Informatique

Cours de compilation INF 553

31 mars 2005

Ce sujet traite de deux modifications du compilateur `zyva` du cours. La première partie traite d'une première modification et les deux parties suivantes d'une seconde modification. Les deux modifications sont indépendantes, et les parties le sont également dans une large mesure.

Première modification : la boucle for.

La boucle `for` est la construction naturelle pour parcourir les tableaux. Par exemple, on affiche un tableau `t` comportant `n` éléments de type **integer** par :

```
procedure affiche(t:array of integer, n:integer) ;
begin
  for i := 0 to n-1 do
    writeln(t[i])
end ;
```

On notera que l'indice de boucle (c'est-à-dire la variable `i`) n'est pas déclaré explicitement. De fait, notre boucle **for** est celle de Caml, dont voici la sémantique informelle.

> L'instruction « **for** i := e_1 **to** e_2 **do** B » évalue d'abord les deux expressions e_1 et e_2 (les bornes) en deux valeurs entières (type **integer**) n et p.
>
> Ensuite, le corps B (une instruction) est exécuté de façon répétée dans un environnement où l'indice i (un nom de variable) est lié successivement aux valeurs entières n, $n+1$, ..., $p-1$, p. Le corps n'est pas exécuté si $n > p$.
>
> L'affectation de l'indice de boucle i dans le corps B est interdite.

Question 1. On essaie tout d'abord d'opérer une tranformation « *source to source* » de la boucle **for** en boucle **while**.

Une procédure contenant une boucle **for** :

```
procedure p(...) ;
var
  ...
begin
  ...
  for i := e1 to e2 do
    B ;
  ...
end ;
```

sera transformée en :

```
procedure p(...) ;
var
  i:integer ;
  ...
begin
```

```
      ...
      i := e1 ;
      while i <= e2 do begin
        B ;
        i := i+1
       end ;
      ...
   end ;
```

Donner un exemple qui montre que la transformation décrite ci-dessus ne respecte pas la sémantique de la boucle **for**.

Plutôt que de suivre la technique de la question 1, on va intégrer complètement la nouvelle construction. La définition des arbres de syntaxe abstraite est étendue par l'ajout des lignes suivantes à la fin du fichier `pp.mli`.

```
| For of string * expression * expression * instruction
(*
   For (i, e1, e2, B) est l'arbre de syntaxe abstraite de
     for i := e1 to e2 do B
*)
```

Cela revient à ajouter une nouvelle instruction à Pseudo-Pascal.

Question 2. Modifier les analyseurs lexical et syntaxique de `zyva` (fichiers `lexer.mll` et `parser.mly`, donnés en annexe) pour intégrer la nouvelle instruction.

Question 3. On se propose de rejeter les programmes qui ne respectent pas la condition « l'affectation de l'indice de boucle dans le corps de la boucle est interdite ». Écrire, en Caml, une fonction `checkCorrect` de type `Pp.instruction -> bool`, qui prend en argument une instruction et renvoie `true`, si et seulement si l'interdiction est respectée.

Question 4. Proposer une sémantique précise de la boucle **for**. Pour répondre à cette question, le plus simple est sans doute de compléter l'interpréteur de `zyva` (fichier `interpret.ml` donné en annexe), mais on pourra aussi donner des règles d'inférence dans le style de celles du poly, section 3.4.2.

Question 5. Modifier le générateur de code intermédiaire pour traiter la boucle **for**. Le générateur est le fichier `trans.ml` donnée en annexe. Les interfaces `code.mli` (définition du code intermédiaire) `gen.mli` (temporaires et étiquettes) et `env.mli` (environnements de Pseudo-Pascal) sont données en annexe.

Question 6. Proposer une transformation correcte de la boucle **for** en boucle **while** équivalente. Concrètement, on expliquera comment transformer la procédure `p` de la question 1.

ATTENTION On veillera à traiter correctement le cas où la procédure `p` utilise une variable de nom i en dehors du corps de la boucle.

Deuxième modification : le coprocesseur flottant.

Dans cette partie (et dans la suivante) on se propose d'étendre `zyva` pour qu'il traite les calculs flottants. Seuls deux points importants seront traités, d'abord l'utilisation du coprocesseur flottant du MIPS, puis, dans la partie suivante, la distinction entre opérations flottantes et entières à l'aide du typage.

Le processeur MIPS peut donc être équipé d'un coprocesseur chargé des calculs flottants. Nous allons uniquement considérer des nombres flottants *double précision* (type **real** de Pseudo-Pascal) dont la taille est de deux mots mémoire ou 8 octets.

1. Le coprocesseur possède 16 registres (doubles) notés `$f0`, `$f2`, ..., `$f30` (de deux en deux). Chaque registre double `$f`$\langle n \rangle$ (n pair) regroupe en fait deux registres simple précision, notés `$f`$\langle n \rangle$ et `$f`$\langle n+1 \rangle$. En pratique, tout se passe le plus souvent comme si le coprocesseur possédait bien 16 registres doubles.
2. Les opérations flottantes sont du type « trois addresses » opérant exclusivement sur les registres flottants. Par exemple, l'instruction **add.d** `$f0, $f2, $f4` range la somme (addition flottante double précision) du contenu des registres `$f2` et `$f4` dans le registre `$f0`.
3. Il existe une instruction **mov.d** de transfert entre registres du copresseur dont l'usage est similaire à celui de l'instruction **move** du processeur. Par exemple, l'instruction **mov.d** `$f0, $f2` copie le contenu du registre (double) `$f2` dans le registre (double) `$f0`.

Question 7. Énoncer des conventions d'appel pour les fonctions qui prennent des arguments flottants et rendent un résultat flottant. On illustrera ensuite ces conventions en « compilant » la fonction Pseudo-Pascal suivante :

```
function fadd(x,y : real) : real ;
begin
  fadd := x + y
end
```

Question 8. On suppose que le compilateur `zyva` est suffisament sophistiqué pour introduire des conversion des entiers (type **integer**) vers les flottants (type **real**) quand c'est utile. Comme par exemple dans la fonction suivante ou l'addition demande la conversion préalable de l'entier `n` :

```
function fiadd(x:real; n:integer):real;
begin
   fiadd := x+n
end ;
```

Pour réaliser ces conversions, le processeur MIPS emploie les instructions suivantes :

1. Il est possible de copier (sans modification) le contenu d'un registre du processeur dans un registre (simple) du coprocesseur, par l'instruction **mtc1** (*Move To Coprocessor 1*). Par exemple, l'instruction **mtc1** `$v0, $f2` copie le contenu du registre `$v0` (du processeur), dans le registre `$f2` (du coprocesseur).
2. Le coprocesseur peut effectuer des conversions de données binaires et en particulier des entiers (32bits) vers les flottants double précision à l'aide de l'instruction **cvt.d.w**. De sorte que, par exemple, l'instruction **cvt.d.w** `$f0, $f2` convertit l'entier contenu dans le registre `$f2` en flottant double précision et range ce résultat dans le registre (double) `$f0` du coprocesseur.

Donner un code possible résultat de la compilation de `fiadd`.

Question 9. Compiler manuellement la fonction suivante :

```
function ffiadd(x : real; n:integer):real;
begin
   ffiadd := fadd(x,fiadd(x, n))
end;
```

ATTENTION : On demande un code tel que produit par le compilateur `zyva` qui n'a pas connaissance des registres modifiés par les fonction appelées, mais s'appuie sur des conventions d'emploi des registres clairement énoncées. Le coprocesseur peut lire un flottant en mémoire par l'instruction **l.d** et écrire un flottant en mémoire par l'instruction `s.d`. Ces deux instructions fonctionnent comme les instructions **lw** et **sw** du processeur. Par exemple **l.d** `$f0, 0($a0)` charge les deux mots mémoire d'adresses `$a0` et `$a0`+4 dans le registre (double) `f0`.

Troisième partie : typage des opérations sur les flottants.

Le programmeur Pseudo-Pascal ne fait aucune distinction entre opérations entières et opérations flottantes. C'est-à-dire qu'il écrit par exemple.

```
function test(n:integer ; x:real) : real ;
begin
  n := n + n ;
  x := x + x ;
  test := x + n
end ;
```

Ici, la première addition est une addition entière (en dernière analyse **add**), tandis que les deux suivantes sont des additions flottantes (en dernière analyse **add.d**). En outre, la dernière addition ne peut s'effectuer qu'une fois l'entier `n` converti (ou « promu ») en flottant. On dit que l'opérateur « `+` » est surchargé, il peut signifier des opérations différentes selon les types de ses arguments.

Question 10. Comment peut-on justifier l'introduction par le compilateur de conversions implicites des entiers vers les flottants ? On remarquera que l'utilisation implicite de la conversion inverse des flottants vers les entiers n'est pas autorisée.

Le compilateur `zyva` modifié procède à la résolution de la surcharge lors d'une phase sémantique de vérification des types. Concrètement, on va réécrire la syntaxe abstraite du programme en *syntaxe abstraite typée*.

La syntaxe abstraite est donnée dans l'annexe `pp2.mli` (attention la syntaxe abstraite a un peu changé par rapport à la première partie), la syntaxe abstraite typée est donnée dans le fichier `tpp.mli`. Les deux syntaxes abstraites non-typée et typée font référence aux types des opérations et des comparaisons définies dans le fichier `primitive.mli`.

Pour ce qui nous intéresse, la syntaxe typée étend et précise la syntaxe non-typéee au sens suivant :

1. Les expressions de la syntaxe typée possèdent toutes un argument supplémentaire (le dernier) qui est le type de l'expression.
2. Les opérateurs (type `binop`) de la syntaxe non-typée sont partitionnés entre calcul (constructeur `Op`) et comparaisons (constructeur `Cmp`). Ils apparaissent comme arguments du constructeur des expressions `Bin`.

   Les opérateurs (type `fun_prim`) de la syntaxe typée sont partitionnés de même et aussi entre opérations entières et flottantes (constructeurs `IOp` et `ICmp` d'une part, et `FOp` et `FCmp` d'autre part). Ils apparaissent comme arguments du constructeur des expressions `Prim_call`, qui représente l'application des opérateurs. Il y a quelques nouveaux opérateurs et notamment la promotion d'un entier en flottant (`ItoF`).

   On notera que `Prim_call` prend une liste d'expressions en second argument, car tous les opérateurs n'ont pas exactement deux arguments. Par exemple, la promotion d'une expression de type entier e en flottant et l'addition de deux expressions entières s'expriment ainsi en syntaxe typée :

   $$\texttt{Prim\_call}(\texttt{ItoF}, [e]) : \texttt{Real} \qquad \texttt{Prim\_call}(\texttt{IOp}(\texttt{Add}), [e_1; e_2]) : \texttt{Integer}$$

   Noter la syntaxe spéciale « $\ldots : t$ » qui met en valeur le dernier argument des constructeurs de la syntaxe typée (le type justement).

Question 11. L'action du vérificateur de type est donc une traduction de la syntaxe abstraite non-typée vers la syntaxe abstraite typée. On peut l'exprimer par des règles d'inférence qui prouvent des propositions de la forme : $\Gamma \vdash e \Rightarrow e' : t$, à lire comme : « dans l'environnement Γ, l'expression e non-typée e se traduit en expression typée e', de type t.

Voici par exemple, comment sont typées les constantes entières, les variables et l'addition entière :

$$\Gamma \vdash \texttt{Int}\,(i) \Rightarrow \texttt{Int}\,(i) : \texttt{Integer} \qquad\qquad \frac{\Gamma(x) = t}{\Gamma \vdash \texttt{Get}\,(x) \Rightarrow \texttt{Get}\,(x) : t}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \texttt{Integer} \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \texttt{Integer}}{\Gamma \vdash \texttt{Bin}\,(\texttt{Op}\,(\texttt{Add}), e_1, e_2) \Rightarrow \texttt{Prim\_call}\,(\texttt{IOp}\,(\texttt{Add}), [e_1' : \texttt{Integer}; e_2' : \texttt{Integer}]) : \texttt{Integer}}$$

Écrire les règles de l'addition flottante.

ATTENTION : Les règles doivent être suffisament puissantes pour typer tous les exemples d'addition de cette partie. Par ailleurs, le typage doit être déterministe, c'est-à-dire autoriser au plus une façon de typer une addition donnée.

Question 12. Le fichier `typer.ml` (la dernière annexe), réalise un vérificateur de type. Il est incomplet dans le sens qu'il ne réalise aucune promotion des entiers vers les flottants. Modifiez-le, afin d'introduire ces conversions partout où c'est possible. Ne pas se laisser impressionner par la longueur du code, une modification mineure permet de répondre à la question.

Corrigé du contrôle INF 553

Question 1. On considère par exemple le programme.

```
program ;

function prec(x:integer):integer ;
begin
  writeln(x) ;
  g := x-1
end ;

procedure affiche(n:integer) ;
begin
  for i := 0 to prec(n) do
   writeln(i)
end ;

begin
  affiche(2)
end.
```

Le programme ci-dessus affiche 3 entiers :

```
2
0
1
```

Alors que le programme transformé affiche 5 entiers :

```
2
0
2
1
2
```

Plus généralement, la sémantique de la boucle **for** demande une et une seule évaluation de la borne supérieure e_2, tandis que le programme transformé évalue cette borne à chaque évaluation du test.

Question 2. Il y a deux nouveau mots-clés **for** et **to** à ajouter (car **:=** et **do** sont déjà connus). On ajoute donc la définition de deux nouveaux lexèmes dans `parser.mly`, par exemple après la ligne 16.

```
%token FOR TO
```

Dans le même analyseur grammatical on ajoute une ligne à la définition du non-terminal `instruction`.

```
instruction:
   ... /* Comme avant */
| FOR IDENT COLONEQUAL expression TO expression DO instruction
     { For ($2, $4, $6, $8) }
;
```

Enfin, pour reconnaître les deux nouveaux mots clés, on ajoute les deux nouveaux mots-clés à la table de hachage `keywords`, en insérant la ligne suivante par exemple entre les lignes 11 et 12 du fichier `lexer.mll`.

```
add_keyword "for" FOR ; add_keyword "to" TO ;
```

Question 3.

```
open Pp
(* 'vars' est la liste des indices actifs *)
let rec doCheck vars stm = match stm with
| Set (x,_)|Read_int (x) -> not (List.mem x vars)
| For (x,_,_,body) -> doCheck (x::vars) body
| Sequence stms -> doChecks vars stms
| While (_,body) -> doCheck body
| _ -> true

and doChecks vars stms = match stms with
| [] -> true
| stm::rem -> doCheck vars stm && doChecks vars rem

let checkCorrect stm = doCheck [] stm
```

Question 4. On ajoute un cas au filtrage définissant la fonction `instr` d'interprétation des instructions.

```
and instr env = function
...
| For (i, e1, e2, body) ->
    let i1 = expr_int env e1 in
    let i2 = expr_int env e2 in
    let icell = ref (Vint i1) in
    let env = {env with locals = (i, xcell)::env.locals} in
(* Je n'utilise pas la fonction 'extend' donn^^e9e qui est mal adapt^^e9e ici *)
    for y=i1 to i2 do
      icell := Vint y ; (* Important : mettre l'indice ^^e0 jour *)
      instr env body
    done
...
```

La principale astuce est d'utiliser la boucle **for** de Caml. Il faut bien penser à ajouter l'indice de boucle dans l'environnement utilisé pour évaluer le corps de la boucle.

Répondre par des règles d'inférence SOS est un peu plus tordu. Rappelons que les jugements d'exécution d'une instruction sont de la forme générale $\rho/\sigma \vdash i \Rightarrow /\sigma'$ pour « *dans l'état-mémoire σ et l'environnement ρ, l'exécution de l'instruction i produit un nouvel état mémoire σ'.* » Pour bien tenir compte du traitement des bornes de la boucle **for**, j'introduis une nouvelle construction de syntaxe abstraite : la boucle « en cours », notée $\texttt{ForI}\,(\ell_x, v_{\max}, B)$, où ℓ_x est une adresse, $v_{\max}$ est un entier et B est une instruction. La sémantique de cette construction est donnée par les deux règles :

$$\frac{\sigma(\ell_x) > v_{\max}}{\rho/\sigma \vdash \texttt{ForI}\,(\ell_x, v_{\max}, B) \Rightarrow /\sigma}$$

$$\frac{\sigma(\ell_x) \leq v_{\max} \qquad \rho/\sigma \vdash B \Rightarrow /\sigma' \qquad \rho/\sigma', \ell_x \mapsto \sigma'(\ell_x) + 1 \vdash \texttt{ForI}\,(\ell_x, v_{\max}, B) \Rightarrow /\sigma''}{\rho/\sigma \vdash \texttt{ForI}\,(\ell_x, v_{\max}, B) \Rightarrow /\sigma''}$$

On donne ensuite la règle du **for** lui-même :

$$\frac{\begin{array}{c}\rho/\sigma \vdash e_1 \Rightarrow v_1/\sigma_1 \\ \rho/\sigma_1 \vdash e_2 \Rightarrow v_2/\sigma_2 \qquad \ell_x \notin dom\,(\sigma_2) \qquad \rho, x \mapsto \ell_x/\sigma_2, \ell_x \mapsto v_1 \vdash \texttt{ForI}\,(\ell_x, v_2, B) \Rightarrow /\sigma_3\end{array}}{\rho/\sigma \vdash \texttt{For}\,(x, e_1, e_2, B)) \Rightarrow /\sigma_3}$$

On remarquera que les deux sémantiques donnent un résultat différent lorsque l'indice de boucle est affecté dans le corps. Ce n'est pas trop grave, au vu du dernier paragraphe de la sémantique informelle qui interdit cette opération.

Question 5. On ajoute un cas au filtrage de la fonction `cinstruction` de compilation des instructions.

```
let rec cinstruction env stm = match stm with
| Set (s,e) ->
... (* Comme avant *)
| For (x, e1, e2, body) ->
  (* Se donner trois ^^e9tiquettes *)
    let lab_test = Gen.new_label ()
    and lab_enter = Gen.new_label ()
    and lab_out = Gen.new_label () in
  (* Se donner deux temporaires (indice et limite) *)
    let t_x = Gen.new_temp ()
    and t_up = Gen.new_temp () in
  (* Renvoyer une seule instruction -> Seq *)
    Seq [
      Move_temp (tx, cexpr env e1) ;
      Move_temp (t_up, cexpr env e2) ;
      Label lab_test ;
      Cjump (Rle, t_x, t_up, lab_enter, lab_out) ;
      Label lab_enter ;
      cinstruction
        (Env.add_local_vars env [x, t_x]) (* Attention ^^e0 l'environnement *)
          body ;
     Move_temp (t_x, Bin (Plus, t_x, Const 1)) ;
     Jump lab_test ;
     Label lab_out ;
   ]
```

Question 6. Il y a en fait deux questions à résoudre, il faut d'abord limiter les évaluations de la borne supérieure à une et une seule, mais aussi respecter la portée de l'indice de boucle. On peut procéder en se donnant deux variables i' et i'' « fraîches », c'est-à-dire n'apparaisant pas dans le programme avant transformation, et en écrivant :

```
procedure p(...) ;
var
  i', i'':integer ;
  ...
begin
  ...
  i' := e1 ;
  i'' := e2 ;
  while i' <= i'' do begin
    B' ;
    i' := i' + 1
  end ;
  ...
end ;
```

L'instruction B' ci-dessus est le corps de boucle original B, avec i remplacé par i', noté $B' = B[i \leftarrow$

$i']$. Attention les règles de définition de cette opération de substitution doivent tenir compte de ce que B peut contenir une boucle liant à nouveau la variable i. Plus précisément, voici les règles de substitution dans les instructions.

$$\begin{array}{rcll}
\texttt{For}\,(i, e_1, e_2, I)[i \leftarrow i'] & = & \texttt{For}\,(i, e_1[i \leftarrow i'], e_2[i \leftarrow i'], I) & \\
\texttt{For}\,(j, e_1, e_2, I)[i \leftarrow i'] & = & \texttt{For}\,(j, e_1[i \leftarrow i'], e_2[i \leftarrow i'], I[i \leftarrow i']) & (j \neq i) \\
\texttt{Set}\,(i, e)[i \leftarrow i'] & & \text{–Interdit–} & \\
\texttt{Set}\,(j, e)[i \leftarrow i'] & = & \texttt{Set}\,(j, e[i \leftarrow i']) & (j \neq i) \\
\texttt{Sequence}\,[I_1; \ldots; I_n][i \leftarrow i'] & = & \texttt{Sequence}\,[I_1[i \leftarrow i']; \ldots; I_n[i \leftarrow i']] & \\
\texttt{If}\,(e, I_1, I_2)[i \leftarrow i'] & = & \texttt{If}\,(e[i \leftarrow i'], I_1[i \leftarrow i'], I_2[i \leftarrow i']) & \\
\texttt{While}\,(e, I)[i \leftarrow i'] & = & \texttt{While}\,(e[i \leftarrow i'], I[i \leftarrow i']) & \\
\texttt{Call}\,(p, e_1, \ldots, e_n)[i \leftarrow i'] & = & \texttt{Call}\,(p, e_1[i \leftarrow i'], \ldots, e_n[i \leftarrow i']) &
\end{array}$$

Il manque quelques instructions et la substitution dans les instructions fait appel à la substitution dans les expressions dont les définition sont sans grande surprise.

$$\begin{array}{rcll}
\texttt{Get}\,(i)[i \leftarrow i'] & = & \texttt{Get}\,(i') & \\
\texttt{Get}\,(j)[i \leftarrow i'] & = & \texttt{Get}\,(j) & (j \neq i) \\
\texttt{Bin}\,(op, e_1, e_2)[i \leftarrow i'] & = & \texttt{Bin}\,(op, e_1[i \leftarrow i'], e_2[i \leftarrow i']) & \\
 & \vdots & &
\end{array}$$

Question 7. Pour le moment il suffit de fixer les registres de passage des arguments et de retour. On suppose par exemple que les arguments de type **real** sont passés dans les registres `$f2`, `$f4`, `$f6` et `$f8`, puis dans la pile ; tandis que le résultat de d'une fonction est rendu dans le registre `$f0` quand il est de type **real**.

```
fadd:
   add.d $f0, $f2, $f4
   j $ra
```

Question 8. En utilisant les convention de passage de `zyva` (premier argument « générique » dans `$a0`), on a par exemple :

```
fiadd:
    mtc1 $a0, $f0 # Transfert vers le coprocesseur ($f1 inchang^^e9)
    cvt.d.w $f0, $f0 # conversion vers la paire $f0, $f1
    add.d $f0, $f2, $f0
    j    $ra
```

Question 9. On décide ici que les 8 registres `$f16` à `$f30` sont traités comme des *callee-save*.

```
    ffiadd_f=12
ffiadd:
    subu $sp, $sp, ffiadd_f
    sw  $ra, 8($sp)
    s.d $f16, 0($sp) # Lib^^e9rer le callee-save $f16
    mov.d $f16, $f2 # L'argument 'x' est dans $f16
    mov.d $f2, $f16 # inutile
    jal  fiadd       # arguments dans $f2 et $a0
    mov.d $f2, $f16
    mov.d $f4, $f0
    jal  fadd        # arguments dans $f2 et $f4
    lw  $ra, 8($sp)
```

```
l.d  $f16, 0($sp)
addu $sp, $sp, ffiadd_f
j    $ra
```

On notera et c'est important que la sauvegarde en pile des registres `$ra` (usage général) et `$f16` (flottant) entraîne la consommation de $1 + 2 = 3$ mots de pile.

Question 10. La conversion d'entiers (32 bits) vers les flottants (64 bits) ne fait perdre aucune information et n'echoue jamais. La conversion inverse est plus problématique, elle fait perdre de l'information (la partie fractionaire du nombre converti), possède plusieures définitions possibles (arrondi par défaut, par excès. . .) et peut échouer (en cas de nombre trop grand).

Question 11. Il y a trois règles, d'abord le cas de deux arguments de type **real**, puis celui où un argument est de type **integer** et l'autre de type **real**.

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \texttt{Real} \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \texttt{Real}}{\Gamma \vdash \texttt{Bin}\,(\texttt{Op}\,(\texttt{Add}), e_1, e_2) \Rightarrow \texttt{Prim\_call}\,(\texttt{FOp}\,(\texttt{Add}), [e_1' : \texttt{Real}; e_2' : \texttt{Real}]) : \texttt{Real}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \texttt{Integer} \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \texttt{Real}}{\begin{array}{c}\Gamma \vdash \texttt{Bin}\,(\texttt{Op}\,(\texttt{Add}), e_1, e_2) \Rightarrow \\ \texttt{Prim\_call}\,(\texttt{FOp}\,(\texttt{Add}), [\texttt{Prim\_call}\,(\texttt{ItoF}, [e_1' : \texttt{Integer}]) : \texttt{Real}; e_2' : \texttt{Real}]) : \texttt{Real}\end{array}}$$

Avec, en plus la règle « premier argument de type **real**, second argument de type entier » symétrique de la seconde règle ci-dessus.

La solution qui consiste à ne retenir que la première règle (addition de deux flottants) et une règle de promotion n'est pas complètement satisfaisante.

$$\frac{\Gamma \vdash e \Rightarrow e' : \texttt{Integer}}{\Gamma \vdash e \Rightarrow \texttt{Prim\_call}\,(\texttt{ItoF}, [e' : \texttt{Integer}]) : \texttt{Real}}$$

Supposons donnée une variable `x` de type **real** et considérons l'affectation `x := 1 + 2`. Les règles autorisent alors, soit de promouvoir d'abord les deux arguments `1` et `2` puis d'effecture une addition flottante, soit d'effectuer une addition entière puis de promouvoir le résultat. Même si la valeur affectée à `x` est la même dans les deux cas, ce n'est pas très déterministe. On notera aussi que, dans le cas de la division`1/2`, alors les deux valeurs peuvent différer. On obtient alors `0.5` dans le premier cas et `0.0` dans le second.

Question 12. En réfléchissant un peu, on voit que la promotion des entiers en flottants s'opère dès que qu'une opération demandant une valeur flottante pour son argument trouve un entier à sa place. Il suffit donc de modifier la fonction `check` du programme donné.

```
(* Introduire une conversion d'un entier en flottant *)
let promote e =  Function_prim (ItoF, [e], Real)

(* Verifier que l'expression 'e' a bien le type 'expected' *)
let check expected e =
  let found = get_type e in
  if expected <> found then match expected, found with
    | Real, Integer -> promote e
    | _ -> raise (Bad (expected, found))
  else
    e
```

Cela fonctionne parce que, dans le cas particulier des opérateurs surchargés (comme l'addition, les comparaisons etc.), le code du typeur traite d'abord le cas d'arguments tous entiers.