

Rapport pour le projet du Compilateur MISC

Nicolas Bonvin, Gilles Diacon, Xavier Perséguers
École Polytechnique Fédérale de Lausanne

2 février 2003

1 Optimisations

Le code généré par le compilateur n'étant pas toujours très optimal, nous avons choisi de mettre au point un certain nombre d'optimisations pour étendre quelque peu les possibilités de notre compilateur.

1.1 Expressions arithmétiques

La génération des expressions arithmétiques, telle que proposée, n'était pas vraiment efficace. Une expression du type 3×4 était codée de façon à stocker dans deux registres chacune des constantes, avant d'effectuer l'opération. Nous avons modifié un peu le code pour tirer partie des instructions comportant une valeur immédiate. Nous n'utilisons donc plus qu'un seul registre pour une expression telle que la précédente.

En outre, nous générons d'abord le sous-arbre le plus profond, ceci afin de lui laisser un maximum de registres et ne pas en perdre un pour le stockage intermédiaire du second sous-arbre.

Multiplication : La multiplication, qui est un opérateur coûteux, est remplacée par un décalage vers la gauche si l'une des opérandes est une puissance de 2.

Problèmes rencontrés : La combinaison de l'utilisation des opérations immédiates et de la génération du sous-arbre le plus profond en premier amène à se retrouver avec les opérandes inversées si le sous-arbre droit est plus profond. Dans le cas des opérateurs “+” ou “*”, ce n'est pas grave, mais pour les autres opérateurs, le résultat est faussé. Deux solutions étaient possibles : ne pas changer l'ordre de génération si les opérandes peuvent être utilisées dans une instruction immédiate, ou garder le comportement initial dans ce cas, à savoir charger les opérandes dans des registres. C'est cette deuxième solution que nous avons préférée.

Nous avons aussi fait attention à utiliser les instructions immédiates signées ou non-signées si elles permettaient de faire l'opération escomptée. Dans le cas de constantes trop grandes, la solution standard — consistant à stocker la valeur immédiate dans un registre — a été retenue.

1.2 Listes

L'algorithme initial que nous avons utilisé pour la création de listes était le suivant :

- Allouer une nouvelle cellule en mémoire (2×4 octets) ;
- Générer le code pour la tête de la liste ;
- Copier le résultat en mémoire dans le premier octet de la nouvelle cellule ;
- Générer le code pour la queue de la liste (appel récursif du visiteur sur la partie droite de l'arbre) ;
- Copier le résultat (adresse) dans la second octet de la nouvelle cellule.

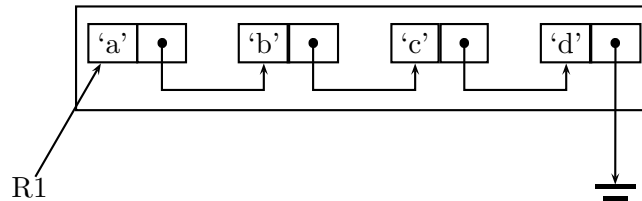
Exemple d'exécution : Avec cet algorithme, et en partant du principe que tous les registres sont libres au moment de la création de la liste, une liste représentant la chaîne "abcd" en mémoire conduit à cette exécution :

- Allocation d'une nouvelle cellule. Adresse stockée dans R1 ;
- Génération du code pour 'a'. Stockage dans R2 ;
- Stockage du résultat R2 dans la cellule ;
- Libération de R2 ;
- Génération du code pour la queue "bcd" :
 - Allocation d'une nouvelle cellule. Adresse stockée dans R2 ;
 - Génération du code pour 'b'. Stockage dans R3 ;
 - Stockage du résultat R3 dans la cellule ;
 - Libération de R3 ;
 - Génération du code pour la queue "cd" :
 - Allocation d'une nouvelle cellule. Adresse stockée dans R3 ;
 - Génération du code pour 'c'. Stockage dans R4 ;
 - Stockage du résultat R4 dans la cellule ;
 - Libération de R4 ;
 - Génération du code pour la queue "d" :
 - Allocation d'une nouvelle cellule. Adresse stockée dans R4 ;
 - Génération du code pour 'd'. Stockage dans R5 ;
 - Stockage du résultat R5 dans la cellule ;
 - Libération de R5 ;
 - Stockage du pointeur 0 (NilLit) dans la cellule ;

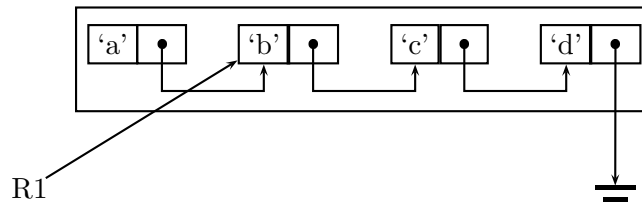
- Stockage du pointeur R4 dans la cellule (R3);
- Libération de R4;
- Stockage du pointeur R3 dans la cellule (R2);
- Libération de R3;
- Stockage du pointeur R2 dans la cellule (R1);
- Libération de R2;

Conséquence : Il apparaît clairement que la taille d’une liste générée de cette façon est limitée non pas par la mémoire disponible, mais par le nombre de registres! Pour nous, cela nous semblait inconcevable, surtout si l’on pense que l’utilisateur voudra certainement écrire des petits textes et pas uniquement stocker statiquement un ensemble de nombres dans une liste.

Première amélioration : Nous avons décidé d’abandonner la génération récursive des listes pour la faire de façon itérative. Plutôt que d’allouer une cellule à la fois, pourquoi ne pas allouer directement une zone mémoire correspondant à la longueur de la liste pour y stocker tous les éléments? Nous l’avons donc fait et, de ce fait, le nombre de registres utilisés est devenu constant quelle que soit la longueur de la liste. C’est à ce moment que nous avons réfléchi aux conséquences d’une telle génération. Pour la chaîne “abcd”, nous arrivons à cet usage de la mémoire (en partant du principe que la liste est pointée par R1) :



Le grand cadre représente la zone mémoire qui a été allouée et qui est vue comme un seul bloc par le garbage collector. Imaginons maintenant que nous appelions la méthode `tail` sur cette liste. Cela nous conduira à cet état de la mémoire :



Dès lors, le premier élément du bloc n'est plus référencé et le garbage collector pensera, à tort, que cette zone mémoire peut être libérée. Cela ne se produira malheureusement que si la mémoire est saturée, générant donc une erreur que dans certains cas particuliers et rendant le débogage extrêmement difficile.

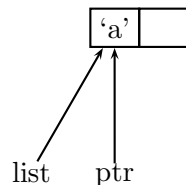
Seconde amélioration : Après avoir constaté cette source d'erreur, nous avons été forcé, pour réussir à ne pas limiter l'utilisateur avec des contraintes liées à notre compilateur, de trouver un autre algorithme, permettant à la fois de s'affranchir du nombre de registres proportionnel à la longueur de la liste et de la gestion de la mémoire par le garbage collector. L'algorithme obtenu est le suivant :

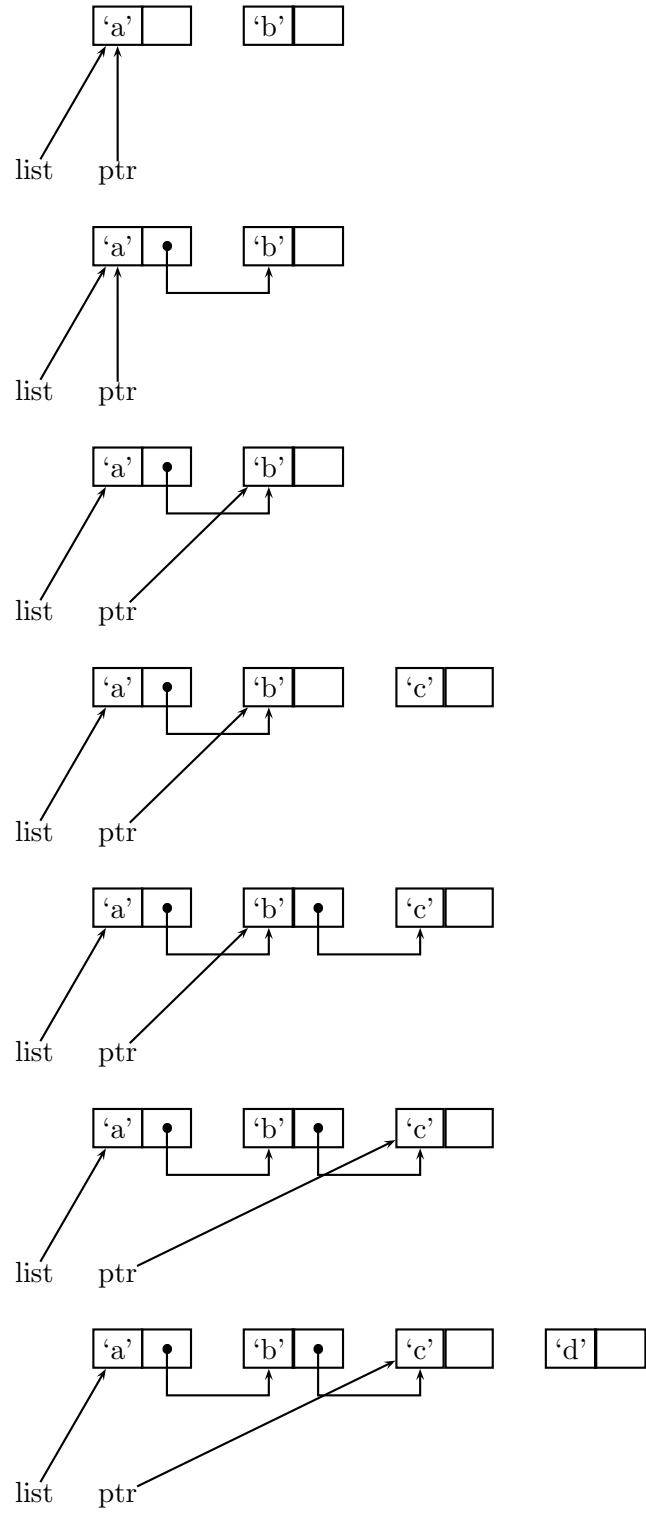
- Allouer une nouvelle cellule en mémoire (2×4 octets). Le pointeur est appelé `list` ;
- Générer le code pour la tête de la liste ;
- Réserver un autre registre, appelé `ptr`, qui pointera toujours sur le dernier élément de la liste. Le faire pointer sur la cellule créée précédemment ;
- Faire une boucle sur tous les éléments de la liste (à partir du deuxième) :
 - Si la fin de la liste est atteinte, placer 0 dans le second octet de la cellule pointée par `ptr` ;
 - Allouer une nouvelle cellule en mémoire ;
 - Générer le code pour l'élément actuel (la tête d'une sous-liste) ;
 - Placer l'adresse de cette cellule dans le second octet de la cellule pointée par `ptr` ;
 - Mettre à jour `ptr` pour pointer vers cette nouvelle cellule.

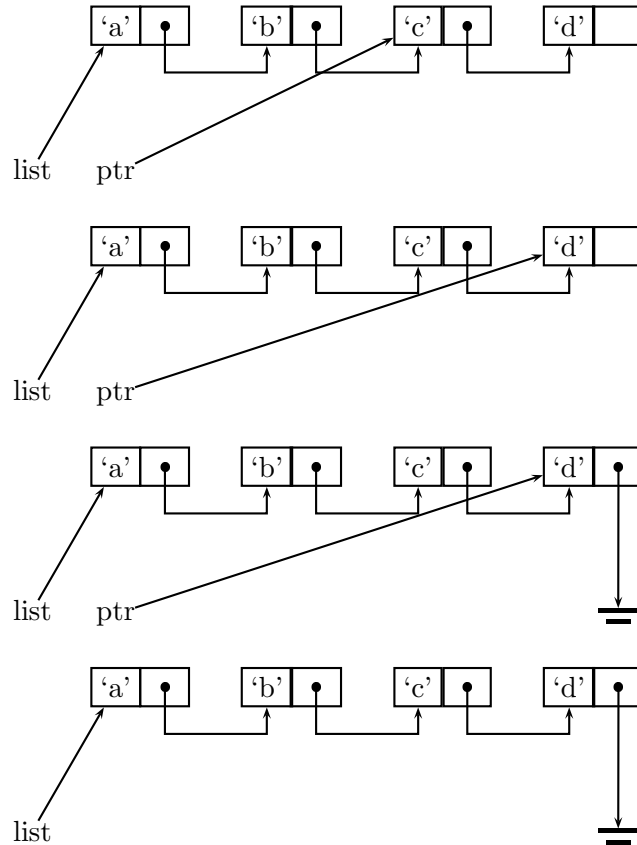
La fin de la liste est aussi atteinte dans le cas où la sous-liste n'est pas directement interprétable (par exemple à cause d'une opération `tail` ou `head`). Dans ce cas, cette sous-expression est générée de façon récursive en appelant le visiteur sur le sous-arbre droit et le résultat est placé dans le second octet de la cellule pointée par `ptr`.

L'avantage de cette méthode par rapport à la précédente est que la liste est créée par allocations successives de cellules, comme dans notre première version, mais de façon itérative, donc avec un nombre de registres constant.

Exemple d'exécution : Voici les différents états de la mémoire pour la création d'une liste représentant la chaîne "abcd" en mémoire :







1.3 Remarque

Le code de génération des valeurs immédiates dans le fichier `Item.java` fait une distinction de cas entre les valeurs positives et négatives. Dans la première version, notre compilateur ne fournissait jamais de valeurs négatives, étant donné que les nombres négatifs ont été remplacés par une opération de type $0 - |valeur|$ lors de la génération de l'arbre. Cette distinction de cas a néanmoins été utilisée dès que nous avons simplifié les expressions arithmétiques (cf. ci-après).

1.4 Optimisations “avancées”

Une phase d'optimisation a été rajoutée dans le compilateur, entre l'analyse et la génération de code. Cette phase est facultative. Pour l'activer, il suffit de passer le paramètre `-optimize` au générateur de code.

1.4.1 Fonctions

Probablement l'optimisation la plus utile. Cette optimisation crée au moment de la phase d'optimisation un graphe des dépendances des différentes

fonctions. Au moment de générer le code, une vérification est faite pour déterminer si la fonction a réellement besoin d'être générée. Dans le cas des fonctions de base, étant donné qu'elles sont "inlinées", elles ne sont générées que si elles sont référencées comme paramètre d'une fonction ou comme élément d'une liste.

1.4.2 Expressions arithmétiques

Les expressions arithmétiques sont évaluées et seul le résultat est retourné. Dans le cas d'une division par zéro au moment de l'exécution, le code est généré avec la division par zéro et un message d'avertissement est affiché sur le terminal.

Dans le cas de valeur immédiate uniquement dans l'un des deux nœuds, l'optimisation tente néanmoins de simplifier l'expression. Par exemple un appel tel que `1 * appel_de_fonction` sera remplacé par l'appel de fonction lui-même. Les multiplications par la constante 0 ne sont pas remplacés par 0 car la fonction peut très bien avoir des effets de bord.

1.4.3 Blocs conditionnels

Les conditions des tests sont vérifiées et dans le cas d'un résultat constant, seule la branche utile du test sera générée. Un message d'avertissement est généré.

1.4.4 Boucles

Cette optimisation est faite pendant la génération de code, mais uniquement si l'optimisation a été activée. Le compilateur détecte les boucles infinies si la condition est une constante différente de zéro. Un message d'avertissement est généré. Dans le cas de condition toujours fausse, le code complet de la boucle est supprimé avec un message d'avertissement.

2 Améliorations

2.1 Grammaire

2.1.1 Caractères

Nous avons rajouté la gestion des caractères interprétés comme leur code ASCII comme entité du langage MISC. En effet, dans la grammaire actuelle, si un utilisateur veut passer le caractère 'A' à la fonction `printChar` par exemple, il doit écrire le code suivant :

```
printChar(head("A"))
```

ou alors passer directement le code ASCII du caractère à cette fonction. Désormais il est possible de taper 'A' en lieu et place de `head("A")`. La

syntaxe est un caractère placé entre deux apostrophes. Il n'existe pas de caractère d'échappement. Il n'est donc pas possible de taper le caractère de retour à la ligne par exemple. En particulier, l'apostrophe est représentée par trois apostrophes consécutives.

2.1.2 Commentaires

La gestion des commentaires sur plusieurs lignes a été ajoutée. La version proposée ne gère pas l'imbrication de commentaires. Ils sont délimités, comme dans C ou Java, par `/*` et `*/`.

2.2 Exécution

Le paramètre `-debug` a été utilisé au niveau de la génération du code pour ajouter des commentaires dans le code source généré, de façon à pouvoir relire plus facilement le code assembleur. Pour pouvoir insérer des instructions, le fichier `Code.java` a dû être enrichi d'un nouveau type d'"instruction" : le commentaire. De cette façon, le commentaire peut être placé exactement à l'endroit voulu dans le code, et non pas avant celui-ci, si nous avons utilisé la syntaxe `System.out.println()`.

Conséquence : Du fait que les commentaires sont considérés comme des instructions, il a fallu réécrire les méthodes `pc()` et `fixup()` pour ne pas avoir de décalage dans les opérations sur les instructions (récupération de l'instruction à une adresse précise, ...).