Algorithmes, Types, Preuves

Jan-Georg Smaus, Martin Strecker

Univ. J.-F. Champollion / IRIT

Année 2014/2015

Plan

Plan

Théorie des langages - pourquoi ?

Permet de répondre aux questions . . .

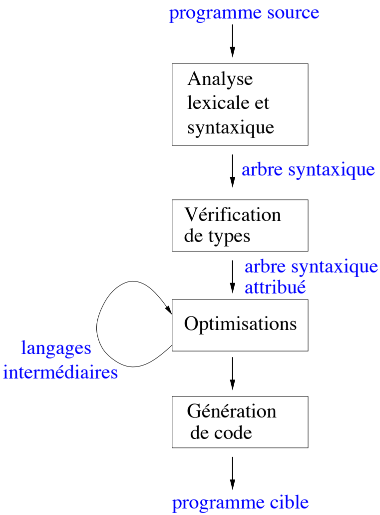
. . . d'un utilisateur d'un langage de programmation :

- 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*
    $\rightsquigarrow$ 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*
  ($\leadsto$ pas d'erreur de type lors de l'exécution)
- Sous-typage
  - classes $\leftrightarrow$ classes
  - interfaces $\leftrightarrow$ interfaces

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

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

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

Présentation des langages

Dans la suite, on étudiera le système de typage d'un langage impératif.
On introduit des langages de complexité croissante :

1. langage avec expressions et instructions simples : $\mathcal{L}_e$
2. langage avec fonctions : $\mathcal{L}_f$
3. langage avec sous-typage : $\mathcal{L}_s$

Plan

Un langage impératif simplifié

Pour voir plus clair, nous distinguons entre

Expressions

- qui produisent une valeur
- qui ne changent pas l'état du programme

Instructions

- qui modifient l'état du programme

Exemple : Contrairement à C, nous n'acceptons pas

- `x + 2; y = 5;`
- `x = x + (x = 4);`

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

Expressions

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

Instructions

C	: :=	$v =$ E	(Affectation)
	|	C ; C	(Séquence)
	|	`if` E `then` C `else` C	
	|	`while` E `do` C	

Expressions et instructions de $\mathcal{L}_e$ (2)

On peut maintenant écrire des expressions

- *bien typées :*
  - $3 + (5 - 2)$
  - $((4 * 2) < 42) \&\& ((4 + 2) == 5)$
- *mal typées :*
  - $((4 * 2) < \mathit{true}) + 7$

Est-ce que $((v1 * 2) < 42) || v2$ *est bien typée ?*

Expressions et instructions de $\mathcal{L}_e$ (2)

On peut maintenant écrire des expressions

- *bien typées :*
  - $3 + (5 - 2)$
  - $((4 * 2) < 42) \&\& ((4 + 2) == 5)$
- *mal typées :*
  - $((4 * 2) < \mathit{true}) + 7$

Est-ce que $((v1 * 2) < 42) || v2$ *est bien typée ?* Ceci dépend des déclarations :

- Oui, si `v1 : int` et `v2 : bool`
- Autrement : Non

Types et déclarations

Types Nous utilisons les types suivants :

- `bool` pour les valeurs de vérité
- `int` pour les entiers
- `void` pour des instructions bien typées

Déclarations

- Une déclaration associe un type (`bool` ou `int`) à une variable
- Il n'est pas possible de déclarer une variable de type `void`

Environnements

Un environnement est une liste (*ordre important !*) qui

- associe un type à une variable
- représente les déclarations en vigueur à une position du programme.

Les environnements peuvent varier, selon la position :

```
int n;
int f1(int b) {
  int n;
  //(1)
  return (b + n); }
```

```
int n;
int f2(int a) {
   bool b;
   //(2)
   n = a;
   return 0; }
```

- Env1 = [(int n), (int b), (int n)]
- Env2 = [(bool b), (int a), (int n)]

Jugements de typage

Format du jugement : $Env \vdash e : T$, où

- Env est un environnement
- e une expression (resp. une instruction)
- T un type

Lecture informelle :

> *Dans l'environnement Env, l'expression e est bien typée et a le type T.*

Exemples :

- $[(int\ n)] \vdash n + 2 : int$
- il n'existe aucun T tel que $[(bool\ b)] \vdash b + 2 : T$

Format des règles de typage

Une règle de typage a la forme

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

où

- $P_1 \ldots P_n$ sont les *préconditions*
- C est la conclusion

Lecture informelle :

> *Si les préconditions $P_1 \ldots P_n$ sont satisfaites, alors on peut conclure C.*

Règles de typage (1)

Définition par induction sur la structure des expressions/instructions :

Constantes :

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

Variables :

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

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

Règles de typage (2)

Opérations unaires et binaires :

$$\frac{Env \vdash e_1 : int \quad Env \vdash e_2 : int}{Env \vdash e_1 + e_2 : int}$$

$$\frac{Env \vdash e : bool}{Env \vdash !e : bool}$$

Compléter !

Règles de typage (3)

Toutes les instructions *c* sont typées avec *void*

Donc, $Env \vdash c : void$ signifie : *c* est bien typée

Affectation :

$$\frac{tp(v, Env) = T \quad Env \vdash e : T}{Env \vdash (v = e) : void}$$

Boucle :

$$\frac{Env \vdash e : bool \quad Env \vdash c : void}{Env \vdash \texttt{while}\ e\ \texttt{do}\ c : void}$$

Compléter !

Exemples de typage

- Montrer : L'expression `!((3 + 4) < 34)` est bien typée
- L'expression `(x == 2) || (y == z) || b` est bien typée dans quels environnements ?
- Montrer : `while (1) do x = x + 1;` n'est pas bien typé

Plan

Syntaxe de $\mathcal{L}_f$ (1)

Déclarations
D ::= $T\ v$ $(v \in \mathcal{V}, T \in \mathit{types})$
Définition de fonctions F ::= T fn (D*) { D* ; C ; `return` E ; }
composée de :

- type de résultat T
- nom de la fonction *fn* (un identificateur)
- une liste $D*$ de déclarations de paramètre (séparées par ',')
- une liste $D*$ de déclarations de variables locales (sép. ' ;')
- une instruction C
- `return` avec une expression E

Syntaxe de $\mathcal{L}_f$ (2)

Définition de programmes

P : := D* F*

composée de :

- une liste D^* de déclarations de variables globales
- une liste F^* de définitions de fonctions

Instructions : on ajoute des appels de fonction avec affectation :

C : := . . .
| v = *fn* (E*) $v \in \mathcal{V}$, *fn* nom de fct.

Environnements pour $\mathcal{L}_f$ (2)

Un environnement *Env* a maintenant deux composants :

- *Env.vars* (comme avant) : liste qui associe des types à des variables
- *Env.fns* (nouveau) : liste composée des profils des fonctions du programme

Un profil de fonction est un en-tête de programme, sans les noms de variable.
Ex. : le profil de la fonction

```
int f (int n, bool b) { ... }
```

est

```
int f (int, bool)
```

Vérification de types (1)

1. Analyse syntaxique du programme P
2. Construction d'un environnement initial Env_{in}, avec
   - $Env_{in}.vars$: déclar. des var. globales de P
   - $Env_{in}.fns$: profil des fonctions de P
3. Vérification des corps des fonctions.
   Pour chaque f définie dans P, faire :
   1. Construire environnement Env_f, avec
      - $Env_f.vars$:
        $Env_{in}.vars$, plus décl. des paramètres et des var. locales de f
      - $Env_f.fns$: le même que $Env_{in}.fns$
   2. Vérifier le corps de f sous l'environnement Env_f
      $\leadsto$ légère adaptation des règles de typage

Vérification de types (2)

Adaptation des règles

- Modifier les règles pour prendre en compte l'environnement plus complexe
- Ajouter une règle pour l'appel des fonctions
- Ajouter une règle pour `return` *E*

Points délicats

- Pourquoi n'est-il pas possible de faire la vérification de types en même temps que l'analyse syntaxique ?
- Comment sont traitées différentes occurrences de variables du même nom ?

Plan

Plan

2. Programmes fonctionnels
   - Types de programmes fonctionnels
   - Inférence de types

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 :

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

Application :

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

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

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)
```

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

Polymorphisme (5)

Définir les types polymorphes

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

Polymorphisme (6)

Une substitution $\sigma = [\alpha_1 \leftarrow S_1, \dots, \alpha_n \leftarrow S_n]$, appliquée à un type T, remplace en parallèle toute occurrence de α_i dans T par S_i.
Notation : $T\sigma$
Exemple : $('\mathtt{a\ list})['\mathtt{a} \leftarrow \mathit{bool}] = \mathtt{bool\ list}$
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`

Polymorphisme (7)

Règle de typage : Modification de la règle d'application :

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

Exemple :

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

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

Plan

2. Programmes fonctionnels
   - Types de programmes fonctionnels
   - Inférence de types

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

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)

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

- *Constantes :* $PT(Env, c) = (Env, T)$
  si T est le type "naturel" de c
- *Variables :* $PT(Env, x) = (Env, T)$
  si $tp(x, Env) = T$
  (sinon erreur : variable non déclarée)
- *Abstraction :* si $PT((x : A) :: Env,\ e) = ((x : A') :: Env', B)$
  pour une nouvelle variable de type A
  alors $PT(Env, \texttt{fun}\ x \rightarrow e) = (Env', A' \rightarrow B)$
- *Application :* si $PT(Env, f) = (Env_1, F)$ et $PT(Env_1, a) = (Env_2, A)$
  et $\sigma = Unif(F, A \rightarrow B)$ pour une nouvelle var. de type B
  alors $PT(Env,\ f\ a) = (Env_2\sigma, B\sigma)$
  si échec de l'unification, alors $PT(Env,\ f\ a) = fail$

Algorithme d'inférence (3)

Substitution

- appliquée à un type ($B\sigma$) : comme avant
- appliquée à un environnement ($Env\sigma$) : appliquer à chaque type de l'environnement :
  $[(x_1 : A_1); \ldots (x_n : A_n)]\sigma = [(x_1 : A_1\sigma); \ldots (x_n : A_n\sigma)]$

Algorithme d'inférence : Exemples (1)

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

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

Unification (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 : Des algorithmes d'unification seront étudiés plus tard.

Unification (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. $\text{Unif}(\texttt{int -> (int -> bool), (int * int) -> bool})$

   $\quad \text{Unif}(\texttt{int, (int * int)}) = \text{fail}$

   $= \text{fail}$

Plan

Plan

3 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 :

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

si F et G sont unifiables avec unificateur σ

Exemple :

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

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

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 $*$: $\sigma = [x \leftarrow 0]$
- $\leadsto$ 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 : $2 + (c * 5) \neq (2 + c) * 5$

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

Termes

Les termes t du langage ont la forme suivante :

$$
\begin{array}{llll}
t & ::= & c & \text{(constantes)} \\
 & | & v & \text{(variables)} \\
 & | & (t\ t) & \text{(application)}
\end{array}
$$

Les *constantes* peuvent être

- des constantes traditionnelles : 2, *true*, *c*, *d*
- des noms de fonctions : *f*, *g*

Conventions :

- Noms des *variables* du fin de l'alphabet : $x, y, z, \ldots$
- l'application associe à gauche : $p\ x\ 2 = ((p\ x)\ 2)$

Substitution (1)

Une substitution σ est une fonction qui associe un terme à chaque variable.
Notation : $\sigma = [x_1 \leftarrow t_1; \ldots x_n \leftarrow t_n]$

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

- Constantes : $c\sigma = c$
- Variables : $v\sigma = \sigma(v)$
- Application : $(t_1\ t_2)\sigma = ((t_1\sigma)\ (t_2\sigma))$

Substitution (2)

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

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

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 :* $(\{t\overset{?}{=}t\} \cup E, S) \Longrightarrow (E, S)$
- *Decompose :* $((s_1\ s_2)\overset{?}{=}(t_1\ t_2) \cup E, S) \Longrightarrow (\{s_1\overset{?}{=}t_1, s_2\overset{?}{=}t_2\} \cup E, S)$
- *Fail :* $((s_1\ s_2)\overset{?}{=}c \cup E, S) \Longrightarrow \mathit{fail}$
- *Clash :* $(c_1\overset{?}{=}c_2 \cup E, S) \Longrightarrow \mathit{fail}$
  si c_1 et c_2 sont des constantes différentes
- *Eliminate :* $(x\overset{?}{=}t \cup E, S) \Longrightarrow (E[x \leftarrow t]; S[x \leftarrow t] \cup \{x = t\})$
  si $t \neq x$ et $x \notin FV(t)$
- *Check :* $(x\overset{?}{=}t \cup E, S) \Longrightarrow \mathit{fail}$
  si $t \neq x$ et $x \in FV(t)$

Algorithme (3)

Notes :

- Les règles *Fail, Eliminate, Check* ont des variantes symétriques :
  Fail' : $(c \stackrel{?}{=} (s_1 \; s_2) \cup E, S) \Longrightarrow$ *fail*
  etc.

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 :

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

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

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

Algorithme (5)

Exemple :

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

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 $\leadsto$ 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 $(E, S) \Longrightarrow$ *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

3 Unification
- Motivation et terminologie
- Unification syntaxique
- Unification modulo théories

Différentes notions d'égalité

Unification syntaxique :

- prend en compte uniquement la structure des termes

Exemple : Syntaxiquement, on ne peut pas unifier :

- $(x_f\ 5) = 5$
- $x \oplus 2 = y \oplus 3$

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 x_f telle que $(x_f\ 5) = 5$
Deux réponses possibles :

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

4 Induction
- Induction structurelle
- Étude de cas : Arbres de recherche
- Rule induction
- Induction fondée

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

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

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
  $\leadsto$ 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.

$\leadsto$ 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 $\texttt{of}$ $T_1 * \dots * T_n$, montrer $\forall x_1 \dots x_n.P(C_b(x_1, \dots, x_n))$
- Pour tout constructeur inductif C_i $\texttt{of}$ $\underbrace{T * \dots T}_{k\times} * T_1 * \dots T_m$, montrer $\forall x_1 \dots x_m.P(t_1) \wedge \dots \wedge P(t_k) \longrightarrow P(C_i(t_1, \dots, t_k,\ x_1, \dots, x_m))$

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

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 : $\forall t.\texttt{swap}(\texttt{swap}(t)) = t$

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

Compléter la preuve !

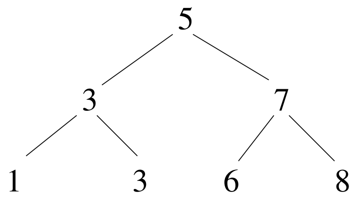
Plan

4 Induction
- Induction structurelle
- Étude de cas : Arbres de recherche
- Rule induction
- Induction fondée

Arbres de recherche simples : Définition

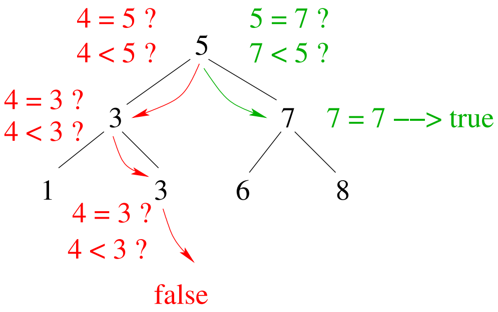
Un arbre binaire de recherche est un arbre binaire dont les noeuds sont ordonnées selon un critère.
Exemple de critère : Récursivement,

- les valeurs du sous-arbre gauche sont inférieures à la racine,
- les valeurs du sous-arbre droit sont supérieures ou égales à la racine



Arbres de recherche simples : Recherche d'un élément

Donné : un élément *e* et un arbre *t*
Requis : Vérifier si *e* est dans *t*
Exemples : recherche de 4 et 7

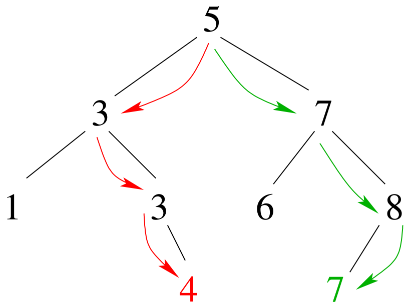


Variation :

- *Donné :* une clé *k* et un arbre *t*
- *Requis :* Récupérer la valeur associe à *k* dans *t*

Arbres de recherche simples : Insertion d'un élément

Donné : un élément *e* et un arbre *t*
Requis : Insérer *e* dans *t*, en préservant l'ordre
Exemples : Insertion de 4 et 7



à noter : l'insertion se fait toujours aux feuilles

Arbres de recherche simples : Propriétés

Dans le meilleur des cas :

- L'arbre est *équilibré :*
  Récursivement, les sous-arbres gauches et droits ont le même nombre de noeuds
- Dans ce cas : Un arbre de $2^n - 1$ noeuds a la *profondeur* n
- . . . la recherche prend au maximum n pas.

Plan

Plan

4 Induction
- Induction structurelle
- Étude de cas : Arbres de recherche
- Rule induction
- Induction fondée

Plan

Plan

Motivation et terminologie (1)

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

1. évaluer les arguments $a_1 \dots a_n$ pour obtenir des valeurs $v_1 \dots v_n$
2. affecter $v_1 \dots v_n$ 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 innécessaires :

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

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 :

$$\begin{array}{ll} & \texttt{(fun x -> x + 2)}\ \underline{\texttt{((fun y -> 3 * y) 4)}} \\ \longrightarrow & \texttt{(fun x -> x + 2) 12} \\ \dots & \end{array}$$

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 ; pair 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)`

$\longrightarrow_s$ `(fun x -> ((fun y -> 3 * y) x)) 7`

$\longrightarrow_s$ `(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`*)*

$\longrightarrow_s$ `(fun x -> 3 * x) ((fun z -> 2 + z) 5)`

(pb. : réduction de non-valeur)

$\longrightarrow_s$ `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

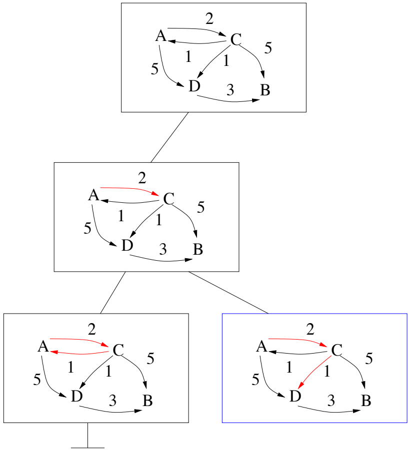
- les noeuds sont les solutions (partielles ou complètes)
- les arcs sont l'application des opérateurs

Il existe différentes manières de construire cet arbre ...

Stratégies de recherche (3)

Recherche en profondeur (*depth first*)

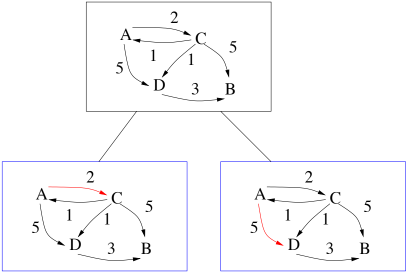
- Un noeud est actif
- Explorer récursivement les fils du noeud actif, de gauche à droite
- Arrêter la descente en cas d'échec



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

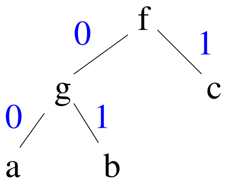
- Variables $x, y, z \ldots$
- Constantes $a, b, c, \ldots, f, g, h \ldots$
- Applications : $(f\ a)$

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 $l_i \longrightarrow r_i \in \mathcal{R}$
- $\xrightarrow{+}_{\mathcal{R}}$ est la fermeture transitive de $\longrightarrow_{\mathcal{R}}$
- $\xrightarrow{*}_{\mathcal{R}}$ est la fermeture reflexive et transitive de $\longrightarrow_{\mathcal{R}}$
- $\xleftrightarrow{*}_{\mathcal{R}}$ 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 $f\ (f\ a) \xrightarrow{*} b$
- $f\ a \xleftrightarrow{*} f\ b$,
  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”

$$\begin{tikzcd} u & s \arrow[l] \arrow[r, bend left] & t \arrow[l, bend left] \arrow[r] & v \end{tikzcd}$$

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 $s \stackrel{*}{\longleftrightarrow} t$:
    1. réduire $s \stackrel{*}{\longrightarrow} s'$, où s' est irréductible (la réduction termine !)
    2. réduire $t \stackrel{*}{\longrightarrow} t'$, où t' est irréductible (de même !)
    3. Si $s' = t'$, alors $s \stackrel{*}{\longleftrightarrow} t$
    4. Si $s' \neq t'$, alors non $s \downarrow t$, donc non $s \stackrel{*}{\longleftrightarrow} t$

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 $\leadsto$ 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 : $(b, f\ b)$

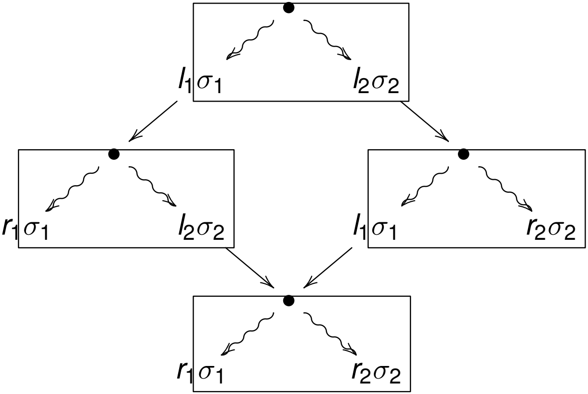
$$\begin{tikzcd}
 & f\ (f\ a) \arrow[dl, "*"'] \arrow[dr, "*"] & \\
b & & f\ b \arrow[ll]
\end{tikzcd}$$

Nouvelle règle : $f\ b \longrightarrow b$

Paires critiques (2)

Cas 1 : Réduction de sous-arbres distincts

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



Jamais de paire critique

Paires critiques (3)

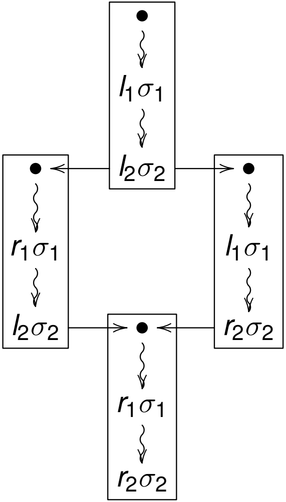
Réduction de sous-arbres distincts

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



- 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

Jamais de paire critique

Paires critiques (5)

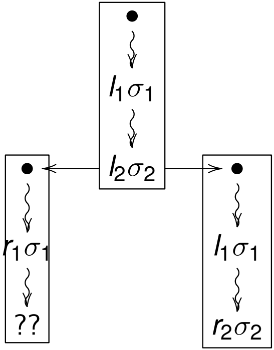
Réduction de sous-arbres distincts

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



- La position de $l_2\sigma_2$ dans l_1 est non-variable
- Le sous-terme $l_2\sigma_2$ peut disparaitre par application de la règle $l_1 \longrightarrow r_1$
- …par conséquent, $l_2 \longrightarrow r_2$ peut ne plus être applicable
- paire critique possible entre $r_1\sigma_1$ et $l_1\sigma_1[r_2\sigma_2]$

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 $(f\ b) \stackrel{*}{\longleftrightarrow} b$ par réduction avec $\mathcal{R}$.
- Pourtant : $(f\ b) \stackrel{*}{\longleftrightarrow} (f\ (f\ a)) \stackrel{*}{\longleftrightarrow} (f\ a) \stackrel{*}{\longleftrightarrow} b$

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

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 :
     $f(f\ x) \xrightarrow{*} x$ et $g(f\ x) \xrightarrow{*} x$
2. Preuve de $(f\ a) \neq g(f\ b)$, pour constantes a, b
   - $(f\ a) \xrightarrow{*} a$
   - $g(f\ b) \xrightarrow{*} b$
   - On conclut $(f\ a) \neq g(f\ b)$, parce que $\longrightarrow$ est confluent et $a \neq b$

Plan

Motivation (1)

Assurer la terminaison d'une fonction est primordial pour

- assurer que la fonction fournit un résultat
- permettre de raisonner sur la correction de la fonction

Contre-exemple : La fonction définie par

```
let rec f n = (f n) + 1
```

ne termine pas.
Vue comme équation, la définition est inconsistante. Elle permet de dériver `0 = 1`.

Motivation (2)

Rappel : Nous avons vu une correspondance entre :

- fonctions définies par récursion structurelle
- preuves par induction structurelle

La terminaison est conséquence immédiate de la réc. structurelle.
Dans la suite : Correspondance entre :

- fonctions définies par récursion bien fondée
- preuve par induction bien fondée

Préalable : Identification d'une *relation bien fondée*

Notions de base

Une relation d'ordre $\longrightarrow$ est bien fondée s'il n'existe pas de chaîne infinie $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$
Cette définition assure l'indépendance par rapport à la stratégie de réduction.
Contre-exemples : La relation suivante n'est pas bien fondée :

$$\begin{tikzcd}
t_0 \arrow[r] \arrow[d] & t_1 \arrow[r] \arrow[dl] & t_2 \arrow[r] \arrow[dll] & \ldots \arrow[dlll] \\
t & & &
\end{tikzcd}$$

Une relation cyclique n'est pas bien fondée :

$$\begin{tikzcd}
 & t_2 \arrow[dl] & \\
t_0 \arrow[rr] \arrow[dr] & & t_1 \arrow[ul] \arrow[dl] \\
 & t &
\end{tikzcd}$$

Récursion bien fondée sur Nat

Rappel : La définition de f par récursion structurelle suit le schéma :

- $f(0)$: cas d'arrêt
- $f(n+1)$ se réduit à $f(n)$

La récursion bien fondée sur les nombres naturels suit le schéma :

- $f(n_0) \dots f(n_k)$: cas d'arrêt
- Pour les $n \notin \{n_0, \dots n_k\}$: $f(n)$ se réduit à $f(m)$, pour $m < n$

Exemple :

```
let rec even n =
  if (n = 0) then true
  else if (n = 1) then false
  else even (n-2)
```

Termine parce que $n \neq 0 \wedge n \neq 1 \Longrightarrow n - 2 < n$
et $<$ est bien fondée sur Nat.

Induction bien fondée sur Nat (1)

Rappel : Induction structurelle sur les nombres naturels :
Pour montrer $\forall n.P(n)$

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

Induction bien fondée sur les nombre naturels :
Pour montrer $\forall n.P(n)$

- montrer pour tout n : $(\forall m.n > m \Longrightarrow P(m)) \Longrightarrow P(n)$

Induction bien fondée sur Nat (2)

Preuve de : $even(n) = (n \mod 2 = 0)$
Distinguer les cas :

- $n = 0$: application de la déf. de `even` et simplification
- $n = 1$: pareil
- $n \neq 0 \wedge n \neq 1$: Application de la règle d'induction fondée, où $P(n)$ est
  $(n \neq 0 \wedge n \neq 1 \Longrightarrow even(n) = (n \mod 2 = 0))$
  Compléter la preuve !

Induction bien fondée

Généralisation : Pour tout ensemble A avec une relation bien fondée $\longrightarrow$, on peut définir une règle d'induction fondée.

Pour montrer $\forall x \in A.P(x)$

- montrer pour tout $x \in A$: $(\forall y \in A.x \xrightarrow{+} y \Longrightarrow P(y)) \Longrightarrow P(x)$

Ici :

- $x \in A, y \in A$ généralisent $n : Nat, m : Nat$
- $\xrightarrow{+}$ sur A généralise $<$ sur Nat

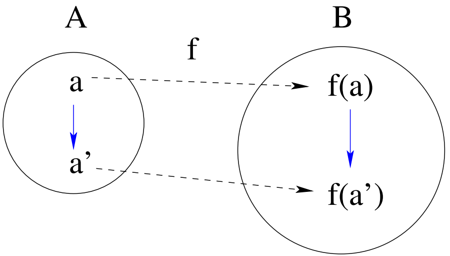
Construction d'ordres bien fondés

Étant donné un ordre bien fondé (tel que $(Nat, >)$), on peut construire d'autres par :

- image inverse
- ordre lexicographique
- ordre multi-ensembliste

Ordres bien fondés : Image inverse (1)

Soit $(A, >_A)$ un domaine avec une relation $>_A$ sur A.
Soit $(B, >_B)$ un domaine avec $>_B$ bien fondée sur B.
Soit $f : A \Rightarrow B$ une fonction *monotone*, c.à.d.
$a >_A a'$ implique $f(a) >_B f(a')$
Alors $>_A$ est bien fondée sur A.



Ordres bien fondés : Image inverse (2)

Exemple : La fonction `even` sur des nombres entiers :

```
let rec even n =
  if (n = 0) then true
  else if ((n = 1) or (n = -1)) then false
  else if (n < 0) then even(n + 2) else even (n-2)
```

Quel est la fonction $f : Int \Rightarrow Nat$ qui justifie la terminaison ?

Ordres bien fondés : Lexicographique (1)

Étant donnés $(A, >_A)$ et $(B, >_B)$, l'ordre lexicographique $_{lex}$ est défini sur $A \times B$ par :
$(a, b) >_{lex} (a', b') \equiv (a >_A a') \vee ((a = a') \wedge (b >_B b'))$

Exemple : avec l'ordre naturel sur les caractères / Nat, on a :
$('b', 2) >_{lex} ('a', 1)$ et $('b', 2) >_{lex} ('a', 3)$ et $('b', 2) >_{lex} ('b', 1)$
mais non pas $('b', 2) >_{lex} ('b', 3)$

Lemme : Si $>_A$ et $>_B$ sont bien fondées, alors aussi $>_{lex}$

Preuve : Par contradiction : Supposez qu'il existe une chaîne infinie
$(a_0, b_0) >_{lex} (a_1, b_1) >_{lex} \dots$
Complétéz la preuve !

Ordres bien fondés : Lexicographique (2)

La fonction de Ackermann est définie par

```
let rec ack = function
    (0, y) -> y + 1
  | (x, 0) -> ack(x - 1, 1)
  | (x, y) -> ack(x - 1, ack(x, y - 1))
```

Historique : Définie en 1928 par W. Ackermann comme exemple d'une fonction

- qui termine – donnez la preuve !
- mais qui n'est pas définissable par le schéma de récursion primitive de premier ordre.

Ordres bien fondés : Multi-ensembliste

Étant donné $(A, >_A)$.
Soit $\mathcal{M}$ l'ensemble des multi-ensembles sur A.
L'ordre $>_{mult}$ sur $\mathcal{M}$ est défini par :
$M >_{mult} N$ s'il existe des multi-ensembles X, Y tels que :

- $N = (M - X) \cup Y$
- X est non-vide
- $\forall y \in Y . \exists x \in X . x >_A y$

Exemples :

- $\{5, 3, 2\} >_{mult} \{5, 2, 2\}$
  ici : $X = \{3\}$, $Y = \{2\}$
- $\{5, 3, 2\} >_{mult} \{4, 4, 4, 2, 2\}$
  ici : $X = \{5, 3\}$, $Y = \{4, 4, 4, 2\}$
- mais non pas : $\{5, 3, 2\} >_{mult} \{5, 5\}$

Refaire la preuve de terminaison de l'algorithme d'unification !