Projet de programmation
Compilateur de Mini-Caml

Stéphane Glondu*

Lundi 24 janvier 2005

Table des matières

*ENS de Cachan, magistère STIC, première année, premier semestre.

1 Organisation générale du projet

1.1 Description des modules

Le projet est articulé autour de sept modules :

- `Mltype` : définition de la syntaxe abstraite (fourni) ;
- `Lexer`, `Parser` : transformation du fichier source en syntaxe abstraite (fourni) ;
- `Projet` : quelques fonctions auxiliaires utilisées par la suite (partiellement fourni) ;
- `Translate` : le compilateur proprement dit ;
- `Typeur` : le typeur ;
- `Compile` : c'est ce module qui rassemble le tout.

Les quatre premiers modules étant plutôt indissociables, je les ai regroupés dans une bibliothèque `parser.cma` utilisée dans le "*toplevel*" d'Objective Caml lors du développement de `Translate` et de `Typeur` (voir le fichier `debug/draft.ml`).

Le module `Mltype` définit la syntaxe abstraite — que je ne rappellerai pas ici — d'une expression et d'un programme Mini-Caml.

La description des modules `Lexer` et `Parser` sort du cadre de ce projet.

Le module `Projet` contient essentiellement deux fonctions :

- `parse : string -> Mltype.letrec list` :
  transforme le fichier source dont le nom est fourni en argument en syntaxe abstraite ;
- `var_libres : Mltype.expr -> Ident.t` :
  renvoie l'ensemble des variables libres de l'expression Mini-Caml passée en argument,

où `Ident` est obtenu à l'aide du module `Set` d'Objective Caml :

```
module Ident = Set.Make(String)
```

Les modules `Translate` et `Typeur` seront décrits en détail dans les sections 3 et 4.

Le module `Compile` contient la séquence d'instructions qui est exécutée lors de l'invocation de `compile`. Précisément :

1. il lit le fichier source et le transforme en syntaxe abstraite ;
2. il vérifie les types ;

3. il génère un fichier assembleur ;
4. il appelle `gcc` sur le fichier généré.

J'ai aussi fait un "*Makefile*". Pour compiler le tout, il suffit de taper `make` à partir du répertoire `src/` du projet. Cela crée (notamment) l'exécutable `compile`.

1.2 Tests effectués

J'ai regroupé dans le sous-répertoire `tests` des programmes Mini-Caml qui testent diverses possibilités de ce langage (ils reprennent notamment ceux disponibles sur la page de Manuel Baclet). Toutes les tournures syntaxiques sont testées, et je pense aussi avoir testé pas mal d'aspects fonctionnels :

- `factorielle.ml`, `fib.ml` : quelques exemples assez classiques... qui testent néanmoins la récursivité, les comparaisons, et les e/s ;
- `arithmetics.ml`, `composition.ml` : teste les fonctions en tant que valeurs ordinaires, les fonctions curryfiées et les opérations arithmétiques ;
- `cat.ml` : teste les appels aux primitives C ;
- `referencep.ml` : encore un autre exemple classique avec les références (le compteur utilisé en 3.2 et en 4.1) ;
- `tuples.ml` : teste les "*tuples*" (tableaux à lecture seule) ;
- `tordu.ml` : un exemple que m'a proposé Manuel Baclet ;
- `tweak.ml` : un programme de test des extensions décrites en 4.5 ;
- `ooops.ml` : un programme qui met en évidence le problème que pose le typage des objets mutables (notamment les références) ; cet exemple devrait normalement être rejeté.

Des tests sont automatiquement réalisés par "*Makefile*" (`make` à partir de `tests/` ou `make tests` à partir de `src/`).

2 Bibliothèque standard

Les programmes Mini-Caml pourront utiliser les valeurs prédéfinies suivantes (non détaillées ici) :

```
afficher_int : int -> unit
int_of_string : string -> int
argc : int
argv : string tuple
r : string
```

où `r` est la chaîne `"r"` (utilisée avec `fopen`).

De plus, le typeur autorise l'utilisation (sûre) des primitives C suivantes :

```
fopen : string -> string -> file
fgetc : file -> int
fclose : file -> unit
putchar : int -> unit
```

3 Compilateur

J'ai codé le compilateur proprement dit dans le module `Translate`. Sa fonction « ultime » est :

```
output_file : bool -> string -> Mltype.letrec list -> unit
```

Elle écrit dans le fichier dont le nom est passé en argument le programme compilé (en assembleur PC, syntaxe `gcc`) correspondant au programme Mini-Caml passé en argument. Le rôle du booléen sera expliqué en 3.6. Détaillons un peu...

3.1 Présentation du problème et solution adoptée

On veut « traduire » un programme Mini-Caml en assembleur. L'idée générale est un parcours en profondeur d'abord de l'arbre du programme : pour évaluer une *expression* non élémentaire, on évalue d'abord (récursivement) ses sous-expressions qu'on empile sur la pile du processeur, puis on dépile toutes ces valeurs intermédiaires et on empile (à leur place) le résultat. Plus précisément, un invariant pour le programme assembleur généré sera : si s désigne la pile *avant* le calcul d'une expression Mini-Caml, la pile *après* ce calcul sera $s \cdot r$, où r est le résultat du calcul (que l'on fera en sorte qu'il tienne toujours sur un mot de 32 bits). Un *programme* étant une liste d'*expressions*, il suffira alors d'évaluer successivement ces expressions.

3.2 Quelques détails techniques

Le code assembleur généré le sera d'abord sous forme de liste d'instructions, qui seront du type :

```
type asm_instr =
  | APOP of binding
  | APUSH of binding
  | AOTHER of string
  | ALABEL of string
  | ACLEANUP of int
```

On verra en 3.3.3 la définition du type `binding`.

Enfin, j'utilise aussi une fonction :

```
etiquette : unit -> string
```

qui renvoie un identifiant frais.

3.3 Compilation d'une expression

3.3.1 Présentation générale

La principale fonction du module est :

```
asm_of_expr :
  binding Environ.t -> int ->
  asm_instr list    -> asm_instr list list ->
  Mltype.expr       -> asm_instr list * asm_instr list list
```

Cette fonction récursive est la fonction de « visite » du parcours en profondeur évoqué ci-dessus. Supposons qu'elle soit appelée alors qu'une certaine portion P du programme Mini-Caml a déjà été compilée — autrement dit, P est un ensemble de nœuds déjà visités. Les arguments de `asm_of_expr` sont :

1. un environnement `table` ;
2. un entier `level` qui représente le nombre de mots (de 32 bits) que P a ajouté à la pile du processeur depuis le début de l'exécution du programme *compilé* ;
3. la liste `accu` (renversée) des instructions assembleur de P ;
4. une liste de routines `accu2` qu'utilise P ;
5. une nouvelle expression à compiler `expr`,

et elle renvoie un couple `(accu, accu2)` mis à jour — c'est-à-dire dans lequel ont été ajoutées de nouvelles instructions et routines qui correspondent à la compilation de `expr`.

L'implémentation de `asm_of_expr` est un (énorme ?) filtrage à 22 cas. Si :

- on représente un entier directement par un mot de 32 bits, un booléen par un entier qui vaut 0 s'il est faux et 1 sinon, et `unit` par un entier quelconque ;
- on se donne un ordre d'évaluation des sous-expressions (qui n'est vraiment canonique que pour `SEQ` — j'ai choisi l'ordre de la droite vers la gauche pour les autres) ;
- on sait allouer de la mémoire en assembleur et y accéder ;
- on sait utiliser les primitives C en assembleur,

alors un certain nombre de cas — `INT`, `UNIT`, `PLUS`, `MINUS`, `TIMES`, `OPP`, `DIV`, `REM`, `EQUAL`, `LESS`, `IF`, `SEQ`, `TUPLE`, `PROJ`, `REF`, `BANG`, `SET` et `CALLPRIM` — se traitent de façon assez simple et naturelle exclusivement sur les registres `%eax`, `%ebx`, `%ecx`, `%edx` (en fait, on pourrait se contenter de deux registres, mais certaines instructions assembleur nous contraignent à utiliser des registres déterminés) : ils ne font que rajouter des instructions à la liste `accu`. Il ne reste donc que quatre cas que je vais (un peu) détailler.

3.3.2 Les fonctions : FUN et APP

Une fonction est représentée formellement par une *clôture*, c'est-à-dire un couple (P, ρ) d'un bout de code assembleur P et d'un environnement ρ. On pourrait, lors de la compilation de `fun x -> a` :

1. allouer un $(n+2)$-uplet $(f, x_0, \ldots, x_n)$ de mots de 32 bits (on peut déterminer n à la compilation) en mémoire ;
2. recopier dans $x_1, \ldots, x_n$ l'environnement, en laissant x_0 vide pour l'argument `x` ;
3. compiler le corps de la fonction en tant que routine dans `accu2` (on pourrait aussi compiler directement ce corps dans `accu`, mais cela obligerait à faire un saut), et en utilisant l'environnement sauvegardé ;
4. stocker dans f l'adresse du corps de la fonction,

mais on gaspillerait ainsi beaucoup de mémoire ! On peut déjà remarquer que seules les variables libres de `a` vont être utilisées dans le corps, donc il n'est pas nécessaire de sauvegarder les autres. Il n'est pas non plus nécessaire de sauvegarder les valeurs définies à la « racine » du programme — c'est-à-dire dans un `let [rec] y = ... ;;` antérieur ou dans la bibliothèque standard — car une fois que ces valeurs rentrent dans l'environnement, elle n'en ressortent plus.

Ainsi, le nombre de valeurs sauvegardées n diminue à de façon significative à n' — par exemple, les valeurs définies à la « racine » du programme prennent chacune un espace mémoire constant au lieu de suivre une progression arithmétique comme dans la première solution.

De plus, plutôt que de réserver un mot en mémoire qui sera souvent vide (l'argument), j'ai choisi de stocker l'argument dans le registre `%edi`. La *valeur* (au sens « opérationnel » du terme) sera l'adresse du $(n'+1)$-uplet $(f, x_1, \ldots, x_{n'})$, qui tient sur un mot de 32 bits. Lors de l'exécution du corps de la fonction — donc du code à l'adresse f — on fera en sorte que l'adresse du $(n'+1)$-uplet soit dans le registre `%esi`, et le corps de la fonction placera sa valeur de retour dans `%edi` de telle sorte que si la pile juste *avant* avant l'appel de f est s (qui se matérialise donc par un `call *(%esi)`), alors la

pile *après* cet appel soit toujours s. Bien sûr, il faudra sauvegarder (sur la pile) les précédentes valeurs de `%esi` et `%edi`, car on peut déjà se trouver dans le corps d'une fonction ! Quand on aura tout restauré, on pourra enfin empiler la valeur de retour de f (en faisant attention à `%esi`).

On a ainsi réglé les cas `FUN` et `APP` (pour `APP`, l'ordre d'évaluation des sous-expressions n'est toujours pas canonique — j'ai choisi d'évaluer l'argument *avant* la fonction).

3.3.3 L'environnement : LET et VAR

On pourrait implémenter `let x = a in b` comme `(fun x -> a) b`, et se ramener ainsi aux cas précédents. Cependant, cela a l'inconvénient de créer une fonction, ce qui est — on l'a vu — un processus assez coûteux pour le programme généré. De plus, cela crée un saut inutile dans le programme assembleur. Je propose donc une autre solution.

Pour pouvoir gérer les expressions `let x = a in b`, j'utilise un environnement qui associe à chaque nom de variable `x` (en fait, une chaîne de caractères) l'emplacement *à l'exécution* de l'expression `a` *calculée*. J'utilise le module `Map` d'Objective Caml :

```
module Environ = Map.Make(String)
```

et pour les emplacements à l'exécution, j'utilise le type suivant :

```
type binding =
  | StackIndex of int
  | HeapIndex of int
  | Absolute of string
  | Argument
```

où :

- `StackIndex i` désigne une valeur sur la pile, `i` étant le `level` auquel elle a été empilée (cela permet donc, avec le `level` courant, calculer un décalage par rapport à l'adresse pointée par `%esp` ;
- `HeapIndex i` désigne x_i dans une clôture $(f, x_1, \ldots, x_n)$ — c'est donc un décalage par rapport à l'adresse pointée par `%esi` ;
- `Absolute s` représente une valeur définie à la racine du programme ;
- `Argument` désigne l'argument de la fonction en cours d'exécution — c'est donc indéfini si on est à la racine du programme, sinon il s'agit du registre `%edi`.

On a ainsi réglé les cas `LET` et `VAR`.

3.3.4 Ne manquerait-il pas quelque chose ?

Le programme généré fait appel à `malloc` en de multiples occasions. Pourtant, il ne fait jamais appel à `free` ! En Objective Caml, les `free` sont gérés par le "*garbage collector*". En concevoir un sort — selon moi — du cadre de ce projet. J'ai néanmoins essayé de voir si on ne pouvait pas déterminer facilement si certains emplacements mémoire n'étaient plus utilisés... mais des problèmes rencontrés avec les références m'ont fait abandonner !

Le problème peut se faire fortement ressentir avec des fonctions curryfiées à au moins deux arguments : à chaque application « totale », au moins un mot est alloué pour une utilisation unique et est définitivement perdu. Ce problème se serait aussi fait ressentir si on avait implémenté les `let ... in` de manière naïve.

3.4 Gestion de la bibliothèque standard

J'ai codé la fonction suivante :

```
stdlib :
  asm_instr list ->
  asm_instr list list ->
  string -> asm_instr list * asm_instr list list
```

Dès qu'une variable libre `stdfun` apparaît dans une expression à la racine du programme alors qu'elle n'a pas été définie précédemment dans le programme, un appel à `stdlib accu accu2 stdfun` est effectué. Il a le même comportement qu'un (hypothétique) :

```
asm_of_expr dum_table dum_level accu accu2 dum_expr
```

où tout ce qui commence par `dum_` est ignoré (toutes les valeurs de la bibliothèque standard sont précompilées dans la fonction `stdlib`), et où le résultat est stocké à l'exécution quelque part en mémoire ailleurs que sur la pile (ici, dans des variables assembleur). S'il n'y a pas de `stdfun` dans la bibliothèque standard, alors une exception `Unbound stdfun` est déclenchée, et la compilation échoue.

Cette gestion permet de n'inclure dans le programme assembleur généré que les fonctions standard vraiment utilisées.

3.5 Compilation d'un programme

Ici, un programme est en fait une liste de valeurs auxquelles ont donne un nom. Ces valeurs peuvent être des fonctions récursives. On procède comme

suit, en partant d'un environnement `table`, d'un programme assembleur `accu` et d'une liste de routines `accu2` initialement vides tous les trois :

1. on alloue un emplacement mémoire μ (cette allocation est faite implicitement par `gcc` par l'intermédiaire de variables dans le programme assembleur) ;
2. pour un élément du programme de la forme `let rec f = a ;;`, on ajoute (ou modifie) dans `table` le lien de `f` vers μ ;
3. si `a` contient des variables libres dans `table`, on tente de voir si on ne fait pas appel à un élément de la bibliothèque standard ;
4. on compile `a` en appelant `asm_of_expr` et en mettant à jour `accu` et `accu2` ;
5. on répète le tout tant qu'il reste quelque chose à compiler...

Une fois que tout est compilé, on renverse toutes les listes et on les écrit dans le fichier de sortie avec les décorations typiques d'un programme assembleur.

3.6 Une (petite) optimisation

En pratique, j'ai remarqué que les programmes générés directement avec `asm_of_instr` présentaient de nombreux `push` immédiatement suivis d'un `pop`, ce qui pourrait se faire directement avec un `mov`... J'ai donc décidé d'arranger un peu cela en effectuant les « simplifications » suivantes :

```
push x          ->      mov x, y
pop  y
```

et :

```
push x
push z          ->      mov x, y
pop  t                  mov z, t
pop  y
```

La deuxième transformation est un peu audacieuse car il faut faire attention aux éventuels effets de bord. On peut aussi supprimer un `mov` quand sa source et sa destination sont les mêmes. Je ne sais pas si cela optimise vraiment l'exécution du programme compilé, mais cela réduit sensiblement le nombre de lignes du programme assembleur généré. Et cela permet notamment de charger directement des constantes dans les registres sans avoir à les stocker temporairement sur la pile, et d'éviter de nombreuses opérations inutiles !

J'ai ainsi défini une fonction :

```
simplify : asm_instr list -> asm_instr list
```

que je "*mappe*" sur toutes les séquences d'instructions assembleur générées avant de les écrire dans le fichier de sortie. Cette optimisation est activée si le booléen passé à `output_file` est vrai. Cette optimisation peut être désactivée en passant l'option `--no-simplify` à `compile`. Elle est activée par défaut.

À titre d'indication, les programmes assembleur générés avec les fichiers de tests ci-dessus totalisent 1208 lignes avec cette optimisation, et 1396 sans.

4 Typeur

Je vais maintenant décrire le module `Typeur`. Il s'agit d'une application directe du cours de typage. Sa fonction « ultime » est :

```
type_prog : Mltype.letrec list -> unit
```

Elle vérifie les types du programme passé en argument et affiche les types inférés de toutes les expressions qui sont liées par un `let`. Détaillons un peu...

4.1 Quelques détails techniques

Les termes de l'algèbre des types seront représentés par le type suivant :

```
type constructeur =
  | Base of string
  | Fun of constructeur * constructeur
  | Tuple of constructeur
  | Ref of constructeur
  | Var of int
```

Les variables de type seront identifiées par des entiers.

La fonction suivante renvoie à chaque appel une variable fraîche :

```
compteur : unit -> int
```

Pour représenter un ensemble de variables, j'utilise le module `Set` d'Objective Caml :

```
module VarSet = Set.Make(Integer)
```

Un type universel sera représenté par un couple :

```
(quantifiees, corps) : VarSet.t * constructeur
```

4.2 Unification

Un algorithme d'unification sera nécessaire pour l'inférence de types. J'utilise les règles suivantes, vues en cours de typage :

$$\frac{E \cup \{s \doteq s\}}{E}\ \textsf{trivial} \qquad \frac{E \cup \{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\}}{E \cup \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}}\ \textsf{decompose}$$

$$\frac{E \cup \{s \doteq \alpha\} \quad s \text{ n'est pas une variable}}{E \cup \{\alpha \doteq s\}}\ \textsf{reverse}$$

$$\frac{E \cup \{\alpha \doteq s\} \quad \alpha \in \mathrm{Var}(E) \quad \alpha \notin \mathrm{Var}(s)}{E\,\{\alpha/s\} \cup \{\alpha \doteq s\}}\ \textsf{replace}$$

avec les notations usuelles. On a vu en cours que ces règles terminent quel que soit l'ordre dans lequel elle sont appliquées. Pour l'implémentation, j'ai choisi de représenter un ensemble d'équations par une liste de couples de `constructeurs`. Initialement, on donne à toutes les équations le statut de *non-résolues*.

Les règles trivial et decompose sont d'abord appliquées aux équations non-résolues. Puis on parcourt *toutes* les équations en appliquant reverse. Si, lors de ce parcours :

1. on trouve une équation de la forme $f(s_1, \ldots, s_n) \doteq g(t_1, \ldots, t_m)$, alors le système n'a pas de solution et une exception `Clash (f, g)` est déclenchée ;
2. on trouve une équation de la forme $\alpha \doteq s$, avec $\alpha \in \mathrm{Var}(s)$, alors le système n'a pas de solution et une exception `Occur_check (alpha, s)` est déclenchée ;
3. on trouve une équation de la forme $\alpha \doteq s$, avec $\alpha \notin \mathrm{Var}(s)$, alors on applique la règle replace, on donne à $\alpha \doteq s$ le statut d'équation *résolue*, et on recommence le processus ;
4. on arrive à la fin de la liste, c'est que le système est sous forme résolue et décrit un plus général unificateur.

En pratique, les équations résolues et non résolues sont dans deux listes distinctes. J'ai choisi de représenter une substitution à l'aide du module `Map` d'Objective Caml :

```
module Substitution = Map.Make(Integer)
```

J'ai donc programmé une fonction :

```
mgu :
  (constructeur * constructeur) list ->
  constructeur Substitution.t
```

4.3 Algorithme de Damas-Milner-Tofte

Un *environnement de typage* est une fonction qui associe à des variables de programme un type polymorphe. J'ai choisi de les implémenter à l'aide du module `Map` d'Objective Caml :

```
module TypingEnv = Map.Make(String)
```

On définit :

$$\mathrm{Inst}(\forall(\alpha_1, \ldots, \alpha_n).A) = A\,\{\alpha_1/\beta_1, \ldots, \alpha_n/\beta_n\}\,,$$

où A est un type sans quantificateur, et $\beta_1, \ldots, \beta_n$ des variables fraîches, et :

$$\mathrm{Gen}(A, \Gamma) = \forall(\alpha_1, \ldots, \alpha_n).A,$$

où A est un type sans quantificateur, Γ un environnement de typage, et $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{VarLib}(A) \setminus \mathrm{VarLib}(\Gamma)$.

L'algorithme W de Damas-Milner-Tofte prend en argument un environnement Γ et un terme `M` et renvoie un type A (sans quantificateur) et une substitution σ telle que $\sigma(\Gamma) \vdash \texttt{M} : A$. Il est défini comme suit :

- $W((\Delta, \texttt{x} : A), \texttt{x}) = (\mathrm{Inst}(A), \mathrm{id})$;
- $W(\Gamma, \texttt{fun x -> m}) = (\rho(\alpha) \to A, \rho)$, où :

$$W((\Gamma, \texttt{x} : \alpha), \texttt{m}) = (A, \rho)$$

  et α est une variable fraîche ;
- $W(\Gamma, \texttt{m (n)}) = (\mu(\alpha), \mu \circ \rho_B \circ \rho_A)$, où :

$$\begin{aligned} W(\Gamma, \texttt{m}) &= (A, \rho_A) \\ W(\rho_A(\Gamma), \texttt{n}) &= (B, \rho_B) \\ \mu &= \texttt{mgu}\big(\rho_B(A) \doteq B \to \alpha\big), \end{aligned}$$

  et α est une variable fraîche ;
- $W(\Gamma, \texttt{let x = m in n}) = (B, \rho_B \circ \rho_A)$, où :

$$\begin{aligned} W(\Gamma, \texttt{m}) &= (A, \rho_A) \\ W\left(\big(\rho_A(\Gamma), \texttt{x} : \mathrm{Gen}(A, \rho_A(\Gamma))\big), \texttt{n}\right) &= (B, \rho_B). \end{aligned}$$

4.4 Implémentation

La principale fonction est :

```
subtype :
  (VarSet.t * constructeur) TypingEnv.t ->
  Mltype.expr -> constructeur * constructeur Substitution.t
```

Cette fonction récursive implémente la fonction W définie précédemment. Comme `asm_of_expr`, c'est un filtrage à 22 cas. Les cas `VAR`, `INT`, `UNIT`, `FUN`, `APP`, `CALLPRIM` et `LET` sont des applications directes du paragraphe précédent ; les autres cas — `PLUS`, `MINUS`, `TIMES`, `DIV`, `REM`, `OPP`, `REF`, `BANG`, `SET`, `PROJ`, `SEQ`, `LESS`, `EQUAL`, `IF` et `TUPLE` — peuvent facilement se ramener à l'application d'une fonction d'un type bien choisi.

Pour vérifier les types d'un programme Mini-Caml, on le tranforme en expression comme suit :

- on remplace chaque déclaration du genre :
  `let rec f = fun x -> a ;;`
  par :
  `let f = fix (fun x -> a) in`
  où `fix` (on fait en sorte que ce nom ne soit jamais utilisé ailleurs) est une fonction de type $\forall\alpha.(\alpha \to \alpha) \to \alpha$;
- on remplace les autres déclarations :
  `let rec f = a ;;`
  par :
  `let f = a in`
- à la fin du programme, on ajoute une constante (comme `()`).

Il suffit alors de typer l'expression obtenue dans un environnement contenant toutes les valeurs de la bibliothèque standard. Avec cette méthode, seules les fonctions peuvent être récursives.

4.5 Quelques extensions

Dans une séquence `a ; b`, `a` devrait être de type `unit`. Mais en réalité, son type n'influe pas sur le typage de l'expression. J'ai donc décidé d'autoriser `a` à avoir n'importe quel type. Néanmoins, si ce type n'est pas `unit`, un message d'avertissement est affiché.

Idéalement, toutes les primitives C utilisées dans un programme Mini-Caml devraient être explicitement autorisées. Cependant, j'ai décidé d'autoriser implicitement l'usage de n'importe quelle primitive C : si elle est inconnue, un message d'avertissement avec le type inféré est affiché. La cohérence entre les différentes utilisations de la même primitive C n'est pas vérifiée.

4.6 Typage des valeurs mutables

Le programme-test `ooops.ml` met en avant l'incorrection de l'algorithme exposé ci-dessus avec les références. Le problème vient de la généralisation faite dans le cas du `LET`. En m'inspirant de la littérature sur le sujet, j'ai décidé de modifier l'algorithme de typage. L'idée est de ne pas généraliser systématiquement les types.

Une expression est dite *expansive* si les deux conditions suivantes sont réunies :

- elle n'est pas de la forme `FUN` ;
- elle est, ou possède une sous-expression, de la forme `APP` (application d'une fonction différente de `fix`), `REF`, `BANG`, `SET`, `CALLPRIM`.

Intuitivement, les expressions non expansives ne manipulent pas de référence, et sont par conséquent « purement fonctionnelles ».

Dans l'algorithme W, la règle pour le `LET` est modifiée ainsi :

- $W(\Gamma, \texttt{let x = m in n}) = (B, \rho_B \circ \rho_A)$, où :

$$\begin{aligned} W(\Gamma, \texttt{m}) &= (A, \rho_A) \\ W\left(\left(\rho_A(\Gamma), \texttt{x} : A'\right), \texttt{n}\right) &= (B, \rho_B), \end{aligned}$$

$$\text{où} \quad A' = \begin{cases} \mathrm{Gen}(A, \rho_A(\Gamma)) & \text{si m est non expansive,} \\ A & \text{sinon.} \end{cases}$$

Cette méthode permet d'éviter les écueils comme `ooops.ml`, mais restreint (un peu) le polymorphisme.

5 Conclusion

Voici les imperfections que j'aurais aimé améliorer :

- le code généré n'exploite pas au mieux tous les registres du microprocesseur ;
- le problème déjà évoqué des `malloc` sans `free` ;
- mon implémentation de $\mathrm{VarLib}(A) \setminus \mathrm{VarLib}(\Gamma)$ dans l'algorithme de Damas-Milner-Tofte est inefficace ;
- les primitives C autorisées sont codées en dur dans le module `Typeur`.

La réalisation de ce projet a néanmoins été très instructive et a en particulier attisé ma curiosité sur les problèmes de typage avec des références.