Université de Montréal

Erreurs et Types dépendants

Évolution des langages liée aux erreurs

Volonté de détecter et éliminer les erreurs

Limité par le désire de généralité

Donc limité aux cas d'"erreur évidente"

Évidence fournie par redondance

On ne peut pas deviner l'intention du programmeur

Détection&élimination d'erreurs

Langage machine: aucune détection d'erreurs

Assembleur: élimine les erreurs d'encodage

Assembleur: détecte les erreurs de syntaxe

C: détecte certaines erreurs sémantiques simples

Java: détecte plus d'erreurs sémantiques simples

Java: élimine les pointeurs fous et accès hors des bornes

Haskell: élimine encore plus d'erreurs sémantiques

STM: détecte les conditions de course et élimine les inter-blocages

¡détection $\Rightarrow$ redondance! ¡élimination $\Rightarrow$ contraintes!

Erreurs à l'exécution

Types éliminent beaucoup d'erreurs d'exécution, mais pas toutes:

Exemple: `head : List` $\alpha \to \alpha$

- Renvoyer une indication d'échec: `List` α $\to$ `Maybe` α
- Demander l'aide de l'appelant: `List` $\alpha \to \alpha \to \alpha$
- Signaler une exception

Autres exemples:

- `x / y` lorsque `y` est 0
- Accès hors des bornes d'un tableau
- Multiplication de matrices de tailles incompatibles

Fonctions totales

Une *fonction totale* est une fonction qui renvoie toujours un résultat

`head : List a` $\rightarrow$ `a` n'est pas totale

Deux manières de rendre une fonction totale:

1. Compléter son co-domaine en renvoyant des erreurs explicites

   E.g. `head : List` $\alpha \rightarrow$ `Maybe` α

2. Restreindre son domaine aux cas qui fonctionnent

   E.g. `head : NonEmptyList` $\alpha \rightarrow \alpha$

On peut considérer les exceptions comme des erreurs explicites,

si elles sont mentionnées dans le type:

E.g. `head : List` $\alpha \rightarrow \alpha$ *`signals`* `Empty`

Type des tableaux

En Java: `int[] myarray`

+ Taille des tableaux dynamique
- Tableaux accompagnés de leur taille
- Accès coûte une vérification de borne

En Modula-2: `ARRAY [0..12] OF INTEGER`

+ Permet une représentation efficace
+ Permet un accès efficace

- - Taille des tableaux fixée à la compilation

Statique et dynamique à la fois?

On veut des tableaux de taille dynamique

Mais on veut que le compilateur la connaisse (assez)

```
make_array : (n : Int) → α → Array α n;
make_matrix : (rows : Int) → (cols : Int)
                 → Matrix rows cols;
make_array (x + 1) 0.0 : Array Float (x + 1);
```

`make_array` a un *type dépendent* parce le *type* de la valeur renvoyée dépend de la *valeur* de son argument!

Types précis

On peut donner des co-domaines plus précis:

```
array_concat : Array α n1 →  Array α n2
               →  Array α (n1 + n2);
array_zip : Array α n1 →  Array β n2
     →  Array (α,β)
               (if n1 < n2 then n1 else n2);
```

ou contraindre le domaine pour éliminer des cas d'erreur:

```
multiply : Matrix rows n → Matrix n cols
            → Matrix rows cols;
array_zip' : Array α n →  Array β n
              →  Array (α,β) n;
```

Manipuler des Types

En Haskell, les types sont implicites

```
id :: α → α                    nil :: List α
```

Mais dans les λ-calculs sous-jacents, on a en fait:

```
id : (α : Type) → α → α;
id α (x : α) = x;

nil : (α : Type) → List α;
list1 : (α : Type) → α → List α;
list1 α x = cons α x (nil α);
```

Les types son reçus et manipulés comme d’autres données!

GADTs et types indexés

Le type `List` α est *paramétrique* en α:

```
type List α
  | nil : List α
  | cons : α → List α → List α;
```

Le *Generalized Algebraic DataType* `List` α `n` est *indexé* par `n`:

```
type Listn α n
  | nil : Listn α 0
  | cons : α → Listn α n → Listn α (n + 1);
```

Arbres équilibrés

On peut utiliser les indexes pour imposer des contraintes fortes:

```
type Tree α depth
  | empty : Tree α 0
  | leaf  : α → Tree α 0
  | node  : Tree α d → α → Tree α d
            → Tree α (d + 1)
```

L'indexe `depth` garde trace de la profondeur

Dans `node` il garanti aussi l'équilibre de l'arbre

Ce n'est pas magique: le système de type ne peut que rejeter du code!

Ces annotations expliquent l'*intention* du programmeur

Red-Black trees

Ça se généralise bien à des contraintes plus complexes:

```
type RBtree α color depth
  | leaf  : RBtree α Black 0
  | rnode : RBtree α Black d → RBtree α Black d
            → RBtree α Red d
  | bnode : RBtree α c1 d → RBtree α c2 d
            → RBtree α Black (d + 1)
```

L'indexe `color` indique la couleur de la racine de l'arbre

L'indexe `depth` garde trace de la profondeur (de nœuds noirs)

Ces déclarations encodent les règles qui définissent un *red-black tree*

Université de Montréal

Tableaux sûrs et efficaces

Des types précis permettent un accès efficace et sûr à la fois:

```
array_ref : (i : Nat) → Array α n
                → (i < n) → α;
```

Le troisième paramètre est censé être une *preuve* que $i < n$

Ainsi l'accès n'a pas besoin de vérifier les bornes:

- Accès aussi rapide qu'en C (ou Modula-2)
- Pas de risque d'erreur
- Pas besoin de stocker la taille du tableau à l'exécution

Le troisième paramètre est seulement présent pendant la compilation

Preuves et tests

Attention, il y a deux notions de `i < n`:

- L'expression booléenne `i < n : Bool`

  Renvoie `true` ou `false`

- La proposition logique `i < n : Type`

  Ne renvoie "rien"; dit que cette propriété doit être vraie

La même chose s'applique à tous les *prédicats*

Une convention est de mettre un `?` aux *tests* booléens:

```
x < y : Type;         x <? y : Bool;
x = y : Type;         x =? y : Bool;
```

Preuves d'inégalité

Comment construire une preuve que `i ≤ n` ?

```
type (≤) (n1 : Int) (n2 : Int)
  | Lt_base : n ≤ n
  | Lt_succ : n1 ≤ n2 → n1 ≤ n2 + 1;
```

Autre option

```
type (≤) (n1 : Int) (n2 : Int)
  | Lt : (diff : Nat) → n1 + diff = n2;
```

Ou encore

```
(≤?) : (n1 : Int) → (n2 : Int)
         → Either (n1 ≤ n2) (not (n1 ≤ n2));
```

Preuves d'égalité

Comment construire une preuve de `n1 + diff = n2` ?

Définition standard de l'égalité propositionnelle:

```
type (=) (x : α) (y : α)
  | Refl : x = x;
```

Exemple d'usage:

```
add5 t (x : t) (P : Int = t)
   = case P
     | Refl -> x + 5;
n = add5 Int 37 Refl;
```

Négation

Un système logique *cohérent*: système où il existe au moins une propriété qu'on ne peut pas prouver

Dans notre cas, cela signifie: un type *non-habité*

```
type False;        False = (P : Type) → P;
```

On peut alors définir la négation:

```
not α = α → False;
```

Que dit la preuve ci-dessous?

```
(λ (P1 : α) (P2 : not α) → P2 P1)
```

Preuves Logiques

Deux preuves "classiques"

```
(λ (P1 : Either α (not α)) (P2 : not (not α))
  → case P1
     | Left P3 -> P3
     | Right P3 -> P2 P3 α)
```

et

```
(λ (P1 : (α : Type) → not (not α) → α)
 → P1 (Either β (not β))
     (λ (P2 : not (Either β (not β))
      → let P4 (P3 : β) = P2 (Left P3)
        in P2 (Right P4))))
```

Définitions, preuves, axiomes

Ce genre de λ-calcul défini donc sa propre logique

Les *axiomes* de la logique classique se traduisent par:

- Les règles de typage du λ-calcul

  Exemple: le modus-ponens

- Des *définitions* dans le langage

  Exemples: *False*, *not*, *Either*

- Des *preuves* dans le langage

  Exemple: $\alpha \rightarrow \texttt{not (not } \alpha\texttt{)}$

Remplacer la théorie des ensembles par un tel λ-calcul ?

Maintenir la logique cohérente

Pour que la logique soit cohérente, *False* doit ne pas être habité

```
evil : False;
evil P = evil P;
```

⇒ ¡Pas de récursion générale!

ainsi que

```
type Bad
  | U (Bad → False);
f : Bad → False;
f x = case x
        | U g => g x;
checkmate = f (U f);
```

⇒ ¡Pas d'*occurences négatives*!

Pas d'effets de bord non plus, SVP

Logique classique

On peut montrer $\alpha \rightarrow$ `not (not` α`)` mais pas l'inverse:

¡L'élimination de la double négation n'est pas généralement vrai!

De même pour le principe du tiers exclus, bien sûr

On appelle une telle logique *constructiviste* ou *intuitioniste*

On peut les ajouter comme *axiomes*

Ou utiliser une définition alternative de la disjonction:

```
A ∨ B = not (not (Either A B));

LEM (P : not (Either A (not A)))
  = let P1 (P2 : A) = P (Left P2)
    in P (Right P1);
```

Expressions en forme normale

On peut représenter une expression nécessairement réduite

La réduction-β s'applique lorsque:

un *destructeur* est appliqué à un *constructeur*

```
type Exp constructor
  | Var : String → Exp false
  | Lam : String → Exp c → Exp true
  | App : Exp false → Exp c → Exp false
  | Num : Int → Exp true
  | Add : Exp c1 → Exp c2
          → (not (c1 && c2 = true))
          → Exp false
```

Typage intrinsèque

Représenter des expressions nécessairement bien typées

```
type SrcType
  | TInt : SrcType          | TBool : SrcType
  | TArrow : SrcType → SrcType → SrcType

type Exp (t : SrcType)
  | Num : Int → Exp TInt
  | App : Exp (TArrow t1 t2) → Exp t1 → Exp t2
  | Add : Exp (TArrow TInt (TArrow TInt TInt))
  | If : Exp Bool → Exp α → Exp α → Exp α
```

Une valeur de type `Exp` α représente une expression bien typée

Exemples d'usage: compilateur certifiant; construction de requêtes SQL