UPMC/master/info/APS-4I503

P. Manoury

janvier 2015

1 Syntaxe

1.1 Concrète

Prog	: :=	`program` Classes Locals InstrList
Classes	: :=	ε | Class Classes
Locals	|	ε | `let` VarDecs `in`
InstrList	: :=	`begin` InstrSeq `end`
Class	: :=	`class` id Extends `is` VarDecList MethodList
Extends	: :=	ε | `extends` ClassType
VarDecList	: :=	ε | `vars` VarDecs
MethodList	: :=	ε | `methods` Methods
ClassType	: :=	`Obj` | `Void` | `Int` | `Bool` | id
VarDecs	: :=	VarDec | VarDecs VarDec
VarDec	: :=	ClassType Ids `;`
Ids	: :=	Ids `,` id
Methods	: :=	Method | Methods Method
Method	: :=	ClassType id `(` ArgDecList `)` Locals InstrList
ArgDecList	: :=	ε | ArgDecs
ArgDecs	: :=	ArgDec | ArgDecs `;` ArgDec
ArgDec	: :=	ClassType id
InstrSeq	: :=	Instr | InstrSeq `;` Instr
Instr	: :=	Exp`.`id `(` ArgList `)`
	|	id `:=` Exp
	|	Exp`.`id `:=` Exp
	|	`return` Exp
	|	`writeln` Exp
	|	`if` Exp `then` InstrList `else` InstrList
	|	`while` Exp `do` InstrList
Exp	: :=	Exp `+` Term | Exp `-` Term | Exp `or` Term
	|	Exp`.`id | Exp`.`id `(` ArgList `)` | `super.`id `(` ArgList `)`
	|	Exp `instanceof` ClassType
Term	: :=	Term `*` Fact | Term `/` Fact | Term `and` Fact | Fact
Fact	: :=	Fact `=` Basic | Fact `<` Basic | Basic
Basic	: :=	`not` Exp | num | id | `true` | `false` | `nil`
	|	`new` ClassType
	|	`(` Exp `)`
ArgList	: :=	ε | Args
Args	: :=	Exp | Args `,` Exp

1.2 Abstraite

On adopte la même convention typographique. On désigne par ITEM* ou item* une suite, possiblement vide de non-terminaux ou de terminaux. On pose op = {`not`, `and`, `or`, `add`, `sub`, `mul`, `div`, `eq`, `lt`}. C'est un ensemble de terminaux.

PROGRAM	: :=	`program` CLASS* VARDEC* INSTR*
CLASS	: :=	`class` id CLASSTYPE VARDEC* METHOD*
CLASSTYPE	: :=	`Obj` | `Void` | `Int` | `Bool` | id
VARDEC	: :=	CLASSTYPE id
METHOD	: :=	`method` id VARDEC* CLASSTYPE VARDEC* INSTR*
INSTR	: :=	`call` EXP id EXP*
	|	`setvar` id EXP
	|	`setfield` EXP id EXP
	|	`return` EXP
	|	`write` EXP
	|	`if` EXP INSTR* INSTR*
	|	`while` EXP INST*
EXP	: :=	`nil` | `true` | `false`
	|	num | id
	|	op EXP*
	|	`getfield` EXP id | `call` EXP id EXP*
	|	`new` CLASSTYPE | `instanceof` CLASSTYPE EXP

Nota : les déclarations «factorisées» de variables sont développées (`Int x,y;` devient `Int x; Int y;`)

Simplification du nombre de non-terminaux

2 Typage

2.1 Les types

Les *types* désignent
- les ensembles des valeurs calculables par un programme.
  Ces valeurs sont
  - *prédefinies* (`Obj`, `Void`, `Int`, `Bool`) ou
  - *définies* par le programme.
- les conditions d'*application* des opérateurs ou des méthodes de manière à ce que les *expressions* des programmes soient à même de calculer effectivement les valeurs calculables.
  Les types sont ici des *types fonctionnels* : relation entre domaine (type des paramètres) et co-domaine (type du résultat). Le domaine peut être un *type produit* (produit cartésien).

Langage de types

TYPE	: :=	CLASSTYPE | FUNTYPE
CLASSTYPE	: :=	`Obj` | `Void` | `Int` | `Bool` | id
FUNTYPE	: :=	ARGTYPE → CLASSTYPE
ARGTYPE	: :=	CLASSTYPE | ARGTYPE × CLASSTYPE

2.2 Programmes biens typés

Un programme est *bien typé* lorsqu'aucune exécution d'aucune partie d'un programme entraîne un calcul dont le résultat est hors du domaine de valeurs attendu par un opérateur ou une *instruction*.

Application d'un opérateur Pour garantir la correction de l'application d'un opérateur (fonction) on pose une *règle de typage* de l'application :

Si f est de type $A \to B$ et x est de type A alors $f(x)$ est de type B.

Notez la définition récursive : pour connaître le type de $f(x)$, il faut connaître le type de f et de x.

Syntaxiquement : pour connaître le type de l'*application d'un opérateur*, il faut connaître le type de ses composantes, fonction et arguments. Une application répond à une instance de la régle syntaxique Exp qui est définie en fonction d'autres régles syntaxiques. Par exemple : juxtatposition du terminal `not` et d'un autre Exp.

En généralisant, pour connaître le type d'une expression, il faut :

1. connaître le type de ses sous-expressions.
2. respecter la règle de typage de la composition des expressions.

On *fonde* la définition récursive en posant, comme prédéfinie, les types des atomes pour les constantes et les opérateurs prédéfinis :

`nil`	est de type	`Void`
`true`	est de type	`Bool`
`false`	est de type	`Bool`
`not`	est de type	`Bool`$\to$`Bool`
`and`	est de type	`Bool`$\times$`Bool`$\to$`Bool`
`or`	est de type	`Bool`$\times$`Bool`$\to$`Bool`
`add`	est de type	`Int`$\times$`Int`$\to$`Int`
`sub`	est de type	`Int`$\times$`Int`$\to$`Int`
`mul`	est de type	`Int`$\times$`Int`$\to$`Int`
`div`	est de type	`Int`$\times$`Int`$\to$`Int`
`eq`	est de type	`Int`$\times$`Int`$\to$`Bool`
`lt`	est de type	`Int`$\times$`Int`$\to$`Bool`

Pour les constantes entières, on pose un *schéma de règle* :

si $n \in$ `num` alors n est de type `Int`

Notez que le critère $n \in$ `num`, où `num` est l'ensemble des mots définis par une expression régulière, est purement lexical (syntaxe).

Notation : l'expression «*truc* est de type *machin*» de cette manière

$$truc : machin$$

C'est une *assignation de type*.

Un ensemble d'assignations de types définit un *contexte de typage*. Un expression est correctement typée vis-à-vis d'un contexte de typage. Une règle de typage pour une expression est donc une relation entre un contexte de typage, une expression et son type.

Un *jugement de typage* est une proposition de la forme «dans le contexte *bidule*, *truc* a le type *machin*». Les règles de typage sont là pour garantir la *validité* des jugements de typage.

Lorsque *truc* est un atome, la seule manière d'établir la validité du jugement «dans le contexte *bidule*, *truc* a le type *machin*», c'est que l'assignation «*truc* a le type *machin*» soit un élément du contexte *bibule*.

Si l'on admet dans le contexte des assignations de la forme «x est de type *machin*» où $x \in$ `id` (lexique) alors on peut juger du type d'une expression contenant des symboles de variables.

Formalisation Si a est un atome et τ un type, on écrit l'assignation de type «a est de type τ» de cette manière :

$$a : \tau$$

On désigne par Γ un contexte de typage, on désigne par e une expression et par τ un type. On écrit le jugement de typage «dans le contexte Γ, e est de type τ» de cette manière :

$$\Gamma \vdash e : \tau$$

Pour signifier que l'assignation $a : \tau$ est un élément d'un contexte de typage ; on écrit

$$[\Gamma; a : \tau]$$

Cette écriture permet de formuler la règle de typage des atomes non numériques :

$$\frac{}{[\Gamma; a : \tau] \vdash a : \tau}$$

Cette règle permet de typer les expressions atomiques définie par les *règles syntaxiques*

$$\textsc{Exp} \quad ::= \quad \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid \texttt{num} \mid \mathsf{id}$$

Le trait horizontal surmontant le jugement de typage signifie qu'il n'y a pas d'autre manière d'obtenir un tel jugement (cas de base de la définition récursive du typage). On parle de *règle axiome*.

On désigne par es une suite d'expressions $e_1 \dots e_n$. On pose que o un élément de op. On désigne par τs une suite non vide de types $\tau_1 \dots \tau_n$ (abréviation du produit cartésien $\tau_1 \times \dots \times \tau_n$, ou simplement τ_1 si τs se réduit au seul τ_1). On définit la règle de typage de l'application d'un opérateur de la manière suivante :

$$\frac{\Gamma \vdash o : \tau s \to \tau \quad \Gamma \vdash es : \tau s}{\Gamma \vdash (o\ es) : \tau}$$

où $\Gamma \vdash es : \tau s$ est une abréviation pour la suite des jugements $\Gamma \vdash e_1 : \tau_1 \dots \Gamma \vdash e_n : \tau_n$.

Le trait horizontal se lit ici de deux manières :

- descendante : si $\Gamma \vdash o : \tau s \to \tau$ et si $\Gamma \vdash es : \tau s$ alors $\Gamma \vdash (o\ es) : \tau$
- ascendente : pour (vérifier) que $\Gamma \vdash (o\ es) : \tau$, il faut (vérifier) que $\Gamma \vdash o : \tau s \to \tau$ et que $\Gamma \vdash es : \tau s$

La lecture ascendente de la règle permet d'obtenir un algorithme récursif de vérification de type d'une expression *dirigé par la syntaxe* (parcours de l'arbre syntaxique de l'expression).

Les traits objets dans les expressions Une expression peut désigner une instance de classe. Règle syntaxique

$$\textsc{Expr} \quad ::= \texttt{new}\ \textsc{ClassType}$$

Si $c \in \textsc{ClassType}$, on pose la règle

$$\frac{}{\Gamma \vdash (\texttt{new}\ c) : c}$$

Un champ d'instance de classe est un identificateur. Son typage est analogue à celui des atomes : sont type doit figurer dans le contexte de typage. Un champ d'instance x est «relatif» à une classe c. On enrichi les contexte de typage de manière à pouvoir contenir des assignations relatives à une classe. On écrit de telles assignations

$$c.x : \tau$$

Cette nouveauté permet également d'assigner les types des méthodes. On pose la nouvelle règle axiome :

$$\frac{}{[\Gamma; c.x : \tau] \vdash c.x : \tau}$$

L'appel de méthode et l'accès à un champ d'instance sont définis par les règles syntaxiques

$$\textsc{Exp} : := \texttt{getfield}\ \textsc{Exp}\ \mathsf{id} \mid \texttt{call}\ \textsc{Exp}\ \mathsf{id}\ \textsc{Exp}^*$$

Soit $e \in \textsc{Exp}$, $x \in \mathsf{id}$ et $es \in \textsc{Exp}*$, on pose

$$\frac{\Gamma \vdash e : c \qquad \Gamma \vdash c.x : \tau}{\Gamma \vdash (\texttt{getfield}\ e\ x) : \tau} \qquad \frac{\Gamma \vdash e : c \qquad \Gamma \vdash c.m : \tau s \to \tau \qquad \Gamma \vdash es : \tau s}{\Gamma \vdash (\texttt{call}\ e\ m\ es) : \tau}$$

Pour compléter les traits objet, pour la règle syntaxique

$$\textsc{Exp} \quad ::= \quad \texttt{instanceof}\ \textsc{Exp}\ \textsc{ClassType}$$

on a également l'axiome :

$$\frac{}{\Gamma\Delta \vdash (\texttt{instanceof}\ c\ e) : \texttt{Bool}}$$

La méthode s'étend aux *instructions*. Pour chacune d'elle on donne les conditions d'application qui permettent de leur attribuer un type en fonction du type des leurs composants.

Syntaxe	Typage
`setvar` id Exp	$\dfrac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{setvar}\ x\ e) : \texttt{Void}}$
`setfield` Exp id Exp	$\dfrac{\Gamma \vdash e_1 : c \qquad \Gamma \vdash c.x : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{setfield}\ e_1\ x\ e_2) : \texttt{Void}}$
`write` Exp	$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{write}\ e) : \texttt{Void}}$
`if` Exp Instr* Instr*	$\dfrac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma \vdash is_1 : \tau \qquad \Gamma \vdash is_2 : \tau}{\Gamma \vdash (\texttt{if}\ e\ is_1\ is_2) : \tau}$
`while` Exp Inst*	$\dfrac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma \vdash is : \tau}{\Gamma \vdash (\texttt{while}\ e\ is) : \tau}$

Un appel de méthode peut figurer comme instruction. Sa règle de typage n'est pas différente de celle posée pour son usage comme expression. Reste l'instruction `return` qui a un rôle particulier dans le typage des suites d'instructions. Pour celles-ci, on pose :

$$\frac{}{\Gamma \vdash [] : \texttt{Void}} \quad \frac{\Gamma\Delta \vdash i : \tau}{\Gamma\Delta \vdash [i] : \tau} \quad \frac{\Gamma\Delta \vdash e : \tau}{\Gamma\Delta \vdash [(\texttt{return}\ e); is] : \tau} \quad \frac{\Gamma\Delta \vdash i : \texttt{Void} \qquad \Gamma\Delta \vdash is : \tau}{\Gamma\Delta \vdash [i; is] : \tau}\ (a)$$

$(a)\ i \not\equiv \texttt{return}$

Dans la dernière régle, on pourrait choisir de ne pas exiger $i : \texttt{Void}$. C'est une affaire de goût.

Les déclarations de variables locales produisent un contexte de typage pour type le corps des programmes. Il en est de même des déclarations de classe. Dans le typage d'une classe, les déclarations de variables d'instances et de méthodes produisent également un contexte. Enfin, dans les déclarations de méthodes, la déclaration de paramètres de la méthode produit également un contexte pour le typage du corps de la méthode.

Si vs est une suite de déclarations de variables (ou de paramètres) on aura des jugements de typage de la forme

$$\vdash vs : \Gamma'$$

Ils représentent simplement la transcription d'une suite de déclarations en contexte. On pose les règles

$$\frac{}{\vdash [] : []} \quad \frac{\vdash vs : \Gamma}{\vdash [(c\ x); vs] : [x : c; \Gamma]}$$

Si Γ est le contexte $[x_1 : \tau_1; \ldots; x_n : \tau_n]$, $c.\Gamma$ est une abréviation pour le contexte $[c.x_1 : \tau_1; \ldots; c.x_n : \tau_n]$.

Avec les classes, il faut tenir compte des deux mots réservés `self` et `super`. On pose les deux axiomes :

$$\frac{}{[\Gamma; \texttt{self} : \tau]\Delta \vdash \texttt{self} : \tau} \quad \frac{}{[\Gamma; \texttt{super} : \tau]\Delta \vdash \texttt{super} : \tau}$$

C'est le typage de chaque classe qui introduira les assignations pour `self` et `super` dans les contextes. Il ne peuvent avoir d'occurrence que dans la définition du corps d'une méthode dans une déclaration de classe.

Une déclaration de méthode produit une assignation de type (relative à une classe). Elles correspondent à des jugements de typage de la forme

$$\Gamma \vdash mt : [c.m : \tau]$$

. On pose la règle

$$\frac{\Gamma \vdash \texttt{self} : c \qquad \vdash vs : [xs : \tau s] \qquad \vdash vs' : \Gamma' \qquad [\Gamma; c.m : \tau s \to \tau; xs : \tau s; \Gamma'] \vdash is : \tau}{\Gamma \vdash (\texttt{method}\ m\ vs\ \tau\ vs'\ is) : [c.m : \tau s \to \tau]}\ (a)$$

Une suite de déclarations de méthodes construit simplement un contexte, à l'instar des déclarations de variables :

$$\frac{\Gamma \vdash mt : [c.m : \tau] \qquad [\Gamma; c.m : \tau] \vdash mts : \Gamma'}{\Gamma \vdash [mt; mts] : [c.m : \tau; \Gamma']}$$

Une déclaration de classe produit un contexte :

$$\frac{\vdash vs : \Gamma' \qquad [\Gamma; \texttt{self} : c; \texttt{super} : c'; c.\Gamma'] \vdash mts : \Gamma''}{\Gamma \vdash (\texttt{class}\ c\ c'\ vs\ mts) : [c.\Gamma'; \Gamma'']}$$

Idem pour les suites de déclarations de classes

$$\frac{\Gamma \vdash cl : \Gamma' \qquad \Gamma; \Gamma' \vdash cls : \Gamma''}{\Gamma \vdash [cl; cls] : [\Gamma; \Gamma'; \Gamma'']}$$

Enfin, le type d'un programme est vérifié dans un contexte Γ_0 qui donne les assignations de types des constantes et opérateurs prédéfinis.

$$\frac{\Gamma \vdash [cls] : \Gamma \qquad \vdash vs : \Gamma' \qquad [\Gamma; \Gamma'] \vdash is : \tau}{\Gamma \vdash (\texttt{program}\ cls\ vs\ is) : \tau}$$

Héritage et sous-typage La relation d'héritage induit un notion de *sous-typage* : une classe fille `c` est sous-type de sa mère `c'`. On note $\texttt{c} \subseteq \texttt{c'}$.

En effet, du point de vue des objets, cela signifie que là où l'on attend une instance de `c`, on peut toujours utiliser une instance d'une ses héritères. En effet, ces dernières seront toujours répondre aux appels de méthodes attendues pour leur classe mère.

Le typage des programmes est vérifié *modulo* les relations de sous-typages induites par l'héritage. On ajoute un environnement de sous-typage contenant des assertions de la forme $\texttt{c} \subseteq \texttt{c'}$. On désigne par Δ un environnement de sous-typage. On pose

$$\frac{}{\Delta, c_1 \subseteq c_2 \vdash c_1 \subseteq c_2} \qquad \frac{\Delta \vdash \tau_1 \subseteq \tau_2 \qquad \Delta \vdash \tau_2 \subseteq \tau_3}{\Delta \vdash \tau_1 \subseteq \tau_3}$$

La relation de sous typage est transitive (*clôture transitive* de la relation d'héritage).

Les règles de typage disposent à présent de deux contextes : le contexte des assignations de types et celui de la relation d'héritage/sous-typage.

Ce dernier permet en particulier de typer les champs et méthodes héritées :

$$\frac{\Gamma\Delta \vdash c'.x : \tau \qquad \Delta \vdash c \subseteq c'}{\Gamma\Delta \vdash c.x : \tau}$$

Co et Contra variances Une fonction de type $\tau_1 \to \tau_2$ peut être utilisée en place d'une fonction de type $\tau_1' \to \tau_2'$ si l'on dispose que $\tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2'$. Une fonction de type $\tau_1 \to \tau_2$ peut-être appliquée à une valeur de type τ_1' si $\tau_1' \subseteq \tau_1$. C'est la *contra-variance* de la flèche fonctionnelle. L'utilisation d'une fonction de type $\tau_1 \to \tau_2$ en place d'une fonction de type $\tau_1' \to \tau_2'$ donne un résultat de type τ_2 là où τ_2' pouvait être attendu. Il faut donc que $\tau_2 \subseteq \tau_2'$: c'est la *covariance* de la flèche fonctionnelle. On exprime cela par la règle

$$\frac{\Delta \vdash \tau_1' \subseteq \tau_1 \qquad \Delta \vdash \tau_2 \subseteq \tau_2'}{\Delta \vdash \tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2'}$$

Dans les langages à objet, la laison retardée et la possibilité de redéfinir des méthodes induit que le type de la méthode redéfini soit contra-variant de celui de la méthode d'originale.

```
class A
  method f(x:T) : S = .. end;
  method g() : S = return f(new T) end;
end.
class B
  method f(x:T') : S' = ... end;
end.
```

Soit `b = new B`. Pour que `b.g()` reste correct vis-à-vis des types, il faut que
contra-variance `(new T) : T'`. Ce que l'on peut garantir si `T` $\subseteq$ `T'` ;
co-variance `b.f(new T) : S`. Ce que l'on obtient avec `S'` $\subseteq$ `S`.
On exprime l'exigence de contra-variance et co-variance sur la règle de typage des déclarations de méthodes

$$\frac{\Gamma \vdash \texttt{self} : c \qquad \vdash vs : [xs : \tau s] \qquad \vdash vs' : \Gamma' \qquad [\Gamma; c.m : \tau s \to \tau; xs : \tau s; \Gamma']\Delta \vdash is : \tau}{\Gamma\Delta \vdash (\texttt{method}\ m\ vs\ \tau\ vs'\ is) : [c.m : \tau s \to \tau]}\ (a)$$

(a) si $\Gamma \vdash c.\texttt{super} : c'$ et $\Gamma \vdash c'.m : \tau s' \to \tau'$ alors $\Delta \vdash \tau s \to \tau \subseteq \tau s' \to \tau'$

Les déclarations de classes affecte l'environnement de sous-typage :

$$\frac{\vdash vs : \Gamma' \qquad [\Gamma; \texttt{self} : c; \texttt{super} : c'; \Gamma'][\Delta; c \subseteq c'] \vdash mts : \Gamma''}{\Gamma\Delta \vdash (\texttt{class}\ c\ c'\ vs\ mts) : ([\Gamma'; \Gamma''], [\Delta; c \subseteq c'])}$$

$$\frac{}{\Gamma\Delta \vdash [] : ([], [])} \qquad \frac{\Gamma\Delta \vdash cl : (\Gamma', \Delta') \qquad \Gamma'\Delta' \vdash [cls] : (\Gamma'', \Delta'')}{\Gamma\Delta \vdash [cl; cls] : (\Gamma'', \Delta'')}$$

Enfin, pour les programmes, on a désormais :

$$\frac{\Gamma_0\Delta_0 \vdash [cls] : (\Gamma, \Delta) \qquad \vdash vs : \Gamma' \qquad [\Gamma; \Gamma']\Delta \vdash is : \tau}{\Gamma_0\Delta_0 \vdash (\texttt{program}\ cls\ vs\ is) : \tau}$$