/SU/FSI/MASTER/INFO/MU4IN503 (APS)

Analyse des Programmes et Sémantique

Janvier 2021

Pascal Manoury – Romain Demangeon
pascal.manoury@lip6.fr

2 : Typage

Langage typé

Deux *domaines* de valeurs : les entiers et les booléens

Des fonctions qui manipulent ces valeurs :

- *domaine* (argument)
- *codomaine* (résultat)

Dans le langage : *expression de type*
`bool int (int -> int) (int * int -> bool)` etc.

Typage : cohérence entre
manipulation et représentation mémoire
calculs et codage

Typage

Analyse statique du code des programmes

(*statique* : préalable à l'exécution)

But :

vérifier que le code écrit respecte les indications de type qu'il contient

Bénéfice :

le code exécuté n'a plus besoin d'information de type

Autres possibilités [1]

- vérification *dynamique* (à l'exécution)
  (le code exécuté contient les informations de type)
- *inférence* de type
  (le code source ne contient pas d'information de type)

1. Que nous n'étudierons pas

Moyen

Assignation de type

A priori

- les constantes booléenne `true false` sont de type `bool`
- les constantes numériques `num` sont de type `int`
- les fonctions primitives ont un *type prédéfini*
  (`eq` est de type `int * int -> bool` etc.)

Explicite déclarations

- type des constantes
- type de retour des fonctions
- types des paramètres

Calculée

Règles de typage

Identificateurs et types

Quel type pour les identificateurs ?

- fixé pour les fonctions prédéfinies
  ou les constantes booléennnes
- donné pour les noms des paramètres
- donné pour les noms des fonctions définies
  (types des paramètres -> type de retour)

Dépend du code source analysé.
Assignation de type *contextuelle*

Contexte de typage

Suite d'association (`ident`, $\textsc{Type}$).

- théorie : fonction `ident` $\rightarrow \textsc{Type}$
- pratique : liste d'association

Principe de l'analyse de type

Guidé par la syntaxe [2]

- Grammaire : définition récursive de la *syntaxe concrète* (non terminal $\textsc{Prog}$)
- Arbre de syntaxe abstraite : structure de donnée récursive (arbre)

Poser une *règle de typage* pour chaque cas de construction syntaxique

2. *syntax directed*

Règle de typage

Définit une *relation* (ternaire) entre

- un contexte de typage, un élément syntaxique et un type
- ou, un contexte, un élément syntaxique et un contexte

Jugement de typage

Système déductif (*si . . .alors . . .*)

- Prémisses :
  - condition externe
  - ou ensemble de jugements de typage
- Conclusion :
  - jugement de typage

Langage de types

Quel type pour l'instruction `ECHO` ?

Suivons l'usage : `void`

Langage de types pour la vérification de type

$$\textsc{Type} \cup \{\texttt{void}\}$$

Remarque :

- aucune constante définie n'est de type `void`
- aucune fonction n'est à valeur dans `void`
- aucun paramètre n'est de type `void`

Formalisation des contextes de typage

L'ensemble Cx des *contextes de typage* est l'ensemble des fonctions (partielles) $\texttt{ident} \to \textsc{Type}$

Notation [3],

Si $x \in \texttt{ident}$ et $\Gamma \in Cx$,
on note $\Gamma(x)$ le type de x donné par Γ.
Si $x \in \texttt{ident}$, $\tau \in \textsc{Type}$ et $\Gamma \in Cx$,
on note $x : \tau$ l'assiciation du type τ à l'identificateur x,
on note $\Gamma[x : \tau]$ le contexte *étendant* Γ avec l'assignation $x : \tau$.

On a
$$\begin{array}{lcll} \Gamma[x:\tau](x) & = & \tau & \\ \Gamma[x:\tau](y) & = & \Gamma(y) & \textit{si } x \neq y \end{array}$$

Abréviation

$[x_1 : \tau_1; \ldots; x_n : \tau_n]$ pour $[x_1 : \tau_1] \ldots [x_n : \tau_n]$

---

3. https://www.lexilogos.com/grec_alphabet.htm

Contexte initial

Donne le type des identificateurs choisis pour les fonctions primitives et les constantes booléennes.

Soit Γ_0 tel que :

$$
\begin{array}{lcl}
\Gamma_0(\texttt{true}) & = & \texttt{bool} \\
\Gamma_0(\texttt{false}) & = & \texttt{bool} \\
\Gamma_0(\texttt{not}) & = & \texttt{bool -> bool} \\
\Gamma_0(\texttt{and}) & = & \texttt{bool * bool -> bool} \\
\Gamma_0(\texttt{or}) & = & \texttt{bool * bool -> bool} \\
\Gamma_0(\texttt{eq}) & = & \texttt{int * int -> bool} \\
\Gamma_0(\texttt{lt}) & = & \texttt{int * int -> bool} \\
\Gamma_0(\texttt{add}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{sub}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{mul}) & = & \texttt{int * int -> int} \\
\Gamma_0(\texttt{div}) & = & \texttt{int * int -> int}
\end{array}
$$

Remarque : `if` n'y figure pas

Relations de typages

On en considère cinq :

Programme $\vdash p : \texttt{void}$
avec $p \in \textsc{Prog}$.

Commandes $\Gamma \vdash_{\textsc{Cmds}} cs : \texttt{void}$
avec $\Gamma \in Cx$ et $cs \in \textsc{Cmds}$

Déclaration $\Gamma \vdash_{\textsc{Dec}} d : \Gamma'$
avec $\Gamma, \Gamma' \in Cx$ et $d \in \textsc{Dec}$.

Instruction $\Gamma \vdash_{\textsc{Stat}} s : \texttt{void}$
avec $\Gamma \in Cx$ et $s \in \textsc{Stat}$

Expressions $\Gamma \vdash_{\textsc{Expr}} e : \tau$
avec $\Gamma \in Cx$, $e \in \textsc{Expr}$ et $\tau \in \textsc{Type}$

Expressions atomiques

Relation $\vdash_{\text{EXPR}}$

Les constantes numériques

(NUM) si $n \in \texttt{num}$
alors $\Gamma \vdash_{\text{EXPR}} n : \texttt{int}$

Les identificateurs

(ID) si $x \in \texttt{ident}$,
si $\Gamma(x) = \tau$
alors $\Gamma \vdash_{\text{EXPR}} x : \tau$

Implicitement : si $\Gamma(x)$ n'est pas défini ($x \notin \text{dom}(\Gamma)$)

- la relation n'est pas satisfaite
- et la vérification de type échouera

Application

Relation $\vdash_{\text{EXPR}}$

Expression de la forme $(e\ e_1 \ldots e_n)$

(APP) si $\Gamma \vdash_{\text{EXPR}} e : (\tau_1$ `*` $\ldots$ `*` τ_n `->` $\tau)$,
si $\Gamma \vdash_{\text{EXPR}} e_1 : \tau_1$,
$\vdots$
si $\Gamma \vdash_{\text{EXPR}} e_n : \tau_n$
alors $\Gamma \vdash_{\text{EXPR}} (e\ e_1 \ldots e_n) : \tau$

Définition récursive

L'application partielle est interdite par typage

Abstraction

Relation $\vdash_{\text{EXPR}}$

Expression fonctionnelle de la forme $[x_1 : \tau_1, \ldots, x_n : \tau_n]\, e$

(ABS) si $\Gamma[x_1 : \tau_1; \ldots; x_n : \tau_n] \vdash_{\text{EXPR}} e : \tau$
alors $\Gamma \vdash_{\text{EXPR}} [x_1 : \tau_1, \ldots, x_n : \tau_n]\, e$
$: (\tau_1 \ * \ \ldots \ * \ \tau_n \ \texttt{->} \ \tau)$

Alternative

Relation $\vdash_{\text{EXPR}}$

Expression de la forme $(\texttt{if}\ e_1\ e_2\ e_3)$

(IF) si $\Gamma \vdash_{\text{EXPR}} e_1 : \texttt{bool}$,
si $\Gamma \vdash_{\text{EXPR}} e_2 : \tau$,
si $\Gamma \vdash_{\text{EXPR}} e_3 : \tau$
alors $\Gamma \vdash_{\text{EXPR}} (\texttt{if}\ e_1\ e_2\ e_3) : \tau$

Opérateur *polymorphe*
(pour tout type τ)

Instruction

Relation $\vdash_{\text{STAT}}$

L'instruction d'affichage (`ECHO` e)

$$(\text{ECHO})\quad \text{si } \Gamma \vdash_{\text{EXPR}} e : \texttt{int}$$
$$\text{alors } \Gamma \vdash_{\text{STAT}} (\texttt{ECHO}\ e) : \texttt{void}$$

(On parenthèse pour la lisibilité)

Restriction purement arbitraire

Déclaration de constante

Relation $\vdash_{\text{DEC}}$

Déclaration de la forme $\texttt{CONST}\ x\ \tau\ e$

(CONST) si $\Gamma \vdash_{\text{EXPR}} e : \tau$
alors $\Gamma \vdash_{\text{DEC}} (\texttt{CONST}\ x\ \tau\ e) : \Gamma[x : \tau]$

Rappel : la règle assigne un (nouveau) *contexte* de typage à la déclaration

Déclaration de fonction

Relation $\vdash_{\text{DEC}}$

Déclaration de la forme $\texttt{FUN}\ x\ t\ [x_1 : \tau_1, \ldots, x_n : \tau_n]\ e$

$$(\text{FUN})\quad \text{si } \Gamma[x_1 : \tau_1; \ldots; x_n : \tau_n] \vdash_{\text{EXPR}} e : \tau$$
$$\text{alors } \Gamma \vdash_{\text{DEC}} (\texttt{FUN}\ x\ \tau\ [x_1 : \tau_1, \ldots, x_n : \tau_n]\ e)$$
$$: \Gamma[x : (\tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n\ \texttt{->}\ \tau)]$$

Notez :

1. la ressemblance avec (ABS)
2. comment l'extension de contexte $[x_1 : \tau_1; \ldots; x_n : \tau_n]$ et le type τ sont ≪lus≫ dans la déclaration.

Notez également :

- l'utilisation de la relation $\vdash_{\text{EXPR}}$

Déclaration de fonction récursive

Relation $\vdash_{\text{DEC}}$

Déclaration de la forme $\texttt{FUN REC}\ x\tau\,[x_1 : \tau_1, \ldots, x_n : \tau_n]\,e$

$$(\text{FUNREC})\quad \text{si } \Gamma[x_1 : \tau_1; \ldots; x_n : \tau_n; x : \tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n\ \texttt{->}\ \tau] \vdash_{\text{EXPR}} e : \tau$$
$$\text{alors } \Gamma \vdash_{\text{DEC}} (\texttt{FUN REC}\ x\ t\ [x_1 : \tau_1, \ldots, x_n : \tau_n]\ e) : \Gamma[x : \tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n\ \texttt{->}\ \tau]$$

On ajoute au contexte le *type déclaré* de la fonction

Suite de commandes

Relation $\vdash_{\text{CMDS}}$

Facilité syntaxique : [4] on marque la fin d'une suite de commandes par la *commande vide* ε

$$cs ::= \varepsilon \mid d\,;cs \mid s\,;cs$$

avec $d \in \text{DEC}$ et $s \in \text{STAT}$

On pose :

(END) $\Gamma \vdash_{\text{CMDS}} \varepsilon : \texttt{void}$.

4. Nous servira par la suite

Suite de commandes et déclaration

Relation $\vdash_{\text{CMDS}}$

Suite de commandes de la forme $d\,;cs$

(DECS) si $d \in \text{DEC}$,
si $\Gamma \vdash_{\text{DEC}} d : \Gamma'$,
si $\Gamma' \vdash_{\text{CMDS}} cs$: `void`
alors $\Gamma \vdash_{\text{CMDS}} (d\,;cs)$: `void`.

Notez :

1. l'utilisation de $\vdash_{\text{DEC}}$
2. comment le contexte Γ' sert au typage de la suite cs

Suite de commandes et instruction

Relation $\vdash_{\text{CMDS}}$

Suite de commandes de la forme $s\,;cs$

(STATS) si $s \in \text{STAT}$,
si $\Gamma \vdash_{\text{STAT}} s : \texttt{void}$,
si $\Gamma \vdash_{\text{CMDS}} cs : \texttt{void}$
alors $\Gamma \vdash_{\text{CMDS}} (s\,;cs) : \texttt{void}$.

Remarque : dans $APS0$ $cs = \varepsilon$

Programme

Relation $\vdash$

Par définition :

si $p \in \textsc{Prog}$ alors, il existe $cs \in \textsc{Cmds}$ telle que $p = [cs]$

On pose

$(\textsc{prog})$ si $\Gamma_0 \vdash_{\textsc{Cmds}} (cs ; \varepsilon) : \texttt{void}$
alors $\vdash [cs] : \texttt{void}$

Notez l'intervention du *contexte initial* Γ_0

Vérification de type

Les règles de typages sont

1. déterministes
2. bien fondées

On en déduit un

algorithme de typage