Projet de *T.E.P.*

Hakim BELHAOUARI & Raimana TEINA
Encadrant : Emmanuel CHAILLOUX

22 novembre 2004

Table des matières

Introduction

Le projet de *T.E.P.* demandé consiste à la réalisation d'une applet *Java* permettant de visualiser la vérification de type. Après un debut prometteur, nous avons eu envie d'évoluer le traitement en réalisant un code à la fois évolutif et permettant de traiter l'instruction `letin`. L'ajout de cette instruction nous a d'ailleurs obligé de modifier entièrement notre programme. En effet, l'ajout du `letin` nous impose par sa construction de savoir faire de l'inférence de type pour l'évaluation de *e1* et la connaissance des schémas de types pour le polymorphisme.

Le projet se décompose en deux grandes parties :

1. La programmation des types et des expressions du λ-calcul
2. La conception d'une interface graphique pour l'applet

1 Implementation

1.1 Hiérarchies de classes

1.1.1 Les termes lambda

Tous les λ-termes sont contenus dans le paquetage `lambdatypes`, tout terme est dérivé de la classe *LambdaType*. Cette classe est abstraite, mais contient des méthodes statiques, concernant la génération de symbole unique. Les fonctions principales de toutes les λ-termes sont :

- `TypeResult visite(Context c,Visiteur v)` : Cette fonction permet à un *visiteur* de réaliser un traitement [1] avec le context passé en argument.
- `String toString()` : permet de renvoyer la forme textuelle du λ-terme.

Voici la hiérarchie de classes correspondant aux λ-terme[2] :

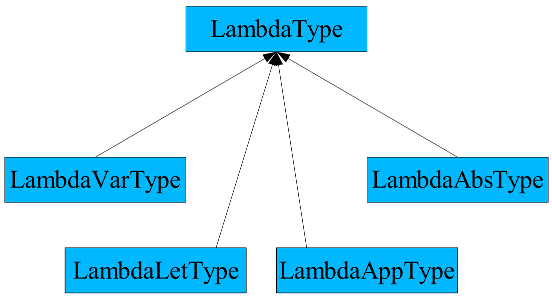


FIG. 1 – Hiérarchie des λ-termes

1.1.2 Les visiteurs

Les visiteurs sont les classes permettant de réaliser un traitement sur un groupe de classes connues à l'avance. Nous nous servons des visiteurs dans la conception de l'interface graphique. Les visiteurs doivents dériver de la classe *Visiteur* et contient les méthodes qui devront être appeler dans les λ-termes. La classe *Visiteur* est écrite en *Java*1.5 et le paramètre est l'élément qui doit renvoyé.

```
public <Resultat> Resultat visite(Context c,Visiteur<T> v) {
    return v.visiteLambdaVarType(this,c);
}
```

FIG. 2 – Exemple d'appel de visiteur dans la classe *LambdaVarType*

[1] par exemple de l'inférence de type
[2] voir la *Javadoc* pour plus de détail

1.1.3 Les types

La super classe *Types* : Les types du λ-calcul sont divisés en deux sous-catégories :

1. Les types simples qui sont formés à partir des *variables de types* et du constructeur $\rightarrow$. Ces types simples sont réprésentés par la classe abstraite *TypesSimple* et peuvent réaliser plusieurs opérations de base :
   – le teste d'occurence de variable
   – le teste d'égalite de deux types simples
   – la substitution d'une variable dans une liste de subtitution
   – l'unification d'un type par rapport à un autre
2. Les schémas de types qui représente un type simple quantifié universellement. Ils sont utilisés dans le polymorphisme, typiquement par l'ajout de la règle `letin`.La classe *SchemaType* permet de généraliser un type simple avec un context. Cela se fait au constructeur ou à l'aide d'une méthode statique appartenant à la classe : `generalize(TypesSimple ts,Context ctx)`. Enfin si l'inférence est amené à instancier des types il peut soit appelé la méthode `instancie()`.

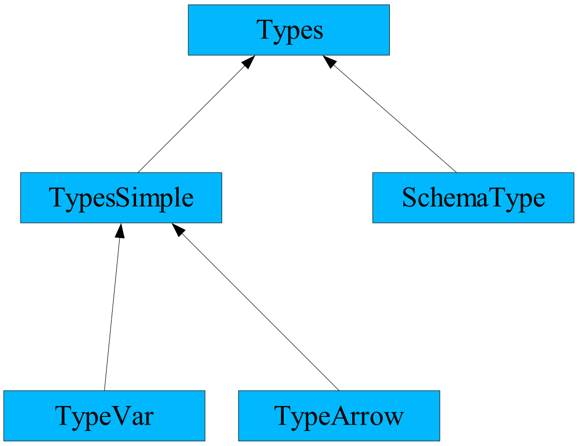


FIG. 3 – Hiérarchie des types

1.2 Typage des termes

1.2.1 Les règles de typage

Maintenant que les termes et les types sont définis, il faut savoir comment déterminer les types de chaque élément, pour cela il existe des *règles de typage* :

(Var)

$$(x_1 : \sigma_1) \ldots (x_n : \sigma_n) \vdash x_i : \tau[\tau_i/\alpha_i]\sigma_i = \forall \alpha_1, \ldots, \alpha_n.\tau$$

(App)

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad\qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

(Abs)

$$\frac{(x : \tau)\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \rightarrow \sigma}$$

(Letin)

$$\frac{\Gamma \vdash N : \tau \qquad \alpha_1, \ldots, \alpha_n = V(\tau) - V(\Gamma) \qquad (x : \forall \alpha_1, \ldots, \alpha_n.\tau)\Gamma \vdash M : \sigma}{\Gamma \vdash let \;\; x = N \;\; in \;\; M : \sigma}$$

1.2.2 Les outils pour les typages

Pour typer un terme, il faut des outils tel que :

– les environnements et les contexts : dans l'applet les environnements et les contexts sont utilisés et construit de façon différente même si leurs rôles sont équivalents. Cela permet d'avoir un masquage de variable de façon très simple dans les contextes et d'avoir un accès optimisé sur les environnements. Les contextes et les environnements sont représentés respectivement par la classe *Context* et par la classe *Env*.
– la substitution : c'est une opération appartenant à un type simple et qui prend un *Env* en paramètre. Elle ramène un nouveau type simple où les substitutions figurant dans l'environnement ont été effectuées.
– la composition des subtitutions : la composition d'une substitution est une méthode statique appartenant à la classe *TypesSimple*, elle prend deux environnements et ramène un nouveau environnement qui est le composé des deux autres (pour déterminer *l'unificateur principale*).
– l'unification : renvoie un environnement dans lequel il y figure la liste des substitutions a réaliser, de manière à ce que pour tout types simples $\tau 1$ $\tau 2$:

$$Env \;\; env = Unify(\tau 1, \tau 2);$$

$$\tau 1.substitution(env) = \tau 2.substitution(env)$$

1.3 Evolution du programme

Les classes sont faîtes en sorte que l'ajout de λ-terme soit simple. L'ajout se fait en deux étapes :

1. La création du nouveau terme dérivant de la classe *LambdaType* et satisfaisant les méthodes abstraites.
2. La création d'un nouveau visiteur dérivant de *Visiteur* et permettant de gérer le nouveau terme.

Bien sûre l'évolution ne prend pas en compte l'évolution pour l'interface graphique.

1.4 Syntaxe des expressions

Maintenant que toutes les classes sont posées. Il faut réaliser l'analyse syntaxique pour l'utilisateur entrant sa formule. Pour cela, le choix du parseur *JavaCC*, réalisé en *Java*, est un bon compromis[3]. Mais il connait un problème pour les structures récursives ceci imposera une écriture complétement parenthésées. Les variables de Types et les noms de variables sont limités aux caractères de l'alphabet et aux chiffres, en syntaxe simplifier les règles donnent :

```
formule() : context() ``|-'' lambdatype()

context() :
  ``('' ident() ``:'' typesimple() ``)'' context()
| empty

lambdatype() :
  ident()
| ``l'' ident() ``.'' lambdatype()
| ( lambdatype() lambdatype() )

typesimple() :
  ident()
| ``('' typesimple() -> typesimple() ``)''
```

[3]le seul défaut est qu'il soit programmé en *Java* 1.4

2 L'interface graphique

2.1 Présentation

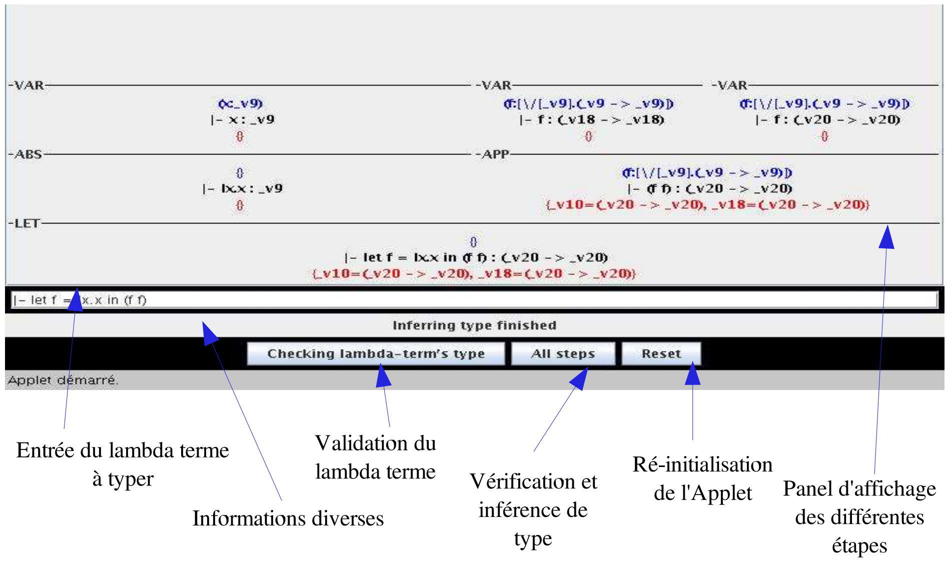


FIG. 4 – Visualisation des étapes

L'Applet est principalement programmée en SWING.
Elle se divise en deux grandes parties :

- Dans la partie du bas, on trouve un `JTextField` dans lequel l'utilisateur entre un λ-terme. En dessous, il y a trois boutons :
  - `Checking lambda-term's type` : Une fois un λ-terme entré, l'utilisateur doit en premier lieu cliquer sur ce bouton afin d'initialiser le parser. De plus, il permet de vérifier si la formulation du terme est correcte.
  - `All steps` : Une instance d'un visiteur va alors faire toutes les étapes de la vérification du λ-terme. A chaque fois qu'il rencontre un λ-terme spécifique, il dessine en conséquence ce qu'il faut dans le panel supérieur de l'Applet.
  - `Reset` : Ce bouton réinitialise entièrement le programme. Le parser est remis à zéro. Tous les panels sont effacés sauf le contenu du `JTextField` afin d'éviter d'avoir à retaper entièrement toute la formule.
- Dans le panel du dessus, l'applet dessinera au fur et à mesure les étapes de la vérification de type.

2.2 La classe *LambdaTermPanel*

La classe *LambdaTermPanel* appartient au paquetage `gui` et étend un *JPanel*. Elle implémente l'interface `MouseListener` afin de permettre à l'utilisateur de voir

le context et le λ-term en entier dans le panel `info` lorsque que ceux-ci deviennent long, en cliquant sur une étape souhaitée.
A chaque étape de l'inférence de type, c'est une instance de cette classe qui est ajouté au panel de visualisation.

2.3 Inférence de type et construction de la visualisation

2.3.1 Retour sur les visiteurs

La classe abstraite *Visiteur*
Nous avons créé une classe abstraite *Visiteur* paramétrée contenant les définitions des méthodes pour tous les visiteurs héritant de cette classe.
Ces définitions de méthodes sont les suivantes :

- `public abstract <Result> visiteLambdaVarType(LambdaVarType lt, Context c) ;`
  Cette méthode sera lancé lorsque le visiteur rencontrera un λ-term de type `LambdaVarType`.
- `public abstract <Result> visiteLambdaAbsType(LambdaAbsType lt, Context c) ;`
  Cette méthode sera lancé lorsque le visiteur rencontrera un λ-term de type `LambdaAbsType`.
- `public abstract <Result> visiteLambdaAppType(LambdaAppType lt, Context c) ;`
  Cette méthode sera lancé lorsque le visiteur rencontrera un λ-term de type `LambdaAppType`.
- `public abstract <Result> visiteLambdaLetType(LambdaLetType lt, Context c) ;`
  Cette méthode sera lancé lorsque le visiteur rencontrera un λ-term de type `LambdaLetType`.

Toutes les classes héritant de *LambdaType* implémente la méthode `visite(Context ctx, Visiteur v)`. Cette méthode appelle une des méthodes citées ci-dessus du visiteur `v` suivant le type du λ-term (i.e *LambdaVarType*, *LambdaAbsType*, *LambdaAppType* ou *LambdaLetType*).

Voici le diagramme des classes des visiteurs.

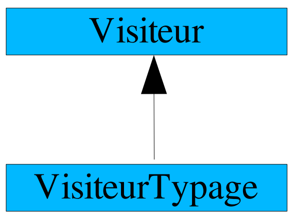


Fig. 5 – Hiérarchie des classes des visiteurs

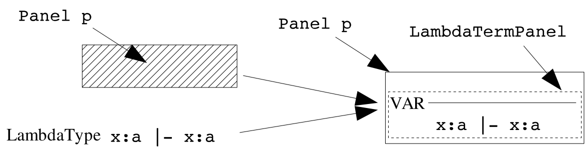
Les visiteur de typage *VisiteurTypage*

Nous avons créé la classe *VisiteurTypage* héritant de la classe *Visiteur* que l'on paramètre avec `<`*TypeResult*`>`. De ce fait les types de retour des méthodes implémentées par ce visiteur seront des *TypeResult*.Ce sont les instances de cette classe qui s'occupe de faire l'inférence de type sur les λ-term et de faire afficher par la même occasion les différents étapes.

Lorque l'on créé une nouvelle instance d'un *VisiteurTypage*, on passe en paramètre du constructeur le panel `p` dans lequel il doit travaillé (i.e mettre les nouveaux panels à afficher). Ensuite, en fonction du type de λ-term sur lequel il travaille, on applique les règles de typage citées plus haut et on construit le panel adéquat. On passe aussi un autre paramètre `info` de type *JLabel* au constructeur. Le texte de cette variable pourra contenir le contexte et le λ-term lorsque que l'utilisateur cliquera sur une étape de l'inférence de type.
Dans tous les cas, le visiteur ajoute dans le panel `p` un panel *LambdaTermPanel* avec le λ-term de type *LambdaType*. Ensuite en fonction du type, il dessine les panels de la manière suivante :

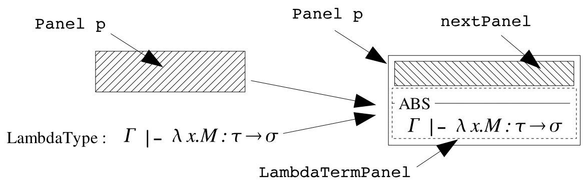
- Cas d'un *LambdaVarType*

  Lorsqu'on peut appliquer une règle `VAR`, c'est que l'inférence de type d'une variable est terminée.



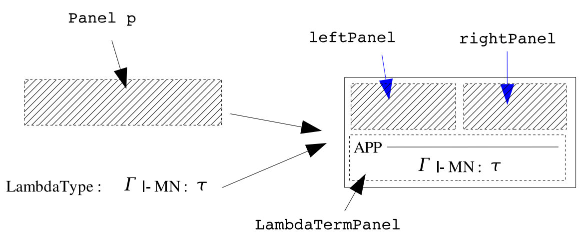
- Cas d'un *LambdaAbsType*

  On cherche ensuite le type de M avec un context augmenté du type de x. Les étapes suivantes sont alors dessinées dans le panel `nextPanel`.



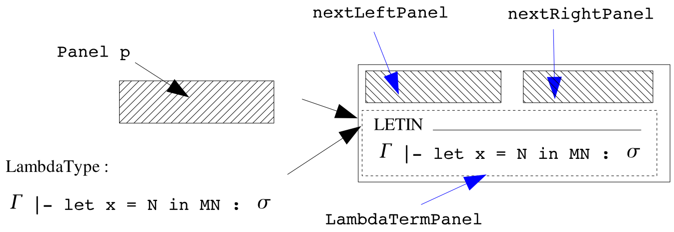
- Cas d'un *LambdaAppType*

  On infère le type de M en dessinant dans `nextLeftPanel` et ensuite celui de N dont on dessine les étapes dans `nextRightPanel`.



- Cas d'un *LambdaLetType*

On recherche le type de N en dessinant dans `nextLeftPanel` et ensuite celui de M dont on dessine les étapes dans `nextRightPanel`.



2.3.2 Explication des codes de couleurs

Lors des étapes d'inférence de type, différentes informations avec des couleurs différentes sont affichées à une étapes données. Voici à ce quoi correspond le code de couleur :

bleu c'est le contexte courant
noir le λ-term et son type
rouge l'environnement

2.4 L'Applet

2.4.1 Sa composition

L'applet se divise en deux parties. Dans la partie du haut, nous avons une visualisation des différentes étapes d'inférence. Dans la partie du bas, nous avons

- un champs de saisie du λ-term que l'on veut typer
- un label affichant des informations diverses.
- un groupe de boutons permettant de valider le λ-term, lancer toutes les étapes de l'inférence de type et réinitialiser le tout.

2.4.2 Utilisation de l'Applet

Avant toute chose, étant donné que l'applet est programmé en Java 1.5, il faut que l'utilisateur ait au moins installé la J2SE 5.0 JRE âfin de pouvoir la faire tourner.
Lorsque l'utilise lance l'applet, un λ-term est mis par défaut (de type *LambdaLetType*), mais il peut toujours la changer. Ensuite il faut qu'il valide le λ-term, en cliquant sur le bouton `Checking lambda term's type`, avant de pouvoir lancer les étapes d'inférence de type (bouton `All Steps`). Afin de pouvoir inférer le type d'un autre λ-term, l'utilisateur doit d'abord cliquer sur le bouton `Reset`.