Conversion de fermetures

Rendre explicite la gestion des variables libres

```
lookup alist x = alist x;
add alist x v
  = lambda y -> case x = y
                  | true => v
                  | false => alist y
```

Le `lambda` a 4 *variables libres*:

- *x*, *v*, *alist*: capturées par la *fermeture*
- `_=_`: variable globale, potentiellement gérée différemment

Éléments de conversion

3 parties distinctes mais interdépendentes

- Transformer $\lambda x \to e$ en une paire $((\lambda(c, x) \to e'), \textit{env})$
  - c est le *contexte* ou *environnement*
  - $\lambda(c, x) \to e'$ est *fermé*, i.e. du code pur
- Transformer les appels f *arg* en $f.0\ (f.1, \textit{arg})$
- Remplacer les variables libres x par $c.n$
  - n est la position de x dans le contexte de la fonction

env doit contenir toutes les *variables libres* de la fonction

Les $\lambda(c, x) \to e'$ peuvent être *hoisted*: plus d'imbrication

Exemple, paire

```
add alist x v
  = lambda y -> case x = y
                  | true => v
                  | false => alist y
```

Devient

```
add alist x v
  = ((lambda (c, y)
       -> case c.0 = y
            | true => c.1
            | false => c.2.0 (c.2.1, y)),
     (x, v, alist))
```

Exemple alternatif

```
add alist x v
  = ((lambda (c, y)
       -> case c.1 = y
            | true => c.2
            | false => c.3.0 (c.3, y)),
     x, v, alist)
```

Au lieu d'une paire, un tuple; passe toute la fermeture en argument

Champ 0 contient du code en lieu et place d'un *tag*:

```
type Alist
   | nil
   | cons Key Value Alist
```

Typage des fermetures

Deux fonctions de même type ($Int \rightarrow Int$):

```
...
let f1 x = x + a;
    f2 y = y + b + c;
in ...
```

Deviennent des fermetures de formes différentes;

```
f1 = ((lambda (c, x) -> x + c.1), a)
f2 = ((lambda (c, y) -> y + c.1 + c.2), b, c)
```

Sous-typage

Fonctionnalité de base des langages OO

Sous-type: A est sous-type de B si tout object de type A peut-être utilisé là où un object de type B est attendu

Différentes variantes:

- Prefix subtyping: $(A, B, C) \subset (A, B, C, D)$
- Depth subtyping: $A \subset B \Rightarrow (A, C) \subset (B, C)$?
- $A \subset B \Rightarrow \mathit{List}\ A \subset \mathit{List}\ B$?

Appel de méthode

Polymorphisme ad-hoc: l'appel d'une méthode

```
o.m (a)
```

devient en réalité un *dispatch*

```
find_method (o, m) (o, a)
```

Le problème est donc d'implanter `find_method` efficacement

Sous-typage statique et héritage simple

Typage garanti le *préfixe* de la structure de x

```
o.m (a)   ==>   o.vtable.m (o, a)
```

Généralement le champ `vtable` est placé à offset 0

Aussi appelé `class`

Sous-typage garanti que la méthode est toujours au même endroit

Implantation typique pour Java

Héritage multiple à la C++

Permettre l'héritage multiple

Maintenir le même appel de méthode:

```
o.m (a)   ==>   o.vtable.m (o, a)
```

Une `vtable` par parent

Up-cast n'est pas gratuit:

```
(P2)o   ==>   o + <offset-of-P2>
```

La majorité des *up-cast* est implicite!

Interfaces Java

Permettre héritage multiple (mais seulement d'interfaces)

Peut ajouter une interface après avoir créé une classe

Up-cast encore plus cher:

```
(I)x    ==>    (vtable, x)
```

Appels de méthode à peine modifés:

```
o.m (a)    ==>    o.0.m (o.1, a)
```

Méthodes par tableau bidimensionnel

Chaque method reçoit un identificateur unique

Chaque classe reçoit un identificateur unique

```
o.m (a)    ==>    mtable[o.classid, m] (o, a)
```

Tableau bidimensionnel de grande taille, *sparse*

Avec typage statique, possibilité de le compresser

Information globale, pas connue à la compilation

Inline cache*: Appels de méthode dynamiques*

Lorsque la sémantique du langage oblige une recherche coûteuse

Utilisation d'un *inline cache*:

```
   o.m (a)
==>
   if (o.class != last_class) {
      last_method = find_method_slow (o, m);
      last_class = o.class;
   }
   last_method (o, a)
```

Majorité des appels utilisés avec 1 seule classe

Monomorphisation

Compilation du polymorphisme par duplication de code

Pour chaque type différent, générer une autre copie du code

`o.class` devient donc connu d'avance!

```
o.m (a)  ==>  the_method (o, a)
```

En général mène à une explosion de taille du code

Impossible en général (e.g. `v[i].m(a)`): dispatch nécessaire

Monomorphisation paresseuse par compilation JIT

Monomorphisation des méthodes

Polymorphisme paramétrique

```
identity : a -> a;
array_ref : Array a -> Int -> a;
map : (a -> b) -> List a -> List b;
```

Comment/où passer paramètres à *identity*?

Comment/où est retourné le résultat?

array_ref et tableaux de *Float*? tableaux de *Bool*?

N'importe quelle fonction peut-être passée à *map*

Polymorphisme par boxing

Représentation uniforme, comme langages dynamiques

```
  lambda (x : Int) -> x + 1
==>
  lambda x -> let x' = unbox-int x
              let r = add x' 1
              let r' = box-int r
              return r'
```

Float devient un pointeur vers un float! *Bool* gros comme *Int*!

unboxing par optimisation classiques

Affecte tout le code, même s'il n'utilise pas de polymorphisme

Monomorphisation

Pour chaque type différent, générer une autre copie du code

Utilisé par les *template*s de C++

Optimisation de type *whole program*

Polymorphisme paramétrique: partage entre types “similaires”

- E.g. 5 types différents: *Float*, *Int*, *Char*, *Bool* et “autre”

Monomorphisation complète possible

Polymorphisme par coercions

Remplacer paramètres de types par un type *Box* (e.g. `void*`)

```
identity : Box -> Box;
identity x = x;
y : Int;
y = coerce [Box => Int]
           (identity (coerce [Int => Box] 5));
```

Pour certains types, *coerce* est gratuit

Pour d'autres, il peut être coûteux ou impossible!

Exemple de coercions

```
map : (Box -> Box) -> List Box -> List Box;
plus_05 : Float -> Float;
plus_05 x = 0.5 + x;
map_plus_05 : List Box -> List Box;
map_plus_05
  = map (lambda b
          -> coerce [Float => Box]
                    (plus_05 (coerce [Box => Float]
                                     b)));
```

Coercion de *List Float* à *List Box* peut être $O(N)$

Coercion de *Array Float* à *Array Box* impossible (mutable!)

Intensional Type Analysis

Paramètres de type transformés en descripteurs de types

```
array_ref : TypeDesc a -> Array a -> Int -> a;
array_ref t v i = case t
                    | Int => aref_int v i
                    | Float => aref_int v i
                    | ...
```

Généralement peu de types vraiment différents

Polymorphisme par OO

Descripteurs de types deviennent des tables de méthodes

```
array_ref : TypeDesc a -> Array a -> Int -> a;
array_ref t v i = t.ref v i
```

Spécialisation de type

Inférence de Hindley/Milner introduit du polymorphisme partout

- Défaire la généralisation là où elle n'est pas nécessaire

Sorte de monomorphisation timide: pas de risque d'explosion

Inutile si on utilise une représentation uniforme