Compilateur Mini-lustre

deuxième partie : analyse sémantique

à rendre avant le 12 novembre 10 heures

1 Analyse sémantique

Cette deuxième partie consiste à vérifier la conformité du fichier source, une fois l'analyse syntaxique effectuée avec succès. Dans tout ce qui suit, les types sont soit des types de base δ, soit des types d'expressions constantes de la forme $\delta^{\mathtt{c}}$, soit des n-uplets :

$$\begin{array}{l}\sigma ::= \tau \mid (\tau_1, \ldots, \tau_n)\\ \tau ::= \delta \mid \delta^{\mathtt{c}}\\ \delta ::= \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{float} \mid \mathtt{string}\end{array}$$

Par la suite, on note $\sigma^{\mathtt{c}}$ un n-uplet composé uniquement de types d'expressions constantes. De même on note $\sigma^{\not c}$, le type σ obtenu en supprimant l'exposant $\mathtt{c}$ des types $\delta^{\mathtt{c}}$ contenus dans σ.

On définie l'opération $*$ sur les types σ de la manière suivante :

$$\begin{array}{rcl}\tau_1 * \tau_2 & = & (\tau_1, \tau_2)\\ \tau * (\tau_1, \ldots, \tau_n) & = & (\tau, \tau_1, \ldots, \tau_n)\\ (\tau_1, \ldots, \tau_n) * (\tau'_1, \ldots, \tau'_k) & = & (\tau_1, \ldots, \tau_{n-1}) * (\tau_n, \tau'_1, \ldots, \tau'_k)\end{array}$$

1.1 Typage des expressions et des motifs

Un contexte Γ est un ensemble de déclarations de types de la forme $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, où $x_i : \tau_i$ signifie que la variable x_i a le type τ_i. Un contexte Δ est une suite de signatures de noeuds ou de primitives (cf. section 1.2) de la forme $[n_1 : \sigma_1 \to \sigma'_1; \ldots; n_k : \sigma_k \to \sigma'_k]$.

On introduit le jugement $\Gamma, \Delta \vdash e : \sigma$ signifiant « dans les contextes Γ et Δ, l'expression e est bien typée et elle a le type σ ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \delta}{\Gamma, \Delta \vdash c : \delta^{\mathtt{c}}} \qquad \frac{x : \tau \in \Gamma}{\Gamma, \Delta \vdash x : \tau}$$

$$\frac{\Gamma, \Delta \vdash e : \mathtt{int}}{\Gamma, \Delta \vdash \mathtt{-}e : \mathtt{int}} \qquad \frac{\Gamma, \Delta \vdash e : \mathtt{float}}{\Gamma, \Delta \vdash \mathtt{-.}\, e : \mathtt{float}} \qquad \frac{\Gamma, \Delta \vdash e : \mathtt{bool}}{\Gamma, \Delta \vdash \mathtt{not}\ e : \mathtt{bool}}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \tau \quad op \in \{\mathtt{=}, \mathtt{<>}\}}{\Gamma, \Delta \vdash e_1\ op\ e_2 : \mathtt{bool}}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \tau \quad op \in \{\mathtt{<}, \mathtt{<=}, \mathtt{>}, \mathtt{>=}\} \quad \tau \in \{\mathtt{int}, \mathtt{float}\}}{\Gamma, \Delta \vdash e_1\ op\ e_2 : \mathtt{bool}}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \mathtt{int} \quad \Gamma, \Delta \vdash e_2 : \mathtt{int} \quad op \in \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{/}\}}{\Gamma, \Delta \vdash e_1\ op\ e_2 : \mathtt{int}}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \texttt{float} \quad \Gamma, \Delta \vdash e_2 : \texttt{float} \quad op \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{\Gamma, \Delta \vdash e_1 \ op \ e_2 : \texttt{float}}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \texttt{bool} \quad \Gamma, \Delta \vdash e_2 : \texttt{bool} \quad op \in \{\texttt{and}, \texttt{or}\}}{\Gamma, \Delta \vdash e_1 \ op \ e_2 : \texttt{bool}}$$

$$\frac{\Gamma, \Delta \vdash e_i : \sigma_i \quad f : (\tau_1, \ldots, \tau_n) \to \sigma \in \Delta \quad \sigma_1 * \cdots * \sigma_k = (\tau_1, \ldots, \tau_n)}{\Gamma, \Delta \vdash f\texttt{(}e_1\texttt{, } \ldots \texttt{,}e_k\texttt{)} : \sigma}$$

$$\frac{\Gamma, \Delta \vdash e_i : \tau_i}{\Gamma, \Delta \vdash \texttt{print(}e_1\texttt{, } \ldots \texttt{,}e_n\texttt{)} : \texttt{unit}}$$

$$\frac{\Gamma, \Delta \vdash e : \texttt{bool} \quad \Gamma, \Delta \vdash e_1 : \sigma \quad \Gamma, \Delta \vdash e_2 : \sigma}{\Gamma, \Delta \vdash \texttt{if} \ \ e \ \texttt{then} \ e_1 \ \texttt{else} \ e_2 : \sigma^{\not c}}$$

$$\frac{\Gamma, \Delta \vdash e_i : \tau_i}{\Gamma, \Delta \vdash \texttt{(}e_1\texttt{, } \ldots \texttt{,}e_2\texttt{)} : (\tau_1, \ldots, \tau_n)}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \sigma^{\mathsf{c}} \quad \Gamma, \Delta \vdash e : \sigma}{\Gamma, \Delta \vdash e_1 \ \texttt{fby} \ e_2 : \sigma^{\not c}}$$

$$\frac{\Gamma, \Delta \vdash e : \delta^{\mathsf{c}}}{\Gamma, \Delta \vdash e : \delta}$$

De même le jugement $\Gamma \vdash_p p : \sigma$ signifie « dans le contexte Γ, le motif p est bien typé et il a le type σ ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_p x : \tau} \qquad \frac{x_i : \tau_i \in \Gamma}{\Gamma \vdash_p (x_1, \ldots, x_n) : (\tau_1, \ldots, \tau_n)}$$

1.2 Primitives

Le langage Mini-lustre comporte un ensemble de primitives réparties en 4 groupes. Le premier regroupe quelques primitives de conversion des types de base,

```
float_of_int : int → float
int_of_float : float → int
float_of_string : string → float
int_of_string : string → int
bool_of_string : string → bool
```

Le second contient deux générateurs de nombres aléatoires et deux primitives de trigonométrie,

```
random_int : int → int
random_float : float → float
cos : float → float
sin : float → float
```

Le troisième regroupe quelques primitives graphiques,

```
draw_point : (int, int) → unit
draw_line : (int, int, int, int) → unit
draw_circle : (int, int, int) → unit
draw_rect : (int, int, int, int) → unit
fill_rect : (int, int, int, int) → unit
get_mouse : unit → (int,int)
```

Enfin, le quatrième contient deux primitives d'entrée/sortie : la fonction `read`, de signature `unit → string`, qui lit des caractères sur l'entrée standard jusqu'au prochain retour-chariot et retourne la chaîne de caractères correspondante, et la primitive `print`, dont la règle de typage particulière a été donnée dans la section précédente, accepte un n-uplet d'arguments de taille quelconque et a une valeur de retour de type `unit`.

1.3 Typage des noeuds

Les noeuds de Mini-lustre ont la forme suivante :

```
node n(x1:τ1;...;xn:τn) returns (o1:τ'1;ok:τ'k);
var y1:τ''1;...;ym:τ''m;
let
  p1 = e1;
     ⋮
  pq = eq;
tel
```

Les variables x_i sont les variables d'*entrées*, les variables o_i celles de *sorties* et enfin les variables y_i sont appelées variables locales. L'environnement Γ permettant de typer un noeud est construit uniquement à partir de ces trois sortes de variables, en associant chaque variable à son type. Le typage d'un noeud consiste alors à vérifier que :

- les variables d'entrées, de sorties et locales ont des noms disjoints deux à deux ;
- les variables de sorties et locales sont bien définies une et une seule fois par les équations du noeud (*i.e.* qu'elles apparaissent dans exactement un motif p_i) ;
- chaque expression e_i est bien typée et a le type du motif p_i ;
- toutes les variables dans les motifs p_i sont déclarées (comme des variables de sorties ou locales) et qu'elles sont distinctes deux à deux ;

Si le noeud est bien typé, on ajoute alors la déclaration $n : (\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_k)$ dans Δ.

1.4 Causalité

Le problème de la causalité a été présenté dans le document d'initiation au langage Mini-lustre. Il consiste à savoir si les équations d'un noeud peuvent être ordonnées (suivant un ordre partiel). Une méthode pour déterminer si un ensemble d'équations n'a pas de problème de causalité est de construire le graphe de dépendance entre les equations du noeud et de vérifier que ce graphe ne contient aucun cycle. Nous pouvons remarquer que les variables qui apparaissent à droite d'un `fby` n'introduisent pas de dépendances.

1.5 Surcharge

Les règles de la section 1.1 ne permettent pas de typer correctement l'équation $1+2.9$ ou encore $x + 3.4$ quand x a le type `int`. En d'autres termes, ces règles n'autorisent pas la surcharge des opérateurs arithmétique. Pour simplifier la syntaxe de notre langage, nous allons modifier la règle de typage des opérateurs `+`, `-`, `/` et `*` par la règle suivante :

$$\frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \Gamma, \Delta \vdash e_2 : \tau_2 \quad \tau 1, \tau_2 \in \{\texttt{int}, \texttt{float}\} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma, \Delta \vdash e_1 \ op \ e_2 : \tau_1 \cup \tau_2}$$

où $\tau_1 \cup \tau_2 = \texttt{int}$ si $\tau_1 = \tau_2 = \texttt{int}$ et `float` sinon.

2 Travail demandé

Adapter votre compilateur pour qu'il effectue la passe d'analyse sémantique, à l'issue de l'analyse syntaxique. En plus du fichier Mini-lustre en argument, votre compilateur devra reconnaître désormais deux options, `-parse-only` et `-type-only`. La première spécifie de s'arrêter juste après l'analyse syntaxique. La seconde est sans effet dans cette deuxième partie et son utilisation sera justifiée par la suite.

En plus des erreurs lexicales et syntaxiques, votre compilateur devra maintenant afficher les erreurs sémantiques et, le cas échéant, terminer avec le code de sortie 1 (`exit 1`). Les erreurs sémantiques seront signalées de la même manière que les erreurs lexicales et syntaxiques ; par exemple :

```
File "test.mls", line 21, characters 13-14:
this expression has type int but is expected to have type string
```

Le comportement en cas d'absence d'erreur ou d'erreur du compilateur ne change pas.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `compilo`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le 12 novembre à 10h00.

[1] `sylvain.conchon@lri.fr`, `stephane.lescuyer@lri.fr`, `louis.mandel@lri.fr`, `luidnel.maignan@inria.fr`, `benjamin.monate@cea.fr`