Mini Ml

Martin Zivojinovic, Raghid Osseiran

1 Parseur simple : compréhension

1. Automate LR(0)

On considère la grammaire simplifiée suivante, où les terminaux `ID`, `INT` et `built_in` sont regroupés sous un unique terminal `T` :

$$\begin{aligned}
&\text{expr} \to \text{FUN T ARROW expr}\\
&\text{expr} \to \text{expr expr}\\
&\text{expr} \to \text{T}\\
&\text{expr} \to (\text{expr})
\end{aligned}$$

Pour construire l'automate LR(0), on ajoute un axiome : $\text{expr}' \to \text{expr}$.
L'automate contient des états pour :

— la lecture d'un terminal `T` ;

— le début d'une définition de fonction ;

— la reconnaissance récursive des applications ;

— l'utilisation des parenthèses.

La présence de la règle récursive gauche $\text{expr} \to \text{expr expr}$ entraîne des conflits classiques dans un tel automate.

2. Conflits dans l'automate SLR

Les conflits apparaissent dans les états où l'automate hésite entre :

— **réduire** l'expression courante via `expr → T`,

— ou **décaler** pour appliquer la règle `expr → expr expr`.

C'est un conflit **shift/reduce** typique des grammaires ambiguës avec application gauche non prioritaire.

3. Exemple de séquence ambiguë

Prenons l'expression suivante :

```
f x y
```

Deux dérivations possibles :

1. `App(App(Var(f), Var(x)), Var(y))` — associativité gauche
2. `App(Var(f), App(Var(x), Var(y)))` — associativité droite

Conclusion : la grammaire est ambiguë, car plusieurs arbres de dérivation sont possibles pour une même suite de tokens.

4. Comportement de `Parser_calc.mly`

Le fichier `Parser_calc.mly` implémente explicitement une règle séparée pour les applications, via le non-terminal `app_expr` :

```
app_expr:
| simple_expr
| app_expr simple_expr
```

Cette construction récursive à gauche force l'associativité gauche :

```
f x y  ->  App(App(f, x), y)
```

C'est donc ce comportement qui est implémenté dans le parseur.

5. Ajout de priorités pour reproduire ce comportement

Pour éviter les conflits SLR tout en forçant cette associativité gauche, on peut ajouter une priorité explicite :

```
%left APP
...
app_expr: app_expr simple_expr %prec APP
```

Cela permet à Menhir de résoudre le conflit automatiquement, en préférant le décalage, ce qui revient à forcer l'associativité gauche.

6. Ajout de toutes les fonctions `built_in`

Si on étend le langage avec toutes les fonctions `built_in`, il faudra :

— distinguer les priorités des opérateurs arithmétiques, logiques, comparatifs, etc.

— ajouter une priorité et une associativité pour chaque groupe d'opérateurs.

On peut estimer qu'il faudra une dizaine de priorités : `UMIN`, `mod`, `+ -`, `::`, `^`, `= < >`, `&&`, `||`, etc.

Cela devient rapidement difficile à maintenir, et très sujet à erreur.

7. Pourquoi utiliser des non-terminaux distincts plutôt que des priorités

Utiliser des non-terminaux séparés pour chaque niveau de priorité permet :

— de factoriser la grammaire proprement, sans ambiguïté,

— d'éviter la dépendance aux directives `%left`, `%right`, etc.,

— de faciliter la maintenance et l'évolution du parseur,

— d'être plus proche du comportement d'OCaml et de ses règles de priorité intégrées.

C'est une solution plus modulaire, plus claire, et plus robuste pour gérer la complexité croissante du langage.

2 Parseur : syntaxe étendue

Sucre syntaxique implémenté dans `Parser.mly`

Nous avons enrichi la grammaire de Mini-ml pour prendre en charge plusieurs formes de sucre syntaxique. Voici les transformations effectuées, accompagnées d'une explication précise de leur implémentation.

— **Notation infixe des opérateurs binaires**
Les opérateurs binaires (`+`, `*`, `<`, `&&`, `::`, etc.) sont intégrés dans la règle principale `expr` sous la forme :

```
| expr binop expr
```

Le désucrage se fait en AST par une double application :

`e1 op e2` $\rightsquigarrow$ `App(App(Cst_func(op), e1), e2)`

Les priorités et associativités sont gérées via des directives Menhir comme :

```
%left ADD SUB
%left MUL DIV MOD
%nonassoc EQ NEQ ...
%right CONCAT CAT APPEND
```

Ce choix permet de conserver une seule règle pour les binop tout en assurant la bonne désambiguïsation des expressions complexes.

— **Let avec arguments multiples**
La syntaxe OCaml-like `let f x y = e` est désucrée via une règle `sugar_func_decl`, combinée à une version alternative des règles `LET` :

```
LET ID arg1 arg2 EQ expr
```

est analysé récursivement comme :

```
Fun(arg1, Fun(arg2, expr))
```

Cela se traduit dans le code par une succession de règles récursives en OCaml :

```
| arg = ID e = sugar_func_decl { Fun(arg, e, ...) }
```

— **- unaire (négation)**
Le `- n` unaire est reconnu spécifiquement avec :

```
| SUB expr %prec UMINUS
```

et désucré en :

```
neg n-> App(Cst_func(UMin), n)
```

Une priorité `%right UMINUS` est ajoutée pour éviter les conflits avec les opérateurs binaires.

— **Listes constantes**
Les expressions de la forme `[1;2;3]` sont traitées par une règle spécifique `list_builder`, définie récursivement :

```
[1;2;3] -> App(App(Cst_func(Cat), 1),
              App(App(Cst_func(Cat), 2),
                  App(App(Cst_func(Cat), 3), Nil)))
```

Le constructeur utilisé est `Cat`, pour correspondre à l'opérateur `::`. Les cas `[]` (liste vide) et `[x]` (liste singleton) sont correctement pris en charge.

Exemple de programme accepté

```
let f x y = x + y * 2 in
print (f 3 (-4))
```

Ceci est traduit en AST comme :

```
Let(false, "f",
  Fun("x", Fun("y",
    App(App(Cst_func(Add), Var("x")),
        App(App(Cst_func(Mult), Var("y")),
            Cst_i(2)))
  )),
  App(Cst_func(Print),
      App(App(Var("f"), Cst_i(3)),
          App(Cst_func(Neg), Cst_i(4)))))
```

Difficultés rencontrées

— Les conflits shift/reduce apparaissent facilement avec les opérateurs binaires si on oublie de bien structurer la grammaire en niveaux de priorité.

— Le `- unaire` est délicat à distinguer du `- binaire` sans un niveau syntaxique dédié.

Exemple de programme accepté par la version corrigée mais pas par une version naïve

```
let f x = x + 1 in f 2 + 3 * 4
```

Une version naïve du parseur sans priorités ou sans structuration correcte renverrait :

— Soit un conflit,

— Soit un arbre incorrect avec `(f 2 + 3) * 4` au lieu de `f 2 + (3 * 4)`.

Comparaison avec le parseur simple

Le parseur simple ne contient aucune notion de priorité, ce qui induit une ambiguïté massive pour les applications et opérateurs. En structurant la grammaire en niveaux hiérarchiques, on obtient une grammaire non ambiguë, claire, et conforme au comportement d'OCaml.

3 Typage naïf et génération des contraintes

Principe de fonctionnement

Le typeur naïf, implémenté dans `typer_naive.ml`, parcourt récursivement l'arbre syntaxique d'une expression et :

— assigne un type universel frais à chaque sous-expression ;

— génère les contraintes entre les types attendus et ceux observés ;

— associe à chaque nœud son type dans l'annotation.

Aucune contrainte n'est résolue à ce stade : on les collecte simplement.

Exemples de programmes testés

Exemple 1 : une fonction d'identité appliquée à un entier.

```
let f = fun x -> x
let     y = f 4
```

Exemple 2 : un test conditionnel.

```
let x = if true then 3 else 4
```

Exemple 3 : une fonction d'application.

```
let apply = fun f -> fun x -> f x
let a = apply (fun n -> n + 1) 3
```

Illustration du typage naïf (exemple 1)

Code source :

```
let f = fun x -> x
let x = f 4
```

Arbre de l'AST annoté avec types (représentation textuelle) :

```
Let(false, "f",
  Fun("x", Var("x")),
  App(Var("f"), Cst_i(4)))
```

Attribution de types :

- T_0 : type de `x`
- T_1 : type de `Var("x")` $= T_0$
- $T_2 = T_0 \to T_0$: type de la fonction `fun x -> x`
- T_3 : type de `f` dans le corps (doit être T_2)
- T_4 : type de `Cst_i(4)` $=$ int
- T_5 : type du résultat de `App(f, 4)` $=$ résultat attendu de type inconnu

Contraintes générées :

$$
\begin{aligned}
T_1 &= T_0 \\
T_2 &= T_0 \to T_0 \\
T_3 &= T_2 \\
T_4 &= \text{int} \\
T_3 &= T_4 \to T_5
\end{aligned}
$$

Comportement du typeur

Le typeur passe bien sur cet exemple, les contraintes sont cohérentes et la substitution (voir section suivante) aboutira à :

$$T_0 = \text{int}, \quad T_5 = \text{int}$$

L'expression entière a donc le type `int`, comme attendu.

4 Résolution des contraintes et polymorphisme faible

Fonction `solve_constraints`

La fonction `solve_constraints` (dans `typer_util.ml`) prend une liste de contraintes de la forme (t_1, t_2) et produit une substitution qui unifie les types.

- Si t_1 est une variable universelle $'a$ et ne figure pas dans t_2, on ajoute la substitution $'a \mapsto t_2$.
- Si t_1 et t_2 sont des fonctions ou des listes, on applique récursivement l'unification sur leurs composants.
- En cas de contradiction (ex. $int = bool$), on renvoie une erreur.

Exemples issus de la section précédente

Exemple 1 (fonction d'identité appliquée à un entier) :

```
let f = fun x -> x
let y = f 4
```

La substitution finale est :

$$\{T_0 \mapsto int, T_5 \mapsto int\}$$

Exemple 2 (test conditionnel) :

```
let x = if true then 3 else 4
```

Les contraintes :

$$T_1 = bool, \quad T_2 = T_3$$

Substitution :

$$\{T_1 \mapsto bool, T_3 \mapsto int\}$$

Exemple 3 (fonction d'application) :

```
let apply = fun f -> fun x -> f x
let y = apply (fun n -> n + 1) 3
```

Substitution calculée (simplifiée) :

$$\{T_f \mapsto int \to int, \quad T_x \mapsto int\}$$

Exemple non typé à cause du polymorphisme faible

Code :

```
let id = fun x -> x
let a = id 1
let b = id "coucou"
```

- `id` est assigné à un type `'a -> 'a`.
- `a` fixe `'a = int`.
- Lors du typage de `b`, on tente d'appliquer `id` (qui est maintenant `int -> int`) à un `string`.

Résultat : erreur de typage. C'est attendu avec le typage faible car `id` n'est pas généralisé à sa définition.

5 Typage fortement polymorphe

Implémentation

Cette étape repose sur l'adaptation de la fonction `type_expr` (dans `typer.ml`) pour :

- distinguer les contraintes internes à une expression `e1` dans une construction `let x = e1 in e2` ;
- résoudre ces contraintes localement pour `e1` uniquement ;
- appliquer la substitution à `e1` et à l'environnement ;
- généraliser le type de `e1` par rapport à l'indice courant du compteur ;
- instancier les types génériques lors de chaque appel de fonction.

Exemple typé uniquement avec le typage fort

Code :

```
let id = fun x -> x
let a = id 1
let b = id "hello"
```

Comportement du typage fort :

- `id` est généralisé : $'a \to' a$ devient $TFunc([0], TUniv(0), TUniv(0))$
- à chaque appel, une `instantiation` fournit une version fraîche :
  - `a` : $int \to int$
  - `b` : $string \to string$

Conclusion : le typage passe, contrairement au typage faible.

Deux programmes typant différemment

Exemple 1 — qui typait faux en faible mais vrai en fort (déjà cité)

```
let id = fun x -> x
let a = id 1
let b = id true
```

Avec typage faible : `'a` fixé à `int`, erreur sur `true`
Avec typage fort : pas de problème, `id` est instancié à chaque appel.

Exemple 2 — fonction d'application polymorphe

```
let apply = fun f -> fun x -> f x
let i = apply (fun n -> n + 1) 2
let s = apply (fun str -> str ^ "!") "yo"
```

Avec typage faible :

— `apply` typé avec `'a -> 'b -> 'c` ;

— premier appel fixe `'a = int -> int, 'b = int` ;

— second appel échoue (conflit avec `string -> string`).

Avec typage fort :

— `apply` est généralisé, donc on a deux instanciations indépendantes ;

— typage accepté pour les deux appels.

Application du typage fort (résumé étape par étape pour le premier appel) :

— Typage de `apply` : $TFunc([0,1,2], TUniv(0), TFunc([], TUniv(1), TUniv(2)))$

— Instantiation : $TUniv(0) \mapsto TFunc([], int, int)$, $TUniv(1) \mapsto int$

— Résultat : $TUniv(2) = int$ donc `i` : int