Drei
syntaxe abstraite et règles de typage

Martin Odersky

11 décembre 2006
version 1.1

Notations

Notation	Interprétation
$\overline{a}$	séquence $a_1, \ldots, a_n$ pour $n \in \mathbb{N}$
ϵ	séquence vide
$\lvert\overline{a}\rvert$	longueur de la séquence $\overline{a}$
$\overline{a}, \overline{b}$	concaténation des séquences $\overline{a}$ et $\overline{b}$
$\overline{a} \mapsto \overline{\sigma}$	$a_1 \mapsto \sigma_1, \ldots, a_n \mapsto \sigma_n$
$dom(\overline{a} \mapsto \overline{\sigma})$	$\overline{a}$
$\Gamma_c; \Gamma_v \vdash \overline{t} : \overline{T}$	$\Gamma_c; \Gamma_v \vdash t_1 : T_1, \ldots, \Gamma_c; \Gamma_v \vdash t_n : T_n$

Notation	Interprétation
$\Gamma \vdash \overline{X} \Rightarrow \Gamma'$	Γ_n pour $\begin{cases} \Gamma \vdash X_1 \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma_{n-1} \vdash X_n \Rightarrow \Gamma_n \end{cases}$

Notation	Interprétation
$\Gamma + (a \mapsto \sigma)$	$\begin{cases} \Gamma, a \mapsto \sigma & \text{si } a \notin dom(\Gamma) \\ \Gamma', a \mapsto \sigma, \Gamma'' & \text{si } \Gamma = \Gamma', a \mapsto \sigma', \Gamma'' \end{cases}$
$\Gamma \uplus \Gamma'$	Γ, Γ' si $dom(\Gamma) \cap dom(\Gamma') = \epsilon$

Notation	Interprétation
$\mathit{fields}(\overline{d})$	$\biguplus_{\texttt{val}\ a:T\ \in\ \overline{d}} (a \mapsto \texttt{Field}(T))$
$\mathit{methods}(\overline{d})$	$\biguplus_{\texttt{def}\ a(\overline{a}:\overline{T}):T=t\ \in\ \overline{d}} (a \mapsto \texttt{Meth}(\overline{T} \mid T))$
$\mathit{params}(\overline{a}, \overline{T})$	$+_{a,T\ \in\ (\overline{a},\overline{T})} (a \mapsto \texttt{Var}(T))$

Grammaire abstraite

nom	a, b			
programmes	P	$::=$	$\overline{D}\ S$	
classes	D	$::=$	`class` a `extends` s $\{\overline{d}\}$	déclaration de classe
	s	$::=$	a | `none`	super classe
membres	d	$::=$	`val` $a : T$	déclaration de champ
		|	`def` $a(\overline{a} : \overline{T}) : T = t$	déclaration de méthode
types	T, U	$::=$	a	type de classe
		|	`Int`	type entier
		|	`None`	type indéterminé

expressions	t, u	$::=$	a	variable
		$\mid$	$\texttt{new}\ a(\overline{t})$	création d'instance
		$\mid$	$t.a$	sélection de champ
		$\mid$	$t.a(\overline{t})$	appel de méthode
		$\mid$	n	nombre entier
		$\mid$	$\mathit{unop}\ t$	opération unaire
		$\mid$	$t\ \mathit{binop}\ t'$	opération binaire
		$\mid$	`readInt`	lecture d'entier
		$\mid$	`readChar`	lecture de caractère
		$\mid$	$\{\ \overline{S}\ t\ \}$	block
		$\mid$	`empty`	

énoncés	S	$::=$	`while` t S	exécution en boucle
		$\mid$	`if` t `then` S `else` S'	exécution conditionelle
		$\mid$	`var` $a : T = t$	déclaration de variable
		$\mid$	`set` $a = t$	définition de variable
		$\mid$	`do` t	instruction
		$\mid$	`printInt`(t)	impression d'entier
		$\mid$	`printChar`(t)	impression de caractère
		$\mid$	$\{ \overline{S} \}$	énoncé composite
op. unaires	*unop*	$::=$	$- \mid !$	
op. binaires	*binop*	$::=$	$+ \mid - \mid * \mid / \mid \%$	
		$\mid$	$= \mid \neq \mid < \mid \leq \mid \geq \mid >$	
		$\mid$	$\wedge$	

Symboles

classe	σ_c	$::=$	$\texttt{Class}(\overline{a}\|\Gamma_f\|\Gamma_m)$ $\overline{a}$: parents, Γ_f : champs, Γ_m : méthodes
champ	σ_f	$::=$	$\texttt{Field}(T)$ T : type du champ
méthode	σ_m	$::=$	$\texttt{Meth}(\overline{T}\|T)$ $\overline{T}$: types des paramètres, T : type de retour
variable	σ_v	$::=$	$\texttt{Var}(T)$ T : type de la variable

Portées

$$
\begin{array}{llll}
\text{classes} & \Gamma_c & ::= & \overline{a} \mapsto \overline{\sigma_c} \\
\text{champs} & \Gamma_f & ::= & \overline{a} \mapsto \overline{\sigma_f} \\
\text{méthodes} & \Gamma_m & ::= & \overline{a} \mapsto \overline{\sigma_m} \\
\text{variables} & \Gamma_v & ::= & \overline{a} \mapsto \overline{\sigma_v}
\end{array}
$$

Règles de typage

$$\frac{\textsc{Program} \quad \texttt{none} \mapsto \texttt{Class}(\epsilon|\epsilon|\epsilon) \vdash \overline{D} \;\Rightarrow\; \Gamma_c \qquad \Gamma_c \vdash \overline{D} \diamond \qquad \Gamma_c;\, \epsilon \vdash S \;\Rightarrow\; \epsilon}{\overline{D}\ S \diamond}$$

- $P \diamond$ means : Program P is well-formed.
- A program $\overline{D}\ S$ is well formed if its declarations $\overline{D}\ S$ generate a class environment Γ_c and its statement S is well-formed in this environment
- The initial environment $\texttt{none} \mapsto \texttt{Class}(\epsilon|\epsilon|\epsilon)$ maps the "undefined" superclass `none` to a symbol denoting an empty class without ancestors.

CLASS1

$$\frac{s \mapsto \texttt{Class}(\overline{a}|\Gamma_f|\Gamma_m) \in \Gamma_c \qquad \Gamma'_f = \Gamma_f \uplus \mathit{fields}(\overline{d}) \qquad \Gamma'_m = \Gamma_m + \mathit{methods}(\overline{d}) \qquad \Gamma'_c = \Gamma_c \uplus (a \mapsto \texttt{Class}(a, \overline{a}|\Gamma'_f|\Gamma'_m))}{\Gamma_c \vdash \texttt{class}\ a\ \texttt{extends}\ s\ \{\overline{d}\} \ \Rightarrow\ \Gamma'_c}$$

- $\Gamma_c \vdash D \ \Rightarrow\ \Gamma'_c$ means : In class environment Γ_c the class declaration D is well-formed, and leads to a new class environment Γ'_c
- Preconditions :
  - The superclass s must be defined in Γ_c.
  - All class member definitions are added to the scopes, but without checking them.

$$\textsc{Class2}\quad \frac{\Gamma_c;\, a \vdash \overline{d} \diamond}{\Gamma_c \vdash \texttt{class}\ a\ \texttt{extends}\ s\ \{\overline{d}\} \diamond}$$

- $\Gamma_c \vdash D \diamond$ means : In class environment Γ_c the class declaration D is well-formed.
- Both this and Class1 must be valid for a particular class to be valid.
- Preconditions :
  - All class member definitions $\overline{d}$ must be well-formed.

$$\textsc{Field}\quad \frac{\Gamma_c \vdash T \diamond}{\Gamma_c;\, b \vdash \texttt{val}\ a : T \diamond}$$

- $\Gamma_c;\, b \vdash d \diamond$ means : Given a class environment Γ_c, definition d is well-formed as a member of class b.
- A field definition is well-formed if its type is well-formed.
- The corresponding rule for methods is much more complicated...

Method

$$\frac{\begin{array}{c}\Gamma_c \vdash T \diamond \qquad \Gamma_c \vdash \overline{T} \diamond \qquad b \mapsto \texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m) \in \Gamma_c \\ \forall c \in \overline{b}\ .\ c \mapsto \texttt{Class}(\overline{c}|\Gamma'_f|\Gamma'_m) \in \Gamma_c \wedge a \mapsto \texttt{Meth}(\overline{U}|U) \in \Gamma'_m \implies \left\{\begin{array}{l}\Gamma_c \vdash \overline{U} <: \overline{T} \\ \Gamma_c \vdash T <: U\end{array}\right. \\ \Gamma_v = params((this, \overline{a}), (b, \overline{T})) \qquad \Gamma_c; \Gamma_v \vdash t : T' \qquad \Gamma_c \vdash T' <: T\end{array}}{\Gamma_c;\ b \vdash \texttt{def}\ a(\overline{a} : \overline{T}) : T = t \diamond}$$

Preconditions :

- Parameter types and result type are well-formed.
- If the method overrides a method a in a base class :
  - Its result type must be a subtype of a's result type.
  - Its parameter types must be supertypes of a's parameter types.
- The body t of the method must have a type which is a subtype of the declared result type.
- The body t is typed in a variable environment Γ_v where
  - All method parameters are bound to their types.
  - The special name *this* is bound to the current class b.

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

$$\textsc{IntType}\quad \Gamma_c \vdash \texttt{Int} \diamond$$

$$\textsc{NoType}\quad \Gamma_c \vdash \texttt{None} \diamond$$

- $\Gamma_c \vdash T \diamond$ means : Type T is well-formed in class environment Γ_c.
- Classes are well-formed if their name is bound in the class environment.
- Types `Int` and `None` are always well-formed.

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

$$\Gamma_c \vdash \texttt{Int} <: \texttt{Int} \; (\textsc{IntRefl})$$

$$\Gamma_c \vdash \texttt{None} <: \texttt{None} \; (\textsc{NoneRefl})$$

- $\Gamma_c \vdash T <: U$ means : Type T is a subtype of type U, assuming a class environment Γ_c.
- A class type is a subtype of all its ancestor classes.
- Note that a class is its own ancestor, as seen from rule Class1.
- Types `Int` and `None` are only subtypes of themselves.

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

$$\textsc{Select}\quad \frac{\Gamma_c; \Gamma_v \vdash t : b \qquad b \mapsto \texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m) \in \Gamma_c \qquad a \mapsto \texttt{Field}(T) \in \Gamma_f}{\Gamma_c; \Gamma_v \vdash t.a : T}$$

- The rule for identifiers demands that every identifier is defined.
- Identifier bindings are found in the value environment Γ_v.
- In a selection $t.a$, the term t must have a class type, b.
- The symbol for a is then found in the field environment of class b.

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

- The rule [Call] is similar to [Select].
- New precondition : The types of function arguments must match (i.e. be subtypes of) the types of formal parameters.
- Implicitly, this demands that the numbers of arguments and formals are the same.

NEW

$$\frac{\Gamma_f = \overline{a \mapsto \texttt{Field}(T)} \quad \text{where } a \mapsto \texttt{Class}(\overline{b}|\Gamma_f|\Gamma_m) \in \Gamma_c \qquad \Gamma_c;\Gamma_v \vdash \overline{t} : \overline{U} \qquad \Gamma_c \vdash \overline{U} <: \overline{T}}{\Gamma_c;\Gamma_v \vdash \texttt{new}\ a(\overline{t}) : a}$$

INTLIT

$$\Gamma_c;\Gamma_v \vdash n : \texttt{Int}$$

UNOP

$$\frac{\Gamma_c;\Gamma_v \vdash t : \texttt{Int}}{\Gamma_c;\Gamma_v \vdash \mathit{unop}\ t : \texttt{Int}}$$

- The name a must refer to a class
- The arguments $\overline{t}$ must match (in number and type) the fields of class a.

$$\textsc{Binop}\quad \frac{\mathit{binop} \notin \{=, \neq\} \qquad \Gamma_c; \Gamma_v \vdash t : \texttt{Int} \qquad \Gamma_c; \Gamma_v \vdash u : \texttt{Int}}{\Gamma_c; \Gamma_v \vdash t \; \mathit{binop} \; u : \texttt{Int}}$$

$$\textsc{ObjComp}\quad \frac{\mathit{binop} \in \{=, \neq\} \qquad \Gamma_c; \Gamma_v \vdash t : T \qquad \Gamma_c; \Gamma_v \vdash u : U \qquad \Gamma_c \vdash T <: U \vee \Gamma_c \vdash U <: T}{\Gamma_c; \Gamma_v \vdash t \; \mathit{binop} \; u : \texttt{Int}}$$

- The operands t and u may have possible different types T and U.
- But one of T, U must be a subtype of the other.
- Are there other possible design choices ?

ReadInt

$$\Gamma_c; \Gamma_v \vdash \texttt{readInt} : \texttt{Int}$$

ReadChar

$$\Gamma_c; \Gamma_v \vdash \texttt{readChar} : \texttt{Int}$$

Block

$$\frac{\Gamma_c; \Gamma_v \vdash \overline{S} \;\Rightarrow\; \Gamma'_v \qquad \Gamma_c; \Gamma'_v \vdash t : T}{\Gamma_c; \Gamma_v \vdash \{\; \overline{S}\; t \;\} : T}$$

Empty

$$\Gamma_c; \Gamma_v \vdash \texttt{empty} : \texttt{None}$$

- The statements $\overline{S}$ must be well-formed, producing a value environment Γ'_v. See slide 3 for notation !
- Γ'_v contains Γ_v and adds bindings resulting from definitions in $\overline{S}$.
- The final expression t is then typed with Γ'_v as environment.
- The type of t is also the type of the block.

IF

$$\frac{\Gamma_c; \Gamma_v \vdash t : \texttt{Int} \qquad \Gamma_c; \Gamma_v \vdash S \;\Rightarrow\; \Gamma_v \qquad \Gamma_c; \Gamma_v \vdash S' \;\Rightarrow\; \Gamma_v}{\Gamma_c; \Gamma_v \vdash \texttt{if}\ t\ \texttt{then}\ S\ \texttt{else}\ S' \;\Rightarrow\; \Gamma_v}$$

- $\Gamma_c; \Gamma_v \vdash S \;\Rightarrow\; \Gamma'_v$ means : In class environment Γ_c and value environment Γ_v the statement S is well-formed and it leads to a new value environment Γ'_v.
- Γ'_v is the same as Γ_v except if the statement S is a variable declaration : in that case, Γ'_v augments Γ_v with a binding for the declared variable.
- In an if-statement `if` t `then` S `else` S' :
  - The condition t must be of type `Int`.
  - The branches S, S' must both be well-formed statements.

$$\textsc{While}\quad \frac{\Gamma_c; \Gamma_v \vdash t : \texttt{Int} \qquad \Gamma_c; \Gamma_v \vdash S \;\Rightarrow\; \Gamma_v}{\Gamma_c; \Gamma_v \vdash \texttt{while}\; t\; S \;\Rightarrow\; \Gamma_v}$$

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

In a variable declaration $\texttt{var}\; a : T = t$:

- The declared variable type T must be well formed.
- The initial expression t must have a type which is a subtype of T.
- The variable declaration then produces a new environment which adds the binding $a \mapsto \texttt{Var}(T)$ to Γ_v.

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

$$\textsc{Do}\quad \frac{\Gamma_c; \Gamma_v \vdash t : T}{\Gamma_c; \Gamma_v \vdash \mathtt{do}\ t \ \Rightarrow\ \Gamma_v}$$

$$\frac{\Gamma_c; \Gamma_v \vdash t : \texttt{Int}}{\Gamma_c; \Gamma_v \vdash \texttt{printInt}(t) \;\Rightarrow\; \Gamma_v}\ \textsc{PrintInt}$$

$$\frac{\Gamma_c; \Gamma_v \vdash t : \texttt{Int}}{\Gamma_c; \Gamma_v \vdash \texttt{printChar}(t) \;\Rightarrow\; \Gamma_v}\ \textsc{PrintChar}$$

$$\frac{\Gamma_c; \Gamma_v \vdash \overline{S} \;\Rightarrow\; \Gamma_v'}{\Gamma_c; \Gamma_v \vdash \{\ \overline{S}\ \} \;\Rightarrow\; \Gamma_v}\ \textsc{Compound}$$

- The rule for compound statements reflects the block structure of Drei :
- After a compound statement, all definitions added to the original value environment Γ_v are forgotten.