Traduction des langages

Typage

Objectif :
- Définir la nouvelle structure d'arbre obtenue après la passe de typage
- Définir les actions à réaliser par la passe de typage

1 Rappel : le typage du langage Rat

1.1 Grammaire du langage RAT

1. $PROG' \rightarrow PROG\$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS$
7. $IS \rightarrow \Lambda$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow \Lambda$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow \Lambda$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$
33. $E \rightarrow\ (\ E\ \ )$

1.2 Jugements de typage du langage RAT

Axiomes

- $\sigma \vdash\ \ : void$
- $\sigma :: \{x : \tau\} \vdash x : \tau$
- $\dfrac{\sigma \vdash x : \tau}{\sigma :: \{y : \tau'\} \vdash x : \tau}$
- $\sigma \vdash true : bool$
- $\sigma \vdash false : bool$
- $\sigma \vdash entier : int$

Expression

— $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash [\ E_1\ /\ E_2\ ] : rat}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash num\ E : int}$

— $\dfrac{\sigma \vdash E : rat}{\sigma \vdash denom\ E : int}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash (E) : \tau}$

— $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$

— $\dfrac{\sigma \vdash E_1 : rat \quad E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$

— $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : bool \quad E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$

— $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$

— On se limitera à ces signatures.

Structures de contrôle

— $\dfrac{\sigma \vdash E : bool \quad \sigma \vdash BLOC_1 : void \quad \sigma \vdash BLOC_2 : void}{\sigma \vdash if\ E\ BLOC_1\ else\ BLOC_2 : void, \{\}}$

— $\dfrac{\sigma \vdash E : bool \quad \sigma \vdash BLOC : void}{\sigma \vdash while\ E\ BLOC : void, \{\}}$

Déclaration / affectation

— $\dfrac{\sigma \vdash TYPE : \tau_1 \quad \sigma \vdash E : \tau_2 \quad (estCompatible\ \tau_1\ \tau_2)}{\sigma \vdash TYPE\ id = E : void, \{id : \tau_1\}}$

— $\dfrac{\sigma \vdash id : \tau_1 \quad \sigma \vdash E : \tau_2 \quad (estCompatible\ \tau_1\ \tau_2)}{\sigma \vdash id = E : void, \{\}}$

Autres instructions

— $\dfrac{}{\sigma \vdash const\ \ id\ =\ entier : void, \{id : int\}}$

— $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash print\ \ E : void, \{\}}$

Déclaration de fonction

— $\dfrac{A \quad B \quad C \quad D \quad E}{\sigma \vdash TYPE\ id\ (DP)\ \{IS\ return\ E;\} : void, \{id : \tau_2 \to \tau_1\}}$

- A : $\sigma \vdash TYPE : \tau_1$
- B : $\sigma \vdash DP : \tau_2, \sigma_p$
- C : $\sigma :: \sigma_p :: \{id : \tau_2 \to \tau_1\} \vdash IS : void, \sigma_l$
- D : $\sigma :: \sigma_p :: \sigma_l :: \{id : \tau_2 \to \tau_1\} \vdash E : \tau_3$
- E : $estCompatible\ \tau_1\ \tau_3$

— $\dfrac{\sigma \vdash TYPE : \tau_1 \quad \sigma :: \{id : \tau_1\} \vdash DP : \tau_2, \sigma_p}{\sigma \vdash TYPE\ id\ DP : \tau_1 \times \tau_2, \{id : \tau_1\} :: \sigma_p}$

— $\dfrac{\sigma \vdash TYPE : \tau_1}{\sigma \vdash TYPE\ id : \tau_1, \{id : \tau_1\}}$

Appel de fonction

— $\dfrac{\sigma \vdash id : \tau_1 \to \tau_2 \quad CP : \tau_3 \quad (estCompatible\ \tau_1\ \tau_3)}{\sigma \vdash call\ (id\ CP) : \tau_2}$

— $\dfrac{\sigma \vdash E : \tau_1 \quad CP : \tau_2 \quad \tau_2 \neq void}{\sigma \vdash E\ CP : \tau_1 \times \tau_2}$

— $\dfrac{\sigma \vdash E : \tau_1 \quad CP : void}{\sigma \vdash E\ CP : \tau_1}$

Suite d'instructions

— $\dfrac{\sigma \vdash IS : void, \sigma'}{\sigma \vdash \{IS\} : void}$

— $\dfrac{\sigma \vdash I : void, \sigma' \quad \sigma :: \sigma' \vdash IS : void, \sigma''}{\sigma \vdash I\ IS : void, \sigma' :: \sigma''}$

Le programme

— $\dfrac{\sigma \vdash FUN : void, \sigma' \quad \sigma :: \sigma' \vdash PROG : void, \sigma''}{\sigma \vdash FUN\ PROG : void, \sigma' :: \sigma''}$

— $\dfrac{\sigma \vdash BLOC : void}{\sigma \vdash id\ BLOC : void}$

2 Passe de typage

Nous rappelons qu'un compilateur fonctionne par passes, chacune d'elle réalisant un traitement particulier (gestion des identifiants, typage, placement mémoire, génération de code,...). Chaque passe parcourt, et potentiellement modifie, l'AST.

La seconde passe est une passe de typage. C'est elle qui vérifie la conformité des types déclarés et associe aux identifiants leurs informations de type.

2.1 Structure de l'AST après la passe de typage

La passe de typage réalise des vérifications de type qui nécessitent une mise à jour des informations de type de identificateurs. Elle prépare également les passes suivantes, par exemple en choisissant la "version" de l'opérateur à utiliser en cas de surcharge des opérateurs.

▷ **Exercice 1** *Définir la structure de l'AST après la passe de typage.*

— **Les informations des identifiants seront mises à jour avec les informations de type.**
— **Les informations de types n'ont plus besoin d'être conservées dans les nœuds de l'arbre.**

— **Les opérateurs surchargés sont "résolus" et remplacés par des opérateurs non surchargés.**

Voici un type possible pour l'arbre après la passe de résolution des identifiants :

```
(* Opérateurs binaires existants dans notre langage - sans surcharge *)
type binaire = PlusInt | PlusRat | MultInt | MultRat | EquInt | EquBool | Inf

(* Expressions existantes dans notre langage *)
(* = expression de AstTds *)
type expression =
  | AppelFonction of string * expression list * Tds.info_ast
  | Rationnel of expression * expression
  | Numerateur of expression
  | Denominateur of expression
  | Ident of Tds.info_ast
  | True
  | False
  | Entier of int
  | Binaire of binaire * expression * expression

(* instructions existantes dans notre langage *)
(* = instruction de AstTds + informations associées aux identificateurs, mises à jour *)
type bloc = instruction list
 and instruction =
  | Declaration of expression * Tds.info_ast
  | Affectation of expression * Tds.info_ast
  | AffichageInt of expression
  | AffichageRat of expression
  | AffichageBool of expression
  | Conditionnelle of expression * bloc * bloc
  | TantQue of expression * bloc
  | Empty (* les nœuds ayant disparus: Const *)

(* informations associées à l'identificateur, liste des paramètres, corps, expr. de retour *)
type fonction = Fonction of Tds.info_ast * Tds.info_ast list *
                            instruction list * expression

(* Structure d'un programme dans notre langage *)
type programme = Programme of fonction list * bloc
```

2.2 Actions à réaliser lors de la passe de typage

La passe de typage est donc une transformation d'un arbre vers un autre.

▷ **Exercice 2** *Définir les actions à réaliser lors de la passe de typage.*

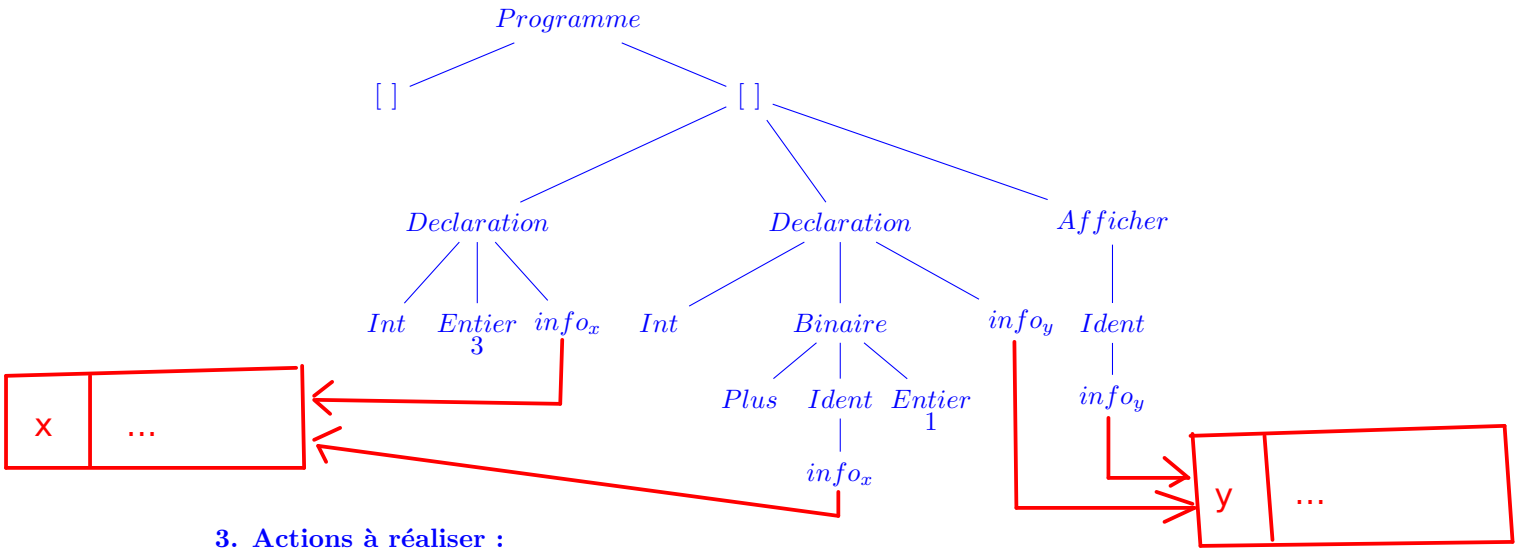
On reprend l'exemple :

1. **Code :**

```
main{
 int x = 3;
 int y = x+1;
```

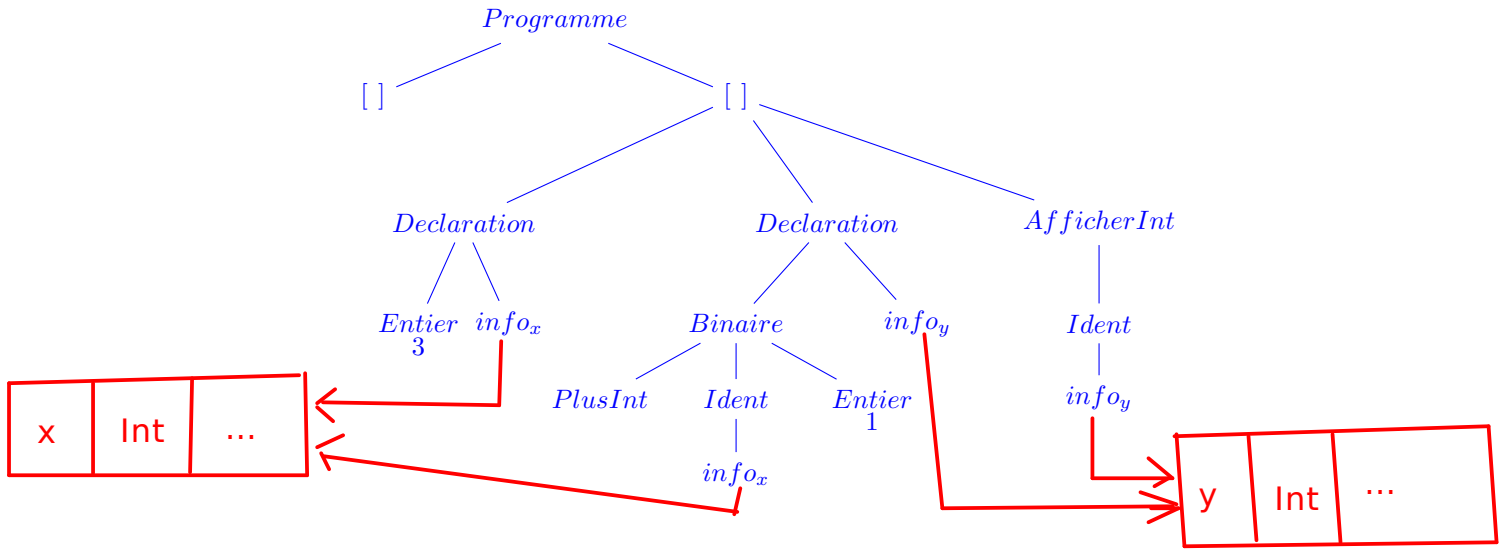
```
 print y;
}
```

2. **AstTds**



3. **Actions à réaliser :**
   (a) **Mettre à jour** $info_x$ **avec le type de** x
   (b) **Vérifier que** $+$ **est utilisé avec une bonne signature**
   (c) **Identifier la version de** $+$ **à utiliser (surcharge)**
   (d) **Vérifier que le type de retour de** $+$ **est compatible avec le type déclaré de** y
   (e) **Mettre à jour** $info_y$ **avec le type de** y
   (f) **Identifier la version d'affichage à utiliser (surcharge)**

4. **AstType**



Comment obtenir ce résultat ? **Les informations de type sont connues au niveau des feuilles, il faut donc que les informations de types remontent dans l'arbre ⇒ valeur de retour des fonctions d'analyse. Nous n'en avons besoin que sur les expressions, car les instructions sont typées en unit / void.**

Commençons par l'analyse des instructions puis des expressions. L'analyse du programme principal et l'analyse des fonctions ne sont pas détaillées et sont à définir.

```
let rec analyse_type_instruction i =
  match i with
  | AstTds.Declaration (t, e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec t [ sinon TypeInattendu ]
        - mettre à jour le type dans info (modifier_type_info t info);
        - retourner Declaration (ne, info)
  | AstTds.Affectation (e, info) ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - vérifier que te est compatible avec le type dans l'info
        - retourner Affectation (ne, info)
    (* Affichage *)
    | AstTds.Affichage e ->
        - Analyser e pour obtenir son type (te) et le nouvel arbre (ne)
        - selon te, construire le nouvel arbre en résolvant la surcharge
    (*Conditionnelle *)
    | AstTds.Conditionnelle (c,t,e) ->
        - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
        - vérifier que tc est Bool [ sinon TypeInattendu ]
        - analyser les blocs t et e
        - retourner Conditionnelle(nc,nt,ne)
    (* Tant Que *)
    | AstTds.TantQue (c,b) ->
      - analyser c pour obtenir son type (tc) et le nouvel arbre (nc)
      - vérifier que tc est Bool [ sinon TypeInattendu ]
      - analyser le bloc b
      - retourner TantQue(nc,nt,ne)
    | AstTds.Empty -> Empty

and analyse_type_bloc b = map analyse_type_instruction b
```

Puis on passe aux expressions :

```
(* Asttds.ast -> (ast,typ) *)
let rec analyse_type_expression e =
  match e with
  | AstTds.AppelFonction (n,le,info) ->                    (* Appel de fonction *)
    - ( info_ast_to_info info) doit être un InfoFun(nom,typeRet,typeParams)
      [ sinon erreur interne, ça doit être garanti par la passe précédente ]
    - analyser chacune des expressions dans le, pour obtenir nle (nouvelle
      liste des arguments) et ltype ( liste des types de ces expressions)
    - vérifier que ltype est compatible avec typeParams
      [ sinon exception TypesParametresInattendus ]
    - retourner le couple (AppelFonction (n,nle,info ), typeRet)
  | AstTds.Rationnel (e1,e2) ->                            (* Définition d'un rationnel *)
    - analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
    - analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
```

```
  – vérifier que t1 et t2 sont Int (sinon TypeInattendu)
  – retourner (Rationnel(ne1,ne2), Rat)
| AstTds.Numerateur e1 ->                                  (* Accès au numérateur *)
  – analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  – vérifier que t1 est Rat (sinon TypeInattendu)
  – retourner (Numerateur ne1, Int)
| AstTds.Denominateur e1 ->                                (* Accès au dénominateur *)
  – analyser l'expression e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  – vérifier que t1 est Rat (sinon TypeInattendu)
  – retourner (Denominateur ne1, Int)
 | AstTds.Ident info ->                                    (* Accès à un identifiant *)
  begin
    match info_ast_to_info info  with
    | InfoVar (_,t,_,_) -> (Ident info, t)
    | InfoConst (_,_) -> (Ident info, Int)
    | _ -> failwith ("Internal error: symbol not found")
  end
| AstTds.Binaire (b,e1,e2) ->                             (* Opération binaire *)
  – analyser e1 pour obtenir ne1 (nouvel arbre) et t1 (type calculé)
  – analyser e2 pour obtenir ne2 (nouvel arbre) et t2 (type calculé)
  – identifier les cas corrects et retourner le couple (arbre,type) correspondant:
    – Int, Plus, Int -> (Binaire (PlusInt,ne1,ne2), Int)
    – Rat, Plus, Rat -> (Binaire (PlusRat,ne1,ne2), Rat)
    – Int, Equ, Int  -> (Binaire (EquInt,ne1,ne2), Bool)
    – Bool, Equ, Bool -> (Binaire (EquBool,ne1,ne2), Bool)
    – etc
    – sinon TypeBinaireInattendu
| AstTds.True -> (True, Bool)
| AstTds.False -> (False, Bool)
| AstTds.Entier i -> (Entier i, Int)
```

Il reste à définir l'analyse du programme principal et l'analyse des fonctions.