Analyse des types

Martin Odersky

version 1.2

Plan du cours

À quoi servent les types

La déclaration des identificateurs (présentée dans le cours précédent) n'est pas la seule chose à vérifier dans un compilateur.

En Vier, comme dans la plupart des langages de programmation, les expressions ont un type. Il faut donc vérifier que les types sont corrects et retourner un message d'erreur dans le cas contraire.

Exemple : Quelques règles de typage de Vier

- Les opérandes de + doivent être des entiers.
- Les opérandes de == doivent être compatibles (`Int` avec `Int` est compatible de même que `List` avec `List`, mais pas `Int` avec `List`).
- Le nombre d'arguments passés à une méthode doit être égal au nombre de paramètres formels de cette méthode.
- etc.

Comment spécifie-t-on les règles de typage ?

Propriétés de typage

Définition : Fortement/faiblement typé

Un langage est dit fortement typé (*strongly typed*) ou sûr si la violation d'une règle de typage entraîne une erreur.

Il est dit faiblement typé (*weakly typed*) ou non-typé dans les autres cas — en particulier si le comportement du programme n'est plus spécifié en cas de typage incorrect.

Définition : Statiquement/dynamiquement typé

Un langage est dit statiquement typé (*statically typed*) s'il existe un système de typage qui peut détecter des programmes incorrects avant que ceux-ci ne soient exécutés.

Il est dit dynamiquement typé dans les autres cas.

Attention : Une langage fortement typé n'est pas un langage statiquement typé !

	fortement	faiblement
statiquement		
dynamiquement		

En pratique, certains tests sont quand même exécutés dynamiquement dans les langages statiquement typés car ils sont difficiles statiquement (par exemple la longueur des tableaux).

Il existe des langages qui sont complètement statiquement sûrs (par exemple la théorie des types de Martin Loef).

- Mais la preuve de leur sûreté doit être explicitement ajoutée au programme.

Exercices

- Le contrôle des types ou la reconstitution des types sont-ils toujours possibles ?

Objectif

On veut donner une spécification exacte de la partie du compilateur chargée de l'analyse des types et des noms.

- On va donc définir formellement, c'est-à-dire mathématiquement, l'ensemble des programmes Vier qui doivent être acceptés par cette phase du compilateur.
- Les programmes qui passeront le test seront dits bien typés car ils correspondent aux programmes pour lesquels il est possible de donner un type à chaque sous-expression.

Cette description mathématique permettra de répondre sans ambiguïté à toutes les questions portant sur l'implémentation de l'analyseur de types.

Dépasser les notations

Il faut seulement être capable de lire les définitions mathématiques :

- La difficulté de lecture des définitions mathématiques ne réside pas dans leur complexité intrinsèque, mais seulement dans les notations utilisées pour rendre leur formulation compacte et élégante.
- Il faut aussi se rappeler qu'il n'y a rien à comprendre dans une définition.

En bref, il ne faut pas se laisser impressionner par les notations.

Un petit langage

Avant d'aborder le typage du langage Vier proprement dit, on se limite à un petit langage composé uniquement de constantes entières, de l'opérateur binaire $+$, et du type **Int**.

Exemple : Le petit langage

La grammaire du petit langage est :

$$\begin{array}{rccl} \text{Constantes entières} & i & = & \mathbf{Int}. \\ \text{Expressions} & e & = & c \\ & & \mid & e_1 + e_2. \\ \textit{Types} & t & = & \mathbf{Int}. \end{array}$$

Typage du petit langage

Une expression e sera dite bien typée de type T si et seulement si :

1. c'est une expression de la forme i et $T = \textbf{Int}$, ou
2. c'est une expression de la forme $e_1 + e_2$, où e_1 et e_2 sont deux expressions bien typées de type **Int**, et $T = \textbf{Int}$.

Comme on le voit cette définition est récursive.

Une manière de représenter de telles définitions récursives est d'écrire des règles d'inférence :

$$(\text{IntLit})\frac{}{\vdash i : \textbf{Int}} \qquad (\text{Binop})\frac{\vdash e_1 : \textbf{Int} \qquad \vdash e_2 : \textbf{Int}}{\vdash e_1 + e_2 : \textbf{Int}}$$

Règles d'inférence

Une règle d'inférence est composée :

- d'une barre horizontale ;
- d'un nom placé à gauche ou à droite de la barre ;
- d'une liste de prémisses placée au dessus de la barre ;
- d'une conclusion placée au dessous de la barre.

Une règle d'inférence dont la liste de prémisses est vide est appelée un axiome.

Arbres de dérivation

Une règle d'inférence se lit de bas en haut.

Exemple

$$(\text{Binop})\ \frac{\vdash e_1 : \textbf{Int} \qquad \vdash e_2 : \textbf{Int}}{\vdash e_1 + e_2 : \textbf{Int}}$$

se lit «Pour montrer qu'une expression de la forme $e_1 + e_2$ a le type **Int**, il suffit de montrer que e_1 et e_2 ont le type **Int**».

- Autrement dit, pour montrer la conclusion d'une règle, on peut se contenter de montrer ses prémisses.
- Ce processus s'arrête naturellement quand on rencontre une règle qui n'a pas de prémisses, c'est-à-dire un axiome.

La représentation de la relation de typage sous forme de règles d'inférence permet d'avoir une représentation graphique de la preuve qu'une expression est bien typée.

Exemple

$$(\text{Bi})\ \frac{(\text{Bi})\ \frac{(\text{In})\ \frac{}{\vdash 1 : \textbf{Int}} \quad (\text{In})\ \frac{}{\vdash 2 : \textbf{Int}}}{\vdash 1 + 2 : \textbf{Int}} \quad (\text{In})\ \frac{}{\vdash 3 : \textbf{Int}}}{\vdash (1 + 2) + 3 : \textbf{Int}}$$

Un tel arbre est appelé un arbre de dérivation.

En conclusion, une expression e est bien typée de type T ssi il existe un arbre de dérivation dont la conclusion est $\vdash e : T$.

Ajout des identificateurs

Supposons que l'on veuille maintenant ajouter au langage les identificateurs :

Exemple : Le petit langage qui grandit

$$\begin{array}{rrcl} \text{Identificateurs} & x & = & \textit{string} \\ \text{Expressions} & e & = & \ldots \\ & & \mid & x \end{array}$$

- Pour déterminer si une expression comme $x + 1$ est bien typée, il faut avoir des informations sur le type de la valeur représentée par l'identificateur x.

On rajoute donc à la relation de typage un environnement Γ (*Gamma*) associant un type à chaque identificateur apparaissant dans l'expression à typer. On écrit maintenant $\Gamma \vdash e : T$.

Environnements

Un environnement Γ est une liste de paires constituées d'un identificateur x et d'un type T notées $x : T$.

Il faut ajouter à nos règles d'inférence une règle pour décider quand une expression consistant en un identificateur x est bien typée :

$$(\text{Ident})\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Exemple : Dérivation d'une expression du petit langage qui grandit

Dans l'environnement $\Gamma = x : \textbf{Int}$, il est possible de typer $x + 1$:

$$(\text{Binop})\frac{(\text{Ident})\dfrac{}{\Gamma \vdash x : \textbf{Int}} \quad (\text{IntLit})\dfrac{}{\Gamma \vdash 1 : \textbf{Int}}}{\Gamma \vdash x + 1 : \textbf{Int}}$$

Exemple : Le langage MFL

Nous allons maintenant présenter les règles de typage pour un langage simple mais complet, le langage MFL (*mini functional language*).

Ce langage dispose :

- de la définition de valeurs et de fonctions ;
- d'identifiants et d'applications de fonction ;
- d'expressions arithmétiques et booléennes ;
- des types primitifs **Int** et **Boolean** ;
- de types fonctionnels $T_1 \Rightarrow T_2$.

Syntaxe abstraite de MFL

Integers	i, j			
Identifiers	$x, f,$	$\ldots$		
Expressions	E, F	$::=$	i	Literal
		$\mid$	x	Identifier
		$\mid$	$E_1 + E_2$	Addition
		$\mid$	$E_1 = E_2$	Comparison
		$\mid$	$E_1(E_2)$	Application
		$\mid$	**if** $(E_1)\, E_2$ **else** E_3	Conditional
		$\mid$	**val** $x : T = E_1;\ E_2$	Value def
		$\mid$	**def** $f(x : T_1) : T_2 = E_1;\ E_2$	Function def
Types	S, T	$::=$	**Int**	
		$\mid$	**Boolean**	
		$\mid$	$T_1 \Rightarrow T_2$	

Le sens des types

Question : Qu'est-ce que signifie un type ?

Réponse : Les types sont des ensembles de valeurs. Par exemple :

$$T \approx \{V \mid V : T\}$$

où la valeur $V = i \mid \textbf{true} \mid \textbf{false} \mid$ "*une fonction*"

Question : A quoi sert un type ?

Réponse : Les jugements obtenus par le typage restent vrais à l'exécution.

Sûreté de type

On peut poser et démontrer le théorème suivant sur MFL :

Théorème

Si $\vdash E : T$ et l'évaluation de E termine, alors eval(E) : T.

Dans ce sens, les types nous fournissent une information sur le résultat d'un programme !

- On peut résumer cette propriété par le slogan :

  "Well-typed programs cannot go wrong"

  (Un programme typé correctement n'ira pas de travers)
- Même si le programme pourrait ne pas terminer ou générer une exception, ce théorème exclut des comportements erratiques (non-déterminés) à l'exécution.

Quelques notes sur l'implantation

On peut généralement établir une correspondance entre les objets mathématiques de la description formelle et les structures de données utilisées dans le compilateur.

Quelques exemples de correspondances en Vier

$(T)_p$	$\approx$	`case class ParamSymbol(dt: DataType) extends Symbol`
$\Gamma_c ::= \ldots$	$\approx$	`type ClassScope = ...`
$\overline{t} \mapsto \overline{\sigma_c}$	$\approx$	`Map[DataType, ClassSymbol]`
$\Gamma_c ; \Gamma_p \vdash e : T$	$\approx$	`val T = validExpr(cScope, pScope, e)`
$a \mapsto (T)_p \in \Gamma_p$	$\approx$	`(pScope get a) match { case Some(p) => ... case None => error }`
$\Gamma' = \Gamma + x \mapsto \sigma$	$\approx$	`val gamma2 = gamma + ((name, symbol))`

En général, chaque jugement correspond à une méthode, chaque règle à un cas de la méthode.

La validation d'un programme à proprement parler est une descente récursive dans l'AST par ces méthodes.

- Le programme à une structure dynamique équivalente à celle de l'arbre de dérivation d'une preuve manuelle.

Exercice

Quel sera la structure dynamique (stack) d'un programme lors de la vérification de $\vdash 1 : \mathbf{Int}$?

$$\dfrac{\dfrac{\dfrac{(\text{In})\dfrac{}{\vdash 1 : \mathbf{Int}} \quad (\text{In})\dfrac{}{\vdash 2 : \mathbf{Int}}}{(\text{Bi})\ \vdash 1 + 2 : \mathbf{Int}} \quad (\text{In})\dfrac{}{\vdash 3 : \mathbf{Int}}}{(\text{Bi})\ \vdash (1 + 2) + 3 : \mathbf{Int}}$$

Cette vision simple de l'analyse souffre toutefois d'un défaut.

- Conceptuellement, toutes les éléments (prémisses d'une règle, membres d'une classe, etc.) sont résolus simultanément,
- dans un programme, ça ne peut être le cas.

Les implantations de systèmes de types doivent souvent

- séparent la vérification en phases successives
  - pour simuler dans une phase la simultanéité des éléments calculés dans une phase antérieure.
- ou utilisent des valeurs gelées (*lazy*)
  - qui sont uniquement calculées lorsqu'elles deviennent nécessaires.

Le système de types de Vier, par contre, prend soin de définir dans ses règles la notion d'ordre lorsqu'elle est importante. Les "phases succesives" sont donc implicites au règles.

Conclusion

- On a spécifié de manière complètement formelle les programmes qui doivent être acceptés par l'analyseur de noms et de types.
- Cette formalisation mathématique nécessitant de définir des relations de typage par récurrence, on a introduit le concept de règles d'inférence.
- On a montré comment passer de la spécification formelle à l'implantation.