Représentation des objets et des types

Introduction

Le contexte dans lequel on étudie les représentations des objets et de leur types est celui où:

- des objets sont créés dynamiquement et
- le typage est dynamique.

Par exemple, ce contexte correspond aux choix de représentation qu'il faut faire pour implanter Scheme.

Introduction

Définitions:

Une représentation est *directe* lorsque la valeur des objets représentés se trouve directement dans la représentation employée.

Par exemple, les données de types `char`, `int` et `float` en C sont représentées directement.

Une représentation est *indirecte* lorsque la représentation consiste en un pointeur (ou un index) et que la valeur des objets est accessible uniquement au travers du pointeur (ou de l'index).

Par exemple, les structures de données allouées dans le tas en C ou les objets en Java.

Introduction

Première tentative de représentation des objets de Scheme:

Représentation directe avec structures, unions et scalaires C.

Cette représentation est forcément inadéquate car un objet peut en contenir un autre.

Par exemple, pour la donnée `(1 2)`, l'objet principal est une paire, disons de type `DATA`, laquelle doit contenir deux champs pour décrire le contenu de la paire. Chacun de ces champs doit être capable d'accommoder un autre objet, de type `DATA`. En particulier, le deuxième champ contient justement une autre paire. Ce qui nous mène à l'inéquation insatisfaisable:

$$\texttt{sizeof(DATA)} \geq 2 \times \texttt{sizeof(DATA)}$$

Donc, il faut absolument employer l'allocation dynamique et les pointeurs.

Introduction

Deuxième tentative de représentation des objets Scheme:

Représentation indirecte avec des pointeurs sur tous les "contenus" possibles.

```
struct Data;

typedef struct Data DATA;

union Contenu
{
  int   entier;
  float reel;
  struct { DATA car; DATA cdr; } paire;
  ...
};

struct Data
{
  int type;
  union Contenu *ctn;
};
```

Les problèmes avec cette représentation sont:

- qu'il y a un gaspillage d'espace car l'union doit être aussi grande que le plus grand de ses membres;
- qu'on ne peut représenter les objets à taille variable comme les `vector` de Scheme.

Introduction

Approche générale:

- Par souci d'efficacité et d'économie de mémoire, on préfère laisser tomber les types C.
- On gère *soi-même* la représentation.
- On suppose que le tas est une longue plage de mémoire composée de mots machine; typiquement, des `int` de C.
- Une allocation dans notre tas consiste essentiellement à réserver quelques mots consécutifs et à conserver l'adresse du premier de ces mots.

Représentation par pointeurs d'objets

Description: Tout objet est représenté par un pointeur vers le tas. La partie du tas qui est pointée consiste en un champ pour indiquer le type de l'objet suivi de 0 ou plus champs en fonction du type.

Par exemple:

Une paire: ⟶ | PAIR | `ref` | `ref` |

Un vecteur: ⟶ | VECTOR | `len` | `ref` | ... |

Un réel: ⟶ | REAL | `float` |

Une liste vide: ⟶ | EMPTY |

Un caractère: ⟶ | CHAR | `char` |

Un booléean: ⟶ | BOOL | `int` |

Un petit entier: ⟶ | FIXNUM | `int` |

Un grand entier: ⟶ | BIGNUM | `len` | `int` | ... |

Une fermeture: ⟶ | CLOS | `C-ptr` | `env` |

...

Représentation par pointeurs d'objets

Avantages et inconvénients:

+ Régularité et simplicité

– Inefficacité

  - Tout objet doit être créé (même un booléen!)
  - Tout test de type ou tout accès à la valeur d'un objet requiert un accès en mémoire

Représentation par pointeurs d'objets

Améliorations possibles:

- Assignation de types spécialisés pour les caractères et les booléens.

  Par exemple:

  Un caractère: $\longrightarrow \boxed{\text{CHAR}_i}$ où $i \in \{T_C, \ldots, T_{C+255}\}$

- Pré-allocation des objets de certains types. Plus de création d'objets de ces types. Comparaison par pointeur possible.

  Par exemple, la liste vide (OK), les booléens (OK), les caractères ASCII (OK), les caractères Unicode (???).

Représentation par pointeurs taggés

Idée:

Certains bits peuvent être constants dans les adresses vers le tas:

- ceux du bas (lsb), à cause de l'alignement;
- ceux du haut (msb), à cause de l'étendue du tas.

On peut profiter de ces bits constants pour:

- encoder de l'information de typage;
- avoir une représentation directe de certains types d'objets.

Représentation par pointeurs taggés

Exemple:

Supposons que les adresses vers le tas sont alignées sur 8 octets et que les mots sont de 32 bits. Donc, les 3 lsb sont connus et sont égaux à 0.

`xx...xx000`	$\longrightarrow$	fixnums	(repr. directe)	(sur 29 bits)
`xx...xx001`	$\longrightarrow$	caractères	(repr. directe)	
`xx...xx010`	$\longrightarrow$	booléens	(repr. directe)	
`xx...xx011`	$\longrightarrow$	liste vide	(repr. directe)	
`xx...xx100`	$\longrightarrow$	paires	(repr. indirecte)	
`xx...xx101`	$\longrightarrow$	fermetures	(repr. indirecte)	
etc.				

Représentation par pointeurs taggés

Définitions:

L'opération d'*emballage* (*wrapping*) d'une adresse ou d'une donnée brute consiste à la convertir sous le format des pointeurs taggés.

L'opération de *déballage* (*unwrapping*) est l'opération inverse.

Il faut éviter de confondre ces opérations avec la création des objets et l'extraction des valeurs contenues par ceux-ci.

Exemples référant à la représentation de la page précédente.

- La création d'un petit entier consiste seulement en un emballage.
- La création d'une paire consiste en l'allocation et l'initialisation de quelques mots dans le tas et en l'emballage de l'adresse.
- L'extraction de la longueur d'un vecteur consiste en le déballage du pointeur taggé, en la lecture du deuxième champ du vecteur et en l'emballage de la valeur brute lue.
- Dans le cas de la représentation par pointeurs d'objets, les opérations d'emballage et de déballage sont inexistantes.

Représentation par pointeurs taggés

Exemple d'implantations des opérations reliées au "tagging" en supposant que les 3 lsb servent au tagging:

Test de type: Pour tester si on a affaire a une fermeture:

$$\texttt{pred\_clos(p)} \equiv \texttt{(p \& 7) == 5}$$

Emballage: Pour emballer un caractère brut:

$$\texttt{wrap(c)} \equiv \texttt{(c << 3) | 1}$$

Déballage: Pour déballer un booléen:

$$\texttt{unwrap(b)} \equiv \texttt{b >> 3}$$

Exercice: refaire des exemples similaires en considérant l'utilisation des 3 msb pour le tagging.

Représentation par pointeurs taggés

L'utilisation de cette représentation combinée avec l'emploi des bits les moins significatifs pour le tagging permet d'avoir une arithmétique des petits entiers efficace.

Supposons que n bits de tagging sont employés par mot de 32 bits et que la valeur en décimal du tag des fixnums soit t. Ainsi, la représentation du nombre x est $2^n x + t \bmod 2^{32}$.

Soient x et y deux petits entiers et soient a et b leurs représentations respectives.

Naïvement, la représentation du résultat d'une opération $\otimes$ entre x et y à partir de a et b est obtenue ainsi:

$$\texttt{wrap}(x \otimes y) \quad \equiv \quad \texttt{wrap}(\texttt{unwrap}(a) \otimes \texttt{unwrap}(b))$$

mais on peut faire mieux:

$$\begin{array}{lcl}
\texttt{wrap}(x + y) & \equiv & a + b - t \\
\texttt{wrap}(x - y) & \equiv & a - b + t \\
\texttt{wrap}(x * y) & \equiv & (a - t) * (b >> n) + t \\
\texttt{wrap}(x / y) & \equiv & \texttt{wrap}((a - t) / (b - t))
\end{array}$$

Noter que ces implantations deviennent particulièrement efficaces si on choisit $t = 0$.

Représentation par pointeurs taggés

Pour mieux utiliser les bits de tagging, on a intérêt à utiliser des tags de longueurs diverses.

En effet, dans un exemple de tagging présenté précédemment, on constate que plusieurs bits sont gaspillés par la représentation (notamment, chez les caractères, les booléens et la liste vide).

De plus, typiquement, les bits de tagging sont rares.

Or, on peut souvent assigner plusieurs types aux mêmes tags.

Exemple:

`xx...xxxxx000`	$\longrightarrow$	fixnums	(repr. directe)
`xx...xx000001`	$\longrightarrow$	caractères	(repr. directe)
`xx...xx001001`	$\longrightarrow$	booléens	(repr. directe)
`xx...xx010001`	$\longrightarrow$	liste vide	(repr. directe)
etc.			
`xx...xxxxx010`	$\longrightarrow$	paires	(repr. indirecte)
`xx...xxxxx011`	$\longrightarrow$	fermetures	(repr. indirecte)
etc.			

Représentation par pointeurs taggés

L'utilisation de $m > 0$ msb et de $n > 0$ lsb à la fois entraîne des coûts supplémentaires.

Par exemple, pour effectuer un test de type sur un objet `p`, on peut faire quelque chose comme:

$$\texttt{((p <<}\ m\texttt{) | (p >>}\ (32-m)\texttt{))}\ \texttt{\&}\ (2^{m+n}-1)\ \texttt{==}\ tag$$

Toutefois, une implantation directement en assembleur peut profiter d'instructions de rotation qui ne sont pas disponibles en C.

$$\texttt{(p rotateleft}\ m\texttt{)}\ \texttt{\&}\ (2^{m+n}-1)\ \texttt{==}\ tag$$

Représentation par partition des codes

L'intervalle $[0, 2^{32}-1]$ est subdivisé en sous-intervalles, un pour chaque type.

Par exemple:

$[\texttt{0x00000000},\texttt{0x20000000}) \longrightarrow$ fixnums
$[\texttt{0x20000000},\texttt{0x60000000}) \longrightarrow$ paires
etc.

Les frontières entre les sous-intervalles ne sont pas nécessairement conçues pour mener à des opérations efficaces.

Représentation par partition des codes

Exemples d'implantations des trois opérations reliées à l'emballage.

Test de type: Pour tester si on a affaire a une fermeture:

$$\texttt{pred\_clos(p)} \equiv (BASE_{\texttt{clos}} \texttt{ <= p) \&\& (p < } BASE_{\texttt{next}}\texttt{)}$$

Emballage: Pour emballer un caractère brut:

$$\texttt{wrap(c)} \equiv \texttt{c + } BASE_{\texttt{char}}$$

Déballage: Pour déballer un booléen:

$$\texttt{unwrap(b)} \equiv \texttt{b - } BASE_{\texttt{bool}}$$

Représentation par partition des codes

Emplacement des objets alloués: 1 tas versus plusieurs tas.

- S'il y a un seul tas, le sous-intervalle de chacun des types indirects doit être assez grand pour fournir un code pour chaque adresse légale dans le tas.
- S'il y a plusieurs tas, il est possible d'assigner un sous-intervalle plus court aux types moins fréquents et de laisser plus de codes aux types directs. Toutefois, la gestion de plusieurs tas augmente la complexité de l'implantation.

Représentation par partition des codes

L'implantation du test de type peut être optimisée grâce au truc suivant:

$$(BASE_{\texttt{type}} \texttt{ <= p) \&\& (p < } BASE_{\texttt{type}'}\texttt{)}$$
$$\equiv$$
$$\texttt{((unsigned) p - } BASE_{\texttt{type}}\texttt{) < ((unsigned) (}BASE_{\texttt{type}'} - BASE_{\texttt{type}}\texttt{))}$$

qui permet de transformer un test comportant deux sauts conditionnels en un test comportant une soustraction et un saut conditionnel.

Représentation par emplacements typés

Description: le type est donné non pas par la référence à l'objet mais par l'endroit où on a trouvé cette référence.

Cette méthode de typage est normalement applicable chez les langages typés statiquement.

Malgré tout, cette forme de typage existe en Scheme. Les chaînes de caractères ne peuvent contenir que des caractères. On peut donc stocker ceux-ci de façon brute dans les chaînes.

Représentation par large wrappers

Description: la référence à un objet est conservée dans plus d'un champ. Par exemple: un champ contient le type et le champ suivant, la valeur ou l'adresse brute.

Avantages et inconvénients:

+ Fait disparaître les coûts d'emballage et de déballage.
+ Sur les architectures comportant de nombreux registres, le passage de la référence à un objet se fait via deux registres.
– Valable surtout pour des données manipulées directement par le processeur et non celles stockées en mémoire où elles causeraient un gaspillage de mémoire important.

Représentation hybride

Par exemple, un représentation hybride utilisant les pointeurs taggés sur 2 bits et les pointeurs d'objets:

`xx...xxxx00`	$\longrightarrow$	fixnums	(repr. directe)	(sur 30 bits)
`xx...xxxx01`	$\longrightarrow$	fermetures	(repr. indirecte)	
`xx...xxxx10`	$\longrightarrow$	autres types /tas	(repr. indirecte)	(sous-type /tas)
`xx...xxx011`	$\longrightarrow$	caractères	(repr. directe)	
`xx...xx0111`	$\longrightarrow$	booléens	(repr. directe)	

etc.

Opérations optimisées diverses

Changement de casse pour des caractères représentés directement:

En minuscules:	`c | (1 <<` j`)`
En majuscules:	`c & (~(1 <<` j`))`
Alternance maj./min.:	`c ^ (1 <<` j`)`

Conversions entre les caractères et les entiers:

De caractère à entier:	`(c >>` k`)` $\hat{}$ $PATCH_{\mathrm{CE}}$
D'entier à caractère:	`(i <<` k`)` $\hat{}$ $PATCH_{\mathrm{EC}}$

Utilité de l'implantation efficace d'un branchement à plusieurs voies sur le type, i.e. sorte de `switch` sur le type:

```
type_switch (exp)
  case fixnum:
    { ... }
  case pair:
    { ... }
  ...
  case string:
    { ... }
```

qui soit plus rapide que:

```
if (pred_fixnum(exp))
  { ... }
else if (pred_pair(exp))
  { ... }
...
else
  { ... }
```

Applicabilité des techniques de représentations

Les techniques de représentation des objets et des types ne servent pas qu'aux langages typés dynamiquement. Ils peuvent aussi servir à l'implantation des langages typés statiquement.

Raisons:

- Présence de petits et grands entiers et conversion implicite entre les deux représentations.
- Types algébriques.

Soit le type algébrique suivant:

```
arbre23 a = Empty
          | Deux (arbre23 a) a (arbre23 a)
          | Trois (arbre23 a) a (arbre23 a) a (arbre23 a)
```

Si, par exemple, la variable 'v' est de type `arbre23 int`, on est garanti que sa valeur est une des trois possibilités énumérées. Toutefois, c'est seulement à l'exécution qu'on sait lequel des trois constructeurs est utilisé.

Il faut donc utiliser une des techniques de représentation pour encoder le type et les données contenues dans un objet de type `arbre23 int`.

Choix d'une représentation

Comment choisir une représentation? Il faut considérer:

- l'importance de la vitesse à l'exécution,
- l'importance de l'espace consommé,
- le nombre de bits constants disponibles (bits *taggables*),
- taille des références (ou mots),
- l'ensemble des types à encoder,
- l'importance de la clarté de l'implantation (???),
- l'importance de l'extensibilité de l'implantation (reste-t-il beaucoup de bits *taggables*?),
- la fréquence d'utilisation des objets des différents types,
- l'interaction avec le gestionnaire de mémoire,
- etc.

