Compilateur *Mini Ada*

Florentin Guth

15 janvier 2017

A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible.
Ada.

Table des matières

1 Arbre de syntaxe abstraite

1.1 R.I.P. (et pas `rip`[1]) foncteur

Finalement, c'était une mauvaise idée (vous nous l'aviez dit après tout...).

On utilise donc deux *AST* différents, afin de permettre plus de flexibilité lors du passage du typage à la production de code.

1.2 Type des identifiants

On conserve la taille de chaque type (bien que ceux-ci n'apparaissent pas en sortie) afin de faciliter le remplissage des champs du nouveau type des identifiants :

ast_typed.ml[50-56]

```
50 type ident =
51   {
52     is_reference: bool;
53     size:   int;
54     level:  int;     (* directly difference from current lvl to ident lvl *)
55     offset: int;     (* offset according to %rbp *)
56   }
```

Listing 1 – Type des identifiants

1.3 Type des expressions

On a rassemblé les valeurs gauches dans les expressions pour simplifier (il n'est plus nécessaire de faire la différence, on sait que l'on a une valeur gauche où il faut grâce au typage). On ne conserve pas les types des expressions mais seulement leur taille (essentiellement pour la compilation du test d'égalité).

On a également rajouté la construction `Ederef` qui correspond à la construction `.all` (qui peut être implicite comme dans l'accès à un champ d'un `access R`. De plus, pour l'accès à un champ on a besoin de connaître uniquement le décalage par rapport à `rbp`.

ast_typed.ml[66-75]

```
66 type expr_desc =
67   | Econst    of const
68   | Ederef    of expr
69   | Eident    of ident
70   | Emember   of expr * int (* address of field *)
71   | Ebinop    of expr * binop * expr
72   | Enot      of expr
73   | Enew      of int (* size *)
74   | Eapp_func of pf_sig * (expr list) (* list is non-empty *)
75 and expr = (expr_desc, int) node (* decorated with size *)
```

Listing 2 – Type des expressions

ast_typed.ml[58-64]

```
58 type pf_sig =
59   {
60     name: ident_desc;
61     modes: mode list;
62     level:    int;
63     size_ret: int;
64   }
```

Listing 3 – Type des appels de fonctions ou procédures

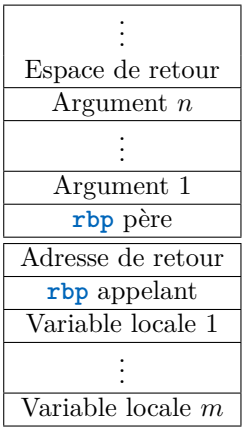
1. où je déplore que `\mintinline` ne fonctionne pas avec %

Le `type pf_sig` contient le nom de la fonction en question (modifié par le typage pour être unique), le mode de passage des paramètres, son niveau (ou plutôt la différence de niveau entre le niveau courant et celui de la fonction) pour calculer le `rbp` du père et la taille du type de retour. En effet, il se produit deux cas :

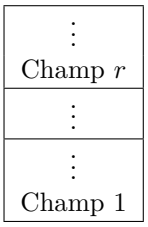
— Soit celui-ci vaut zéro (cas d'une procédure) et alors la valeur de retour (nécessairement `null`, i.e. 0) est renvoyée dans `rax` (utile uniquement pour le code d'erreur de la procédure principale),

— Soit il est strictement positif, et alors l'appelant va allouer la taille correspondante sur la pile (avant d'empiler les paramètres), espace dans lequel la fonction appelée copiera sa valeur de retour (et ainsi lorsqu'on dépilera les paramètres, la valeur de retour sera directement sur le sommet de la pile).

On obtient donc le tableau d'activation suivant :



4.1 Tableau d'activation



4.2 Représentation en mémoire des registres

TABLE 4 – Tableaux pas très jolis avec `\tabular`, parce que je n'ai pas le temps pour TikZ

Dernière modification en ce qui concerne les expressions : lors du *parsing*, lorsqu'on rencontre un moins unaire et que l'expression qui suit est une constante `Econst (Cint n)`, on produit `Econst (Cint (-n))` afin de compiler correctement -2^{31}. On rejette la constante 2^{31} lors du typage.

1.4 Type des instructions

Le seul fait notable est que les boucle `for` ont disparues : elles sont transformées en boucle `while` lors du typage. Celui-ci s'occupe de rajouter trois variables : une pour le compteur et une pour chaque borne (afin de ne les calculer qu'une fois) et donc de leur allouer la place nécessaire dans le tableau d'activation.

1.5 Type des déclarations

On ne garde que les initialisations de variables et les déclarations de fonctions. Dans le cas de ces dernières, on conserve leur signature, la taille du tableau d'activation (pour l'allouer en une seule fois), le décalage par rapport à `rbp` de l'espace de retour en plus du code.

ast_typed.ml[91-98]

```
91 and proc_func =
92   {
93     pf_sig:  pf_sig;
94     frame:   int;
95     ret:     int;
96     decls:   decl list;
97     stmt:    stmt;
98   }
```

LISTING 5 – Type des déclarations de fonctions ou procédures

2 Changements dans le typeur

2.1 Nouvel environnement

L'environnement a été modifié pour être capable de générer des identifiants uniques pour la production de code, et de calculer les différents décalages pour les paramètres et les variables locales. Il atteint désormais le nombre effrayant de 9 champs :

typer.ml[82-93]

```
82  type t =
83    {
84      objs    : map;    (* Maps an identifier to its type *)
85      consts  : Sset.t; (* Set of variables that shouldn't be modified *)
86      notdef  : Sset.t; (* Set of types that haven't been defined yet *)
87      byref   : Sset.t; (* Set of variables that have been passed by reference *)
88      subtypes: Tset.t; (* Contains (t1, t2) pairs where t2 is a subtype of t1 *)
89      current_level: int;
90      offset_params: int;
91      offset_vars:   int;
92      unique_ids: (ident_desc * int) list Smap.t;
93    }
```

Listing 6 – Type de l'environnement

Certains champs pourraient (devraient...) sûrement être regroupés. De plus, je me rends compte que la structure choisie pour représenter les variables déclarées `in`[2], `in out`[3] ou les types non encore déclarés reposent uniquement sur le nom donné par l'utilisateur. Cela pourrait poser problème si une nouvelle variable *shadow* un paramètre passé `in` à un niveau de déclaration supérieur... (mais restructurer une $n^{\text{ème}}$ fois l'environnement quelques heures avant de rendre le projet n'est pas une bonne idée).

2.2 Autres

Le typeur produit désormais un *AST* correct même dans le cas des fonctions sans paramètres (elles étaient correctement typées mais pas correctement représentées).

Le déréférencement est explicité lorsque c'est nécessaire.

Les boucles `for` sont remplacées en boucles `while`.

On rajoute automatiquement une instruction `return null;` dans le corps des procédures.

2. étrange, pas de coloration...
3. le mystère s'épaissit...

3 Production de code

3.1 Commentaires

On a rajouté la possibilité d'écrire en commentaires les opérations de l'arbre de syntaxe abstraite correspondant aux instructions de l'assembleur produit, afin de faciliter le débogage (remplacer la ligne 7 de `compiler.ml` par `let () = allow_comments true`).

3.2 Fonctions auxiliaires

Conventions : toutes les tailles et décalages sont en *quads* et doivent donc être multipliés par huit lorsque c'est nécessaire. Tous les pointeurs sur un objet pointe au bas de la zone correspondante (i.e. la zone d'adresse la plus petite).

On dispose d'une fonction `copy` permettant de recopier une zone de la mémoire de taille quelconque à un autre endroit, ainsi que d'une fonction `follow_rbp` permettant de mettre la valeur du `rbp` père dans `rsi`.

3.3 Compilation des expressions

Un paramètre `~push` permet d'indiquer si le résultat doit être empilé ou si son adresse doit être mise dans `rsi` (possible lorsque c'est nécessaire d'après le typage).

Lorsqu'on compile `R.field`, on fait attention de recopier le champ au dessus de la pile et d'écraser le reste en manipulant correctement `rsp`.

Les opérations arithmétiques sont effectuées sur 32 bits avec extension de signe. Les tests d'égalité sont fait *quad* par *quad* avec un `jmp` dès que le test n'est plus vérifié. Les opérateurs paresseux sont compilés correctement avec une valeur par défaut dans le registre contenant le résultat, qui est modifiée ou non suivant le test de la première opérande (avec éventuellement un `jmp`).

Les registres allouées avec `new` sont sauvegardés sur le tas. On initialise la mémoire avec des 0 afin qu'un test du type `L.Next = null` fonctionne correctement pour une liste chaînée `L`.

Lors d'un appel de fonction, on alloue éventuellement l'espace de retour, et on empile ou bien la valeur du paramètre ou son adresse suivant son mode de passage (cette information ayant été répercutée dans les décalages lors du typage). On empile ensuite la valeur du `rbp` père.

3.4 Compilation des instructions

On ne compile les boucles `while` qu'avec un seul saut au lieu de 2.

Pour l'instruction `return`, on met soit le résultat de l'expression (ici toujours 0, mais on pourrait imaginer que ce soit également le cas si la taille de la valeur de retour est 1) dans `rax`, soit on copie son résultat dans l'espace de retour. On libère ensuite le tableau d'activation, restaure `rbp` puis `ret`.

3.5 Compilation des déclarations

Pour une déclaration de fonction, on sauvegarde le `rbp` de l'appelant, et alloue le tableau d'activation en l'initialisant avec des 0. On effectue ensuite les différentes initialisations des variables locales puis exécute le corps de la fonction.

4 *Bugs*

EDIT : tous les tests passent ! J'avais oublié d'empiler le résultat de `not B` sur la pile (qui l'eût cru ?). Je laisse néanmoins les paragraphes suivants, pour leur indéniable valeur historique.

4.1 `nathanael-pi.adb` (si c'est bien un test officiel, sinon tout marche très bien ;))

Le programme produit semble boucler puisque après quelques secondes, *bash* répond « Processus arrêté » (le code produit avec *gnat* s'éxécute en moins d'une seconde sur ma machine, en guise de comparaison).

L'affichage de la valeur de `rsp` à chaque appel de fonction (ça fait beaucoup) semble suggérer qu'il ne s'agit pas d'un débordement de pile.

4.2 `queens.adb`

Le comportement du code produit est extrêmement étrange :

— avec probabilité (!) $\frac{1}{2}$, le programme effectue une division par 0,

— avec probabilite $\frac{1}{2}$, le programme ne rencontre aucune erreur mais trouve 0 solutions pour toutes les valeurs de N.

Ce n'est pas un problème d'initialisation de mémoire puisqu'il n'y a aucun pointeurs dans le code (et celle-ci est a été initialisée exprès pour ce test, sans succès). Un examen minutieux de l'AST produit et des décalages et niveaux calculés par le typeur indique que l'erreur ne provient pas de là. De plus, la technique consistant à commenter du code pour trouver un exemple minimal ne marche pas : la probabilité de diviser par 0 augmente ou diminue en commentant certaines lignes sans que je comprenne pourquoi. Afficher la valeur de la variable `I` de la fonction `T` à chaque tour de boucle affiche parfois uniquement 0, parfois uniquement 61, parfois 0 puis 61 puis 421678 puis ')' puis *segfault*... Bref, mystère !