Analyse des types

Martin Odersky

12, 13 et 19 décembre 2005
version 1.1

En collaboration avec :

Vincent Cremet, Gilles Dubochet et Michel Schinz

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 ZWEI, 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 Zwei

- 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.

Exercice

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

- Qu'est-ce qu'un type signifie et qu'est-ce qu'il garantit ?
  ou : qu'est-ce qu'est la sûreté dans le contexte des types ?
- 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 ZWEI 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 ZWEI 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 :

$$\begin{array}{rrcl} \text{Constantes entières} & i & = & int. \\ \text{Expressions} & e & = & c \\ & & | & e_1 + e_2. \\ \textit{Types} & t & = & \textbf{int}. \end{array}$$

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

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

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

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

- 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 MLF

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
		$\mid$	$\textbf{if}\,(E_1)\,E_2\,\textbf{else}\,E_3$	Conditional
		$\mid$	$\textbf{val}\,x : T = E_1;\ E_2$	Value def
		$\mid$	$\textbf{def}\,f(x : T_1) : T_2 = E_1;\ E_2$	Function def
Types	S, T	$::=$	$\textbf{int}$	
		$\mid$	$\textbf{boolean}$	
		$\mid$	$T_1 \Rightarrow T_2$	

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 $\text{eval}(E) : T$.

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.

Typage de ZWEI : 4 sortes d'identificateurs

En ZWEI, les identificateurs représentent quatre sortes de choses : des classes, des champs, des méthodes, des variables.

Chaque sorte d'identificateur a ses propres attributs. On regroupe ces attributs dans des symboles. Un symbole σ peut être :

- un symbole de classe $\texttt{Class}(\overline{a}|\Gamma_f|\Gamma_m)$ où $\overline{a}$ représente les parents, Γ_f les champs et Γ_m les méthodes,
- un symbole de champ $\texttt{Field}(T)$ où T représente le type du champ,
- un symbole de méthode $\texttt{Meth}(\overline{T}|T)$ où $\overline{T}$ représente le type des paramètres et T le type de retour, ou
- un symbole de variable $\texttt{Var}(T)$ où T représente le type de la variable.

4 sortes d'environnements de typage

Les environnements de typage sont maintenant des listes associatives qui associent un symbole à un identificateur : $\Gamma = \overline{a} \mapsto \overline{\sigma}$.

A chaque sorte d'identificateur correspond une sorte d'environnement : on note resp. Γ_c, Γ_f, Γ_m et Γ_v un environnement de classes, champs, méthodes et variables.

Un identificateur seul représente forcément une variable, on modifie donc la règle (Ident) vue précédemment pour chercher cet identificateur dans l'environnement des variables.

$$(\text{Ident})\frac{a \mapsto \texttt{Var}(T) \in \Gamma_v}{\Gamma_c; \Gamma_v \vdash a : T}$$

La sélection de champ

La règle permettant de typer une expression représentant la sélection d'un champ sur un objet est la suivante :

$$\text{(Select)}\frac{\begin{array}{c}\Gamma_c;\Gamma_v \vdash t : b \\ b \mapsto \texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m) \in \Gamma_c \\ a \mapsto \texttt{Field}(T) \in \Gamma_f\end{array}}{\Gamma_c;\Gamma_v \vdash t.a : T}$$

- Le terme t doit avoir un type classe b.
- Le nom b doit être associé avec le symbole $\texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m)$ dans l'environnement de classes Γ_c.
- L'environnement de champs Γ_f de ce symbole doit contenir une liaison pour le nom a.

L'appel de méthode

L'appel de méthode est traité comme la sélection de champ :

$$\text{(Call)}\frac{\begin{array}{c}\Gamma_c;\Gamma_v \vdash t : b \quad b \mapsto \texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m) \in \Gamma_c \\ a \mapsto \texttt{Meth}(\overline{T}|T) \in \Gamma_m \\ \Gamma_c;\Gamma_v \vdash \overline{t} : \overline{U} \quad \Gamma_c \vdash \overline{U} <: \overline{T}\end{array}}{\Gamma_c;\Gamma_v \vdash t.a(\overline{t}) : T}$$

Il y a deux contraintes supplémentaires :

- les arguments $\overline{t}$ doivent être bien typés de types $\overline{U}$.
- les types $\overline{U}$ doivent être sous-type des types $\overline{T}$ attendus par la méthode. La notation $\Gamma_c \vdash \overline{U} <: \overline{T}$ représente une séquence de contraintes de sous-typage $(\Gamma_c \vdash U_i <: T_i)_i$. Cette notation n'est bien définie que si les deux séquences de types $\overline{U}$ et $\overline{T}$ ont la même longueur.

Bonne formation des énoncés

- On veut maintenant spécifier l'analyse des énoncés (*statements* en anglais).
- Contrairement aux expressions, les énoncés n'ont pas de type car ils agissent par effet de bord et ne représentent pas une valeur. Par contre, ils doivent être bien formés (*well formed* en anglais).
- Par ailleurs un énoncé peut introduire un nouvel identificateur. C'est le cas de ($\texttt{var}\ a : T = t$) qui déclare une nouvelle variable locale.
- Les règles de bonne formation d'un énoncé ont donc la forme ($\Gamma_c; \Gamma_v \vdash S \Rightarrow \Gamma'_v$) où Γ'_v est égal à l'environnement Γ_v éventuellement enrichi d'une liaison introduite par l'énoncé S.

La déclaration de variable locale

- Voyons comment on modélise l'ajout d'un identificateur dans l'environnement lors du typage d'une déclaration de variable locale.

$$(\text{Var})\ \frac{\begin{array}{c}\Gamma_c \vdash T \diamond \\ \Gamma_c; \Gamma_v \vdash t : U \quad \Gamma_c \vdash U <: T \\ \Gamma'_v = \Gamma_v\{a \mapsto \texttt{Var}(T)\}\end{array}}{\Gamma_c; \Gamma_v \vdash \texttt{var}\ a : T = t \Rightarrow \Gamma'_v}$$

- La notation $\Gamma_c \vdash T \diamond$ spécifie que le *type* T doit être bien formé (voir plus loin).
- La notation $\Gamma_v\{a \mapsto \texttt{Var}(T)\}$ représente l'environnement Γ_v étendu par la liaison $a \mapsto \texttt{Var}(T)$. Cette notation n'est définie que si l'identificateur a n'est pas déjà lié dans Γ_v.

Typage des blocs

- On revient au typage des expressions. Un bloc est constitué en partie d'énoncés, son typage utilise donc la bonne formation des énoncés vue précédemment.

$$\text{(Block)}\frac{\begin{array}{c} n = |\overline{S}| \quad \Gamma_0 = \Gamma_v \\ \forall\, i \in [0, n-1],\ \Gamma_c; \Gamma_i \vdash S_i \Rightarrow \Gamma_{i+1} \\ \Gamma_c; \Gamma_n \vdash t : T \end{array}}{\Gamma_c; \Gamma_v \vdash \{\overline{S}\ t\} : T}$$

- Le typage d'un bloc nécessite de définir une suite d'environnements $\Gamma_0, \ldots, \Gamma_n$ telle que chaque environnement Γ_i dans cette liste est étendu par l'énoncé S_i pour donner Γ_{i+1}. Pour abréger, on note aussi : $\Gamma_c; \Gamma_v \vdash \overline{S} \Rightarrow \Gamma'_v$.

Types bien formés

- On a déjà vu comment la règle (Ident) interdit d'utiliser dans le programme un identificateur qui n'a pas été préalablement déclaré.
- Il y a une règle similaire pour les identificateurs de type, appelée règle de bonne formation (pour les types) :

$$(\text{ClassType})\frac{a \mapsto \texttt{Class}(\overline{a}|\Gamma_f|\Gamma_m) \in \Gamma_c}{\Gamma_c \vdash a \diamond}$$

- Un identificateur a ne peut donc être utilisé en tant que type que s'il appartient à l'environnement de classes Γ_c.

Sous-typage entre types classes

Quand on spécialise une classe A en définissant une sous-classe B, on aimerait pouvoir utiliser une instance de B partout où une instance de A est attendue.

Pour parvenir à ce résultat, on définit une relation de sous-typage entre types classes.

$$(\text{SubClass})\frac{a \mapsto \texttt{Class}(\overline{a}|\Gamma_f|\Gamma_m) \in \Gamma_c}{\Gamma_c \vdash a <: a_i}$$

Sous-typage avec le type `Null`

Il est impératif que l'expression `null` puisse être utilisée partout où une expression d'un type objet est attendue.

Exemple :

Pour déclarer une variable de type `List` dont on ne connait pas encore la valeur, on veut pouvoir écrire `List x = null ;`

Plutôt que de dire que `null` peut avoir n'importe quel type classe, on a choisi de lui donner le type `Null` et de faire de `Null` un sous-type de tous les autres types classes :

$$(\text{SubNull})\frac{}{\Gamma_c \vdash \texttt{Null} <: a}$$

Borne supérieure de 2 types

Le type de la valeur effectivement retournée par une expression conditionnelle `if` (t') t `else` u dépend de la valeur de la condition.

- Cette valeur étant inconnue à la compilation on peut juste approximer le type du `if` en disant que c'est un sous-type des types de chaque branche (appelons-les T et U).
- Pour garder un maximum de précision, on veut aussi que ce soit le plus petit type qui satisfait cette condition. Un tel type s'appelle la borne supérieure (*least upper bound* en anglais) des types T et U. On la note $\text{lub}_{\Gamma_c}(T, U)$.

Exemple :

$$\text{lub}_{\Gamma_c}(\texttt{Null}, \texttt{List}) = \texttt{List}$$

Exemple de dérivation de typage

Montrons que :

$$\epsilon;\ \epsilon \vdash \{\texttt{var}\ x : \texttt{Int} = 1;\ x + 2\} : \texttt{Int}$$

Pour ce faire, nous aurons besoin d'un axiome supplémentaire concernant le sous-typage :

$$(\text{IntRefl})\frac{}{\Gamma_c \vdash \texttt{Int} <: \texttt{Int}}$$

et d'un axiome concernant la bonne formation des types :

$$(\text{IntType})\frac{}{\Gamma_c \vdash \texttt{Int}\ \diamond}$$

Redéfinition de méthode (*overriding* en anglais)

Contrairement aux champs, les méthodes peuvent être définies plusieurs fois dans une même hiérarchie de classes.

A l'exécution, pour déterminer quelle méthode est appelée, on part de la classe dont l'objet est une instance et on remonte dans les super-classes jusqu'à trouver une définition de la méthode.

A la compilation, une définition de méthode m n'est acceptée que :

- s'il n'existe pas déjà dans l'environnement de méthodes Γ_m une méthode avec le même nom, ou
- s'il existe une méthode avec le même nom, mais la nouvelle méthode a une signature "plus précise", c.-à-d. son type de retour est plus petit (covariance) et les types de ses arguments plus grands (contravariance).

Résumé des relations de typage

- Nous n'avons vu qu'une partie des 33 règles d'inférence qui définissent formellement le typage de ZWEI.
- Cependant les autres règles ne font pas intervenir de notations supplémentaires et devraient donc être accessibles.
- Résumé des relations nécessaires au typage de ZWEI :

typage des termes	$\Gamma_c; \Gamma_v \vdash t : T$
bonne formation des types	$\Gamma_c \vdash T \diamond$
sous-typage	$\Gamma_c \vdash T <: T'$
bonne formation des énoncés	$\Gamma_c; \Gamma_v \vdash S \Rightarrow \Gamma'_v$
bonne formation des membres	$\Gamma_c, a \vdash d \Rightarrow \Gamma'_c$
bonne formation des classes	$\Gamma_c \vdash D \Rightarrow \Gamma'_c$
bonne formation des programmes	$P \diamond$

Correspondances

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

$\mathtt{Var}(T)$	$\approx$	`case class VarSymbol(pos: Int, name: String, vartype: Type)`
environnement Γ_v	$\approx$	`varScope: Map[String,VarSymbol]`
$\Gamma_c; \Gamma_v \vdash t : T$	$\approx$	`val T = analyzeExpr(varScope, t);`
$\Gamma_c \vdash T_1 <: T_2$	$\approx$	`T1.isSubtype(T2)`
$a \mapsto \mathtt{Var}(T) \in \Gamma_v$	$\approx$	`varScope.get(a.name) match { case Some(v) => a.sym = v case None => Report.error(...)}`
$\Gamma'_v = \Gamma_v\{a \mapsto \mathtt{Var}(T)\}$	$\approx$	`a.sym = VarSymbol(pos, a.name, T); varScope1 = varScope + a.name -> a.sym;`

Pour montrer comment traduire une règle en code, prenons la règle pour l'affectation :

$$(\text{Set})\frac{a \mapsto \texttt{Var}(T) \in \Gamma_v \qquad \Gamma_c; \Gamma_v \vdash t : U \qquad \Gamma_c \vdash U <: T}{\Gamma_c; \Gamma_v \vdash \texttt{set}\ a = t \Rightarrow \Gamma_v}$$

```
def analyzeStat(varScope: VarScope, tree: Stat): VarScope =
  tree match {
    case Set(ident, expr) =>
      varScope.get(ident.name) match {
        case Some(v) =>
          ident.sym = v;
          if (!analyzeExpr(varScope, expr).isSubtype(v.vartype))
            Report.error(tree.pos, "Incompatible types");
        case None =>
          Report.error(tree.pos, "Unknown variable " + ident.name);
      }
      varScope
    ...
}
```

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 passé en revue les différentes relations de typage nécessaires pour analyser un langage comme ZWEI. On a expliqué en détails certaines règles.
- On a montré comment passer de la spécification formelle à l'implémentation.