Inférence de type pour MINIML

Projet Programmation Fonctionnelle 2SN - Systèmes Logiciels

Session 1 - 2022-2023

1 Sujet d'étude

L'inférence de types de OCAML permet de déterminer les types des expressions manipulées ou de détecter une potentielle mauvaise utilisation des données qui provoquerait une erreur à l'exécution. Le but de ce projet est de reproduire simplement le comportement de l'algorithme de Hindley-Milner utilisé pour réaliser l'inférence de types de OCAML (ainsi que d'autres langages) pour le langage MINIML un sous-ensemble de OCAML compatible avec celui-ci.

2 Travail à réaliser

Les différentes tâches à réaliser sont les suivantes :

- À partir d'une grammaire simplifiée de MINIML, définir un *parser* pour ce langage. Vous utiliserez le *lexer* fourni pour reconnaître les lexèmes du langage. Le langage n'étant pas ambigu, il existe au plus une solution de *parsing* à récupérer.
- À partir des règles de typage présentées dans la figure 3, définir un algorithme qui donne le type d'une expression et accumule les équations entre types qui garantissent le bon typage, si et seulement si ces équations ont une solution.
- Résoudre le système d'équations obtenu, à l'aide des règles de la figure 4, pour obtenir une forme normale du type calculé (ou bien détecter l'absence de solutions).
- Définir enfin un programme principal, qui lit un programme MINIML dans un fichier, applique les règles de typage afin d'obtenir son type et les équations associées, détermine la forme la plus générale des solutions ou l'absence de solution, puis affiche le type obtenu ou une erreur de typage.

3 Reconnaissance du langage MINIML

La grammaire de MINIML pour lequel l'inférence de types sera implantée est présentée figure 1. On remarque que cette grammaire contient des parenthèses qui ne sont pas toujours nécessaires en OCAML, afin que les parsers associés ne soient pas récursifs à gauche et non ambigus. De plus, les définitions de fonctions s'écriront sous la forme **let** f = **fun** x −> ... et non sous la forme classique **let** f x = ... Enfin, le langage MINIML ne contient pas de définitions globales de la forme **let** x = ...;; mais seulement des définitions locales de la forme **let** x = ... **in** ... .

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textbf{let}\ \textit{Liaison}\ \textbf{in}\ \textit{Expr} \\
& | & \textbf{let}\ \textbf{rec}\ \textit{Liaison}\ \textbf{in}\ \textit{Expr} \\
& | & (\textit{Expr}\ \textit{Binop}\ \textit{Expr}) \\
& | & (\textit{Expr}) \\
& | & (\textit{Expr}\ \textit{Expr}) \\
& | & \textbf{if}\ \textit{Expr}\ \textbf{then}\ \textit{Expr}\ \textbf{else}\ \textit{Expr} \\
& | & (\textbf{fun}\ \textit{ident} -> \textit{Expr}) \\
& | & \textit{ident} \\
& | & \textit{Constant}
\end{array}
\qquad
\begin{array}{lcl}
\textit{Liaison} & \rightarrow & \textit{ident} = \textit{Expr} \\
\textit{Binop} & \rightarrow & \textit{Arithop} \mid \textit{Boolop} \mid \textit{Relop} \mid @ \mid :: \\
\textit{Arithop} & \rightarrow & + \mid - \mid * \mid / \\
\textit{Boolop} & \rightarrow & \&\& \mid || \\
\textit{Relop} & \rightarrow & = \mid <> \mid <= \mid < \mid >= \mid > \\
\textit{Constant} & \rightarrow & \textit{entier} \mid \textit{booleen} \mid [] \mid ()
\end{array}
$$

FIGURE 1 – Grammaire des expressions de MINIML

4 Règles de typage

La syntaxe des types de MINIML est définie par la grammaire de la figure 2.

$$
\begin{array}{lcl}
\textit{Type} & \rightarrow & \text{'}\textit{ident} \mid \text{int} \mid \text{bool} \mid \text{unit} \\
& | & \textit{Type} -> \textit{Type} \\
& | & \textit{Type} * \textit{Type} \\
& | & \textit{Type}\ \text{list}
\end{array}
$$

FIGURE 2 – Grammaire des types de MINIML

Le principe de l'algorithme de calcul du type d'une expression MINIML est de parcourir récursivement cette expression et de produire pour chaque sous-expression un type, ainsi que des équations entre types garantissant le bon typage. Les types des sous-expressions seront utilisés pour construire le type de l'expression complète. Les équations associées aux sous-expressions sonet simplement accumulées dans le résultat.
Dans la suite, α, β, γ ... sont des variables de type et τ, σ, ρ, ... sont des types quelconques.

4.1 Jugement de typage

Les jugements de typage sont de la forme $\Gamma \vdash e : \tau$, où Γ est un environnement sous la forme d'une liste ordonnée de couples $\{(x_1 : \tau_1); \ldots ; (x_n : \tau_n)\}$ qui définissent les types τ_i des variables x_i, e est l'expression à typer et τ son type. L'algorithme de typage part d'une expression e et d'un environnement Γ_0 initial et calcule le type de e ainsi que les équations associées, tel que présenté dans la section 4.3.

4.2 Gestion de l'environnement

L'environnement initial Γ_0 devra contenir les types des opérations standard prédéfinies, i.e. au moins :

- l'opérateur de concaténation de listes @ : 'a list -> 'a list -> 'a list
- l'opérateur de construction de liste "::" : 'a -> 'a list -> 'a list
- l'opérateur de construction de paire ",": 'a -> 'b -> 'a * 'b
- les opérateurs binaires arithmétiques *Arithop* : int -> int -> int
- les opérateurs binaires relationnels *Relop* : 'a -> 'a -> bool
- les opérateurs binaires booléens *Boolop* : bool -> bool -> bool

— les fonctions unaires **not** : bool -> bool, fst : 'a * 'b -> 'a, snd : 'a * 'b -> 'b, hd : 'a list -> 'a et tl : 'a list -> 'a list .

En ce qui concerne le typage, les opérateurs binaires seront traités comme de simples applications de fonctions à deux paramètres. Pour pouvoir appliquer ces fonctions à des paramètre de types différents (dans le cas où elles sont polymorphes), il faut dupliquer/copier les variables de type à chaque utilisation. Ainsi, une occurrence de fst aura le type 'a * 'b -> 'a et une autre le type 'c * 'd -> 'c.

Dans tout jugement, l'environnement Γ contient normalement les types des variables libres de e (définies dans un **let**, **let rec** ou comme paramètre d'une fonction **fun** englobante). Ainsi, la déduction de type n'est possible que si toutes les variables de l'expression initiale sont définies. Par exemple, l'expression **let** x=1 **in** x+y ne peut pas être typée car y n'est pas définie. Plusieurs variables locales ou paramètres peuvent porter le même nom et ainsi apparaître plusieurs fois dans un environnement de typage. L'occurrence la plus récente d'une variable x masque toutes les variables x la précédant dans l'environnement.

Les constantes de MINIML (par exemple 1, true, [], (), etc) ne sont pas introduites dans Γ_0. On suppose défini $Type(c)$, le type de toute constante $c \in Const$, le *lexer* permettant de déterminer le résultat dans certains cas.

4.3 Inférence de type

Les règles d'inférence de type présentées dans la figure 3 sont **structurelles** et ont la forme suivante :

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma_n \vdash e_n : \tau_n}{\Gamma \vdash Production(e_1, \ldots, e_n) : \tau} \; (\sigma_1 \equiv \sigma'_1, \quad \ldots \quad , \sigma_p \equiv \sigma'_p)$$

Le principe général des règles est alors le suivant :

1. on part d'un environnement Γ et d'une expression $e = Production(e_1, \ldots, e_n)$ à typer.
2. on calcule récursivement le type τ_i des sous-expressions e_i de e, dans des environnements Γ_i dérivés de Γ par ajout éventuel de définitions locales, suivant la production du language utilisée $Production$.
3. on récupère les types résultats et on construit le type global τ de e, éventuellement à l'aide de variables de type **fraîches** α, β, δ, etc. Une variable est dite fraîche si elle est distincte de toutes les autres variables présentes. On utilisera pour cela un générateur de variable fraîche.
4. les équation de types $(\sigma_j \equiv \sigma'_j)$ servent d'une part à définir et contrôler les relations entre les différents types τ_i et d'autre part à définir à partir des τ_i les variables de type α, β, δ, etc, utilisées dans τ.
5. les équations $\sigma_j \equiv \sigma'_j$ obtenues à chaque étape d'inférence sont accumulées et constituent également un résultat de l'algorithme de typage.

5 Résolution des équations

Une fois le type τ et les équations $Eqs = \{\sigma_i \equiv \sigma'_i\}$ obtenues en appliquant les règles d'inférence à une expression dans l'environnement initial Γ_0, on doit vérifier que les équations admettent une solution et présenter l'ensemble des solutions sous forme normalisée. Cette normalisation transforme progressivement les équations générales en définitions de variables de types, i.e. de la forme $\alpha \mapsto \tau$, selon les règles de la figure 4. Les équations non présentes ne sont pas normalisables et doivent déclencher une erreur de type. C'est le cas par exemple de int $\equiv$ bool, ou bien de $\alpha \equiv$ bool->α qui correspond à un type récursif α mal fondé. On notera $[\alpha \mapsto \tau]X$ la substitution de la variable α par sa définition τ dans X, X pouvant être un

$$Const\ \frac{c \in Const}{\Gamma \vdash c : Type(c)} \qquad Var_1\ \frac{}{(x:\tau) :: \Gamma \vdash x : \tau} \qquad Var_2\ \frac{\Gamma \vdash x : \tau}{(y:\sigma) :: \Gamma \vdash x : \tau}$$

$$Cons\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 :: e_2) : \tau_2}\ (\tau_2 \equiv \tau_1\ \text{list}) \qquad Pair\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$Fun\ \frac{(x:\alpha) :: \Gamma \vdash e : \tau}{\Gamma \vdash (\textbf{fun}\, x{->}e) : \alpha \to \tau}$$

$$App\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \alpha}\ (\tau_1 \equiv \tau_2 \text{ -> } \alpha)$$

$$IfThenElse\ \frac{\Gamma \vdash b : \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{if}\, b\, \textbf{then}\, e_1\, \textbf{else}\, e_2 : \tau_1}\ (\tau \equiv \text{bool}, \tau_1 \equiv \tau_2)$$

$$Let\ \frac{\Gamma \vdash e_1 : \tau \quad \{x:\tau\} :: \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau'}$$

$$LetRec\ \frac{\{x:\alpha\} :: \Gamma \vdash e_1 : \tau \quad \{x:\alpha\} :: \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \textbf{let rec}\, x = e_1\, \textbf{in}\, e_2 : \tau'}\ (\alpha \equiv \tau)$$

FIGURE 3 – Règles de typage

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{int} \equiv \text{int}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{bool} \equiv \text{bool}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{unit} \equiv \text{unit}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{\{\tau_1 \equiv \tau_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 \text{ list} \equiv \tau_2 \text{ list}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{\{\tau_1 \equiv \sigma_1, \tau_2 \equiv \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 -> \tau_2 \equiv \sigma_1 -> \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{\{\tau_1 \equiv \sigma_1, \tau_2 \equiv \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 * \tau_2 \equiv \sigma_1 * \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\alpha \equiv \alpha\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{[\alpha \mapsto \rho]Eqs \vdash_N \tau \Rightarrow \sigma \quad (\alpha \text{ non libre dans } \rho \neq \alpha)}{\{\alpha \equiv \rho\} \cup Eqs \vdash_N \tau \Rightarrow [\alpha \mapsto \rho]\sigma}$$

$$\frac{}{\emptyset \vdash_N \tau \Rightarrow \tau} \qquad \frac{\{\alpha \equiv \rho\} \cup Eqs \vdash_N \tau \Rightarrow \sigma \quad (\rho \text{ n'est pas une variable})}{\{\rho \equiv \alpha\} \cup Eqs \vdash_N \tau \Rightarrow [\alpha \mapsto \rho]\sigma}$$

FIGURE 4 – Règles de normalisation

type, une équation ou un ensemble d'équations. On utilisera le jugement de normalisation $Eqs \vdash_N \tau \Rightarrow \sigma$, qui indique que le type σ est obtenu en normalisant les équations Eqs et en substituant les définitions des variables de types dans τ. À l'issue de l'application de ces règles, le type normalisé obtenu est le type le plus général, représentant toutes les solutions.

6 Généralisation des types (bonus)

En réalité, les types calculés ne sont pas assez généraux et ne suivent pas totalement l'algorithme de Hindley-Milner. De la même façon qu'on doit dupliquer les variables de type des opérateurs prédéfinis polymorphes si on veut les utiliser dans des contextes différents, comme illustré dans la section 4.2, il est nécessaire de le faire pour chaque définition. Mais toutes les variables de type ne sont pas arbitraires et duplicables librement. Le principe est de manipuler dans les environnements Γ non pas des types τ mais des schémas de types de la forme $\forall\alpha \ldots \forall\gamma.\tau$, où certaines variables de type sont quantifiées universellement et donc instantiables/-duplicables librement. On aura besoin de définir :

- $Inst$ qui donne un type instantié "frais" à partir d'un schéma, i.e. $Inst(\forall\alpha.\tau) = Inst(\tau[\alpha \mapsto \alpha^*])$ avec α^* variable fraîche et $Inst(\tau) = \tau$.
- Gen qui généralise les variables non-contraintes par l'environnement. $Gen(\Gamma, \tau) = \forall\alpha_1 \ldots \forall\alpha_n.\tau$ avec $\alpha_1, \ldots, \alpha_n$ apparaissant librement dans τ mais pas librement dans Γ. De plus, pour appliquer Gen correctement dans les définitions MINIML, on doit manipuler le type τ de l'expression sous sa forme résolue, qu'on note $[\tau]$, i.e. à l'issue de la phase précédente décrite dans la section 5. On doit donc normaliser les équations obtenues à ce point (correspondant au typage de la règle courante plus le typage de ses sous-expressions) et substituer les équations normalisées dans τ. Il n'est plus possible de différer la génération des équations de leur résolution lorsqu'on type une définition MINIML.

On modifiera les règles de typage suivantes :

$$Var_1\ \frac{}{(x:\tau)::\Gamma \vdash x : Inst(\tau)} \qquad Let\ \frac{\Gamma \vdash e : \tau \quad \{x : Gen(\Gamma,[\tau])\}::\Gamma \vdash e' : \tau'}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e : \tau'}$$

$$LetRec\ \frac{\{x:\alpha\}::\Gamma \vdash e : \tau \quad \{x : Gen(\Gamma,[\alpha])\}::\Gamma \vdash e' : \tau'}{\Gamma \vdash \textbf{let}\ \ \textbf{rec}\ x = e\ \textbf{in}\ e : \tau'}\ (\alpha \equiv \tau)$$

FIGURE 5 – Règles de typage modifiées avec généralisation des types

7 Réflexion sur l'implantation

Les algorithmes proposés sont assez complexes pour tenter de simplifier leur écriture en séparant les différentes difficultés, par exemple :

- Il faut gérer l'environnement Γ qui est un paramètre supplémentaire (changeant) des fonctions de typage, selon la figure 3.
- Il faut gérer les équations renvoyées en plus du type "brut" calculé, le véritable type étant le résultat de la substitution des définitions de variables présentes dans les équations, appelée normalisation, selon la figure 4.
- Il faut également se préoccuper des échecs de typage qui peuvent intervenir quand il est impossible d'appliquer les règles de typage ou les règles de normalisation des équations produites.

Il est par exemple possible d'utiliser à bon escient les structures monadiques, voire les continuations pour séparer tous ces aspects et améliorer l'architecture de votre code.

8 Matériel fourni

Voici les différents modules OCAML fournis dans l'archive `miniml_distrib_etud.tar` :

- `lib/lazyflux` : contenant une implantation des flux paresseux, utilisée par le *lexer* et le *parser*.
- `lib/miniml_types` : contenant les définitions des lexèmes, expressions et types de MINIML.
- `lib/miniml_printer` : permettant d'afficher notamment les expressions MINIML.
- `lib/miniml_lexer` : permettant l'analyse lexicale du langage MINIML.
- `lib/miniml_parser` : **à compléter**, permettant l'analyse syntaxique de MINIML.
- `lib/miniml_typer` : **à compléter**, permettant le typage de MINIML.
- `bin/miniml_principal` : **à compléter**, le programme principal.

9 Critères de jugement

- dans tous les cas, votre application doit s'exécuter.
- la lisibilité du code et la qualité des contrats et des commentaires sera évaluée.
- la présence de tests (unitaires et d'intégration) sera évaluée.
- le code écrit devra être **purement fonctionnel**, sauf ce qui concerne les entrées-sorties.

— la modularité sera évaluée, les différentes phases du typage devant être bien séparées et spécifiées.
— tout type d'utilisation avancée du langage OCAML sera appréciée, cf. section 7.
— votre code devra être commenté, en faisant notamment apparaître les types et rôles des arguments et des résultats des différentes fonctions implantées.
— un rapport expliquant vos choix de conception et de programmation est obligatoire. Il est rappelé que paraphraser l'énoncé ou simplement énumérer des élements de votre code sans les expliquer ne sera pas suffisant.
— la partie **bonus** n'est pas requise, mais réaliser cette partie peut permettre de rattraper des points perdus par ailleurs.

10 Travail et Rendu

— Ce projet sera effectué par groupes de 3 ou 4 personnes.
— À l'issue de celui-ci, une archive contenant vos fichiers sources, vos tests et le rapport sera remise sous Moodle.
— Le projet doit être rendu au plus tard le dimanche 22 janvier 2023 à 23h59.