CTD3 : Typage simple et Polymophisme

1 Les types

Il existe des termes syntaxiquement corrects mais qui n'ont aucun sens. Par exemple, le terme `2 1` qui applique la constante `2` à la constante `1` est syntaxiquement correct mais ce n'est pas une valeur et on ne peut pas le réduire... Il est donc naturellement *inacceptable*. Pour restreindre l'ensemble des termes syntaxiquement corrects aux seuls termes acceptables (i.e. les valeurs et les termes réductibles), on utilise les notions de **type** et de **terme typable**.

Un **type** est l'abstraction d'un terme qui « cache » ses détails internes. On peut aussi *interpréter* un type comme un ensemble de valeurs. Par exemple, le type `int` est l'ensemble des valeurs $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ et les termes de type `int` sont les termes qui se réduisent en une valeur de `int`. Ainsi, les termes 2, 2+2, 2+3-5/2 ont tous le même type `int`.

En réalité, le problème de déterminer si un terme est acceptable est un problème *indécidable*. Par conséquent, l'ensemble des termes typables est un sous-ensemble strict de l'ensemble des termes acceptables.

Les types sont définis par :

$$\tau \ ::= \ \tau_1 \to \tau_2 \mid g(\tau_1, \ldots, \tau_n)$$

où g est un constructeur de types d'arité n. Parmi les constructeurs de types d'arité 0 citons par exemple `int`, `bool`, `string`. Aussi, `*` est un constructeur de types d'arité 2[1] et `list` est un constructeur de types d'arité 1[2].

2 Le typage

Une **règle d'inférence**

$$\frac{P_1 \ldots P_n}{C}$$

signifie : si les premisses $P_1, \ldots, P_n$ sont satisfaites, alors on peut déduire C. On peut par exemple utiliser les règles d'inférence pour définir des *systèmes de déduction logique* (propositionnelle) :

$$\frac{}{\text{TRUE}}\ (\text{Axiome}) \qquad \frac{\text{A} \quad \text{B}}{\text{A} \wedge \text{B}}\ (\wedge) \qquad \frac{\text{A}}{\text{A} \vee \text{B}}\ (\vee_G) \qquad \frac{\text{B}}{\text{A} \vee \text{B}}\ (\vee_D)$$

Dans un tel système on peut par exemple prouver la formule FALSE∨(TRUE∧(TRUE∨FALSE)) en construisant son *arbre d'inférence*.

Un **environnement de typage** Γ est une application partielle des variables et constantes dans les types :

$$\Gamma \ ::= \ \emptyset \mid \Gamma, x : \tau \mid \Gamma, \mathtt{f} : \tau \mid \Gamma, \mathtt{c} : \tau$$

On dénote par Γ_0 l'environnement qui assigne un type à tout symbole de fonction primitive f et de constructeur de données c (rappel : ceci inclut les entiers, reels, et booleens). On fera l'hypothèse que pour tout environnement Γ on a $\Gamma_0 \subseteq \Gamma$ et que Γ est consistant avec les arités des constantes, c'est-à-dire si $\mathtt{f} \in \mathrm{dom}(\Gamma)$ (resp. $\mathtt{c} \in \mathrm{dom}(\Gamma)$) est d'arité n alors $\Gamma(\mathtt{f})$ (resp. $\Gamma(\mathtt{c})$) est de la forme $\tau_1 \to \ldots \to \tau_n \to \tau$.

Question 1 Donnez l'environnement de typage Γ_0 ! Traitez les fonctions arithmethiques, les booleens, et quelques nombres.

Un **jugement de typage**

$$\Gamma \vdash t : \tau$$

signifie que t a le type τ dans l'environnement de typage Γ.

Les **règles de typage** pour Core ML (monomorphe) sont les suivantes :

$$\frac{\mathtt{f} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{f} : \Gamma(\mathtt{f})}\ (\text{Fun prim}) \qquad \frac{\mathtt{c} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{c} : \Gamma(\mathtt{c})}\ (\text{Data const}) \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ (\text{Var})$$

[1] ne pas confondre avec la fonction primitive `*`. Notez que dans Caml ce constructeur de type apparaît en notation infixe.

[2] Dans Caml, le type auquel `list` est utilisé en notation postfixe alors qu'ici il apparaîtra en notation préfixe.

$$\frac{\Gamma, x:\tau \vdash t:\tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'}\ (\textsc{Fun}) \qquad \frac{\Gamma \vdash t:\tau' \to \tau \qquad \Gamma \vdash u:\tau'}{\Gamma \vdash t\ u : \tau}\ (\textsc{App})$$

$$\frac{\Gamma \vdash u:\tau' \qquad \Gamma, x:\tau' \vdash t:\tau}{\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : \tau}\ (\textsc{Let}) \qquad \frac{\Gamma \vdash u:\tau \qquad u =_\alpha t}{\Gamma \vdash t : \tau}\ (\textsc{Alpha})$$

3 Exemples

Par exemple, le typage de l'addition et de la disjonction sont directement obtenu par (`Fun prim`) :

$$\frac{}{\Gamma_0 \vdash \mathtt{+} : \mathtt{int} \to \mathtt{int} \to \mathtt{int}} \qquad \frac{}{\Gamma_0 \vdash \mathtt{or} : \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool}}$$

Typage de $\mathtt{let}\ inc = \lambda x.\mathtt{+}\,x\,\mathtt{1}\ \mathtt{in}\ inc\,\mathtt{1}$

$$\dfrac{\dfrac{(\star)}{\Gamma_0 \vdash \lambda x.\mathtt{+}\,x\,\mathtt{1} : \mathtt{int} \to \mathtt{int}} \qquad \dfrac{\dfrac{\overset{(\textsc{Var})}{\overline{\Gamma_0, inc : \mathtt{int} \to \mathtt{int} \vdash inc : \mathtt{int} \to \mathtt{int}}} \qquad \overset{(\textsc{Data Const})}{\overline{\Gamma_0, inc : \mathtt{int} \to \mathtt{int} \vdash \mathtt{1} : \mathtt{int}}}}{\Gamma_0, inc : \mathtt{int} \to \mathtt{int} \vdash inc\,\mathtt{1} : \mathtt{int}}\ (\textsc{App})}{}}{\Gamma_0 \vdash \mathtt{let}\ inc = \lambda x.\mathtt{+}\,x\,\mathtt{1}\ \mathtt{in}\ inc\,\mathtt{1} : \mathtt{int}}\ (\textsc{Let})$$

où $(\star)$ est la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\overset{(\textsc{Fun prim})}{\overline{\Gamma_0, x : \mathtt{int} \vdash \mathtt{+} : \mathtt{int} \to \mathtt{int} \to \mathtt{int}}} \qquad \overset{(\textsc{Var})}{\overline{\Gamma_0, x : \mathtt{int} \vdash x : \mathtt{int}}}}{\Gamma_0, x : \mathtt{int} \vdash \mathtt{+}\,x : \mathtt{int} \to \mathtt{int}}\ (\textsc{App}) \qquad \overset{(\textsc{Data const})}{\overline{\Gamma_0, x : \mathtt{int} \vdash \mathtt{1} : \mathtt{int}}}}{\Gamma_0, x : \mathtt{int} \vdash \mathtt{+}\,x\,\mathtt{1} : \mathtt{int}}\ (\textsc{App})}{\Gamma_0 \vdash \lambda x.\mathtt{+}\,x\,\mathtt{1} : \mathtt{int} \to \mathtt{int}}\ (\textsc{Fun})$$

Question 2 Donnez l'arbre de typage pour les termes suivants

1. $\lambda y.\mathtt{or}\ y\ \mathtt{true}$
2. $(\lambda y.\mathtt{or}\ y\ \mathtt{true})\ true$

Question 3 Donnez l'arbre de typage pour les termes suivants

1. $(\lambda y.\mathtt{or}\ y\ \mathtt{true})\ 3$
2. $3\ \ 1$
3. $(\lambda x.xx)(\lambda x.xx)$
4. $\lambda z.z$

Des termes avec plusieurs types ($\lambda x.x$). Les variables abstraites n'étant pas explicitement typées, certains termes peuvent admettre plusieurs types. Ainsi, on peut dériver $\Gamma_0 \vdash \lambda x.x : \mathtt{int} \to \mathtt{int}$ et $\Gamma_0 \vdash \lambda x.x : \mathtt{bool} \to \mathtt{bool}$. Cependant, le terme

$$\mathtt{let}\ id = \lambda x.x\ \mathtt{in}\ \mathtt{paire}\,(id\,\mathtt{true})\,(id\,\mathtt{1})$$

n'est pas typable[3] ! Ce problème sera résolu avec le typage polymorphe.

4 Propriétés du typage

On suppose que les environnements de typage sont consistants avec la réduction des fonctions primitives c'est-à-dire, si $\Gamma \vdash \mathtt{f} : \tau_1 \to \ldots \to \tau_n \to \tau$ et $\Gamma \vdash v_i : \tau_i$ pour tout $i \in \{1, \ldots, n\}$, alors $\mathtt{f}\,v_1 \ldots v_n \underset{(\delta)}{\longrightarrow} v$ avec $\Gamma \vdash v : \tau$.

Le théorème suivant est souvent appelé *subject reduction*.

Théorème 1 (Préservation du typage) *Si $\Gamma \vdash t : \tau$ et $t \to u$ alors $\Gamma \vdash u : \tau$.*

Preuve (indications) : On commence par montrer un *lemme de substitution* :

$$\text{si } \Gamma, x : \tau' \vdash t : \tau \text{ et } \Gamma \vdash u : \tau' \text{ alors } \Gamma \vdash [u/x]t : \tau.$$

[3] on suppose ici que `paire` est un constructeur de paires de type `(* bool int)`.

On montre ensuite le théorème pour les réductions de bases et on montre $(Cont)$ par induction sur la définition des contextes d'évaluation. □

On suppose que toute application bien typée d'une fonction primitive à des valeurs est réductible, c'est-à-dire si $\Gamma \vdash \mathtt{f}\, v_1 \ldots v_n : \tau$ alors $\mathtt{f}\, v_1 \ldots v_n \underset{(\delta)}{\longrightarrow} v$.

On dit qu'un terme t est **clos** ssi $\mathit{fv}(t) = \emptyset$.

Théorème 2 (Progression) *Si t est un terme clos bien typé, alors soit t est une valeur soit il existe un terme u tel que $t \to u$.*

Preuve (indications) : par induction sur la dérivation du typage de t, i.e. $\Gamma \vdash t : \tau$. □

5 Typage polymorphe

L'objectif du polymorphisme est de typer des termes acceptables tels que

$$\mathtt{let}\ \ id = \lambda x.x\ \ \mathtt{in}\ \mathtt{paire}\,(id\,\mathtt{true})\,(id\,\mathtt{1})$$

mais non simplement typable. Dans ce dernier terme, on assigne ainsi à la variable *id* le **schéma de type** $\forall\alpha.\alpha \to \alpha$ où α est une **variable de type** (notée `'a`, `'b`, ... en OCaml). Le type de *id* est en effet indépendant du type de son argument pourvu que le résultat soit aussi du type de l'argument ! Lorsque *id* est appliqué, son type « concret » est le type $\alpha \to \alpha$ où on substitue α au type « concret » de l'argument. Ainsi, le schéma de type $\forall\alpha.\alpha \to \alpha$ représente donc l'ensemble de toutes les instances de $\alpha \to \alpha$. Par exemple, du point de vue du terme $(id\,\mathtt{true})$ le type de id est $\mathtt{bool} \to \mathtt{bool}$ alors que du point de vue de $(id\,\mathtt{1})$ son type est $\mathtt{int} \to \mathtt{int}$. Dans Core ML, le polymorphisme est introduit par la contruction `let`.

Les types sont maintenant aggrémentés de variables de type :

$$\tau \quad ::= \quad \alpha \mid \tau_1 \to \tau_2 \mid g(\tau_1, \ldots, \tau_n)$$

Un *schéma de type* est noté $\forall\bar{\alpha}.\tau$ où $\bar{\alpha}$ est liste de variables de type. On notera simplement τ pour $\forall.\tau$ (i.e. une liste vide de variables de types).

Un environnement de typage Γ associe maintenant des schémas de type aux variables et constantes :

$$\Gamma \ ::= \ \emptyset \mid \Gamma, x : \forall\bar{\alpha}.\tau \mid \Gamma, \mathtt{f} : \forall\bar{\alpha}.\tau \mid \Gamma, \mathtt{c} : \forall\bar{\alpha}.\tau$$

mais les jugements de types associent toujours un type à un terme : $\Gamma \vdash t : \tau$. Par conséquent, les règles de typage des constantes et variables sont modifiées ainsi :

$$\frac{\mathtt{f} \in \mathrm{dom}(\Gamma) \qquad \Gamma(\mathtt{f}) = \forall\bar{\alpha}.\tau}{\Gamma \vdash \mathtt{f} : [\bar{\tau}'/\bar{\alpha}]\tau}\ (\textsc{Fun prim}) \qquad \frac{\mathtt{c} \in \mathrm{dom}(\Gamma) \qquad \Gamma(\mathtt{c}) = \forall\bar{\alpha}.\tau}{\Gamma \vdash \mathtt{c} : [\bar{\tau}'/\bar{\alpha}]\tau}\ (\textsc{Data const})$$

$$\frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma(x) = \forall\bar{\alpha}.\tau}{\Gamma \vdash x : [\bar{\tau}'/\bar{\alpha}]\tau}\ (\textsc{Var})$$

où $[\bar{\tau}'/\bar{\alpha}]\tau$ est le type τ dont les variables de type la liste $\bar{\alpha}$ ont été simultanément substituées par la liste de types $\bar{\tau}'$. La contruction `let` introduit de nouveaux schémas de types dans l'environnement de typage :

$$\frac{\Gamma \vdash u : \tau \qquad \Gamma, x : \forall\bar{\alpha}.\tau \vdash t : \tau' \qquad \bar{\alpha} = \mathit{ftv}(\tau) \backslash \mathit{ftv}(\Gamma)}{\Gamma \vdash \mathtt{let}\ \ x = u\ \ \mathtt{in}\ \ t : \tau'}\ (\textsc{Let})$$

Les autres règles restent inchangées. Notez bien que les quantifications sont toujours en tête de schéma, ainsi $\forall\alpha.\alpha \to \forall\alpha.\alpha$ n'est ni un type ni un schéma de type. C'est la raison pour laquelle le typage de l'abstraction n'introduit pas de schémas de type dans l'environnement de typage !