}) \longrightarrow (f\ b)$
- $(f\ (f\ \underline{(f\ a)})) \longrightarrow \underline{(f\ (f\ b))} \longrightarrow (f\ b)$

Paires critiques (6)

Cas 3 : Sous-arbres qui se chevauchent



Paires critiques (7)

Réduction de sous-arbres distincts

Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $\underline{(f\ (f\ a))} \longrightarrow \underline{(f\ a)} \longrightarrow b$
- $(f\ \underline{(f\ a)}) \longrightarrow \underline{(f\ b)}$

Pair critique : $((f\ b),\ b)$

Signification informelle :

- On ne peut pas prouver $(f\ b) \overset{*}{\longleftrightarrow} b$ par réduction avec $\mathcal{R}$.
- Pourtant : $(f\ b) \overset{*}{\longleftrightarrow} (f\ (f\ a)) \overset{*}{\longleftrightarrow} (f\ a) \overset{*}{\longleftrightarrow} b$

Solution : Ajouter règle $(f\ b) \longrightarrow b$ à $\mathcal{R}$

Paires critiques : Définition formelle

Soient $l_1 \longrightarrow r_1$, $l_2 \longrightarrow r_2$ deux règles tq. $fv(l_1) \cap fv(l_2) = \{\}$.
Soit p une position de l_1 tq. $l_1|_p$ n'est pas une variable.
Soit $\sigma = mgu(l_1|p, l_2)$.
Alors, $(r_1\sigma, l_1\sigma[p \leftarrow r_2\sigma])$ est la paire critique des deux règles.

Attention aux occurrences multiples de variables :
$\mathcal{R} = \{f\ (f\ x\ y)\ z \longrightarrow f\ x\ (f\ y\ z),\ \ f\ (i\ x')\ x' \longrightarrow e\}$

$$
\begin{tikzcd}
 & f\ (f\ (i\ x')\ x')\ z \arrow[dl] \arrow[dr] & \\
f\ (i\ x')\ (f\ x'\ z) & & f\ e\ z
\end{tikzcd}
$$

Algorithme de complétion (1)

Procédure de Knuth-Bendix

Entrée : un ensemble E d'équations

```
si tous les s = t ∈ E sont orientables
     R := orienter E
sinon échec;

faire
  R' := R;
  pour tout (s, t) ∈ CP(R)
    si on peut orienter (s, t) en l ⟶ r
          R' := R' ∪ {l ⟶ r};
    sinon terminer avec échec
tant que R' ≠ R

renvoyer R'
```

Algorithme de complétion (2)

La procédure $CP(R)$ calcule toutes les paires critiques de R

Résultats possibles de l'algorithme de complétion :

- Un ensemble R. *Signification :*
  - R est confluent
  - La fermeture réflexive, symmétrique et transitive de R est E
- non-terminaison de l'algorithme
- échec : il existe des paires critiques qui ne sont pas orientables. *Signification :* ordre sur des termes éventuellement mal choisi

Algorithme de complétion (3)

Exemple : Soit $E = \{f(f\ x) = f\ x,\ f(f\ x) = g\ x,\ g(g\ x) = x\}$

- Orientation : $R_0 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x\}$
- Paire critique de la superposition de $f(f\ x)$ et $f(f\ x)$ à la pos. [ ] donne $((f\ x), (g\ x))$.
- $R_1 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x,\ (f\ x) \longrightarrow (g\ x)\}$
- Paire critiques de la superposition
  - de $f(f\ x)$ et $(f\ x)$ à la pos. [0] donne $((f\ x), x)$
    réduction : $f(f\ x) \longrightarrow (f\ x)$ et $f(f\ x) \longrightarrow f(g\ x) \longrightarrow g(g\ x) \longrightarrow x$
  - de $f(f\ x)$ et $(f\ x)$ à la pos. [0] donne $((g\ x), x)$
    réduction : $f(f\ x) \longrightarrow (g\ x)$ et $f(f\ x) \xrightarrow{*} x$

Algorithme de complétion (4)

Exemple continué :

- $R_2 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x,\ (f\ x) \longrightarrow (g\ x),\ (f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$
- Pas d'autres paires critiques

On peut simplifier les règles de R_2, par exemple :

- $g(g\ x) \longrightarrow x$ superflu parce que simulable par deux applications de $(g\ x) \longrightarrow x$
- $f(f\ x) \longrightarrow f\ x$ superflu parce que instance de $(f\ x) \longrightarrow x$
- $f(f\ x) \longrightarrow g\ x$ superflu parce que $f(f\ x) \xrightarrow{*} x$ et $g\ x \longrightarrow x$
- $f\ x \longrightarrow g\ x$ superflu parce que $f\ x \longrightarrow x$ et $g\ x \longrightarrow x$

Résultat : $R = \{(f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$

Algorithme de complétion (4)

Exemple continué : Pour
$E = \{f(f\ x) = f\ x,\ f(f\ x) = g\ x,\ g(g\ x) = x\}$
et $R = \{(f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$

1. Preuve de $f(f\ x) = g(f\ x)$
   - Par raisonnement équationnel :
     $f(f\ x) = f(f(f\ x)) = g(f\ x)$
     Nécessite la découverte d'un terme intermédiaire plus complexe
   - De manière automatique, par réduction :
     $f(f\ x) \stackrel{*}{\longrightarrow} x$ et $g(f\ x) \stackrel{*}{\longrightarrow} x$
2. Preuve de $(f\ a) \neq g(f\ b)$, pour constantes a, b
   - $(f\ a) \stackrel{*}{\longrightarrow} a$
   - $g(f\ b) \stackrel{*}{\longrightarrow} b$
   - On conclut $(f\ a) \neq g(f\ b)$, parce que $\longrightarrow$ est confluent et $a \neq b$