Programmation fonctionnelle: notes de cours 2

S5 – 2023/2024

1 Un langage typé

Nous avons vu précédemment que le lambda-calcul permettait de coder différents types de données, comme des booléens ou des entiers. Ce codage est suffisant en théorie, mais peu pratique à manipuler. Nous allons donc étendre le langage avec des constantes booléennes (*true* et *false*) et numériques (0, 1, 2, ...), ainsi que des fonctions arithmétiques (+, <) et une fonction ternaire *if* _ *then* _ *else* _. Chacun de ces symboles de fonction aura ses propres règles d'évaluation, mais celles-ci sont intuitives ($1 + 1$ se réduit à 2, $3 < 1$ se réduit à *false*, *if false then* t_1 *else* t_2 se réduit à t_2, etc.)

Jusqu'ici nous avons considéré la version non-typée du lambda-calcul : n'importe quel lambda-terme peut être appliqué à un autre. Rien n'interdit donc d'écrire des lambda-termes tels que $\lambda x.3 + \mathit{true}$, qui ne pourront pourtant pas être évalués. Afin d'éviter ce problème nous allons restreindre les termes acceptables à l'aide d'un système de typage : chaque terme sera dôté d'un type, et pour être bien typé (c.-à-d. correct vis-à-vis du système de typage), un terme correspondant à une fonction ne pourra être appliqué qu'à des arguments d'un type compatible.

1.1 Les types

Nous allons à présent ajouter des règles de typage au lambda-calcul vu précédemment. Pour cela nous devons établir quels sont les types que peuvent prendre les lambda-termes. Ceux-ci sont définis récursivement :

— $\mathbb{B}$ est un type,
— $\mathbb{N}$ est un type,
— si T_1 et T_2 sont des types, alors $T_1 \to T_2$ est un type.

Le premier type est associé aux expressions booléennes, telles que *true* ou $5 < 3$. Le deuxième type est associé aux expressions arithmétiques, telles que 3 ou $2 + 5$. Le troisième cas correspond aux lambda-abstractions : il dénote le type des fonctions qui prennent un argument de type T_1 et retournent un terme de type T_2. Parmi ces types, on trouvera $\mathbb{N} \to \mathbb{B}$ (le type d'une fonction des entiers vers les booléens, par exemple la fonction qui détermine si un entier est pair), mais aussi $(\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$ ou encore $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. Notez que l'opérateur $\to$

est associatif à droite, donc les parenthèses dans ce dernier exemple ne sont pas nécessaires et seront en général omises.

1.2 Typage des expressions arithmétiques et booléennes

Maintenant que les types sont établis, nous pouvons mettre en place les règles de typage qui permettent de déterminer le type d'un terme. On utilise la notation $t : T$ pour indiquer que le terme t est de type T. Une règle de typage se présente sous cette forme :

$$\frac{n_1 : \mathbb{N} \qquad n_2 : \mathbb{N}}{n_1 + n_2 : \mathbb{N}}$$

au dessus de la barre horizontale se trouvent les conditions de la règle, et en dessous, sa conclusion. La règle ci-dessus peut être comprise ainsi : "si le terme n_1 est de type $\mathbb{N}$ et si le terme n_2 est de type $\mathbb{N}$, alors le terme $n_1 + n_2$ est de type $\mathbb{N}$". Une règle peut ne pas avoir de condition, par exemple

$$\frac{}{\mathit{true} : \mathbb{B}}$$

indique simplement que le terme *true* est de type $\mathbb{B}$.

On définit ainsi un système de typage pour les expressions booléennes et arithmétiques, illustré dans la Figure 1. Notez que la fonction *if* _ *then* _ *else* _ est polymorphe, puisque T peut correspondre à n'importe quel type. Toutefois pour être correctement typée, cette expression doit prendre deux arguments t_1 et t_2 de même type.

$$\frac{}{\mathit{true} : \mathbb{B}} \qquad \frac{}{\mathit{false} : \mathbb{B}} \qquad \frac{}{0 : \mathbb{N}} \qquad \frac{}{1 : \mathbb{N}} \qquad \frac{}{2 : \mathbb{N}} \qquad \cdots$$

$$\frac{n_1 : \mathbb{N} \qquad n_2 : \mathbb{N}}{n_1 + n_2 : \mathbb{N}} \qquad \frac{n_1 : \mathbb{N} \qquad n_2 : \mathbb{N}}{n_1 < n_2 : \mathbb{B}}$$

$$\frac{b : \mathbb{B} \qquad t_1 : T \qquad t_2 : T}{\mathit{if}\ b\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : T}$$

Figure 1: Les règles de typage pour les expressions arithmétiques et booléennes

Pour vérifier qu'un terme est d'un type donné, on utilise les règles du bas vers le haut : on part de la conclusion que l'on cherche à prouver, et on vérifie si les conditions sont satisfaites. Si la règle n'a pas de condition, la preuve est terminée, dans le cas contraire, il faut prouver récursivement ces conditions, en formant ainsi un arbre de preuve. Si la racine de cet arbre contient la conclusion $t : T$ et si les feuilles ne sont constituées que de règles sans condition, alors on a une preuve que le terme t est de type T. Par exemple l'arbre suivant arbre est une preuve de typage pour l'expression *if* $(0 < 1)$ *then* 3 *else* $(5 + 1)$

$$\dfrac{\dfrac{\dfrac{}{0:\mathbb{N}} \quad \dfrac{}{1:\mathbb{N}}}{0<1:\mathbb{B}} \quad \dfrac{}{3:\mathbb{N}} \quad \dfrac{\dfrac{}{5:\mathbb{N}} \quad \dfrac{}{1:\mathbb{N}}}{5+1:\mathbb{N}}}{\textit{if } (0<1) \textit{ then } 3 \textit{ else } (5+1):\mathbb{N}}$$

Inversement si un terme est mal typé, ou si on essaye de prouver qu'il est d'un autre type que le sien, il sera impossible de construire un tel arbre.

1.3 Typage des lambda-termes

Sur le même principe, nous pouvons typer les lambda-termes. Pour les lambda-termes formés par application, la règle est la suivante :

$$\frac{t_1 : T \to T' \qquad t_2 : T}{t_1\, t_2 : T'}$$

La règle correspond à l'intuition : si une fonction t_1, dont le type indique qu'elle prend en entrée un terme de type T et retourne un terme de type T', est appliquée à un argument t_2 de type T, on obtient bien un terme de type T'.

Pour les lambda-abstractions, il va être nécessaire d'avoir plus d'informations sur les variables. En effet, considérons le terme $\lambda x.x$: il est clair que son type doit ressembler à $T \to T$, mais en l'état il est impossible de savoir quel est le type T. Cette information devra donc être indiquée dans la lambda-abstraction elle-même. Dans le lambda-calcul typé, la syntaxe pour une lambda-abstraction est la suivante :

$$\lambda(x : T).\, e$$

où x est une variable, T un type, et e un lambda-terme. Par ailleurs, le type de e va dépendre de l'hypothèse sur le type de x, et puisque les lambda-abstractions peuvent être imbriquées, il va parfois falloir considérer une collection d'hypothèses Γ. Pour noter ceci, nous écrirons $\Gamma \vdash t : T$, où Γ est une collection d'hypothèses sur les types des variables. Cette notation signifie que le terme t est de type T sous les hypothèses Γ. La règle de typage pour les lambda-abstractions est alors la suivante

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda(x : T).\, e : T \to T'}$$

Si on utilise la règle du bas vers le haut pour typer un terme, cette règle revient à enrichir nos hypothèses Γ avec une nouvelle hypothèse concernant le type de x avant de vérifier le type du terme e.

La règle suivante permet d'utiliser une hypothèse pour typer une variable

$$\frac{}{\Gamma, x : T \vdash x : T}$$

Puisqu'il s'agit de la seule façon de typer une variable, et que seule la règle de typage des lambda-abstractions permet d'introduire une hypothèse, on constate que seule une variable liée peut être typée. Sans hypothèse de typage préalable, un lambda-terme contenant une variable libre (par exemple $\lambda(x : T).\, y\, x$) ne peut pas être typé.

$$\frac{}{\Gamma, x : T \vdash x : T}$$

$$\frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\, t_2 : T'} \qquad \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda(x : T).\, e : T \to T'}$$

Figure 2: Les règles de typage du lambda-calcul

En adaptant la règle pour le typage de l'application pour tenir compte des hypothèses, on obtient le système de typage du lambda-calcul en Figure 2.

L'arbre de typage suivant donne un exemple qui combine lambda-abstraction, expressions booléennes et arithmétiques. La différence principale avec l'arbre précédent est la présence d'hypothèses dans les formules de typage. On suppose que les règles de typage pour les expressions booléennes et arithmétiques ont été modifiées pour tenir compte de cette caractéristique.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathbb{N} \vdash x : \mathbb{N}} \quad \dfrac{}{x : \mathbb{N} \vdash 3 : \mathbb{N}}}{x : \mathbb{N} \vdash x + 3 : \mathbb{N}} \quad \dfrac{}{x : \mathbb{N} \vdash 5 : \mathbb{N}}}{x : \mathbb{N} \vdash x + 3 < 5 : \mathbb{B}}}{\vdash \lambda(x : \mathbb{N}).x + 3 < 5 : \mathbb{N} \to \mathbb{B}} \quad \dfrac{}{\vdash 6 : \mathbb{N}}}{\vdash (\lambda(x : \mathbb{N}).x + 3 < 5)\ 6 : \mathbb{B}}$$

1.4 Types et programmation

Presque tous les langages de programmation utilisent des types, plus ou moins descriptifs. On dit d'un langage qu'il est typé *statiquement* si la vérification des types est faite avant l'exécution (par exemple à la compilation), ou typé *dynamiquement* si cette vérification est faite au fur et à mesure de l'exécution.

Jusqu'ici nous avons vu la question de la *vérification de type*, à savoir vérifier qu'un terme est d'un type donné (en général par le programmeur, via une annotation). Un problème proche est celui de l'*inférence de type*, qui consiste à déterminer le type d'un terme. L'inférence de type permet de vérifier que le programme est correctemment typé sans que le programmeur n'ait à fournir le type des expressions (à l'exception des variables).

L'utilisation d'un langage avec un système de type fort et vérifié statiquement a de multiples avantages pour un programmeur. Les types permettent à un compilateur de détecter des termes incorrects sans les évaluer, ce qui est particulièrement utile dans un langage fonctionnel où des programmes peuvent être combinés de manière complexe. Les types fournissent aussi une abstraction des fonctions qui permet de raisonner uniquement sur les valeurs d'entrée et de sortie (*quoi ?*) plutôt que sur les algorithmes (*comment ?*). Enfin les types documentent le code, permettant souvent de deviner ce qu'une fonction calcule.