Projet de compilation – L3 Informatique
Un traducteur vers C pour un petit ML

Nicolas Bedon, Arnaud Lefebvre

7 février 2018

1 Présentation générale

Il s'agit ici d'écrire un compilateur qui prend en entrée du code source pour un langage fonctionnel (qu'on appelera MINIML) à la ML, et qui génère en sortie du code C [1] (en version C11) qui, une fois compilé à son tour par `gcc`, s'exécutera comme l'aurait fait le programme de départ.

MINIML est un sous-ensemble de ocaml. Il contient les types de base `int`, `float`, `bool` et `string`. Pour toute valeur d'entier n, on peut y construire des n-uplets. Dans sa version de base, on ne peut pas y définir de nouveaux types, et le filtrage n'existe pas. En revanche, MINIML est équipé d'un système de typage (polymorphe, comme en ocaml). Les constructions syntaxiques sont, en plus de celles pour les expressions à base d'opérateurs arithmétiques, logiques et sur les chaînes de caractères :

- le `let` et le `let...in` ;
- le `if` avec son `else` ;
- la possibilité de considérer un opérateur comme une fonction (exemple : `(+)`).

On peut écrire des fonctions, mêmes récursives. En revanche, la définition simultanée `let...and...` n'existe pas, donc il n'est pas possible de définir des fonctions mutuellement récursives. Pour simplifier, nous supposerons que toutes les fonctions n'ont qu'un paramètre. On rappelle le principe de *currification* : toute fonction à $n > 1$ paramètres peut se voir comme une fonction à 1 paramètre qui retourne une fonction à $n - 1$ paramètres.

Voici un exemple de code en MINIML. Des couleurs ont été ajoutées pour rappeler la portée des noms symboliques.

```
# let x=1;;
# let x=2 in x*x;;
# x;;
# (let x=2 in x*x)+x;;
# (let x=2 in let y=3 in x+y)+x;;
# (let x=2 in let y=3 in x+y)+x+(let x=5 in x*x);;
# let y=2 in let z=y in let y=3 in x+y+z;;
# let rec fact n = if n=0 then 1 else n*(fact (n-1));;
# fact 5;;
# (function x -> x+1) (fact x);;
```

La sortie du programme compilé et exécuté est :

```
val x : int = 1
- : int = 4
- : int = 1
- : int = 5
```

1. Si générer du C ne vous sied pas, vous pourrez générer du Java ou du C++ à la place.

```
- : int = 6
- : int = 31
- : int = 6
val fact : int -> int = <fun>
- : int = 120
- : int = 2
```

Attention : dans cet exemple la notation `->` dans `function x -> x+1` sert uniquement à exprimer que la fonction associe $x+1$ à x. Ca n'est pas la notation de filtrage comme en ocaml.

2 Explications détaillées

2.1 Évaluation des expressions miniML

En miniML comme en ocaml, les expressions s'évaluent dans un *environnement*, qui permet d'associer une valeur et son type à un nom symbolique. Ici l'environnement est noté Γ. Il contient des triplets (nom de symbole, valeur, type). L'insertion d'un nouveau triplet (x, v, τ) dans un environnement Γ est notée $(x, v, \tau) :: \Gamma$. La notation $\Gamma(x)$ désigne le triplet (x, v, τ) que Γ associe au nom x : il s'agit de l'entrée (x, v, τ) la plus récemment insérée dans Γ pour x en première composante. La notation $\Gamma@\Gamma'$ désigne l'environnement obtenu en insérant dans l'environnement Γ' toutes les entrées de Γ dans l'ordre où elles ont été insérées dans Γ. Si les environnements sont implantés par des listes, les notations :: et @ correspondent à leurs analogues ocaml sur les listes, et $\Gamma(x)$ retourne le résultat de la recherche du premier triplet (x, v, t) dans Γ.

L'évaluation des expressions est récursive et obeit aux règles (elles ne sont pas toutes données, on peut en déduire les manquantes) suivantes :

$$(Const) : \frac{}{\Gamma \vdash \text{constante} : \tau \rightsquigarrow \text{constante} : \tau}$$

$$(Nom) : \frac{\Gamma(x) = (x, e, \tau) \qquad \Gamma \vdash e \rightsquigarrow v : \tau}{\Gamma \vdash x \rightsquigarrow v : \tau}$$

$$(Op) : \frac{\Gamma \vdash e \rightsquigarrow v : \texttt{int} \qquad \Gamma \vdash e' \rightsquigarrow v' : \texttt{int}}{\Gamma \vdash e + e' \rightsquigarrow v + v' : \texttt{int}}$$

$$(If) : \frac{\Gamma \vdash c \rightsquigarrow \texttt{true} : \texttt{bool} \qquad \Gamma \vdash e \rightsquigarrow v : \tau \qquad \Gamma \vdash e' \rightsquigarrow v' : \tau}{\Gamma \vdash \texttt{if } c \texttt{ then } e \texttt{ else } e' \rightsquigarrow v : \tau}$$

$$(LetIn) : \frac{\Gamma \vdash e \rightsquigarrow v : \tau \qquad (s, v, \tau) :: \Gamma \vdash e' \rightsquigarrow v' : \tau'}{\Gamma \vdash \texttt{let } s = e \texttt{ in } e' \rightsquigarrow v' : \tau'}$$

$$(App) : \frac{\Gamma \vdash e \rightsquigarrow \text{fun} < s, x, \Gamma' > : \tau \to \tau' \qquad \Gamma \vdash e' \rightsquigarrow v : \tau \qquad (s, v, \tau) :: \Gamma' \vdash x \rightsquigarrow v' : \tau'}{\Gamma \vdash e\ e' \rightsquigarrow v' : \tau'}$$

$$(Fun) : \frac{}{\Gamma \vdash \text{fun} < x, c, \Gamma' > : \tau \to \tau' \rightsquigarrow \text{fun} < x, c, \Gamma'@\Gamma > : \tau \to \tau'}$$

La notation fun $< x, c, \Gamma >$ signifie « fonction dont le paramètre a nom x et dont le corps est c, définie dans l'environnement Γ ». Les règles se lisent de la manière suivante. Nous n'en lisons que (App), la lecture des autres s'en déduit. Si

- dans l'environnement Γ l'expression e s'évalue en une fonction de type $\tau \to \tau'$, de paramètre nommé s, de corps x et d'environnement Γ' ;
- dans Γ l'expression e' s'évalue en v de type τ ;
- dans $(s, v, t) :: \Gamma$ l'expression x s'évalue en v' de type τ'

alors dans Γ l'application de e à e' s'évalue en v' de type τ'.

2.2 Typage

Dans un langage fonctionnel comme miniML, le type d'une expression e est *inféré* par la machine, avant son évaluation. L'inférence du type de e est réalisée en 3 étapes.

2.2.1 Annotation des sous-expressions et des variables

La première consiste à typer chaque sous-expression (au sens large) e' (ou variable) de e par une nouvelle inconnue de type $\tau_{e'}$.

Exemple 1 $\texttt{function}\ f \to f\ (f\ 3)$
Annotations : $(\texttt{function}\ f_\alpha \to (f_\alpha\ (f_\alpha\ 3_\beta)_\gamma)_\delta)_\eta$
La lettre grecque en indice de chaque sous-expression ou variable est la nouvelle variable de type associée à la sous-expression ou à la variable.

2.2.2 Contraintes

La seconde étape consiste à calculer les contraintes existantes entre tous les types, sous la forme d'un système d'équations de types. Par exemple, dans l'expression annotée $(e_{\tau_e}\ e'_{\tau_{e'}})_\gamma$, τ_e est nécessairement une fonction $\tau_{e'} \to \gamma$, ce qu'on note $\tau_e = \tau_{e'} \to \gamma$.

Exemple 2 *Les contraintes calculées à partir de l'Exemple 1 sont :* $\beta = \texttt{int}, \alpha = \beta \to \gamma, \alpha = \gamma \to \delta, \eta = \alpha \to \delta$.

2.2.3 Résolution

Il faut maintenant trouver une solution au système d'équations calculé à l'étape précédente. Elle peut être calculée par un algorithme d'unification, par exemple celui de Martelli et Montanari : bien que peu efficace, il a l'avantage d'être simple. Si possible, il transforme un ensemble $G = \{t_1 = t'_1, \ldots, t_n = t'_n\}$ d'équations en un autre ensemble $\{x_1 = u_1, \ldots, x_m = u_m\}$ d'équations représentant le système est résolu. Sinon, il calcule une erreur (notée $\bot$ ci-dessous). On transforme ($\rightsquigarrow$) G d'après sa forme par une des règles ci-dessous :

$$G \cup \{t = t\} \rightsquigarrow G \tag{1}$$
$$G \cup \{t_1 \to t_2 = t'_1 \to t'_2\} \rightsquigarrow G \cup \{t_1 = t'_1, t_2 = t'_2\} \tag{2}$$
$$G \cup \{t = x\} \rightsquigarrow G \cup \{x = t\} \text{ si } x \text{ est une variable} \tag{3}$$
$$G \cup \{x = t\} \rightsquigarrow G[x/t] \cup \{x = t\} \text{ si } x \notin vars(t) \text{ et } x \in vars(G) \tag{4}$$
$$G \cup \{x = t_1 \to t_2\} \rightsquigarrow \bot \text{ si } x \in vars(t_1 \to t_2) \tag{5}$$

jusqu'à ce qu'on ne puisse plus. La notation $G[x/t]$ désigne G dans lequel chaque apparition de x est remplacée par t.

Exemple 3 *Le système de l'Exemple 2 est ainsi successivement transformé de la manière suivante :*
(-) : $\beta = \texttt{int}, \alpha = \beta \to \gamma, \alpha = \gamma \to \delta, \eta = \alpha \to \delta$
(4) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \gamma, \alpha = \gamma \to \delta, \eta = \alpha \to \delta$
(4) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \gamma, \texttt{int} \to \gamma = \gamma \to \delta, \eta = (\texttt{int} \to \gamma) \to \delta$
(2) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \gamma, \texttt{int} = \gamma, \gamma = \delta, \eta = (\texttt{int} \to \gamma) \to \delta$
(3) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \gamma, \gamma = \texttt{int}, \gamma = \delta, \eta = (\texttt{int} \to \gamma) \to \delta$
(4) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \texttt{int}, \gamma = \texttt{int}, \texttt{int} = \delta, \eta = (\texttt{int} \to \texttt{int}) \to \delta$
(3) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \texttt{int}, \gamma = \texttt{int}, \delta = \texttt{int}, \eta = (\texttt{int} \to \texttt{int}) \to \delta$
(4) : $\beta = \texttt{int}, \alpha = \texttt{int} \to \texttt{int}, \gamma = \texttt{int}, \delta = \texttt{int}, \eta = (\texttt{int} \to \texttt{int}) \to \texttt{int}$

2.3 Code C généré

Vous devrez définir entre autres :
— un type de données (appelons-le `expr`), typiquement un arbre ou une structure plus complexe, pour représenter une expression MINIML dans le programme C généré ;
— un type de données (appelons-le `type`), typiquement un arbre ou une structure plus complexe, pour représenter un type MINIML dans le programme C généré ;

— un type de données (appelons-le `environment`), typiquement une liste chaînée mais ça peut être autre chose, pour représenter un environnement d'exécution miniML dans le programme C généré.

Vous devrez également écrire des fonctions qui typiquement ressemblerons à :

— `int type_expr(const expr * expr, const environment * env, type ** res)` : évalue le type de l'expression en argument dans l'environnement spécifié, et stocke le résultat dans `*res`. La fonction retourne un entier qui indique si elle s'est bien passée ou non ;

— `int evaluate_expr(const expr * expr, environment * env, expr ** res)` : évalue l'expression en argument dans l'environnement spécifié ; l'expression évaluée est une autre expression, normalement plus simple, que `evaluate_expr` stocke dans `*res`. La fonction retourne un entier qui indique si elle s'est bien passée ou non. Elle peut dans certains cas modifier l'environnement (par exemple si l'expression évaluée est un `let`).

— des fonctions d'ajout/suppression/recherche dans l'environnement

et bien d'autres ! Ces fonctions feront partie du programme généré. Comme ce sont les mêmes pour chaque programme généré, vous pourrez les mettre à part dans une bibliothèque qui sera liée avec le reste du programme généré.

Le reste du programme généré pourra ressembler à ce qui suit. Supposons que le programme miniML en entrée soit composé de 10 expressions (chacune se termine par `;;`, il y a donc au total 10 `;;` dans le programme miniML). En sortie de votre projet, on pourra retrouver du code C ressemblant à :

```
#define NB_EXPR 10 /* #exprs in miniML source */

int main(int argc, char *argv[]) {
  expr exprs[NB_EXPR];
  environment env;

  init_env( &env );

  exprs[0] = <data structure for 1st expression>;
  ...
  exprs[NB_EXPR-1] = <data structure for last expression>;

  for (size_t i=0; i<NB_EXPR; ++i) {
    type * res_type = NULL;
    expr * res_eval = NULL;
    if ( type_expr( &exprs[i], &env, &res_type ) != 0 ) {
      // An error has occured.
      ...
    } else {
      // No type error, proceed to evaluation
      if ( evaluate_expr( &exprs[i], &env, &res_eval ) != 0 ) {
        // An error has occured.
        ...
      } else {
        // Type and evaluation are good, send them to output.
        ...
      }
    }
    // Free resources allocated in this loop turn
    ...
  }

  // Free all resources
```

```
  ...

  return EXIT_SUCCESS;
}
```

3 Ce qu'il vous est demandé

Vous devez écrire en C11, et en utilisant `flex` et `bison` pour les parties concernant les analyses lexicales et syntaxiques, un compilateur ayant les capacités présentées dans la section précédente.

Bien entendu, votre projet devra être écrit le plus proprement possible : algorithmique adaptée, code clair et commenté.

Vous pouvez étendre le projet si vous le souhaitez, en rajoutant des fonctionnalités par exemple (définition de types utilisateur ou d'autres types du langage, exceptions, filtrage, définitions simultanées, etc). Cependant, ne le faites que si la base qui vous est demandée est implantée et fonctionne correctement : il est préférable d'avoir un projet qui fait correctement le minimum plutôt que d'avoir un projet étendu dont le minimum demandé ne fonctionne pas.

Votre projet devra être rendu avec un jeu d'exemples illustrant le mieux possible ses fonctionnalités, ainsi qu'un rapport contenant un rapport de développement et un manuel d'utilisation.

Il devra être développé individuellement ou par binôme, et rendu au plus tard le *25 mars 2018* au soir, dans une archive au format `tar` gzippé de nom `FrancoisDupontJacquesDurant.tar.gz` pour un binôme (si Francois Dupont et Jacques Durant sont vos noms), envoyée en pièce jointe à un courriel de sujet « Projet de compilation L3 Info » à l'adresse de courriel de votre chargé de TP (`Nicolas.Bedon@univ-rouen.fr`, `Arnaud.Lefebvre@univ-rouen.fr`). Si vous avez fait le projet en binôme à cheval sur deux groupes de TP, votre projet est à envoyer aux deux chargés de TP. Vous vous mettrez en copie du courriel pour vérifier que vous n'oubliez pas la pièce jointe. L'extraction du fichier d'archive devra produire un répertoire de nom `FrancoisDupontJacquesDurant` contenant le code source de votre projet, un makefile, des jeux d'exemples et un rapport de projet.

Votre projet fera l'objet d'une soutenance sur machine. Il devra en particulier compiler et s'exécuter sans erreur et sans avertissement dans les salles de travaux pratiques.