Travail pratique

IFT-6172

October 15, 2020

1 Survol

Dans ce TP, vous allez étendre un langage existant avec l'extension de votre choix. Le langage existant sur lequel le travail construit est une version simplifiée du langage expérimental Typer, sorte de croisement entre Coq et Lisp.

2 Typer

L'implémentation de Typer est écrite en OCaml; elle est structurée comme suit:

1. Analyse lexicale.
2. Analyse syntaxique, qui renvoie un arbre de syntaxe de type `sexp` (une variante des S-expressions de Lisp).
3. Élaboration, qui transforme la `sexp` en une `lexp` ("lambda-expression"), qui est le "langage central" de Typer, un Pure Type System (PTS) avec quelques extensions.
4. Vérification de types.
5. Effacement, prend une `lexp` et la transforme en une forme simplifiée `elexp` où les annotations de types ont été effacées.
6. Interpréte de `elexp`.

2.1 Analyse lexicale

L'analyse lexicale se veut simpliste: elle sait reconnaître les commentaires, les chaînes de caractères, les entiers, les flottants, et tout le reste est considéré comme des "symboles" qui sont séparés par des espaces, à l'exception de quelques caractères qui sont leur propre symbole.

Le seul paramètre que vous pourriez avoir envie de modifier est le tableau `default_stt`, défini dans `src/grammar.ml` qui indique quels caractères sont leur propre symbole. À l'heure actuelle il s'agit des parenthèses, de la virgule et du point-virgule. Par exemple `a+2:` sera considéré comme 1 symbole.

2.2 Analyse syntaxique

L'analyse syntaxique est elle aussi simpliste, basée sur les *operator precedence grammars* (OPG), une classe de grammaires très restrictive. Elle est paramétrée par une grammaire qui est une table de niveaux de précédence des "mots clés", la table utilisée par défaut est `default_grammar` dans `src/grammar.ml`. À remarquer que cette table vient en réalité de `typer-smie-grammar`, dans `emacs/typer-mode.el` (où elle est utilisée par le mode Emacs pour aider à naviguer et auto-indenter le code), qui est générée mécaniquement à partir d'une représentation plus-ou-moins BNF.

Elle est utilisée pour permettre l'usage de notations infixes, mais le programmeur peut aussi utiliser une notation préfixe à la Lisp, qui lui est 100% équivalente. E.g. Lorsque le programmeur écrit:

```
type List (a : Type)
 | nil
 | cons a (List a)
```

le résultat est le même que s'il avait écrit:

```
type_ (_|_ (List (_:_ a Type))
           nil
           (cons a (List a)))
```

Parce que `type` dans la grammaire à une précédence de `None` à gauche, ce qui indique que c'est un mot-clé préfixe, alors que `|` à la même précédence à gauche qu'à droite ce qui indique qu'il est non seulement infixe mais se combine avec lui-même (on dit qu'il est "associatif").

Le résultat de l'analyse est une structure de donnée de type `sexp`, définie dans `src/sexp.ml`:

```
type sexp =
  | Block of location * pretoken list * location
  | Symbol of symbol
  | String of location * string
  | Integer of location * integer
  | Float of location * float
  | Node of sexp * sexp list
```

Remarquez qu'il n'y a à ce niveau aucune notion de sémantique, c'est juste un arbre de symboles; cette partie du code ne sait pas ce qui est une fonction, un appel de fonction, une définition de type, ... [1]

[1] Les `Block` ne sont probablement pas importants pour votre TP, mais si vous voulez savoir, c'est les blocks de code délimités par des accolades: ils sont gardés "tels quels" sans en faire l'analyse syntaxique (cela permet aux macros d'en faire l'analyse syntaxique manuellement plus tard avec une grammaire de leur choix).

2.3 Élaboration

C'est le cœur de Typer, qui se trouve dans `src/elab.ml`. Il prend une `sexp` et la transforme en une lambda-expression `lexp` (défini dans `src/lexp.ml`), ce qui nécessite de propager les informations de typage (qu'il fait sans faire vraiment d'inférence mais plutôt de la propagation bi-directionnelle) ainsi que de faire l'expansion des macros.

```
type ltype = lexp
 and lexp =
   | Imm of sexp (* Used for strings, ...  *)
   | Sort of U.location * int
   | Builtin of symbol * ltype
   | Var of vref
   | Susp of lexp * subst
   | Let of U.location * (vname * lexp * ltype) list * lexp
   | Arrow of vname * ltype * U.location * ltype
   | Lambda of vname * ltype * lexp
   | Call of lexp * lexp list
   | Inductive of U.location * ((vname * ltype) list) SMap.t
   | Cons of lexp * symbol
   | Case of U.location * lexp
             * ltype
             * (U.location * vname list * lexp) SMap.t
             * (vname * lexp) option
```

Le type `U.location` garde les informations de position dans le code source. Les constructeurs `Arrow`, `Lambda`, et `Call` correspondent respectivement au type, constructeur et éliminateur des fonctions (remarquez que `Call` est *curried*: les fonctions ne prennent qu'un argument à la fois).

Le `Let` décrit une liste de définitions qui peuvent être mutuellement récursives; Cela permet aussi de définir des types et des fonctions sur les types, sans aucune restriction qui permettrait de garantir une forme de normalisation forte. Donc on ne peut pas utiliser ce langage comme une logique (elle serait incohérente) et la vérification de type n'est pas décidable mais ça ne nous préoccupe pas.

le `Imm` correspond à des constantes "imm"édiates comme des nombres ou des chaînes de caractères; le `Builtin` fait référence à une fonction ou un type implémenté dans le code OCaml. `Sort` est la *sorte* au sens des PTS, où on a une tour d'univers prédicatifs (quoique seuls les univers 0 et 1 ont un nom à l'heure actuelle: `Sort(1,0)` s'appelle `Type` et `Sort(1,1)` s'appelle `Kind`).

Le `Susp(e,s)` est une "substitution suspendue", i.e. une `lexp e` où on a pas encore appliqué la substitution `s` (on fait ça pour appliquer les substitutions de manière paresseuse). Chaque fois que vous rencontrer un tel élément, la chose à faire est d'appeler `push_susp e s` qui vous renvoie l'expression cachée derrière cette suspension.

Le `Var` est bien sûr une référence à une variable. Un `vref` (défini dans `src/util.ml`) contient deux parties:

```
type vref = (location * string list) * db_index
```

La première partie contient le (voire les) noms de la variable et la deuxième est un entier: l'*index de De Bruijn* qui est la position de la variable dans le contexte. ¡Important! le nom n'est pas significatif, seul l'index l'est: le nom des variables dans `lexp` est toujours optionnel et ne sert que pour des fins de débogage et pour afficher des messages d'erreur plus compréhensibles.

2.3.1 Types algébriques

`Inductive` décrit un type algébrique: la `Smap` est une table indexée par le nom du constructeur qui liste les arguments du constructeur.

`Cons` décrit une référence à un constructeur d'un type algébrique: son premier argument est le type algébrique et le deuxième est le nom du constructeur auxquel il se réfère.

La déclaration de type:

```
type List (a : Type)
 | nil
 | cons a (List a);
```

est en fait un appel à la macro `type_` qui transforme ce code en:

```
List : (a : Type) -> Type;
List (a : Type) = typecons nil (cons a (List a));
nil (a : Type) = datacons (List a) nil;
cons (a : Type) = datacons (List a) cons;
```

où `typecons` se traduit directement par `Inductive` et `datacons` se traduit par `Cons`.

Le `Case(l, target, ret, branches, default)` correspond bien sûr à une analyse par cas, où `target` est l'expression que l'on analyse, `ret` est le type du résultat (donc aussi le type de chacune des branches), `branches` est une table qui associe une branche à chaque nom de constructeur, et `default` est une branche par défaut (pour les constructeurs non mentionnés dans `branches`).

2.3.2 Macros

Lorsque l'élaborateur trouve une expression $\texttt{e}_1$ $\texttt{e}_2$ $\texttt{e}_3$ `...` qui ressemble à un appel de fonction mais où $\texttt{e}_1$ est une expression de type `Macro`, il s'agit en fait d'un appel de macro, et l'élaborateur appelle alors la fonction contenue dans l'objet $\texttt{e}_1$ en lui passant les `sexp` fournies en argument $\texttt{e}_2$ $\texttt{e}_3$ `...`, qui renvoie l'expansion sous forme d'une nouvelle `sexp` sur laquelle l'élaboration se poursuit. Cela fonctionne donc de manière très similaire au `defmacro` de Lisp.

2.4 Vérification de types

L'élaboration propage les types et fait ainsi fondamentalement le travail de la vérification de types. Cependant, l'élaboration est une phase relativement complexe à laquelle on préfère ne pas faire trop confiance. Donc on vérifie le code qu'elle renvoie un appelant la fonction `check` définie dans `src/opslexp.ml`. Cette fonction aimerait être aussi simple que possible est aussi proche que possible de la formulation théorique des règles de typage, pour minimiser les risques d'erreurs.

2.5 Effacement

Cette phase, implémentée dans la fonction `erase_type`, définie dans `src/opslexp.ml`, est très simple et efface simplement les annotations de types, qui ne sont pas nécessaires pour l'interpréteur. Vu que les types peuvent être manipulés comme des valeurs, il arrive qu'il en reste à ce stade, mais ils ne "font" rien, auquel cas on les garde parce que c'est plus simple que d'essayer de les enlever et ça permet de les imprimer au besoin. Le résultat est de type `elexp` ("erased" lexp), défini dans `src/elexp.ml`.

2.6 Interpréteur

Finalement le code est exécuté par un interpéteur sans sophistication implémenté dans `src/eval.ml`. C'est là aussi que sont définies la majorité des primitives "built-ins".

2.7 Différence avec Typer

Ne pas confondre Typer et Typer: celui que vous utiliser pour le cours est un peu plus simple que le Typer officiel. Il vous arrivera probablement de trouver des références à des fonctionnalités du Typer officiel, donc voici une liste succincte des fonctionnalités enlevées:

- Les arguments implicites et effaçables, qui utilisent des fonctions avec des flèches de la forme `=>` et `≡>`.
- L'inférence de types.
- Les monades: Le langage Typer officiel se veut un langage fonctionnel pur qui utilise les monades pour confiner les effets de bords, alors que celui que vous avez n'est pas pur, donc il est plus comme OCaml que comme Haskell.
- Le polymorphisme d'univers.
- Le type égalité (qui permet de faire des *cast*).
- Un système de modules (très primitif).

3 Devoir

Le travail peut se faire seul ou par groupe de deux. Il se décompose en trois étapes:

1. Choix d'une (voire plusieurs) extensions.
2. Conception de cette extension.
3. Implémentation.

3.1 Choix

Vous êtes libre de choisir la ou les extensions que vous voulez ajouter à Typer, indépendemment du choix des autres. Il faut bien sûr que ce soit une extension au langage et pas seulement à son implémentation (c'est pas un cours de compilation; on n'est pas intéressé à générer du code plus efficace, par exemple).

Exemples d'extensions:

- Ajouter un système d'objet à base de classes.
- Ajouter un concept d'aspect.
- Ajouter une notion de linéarité ou de types *ownership*.
- Du typage graduel.
- Ajouter des macros hygiéniques.
- Ajouter des continuations délimitées.
- Ajouter un système de module.
- Des classes de types.
- ...

À ce stade, il suffira de décrire brièvement (maximum 1 page) à quoi pourrait ressembler l'extension que vous envisager. Cela me permettra de vous indiquer si ça me semble un choix suffisant (ou trop ambitieux) et de vous donner quelques directives pour essayer d'ajuster la complexité.

3.2 Conception

À cette étape, vous devrez me rendre un rapport en LaTeX(code source, pas de PDF ou autre, maximum 5 pages) qui décrit comment cette extension s'intègre au langage, avec une description formelle des éléments syntaxiques ajoutés (ou modifiés), de la sémantique statique et dynamique de ces éléments (i.e. règles de typage et règles de réduction) ainsi qu'une description de comment les règles préexistantes sont affectées (si nécessaire). La présentation formelle devrait être

accompagnée de quelques exemples, où il sera très important d'expliquer ce que ces exemples sont sensés faire.

Il est possible/probable (selon les choix) qu'il soit difficile d'intégrer votre extension avec Typer de manière complètement satisfaisante. Donc une partie importante de la conception est aussi de prendre conscience de ces limitations et de les mentionner dans le rapport.

Quoique cette étape n'inclut aucun code, il va de soit qu'à ce point du travail, vous devriez déjà être en train de coder, vu que c'est aussi en codant que vous aller découvrir des problèmes qui nécessitent de changer la conception.

3.3 Implémentation

Étape finale où vous devrez me fournir le code (sous forme de *patch*) ainsi qu'un rapport (là encore en LaTeX, maximum 10 pages) qui décrit la conception finale (potentiellement identique à celle de l'étape précédente) ainsi que l'approche, les choix, et les limitations de l'implémentation.

Ajouter ces fonctionnalités à une implémentation comme celle de Typer peut être non-trivial, donc votre implémentation sera probablement un prototype limité qui ne couvre que la partie principale des fonctionnalités. C'est normal, mais c'est important de décrire ces limites clairement dans le rapport.

SVP, écrire votre code clairement, bien indenté, n'utilisez pas plus que 80 colonnes, ...

4 Note

Le TP compte pour 50% de la note finale. Ces 50% se décomposent comme suit: 5% pour le choix, 20% pour la conception, 10% pour le rapport final et 15% pour l'implémentation.