Lutin

Sémantique et compilation

Pascal Raymond, VERIMAG

Plan

Le langage

Systèmes

- **Lutin permet de décrire des systèmes réactifs indéterministes.**

  Un système est déclaré avec ses paramètres, par exemple :

  ```
  node toto (x:  int; y:  bool)
  returns (a,b:bool; c:int) =  trace-exp
  ```

- **Le corps *trace-exp* décrit les comportements possibles du système, sous la forme d'un "langage" dont les mots sont des *contraintes* sur les variables du programmes (dites variables *support*).**

Réactions

- **Un comportement atomique (une réaction indéterministe) du système est décrit par une contrainte sur les variables support du programme et leurs valeurs précédentes (`pre x`) :**
  alg-exp **::= une expression algébrique booléenne**
- **Il y a donc des variables :**
  - ⋆ **contrôlables (les sorties),**
  - ⋆ **incontrôlables (les entrées, les `pre`).**
- **Faire une réaction atomique, pour une valeur *donnée* des incontrôlables, c'est générer *aléatoirement* une valeur des sorties qui satisfait la contrainte.**
- **N.B. s'il n'y a pas de solution, le système bloque**

- **Exemple.** Si `x` est une entrée booléenne et `c` est une sortie réelle, "l'exécution de la réaction atomique :

  ```
  (x and (c <= 10.0) and (c > pre c))
  ```

  - ⋆ produit, si `x` est vraie et `pre c`, une valeur `c` aléatoire dans $]$`pre c`$, 10.0]$,
  - ⋆ sinon se bloque.

Enchaînement des réactions

- **Les réactions atomiques sont combinées avec des opérateurs inspirés des expressions régulières :**

  trace-exp ::= *trace-exp* **fby** *trace-exp*

  | **loop** *trace-exp*

  | **{***trace-exp* **|...|***trace-exp* **}**

- **Plus des constructions spécifiques :**

  ⋆ **assert** *alg-exp* **in** *trace-exp*

  distribue la contrainte *alg-exp* **(expression booléenne) dans le comportement** *trace-exp*

  ⋆ **exist** *ident* **:** *type* **in** *trace-exp*

  introduit une variable support contrôlable (sortie cachée)

  ⋆ **try** *trace-exp* **do** *trace-exp*

  si le pemier *trace-exp* **se bloque, passe le contrôle au deuxième**

Contrôle de l'indéterminisme

- **poids relatifs sur les choix (défaut 1) :**

  `{`*trace-exp* `weight` *w1* `|`**...** `|`*trace-exp* `weight` *wn* `}`

  où les *w* **sont des expressions entières** ***incontrôlables***

- **nombre de boucles contraint par un intervale :**

  `loop [`*min*`,`*max*`]`*trace-exp*

  où *min* **et** *max* **sont des expressions entières** ***statiques*****.**

- **nombre de boucles avec moyenne et écart type :**

  `loop ~`*moy*`:`*ec trace-exp*

  où *moy* **et** *ec* **sont des expressions entières** ***statiques*****.**

Boucles instantanées

- Le passage dans une boucle peut être instantané
- Pire: `loop loop` *c* boucle indéfiniment, sans rien faire, si *c* est insatisfiable
- Les programmes qui génèrent des boucles instantanées sont incorrect
- On peut les rejeter statiquement, sur des critères syntaxique, mais alors on en rejette beaucoup d'autres ...
- On peut accepter a priori tout programme et générer une run-time erreur en cas de problème.
  C'est ce que fait la sémantique opérationnelle de référence, mais ça peut varier d'une implémentation à une autre

Indéterminisme, blocage et probabilités

- **Réactivité : un choix ne bloque que *si toutes les possibilités* bloquent.**

- **N.B. la réactivité prévaut sur les poids :**

  {*t1* `weight` 1000000 |*t2* `weight` 1 }

  si *t1* bloque et *t2* non, alors *t2* est choisi.

- **Exemple : {*X* `weight 3 |` *Y* `weight 5 |` *Z* }**

  Les probabilités d'exécution sont, par exemple :

  - ⋆ **si aucun ne bloque, $3/9$ pour *X*, $5/9$ pour *Y*, $1/9$ pour *Z*,**
  - ⋆ **si *Y* bloque, $3/4$ pour *X*, $1/4$,**
  - ⋆ **si *X* et *Y* bloquent, 1 pour *Z***
  - ⋆ **si tous bloquent, le choix bloque.**

Indéterminisme, blocage et priorité

Même avec un poid très faible, une branche non bloquante a toujours une chance d'être choisie, d'où l'utilité d'un opérateur de *choix prioritaire*.

- **choix prioritaire (c.a.d "ou sinon") :**

  {*t1* |>*t2* |>... |>*tn* }

- **Exemple typique :** {*normal* |>*dégradé* |>*sauvetage* }

Parallélisme

- **Syntaxe :**

  { *trace-exp* &> ... &> *trace-exp* }

- **Tout au long de l'exécution, chaque branche produit sa propre contrainte ; la conjonction donne la contrainte globale.**

- **Si une branche termine (normalement) les autres continuent (cf. parallèle Esterel).**

- **Le tout termine si et quand le dernier termine.**

- **Si (au moins) une branche bloque, le tout bloque.**

Parallélisme et probabilités

Ils se marient très mal :

```
{ {X weight 1000 |Y } &> {A weight 1000 |B } }
```

Si *X* et *A* sont non bloquants, mais pas leur conjonction :

- **le plus probable peut être *X*&>*B*, ce qui “lèse” le premier ,**
- **ou bien *X*&>*B*, ce qui “lèse” le deuxième.**

Choix sémantique :

- **la première branche fait son choix en priorité,**
- **le deuxième essaie de faire avec, etc.**
- **i.e., le traitement des poids est séquentiel.**
- **N.B. La notation &> insiste sur le fait que le traitement n’est pas commutatif.**

Exceptions

Permettent de détourner le flot de contrôle, tiennent des exceptions classiques (caml, java), et des signaux de trap d'Esterel.

- **Déclation/portée :**

  `exception` *ident* `in` *trace-exp*

  les exceptions globales, déclarées en dehors du *node* **sont par définition globales et non-masquables.**

- **Levée :** `raise` *ident*

- **Point de rattrapage :**

  `catch` *ident* `in` *t1* `do` *t2*

  si *ident* **est levée dans** *t1*, **le contrôle passe immédiatement à** *t2*.

- **Raccourci :** `trap` *x* `in` *t1* `do` *t2*

  pour : `exception` *x* `in` `catch` *x* `in` *t1* `do` *t2*

- **Blocage : le blocage est vu comme la levée d'une exception *prédéfinie, non-masquable, non émissible*.**

  `catch DeadLock in` *t1* `do` *t2*

  est équivalent à : `try` *t1* `do` *t2*

- **Exception et parallèlisme :**
  - ⋆ **il n'y a pas de multi-émission,**
  - ⋆ **comme pour les poids, le traitement des** `raise` **est séquentiel, de gauche à droite.**
  - ⋆ **ex.** `{raise E &>`*X*`}` **équiv. à** `raise E`
  - ⋆ **ex.** `{`*X* `&>`*Y* `&>raise X }` **équiv. à** `{{`*X* `&>`*Y* `}|>raise X }`

Modularité

Le langage propose une couche “fonctionnelle” qui permet :

- **de factoriser les définitions,**
- **de définir et réutiliser ses propres opérateurs, que ce soit sur les données ou les comportements.**
- **On défini un type (abstrait) `trace` pour caractériser les combinateurs de comportements (et leurs paramètres)**
- **La sémantique (n’)est définie (que) par substitution (on parle de macro, plutôt que de fonction).**

- **Une macro peut être déclarée glogalement (en dehors d'un `node`) :**

  `let` *ident* `(` *params* `):` *type* `=` *exp*

  exp **est** *trace-exp* **ou** *data-exp*, **selon le** *type*.

- **ou déclarée inline dans une** *trace-exp* **:**

  `let` *ident* `(` *params* `):` *type* `=` *exp* `in` *exp*

  auquel cas les règles classiques de portée s'appliquent.

- **Les** *params*, **le** *type* **sont optionnels.**

- **Attention à ne pas mélanger les variables support et les macros sans paramètres (appelés *alias*).**

Exemple commenté

```
let assert_init (
-- prend en entrée une expression booléene...
-- et une expression de trace (sur n'importe quel support)
(init: bool; t: trace)
-- et renvoie une trace (sur le même support)
: trace = trap Stop in {
            -- cast implicite bool ⟶ trace de longeur 1
            init
          &>
            -- si t termine immédiatement, on n'impose pas init
            t fby raise Stop
          }
```

Paramètres et variables support

- Le type `trace` est très abstrait : quid des variables support ?
- En fait, ca n'a généralement aucune importance :
  les opérateurs de traces, définis ou prédéfinis sont polymorphes.
- Si on tient à spécifier qu'on va faire référence à une variable support dans une macro, on peut tout de même *sur-spécifier* son type :
  x `:` *type* `ref`
  Dans ce cas, à l'appel, on vérifie que l'argument est bien une variable support.
- N.B. le `ref` n'est vraiment nécessaire que si on veut utiliser un `pre` dans la macro :

```
let foo (x: bool)= ... pre x ... -- ERREUR DE TYPE
let foo (x: bool ref)= ... pre x ... -- OK
```

Exemples

Le combinateur parallèle qui termine dès qu'une branche termine :

```
let race (X,Y:trace )= {
    trap End in {
        X fby raise End
    &>
        Y fby raise End
    }
}
```

Le front-end du compilateur

Type/binding check

- très classique

Expansion

- Dans un langage intermédiaire noyau.
- Pas forcément nécessaire, mais une compilation modulaire serait très sophistiquée et nécessiterait un exécutif complexe à la ml (on est “un peu” dans l’ordre supérieur).

La sémantique opérationnelle est formellement définie sur le langage noyau dont on va voir la syntaxe abstraite.

Syntaxe abstraite

Les expressions de trace (comportements) sont :

vide :	ε	**filtre du vide :**	$t \setminus \varepsilon$
contrainte:	c	**catch :**	$[t \overset{x}{\hookrightarrow} t']$
levée :	$\uparrow^x$	**choix :**	$t/w \mid t'/w'$
séquence :	$t \cdot t'$	**boucle contrainte :**	$t_i^{(\omega_c,\omega_s)}$
priorité :	$t \succ t'$	**boucle prioritaire :**	t^*
parallèle :	$t \,\&\, t'$		

- **ε et $t \setminus \varepsilon$ n'existe pas dans la syntaxe concrète mais sont bien pratiques pour difinir la sémantique.**
- **La syntaxe abstraite de la boucle contrainte sera expliquée plus loin.**

Sémantique

Environnement abstrait

Le problème de la résolution des contraintes, ainsi que celui des poids et de la sélection aléatoire sont “confiés” à l’environnement.

Pour un environnement abstrait e on utilise les prédicats suivants :

- $e \models c$ **vrai ssi** c **est satisfiable dans e**
- $e : w = 0$ **le comportement associé au poids** w **est interdit.**
- $e : w \succ w'$ **le comportement associé au poids** w **est prioritaire sur celui associé à** w'

Action atomique

- L'exécution d'une trace t dans un environnement e ($Run(e,t) = \alpha$), produit une action α qui peut être :
  - $\star$ $\stackrel{c}{\longrightarrow}$n : transition normale, la contrainte c (réalisable) est produite et la trace se réécrit en n.
  - $\star$ $\int^{x}$: terminaison décorée par un indicateur qui peut être
    - $*$ ε, terminaison normale,
    - $*$ δ, deadlock (contrainte insatisfiable),
    - $*$ $\emptyset$, run-time erreur fatale (boucle instantanée),
    - $*$ une exception programmée pas l'utilisateur
- Étant donnée une contrainte *satisfiable* c, un environnement e se réécrit (après tirage aléatoire, mémorisations etc.) en un environnement suivant e' : $e \stackrel{c}{\longrightarrow} e'$.

Enchaînement des actions

L'exécution d'une trace t_0 dans un environnement intitial e_0 est définie comme une séquence d'environnements :

$(e_0, e_1, \cdots, e_n)$

avec :

- $\exists c_0, \cdots, c_{n-1}\ \exists t_1, \cdots, t_n$ **tels que**
- $\forall i = 0 \cdots n-1$

  $Run(e_i, t_i) = \xrightarrow{c_i} t_{i+1}$ **et** $e_i \xrightarrow{c_i} e_{i+1}$
- **et** $\exists x \quad Run(e_n, t_n) = \updownarrow^x$

La fonction sémantique

- **Le fait d'avoir "caché" les problèmes de résolution et de tirage aléatoire permet de définir un Run parfaitement déterministe.**
- **Run est définie de manière inductive via une fonction $\mathcal{R}_e$ (*run dans e*) qui prend, en plus de la trace à traiter t, 2 fonctions *de continuations* qui rendent des actions :**
  - ⋆ **$g(c, n)$ est appelée pour traiter le cas des transitions (goto),**
  - ⋆ **$s(x)$ est appelée pour traiter le cas d'une termination décorée par l'indicateur x.**

La sémantique top-level

Au top-level, on a $Run(e,t) = \mathcal{R}_e(t,g,s)$ avec simplement :

- $g(c,n) = \xrightarrow{c} n$
- $s(x) = \uparrow^x$

Vide

$$\mathcal{R}_e(\varepsilon, g, s) = s(\varepsilon)$$

Levée d'une exception

$$\mathcal{R}_e(\uparrow^x, g, s) = s(x)$$

Contrainte

C'est là qu'on teste la satisfiabilité:

$$\mathcal{R}_e(c, g, s) = (e \models c)?\ g(c, \varepsilon)\ :\ s(\delta)$$

Séquence

$\mathcal{R}_e(t \cdot t', g, s) = \mathcal{R}_e(t, g', s')$ **avec :**

- $g'(c, n) = g(c, n \cdot t')$
- $s'(x) = (x = \varepsilon)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$

Choix prioritaire

$\mathcal{R}_e(t \succ t', g, s) = \mathcal{R}_e(t, g, s')$ **avec :**

- $s'(x) = (x = \delta)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$

Boucle prioritaire

- **on utilise le filtre du vide, qui lève une erreur fatale en cas de terminaison normale :**

  $\mathcal{R}_e(t \setminus \varepsilon, g, s) = \mathcal{R}_e(t, g, s')$ **avec :**

  $\star\ s'(x) = (x = \varepsilon)?\ \uparrow^{\emptyset}\ :\ s(x)$

- **La boucle se ramène à un choix prioritaire :**

  $t^* \quad = \quad (t \setminus \varepsilon) \cdot t^* \quad \succ \quad \varepsilon$

Rattrapage

N.B. par construction, ne concerne que δ et les exceptions utilisateur.

$\mathcal{R}_e([t \overset{z}{\hookrightarrow} t'], g, s) = \mathcal{R}_e(t, g', s')$ **avec :**

- $g'(c, n) = g(c, [n \overset{z}{\hookrightarrow} t'])$
- $s'(x) = (x = z)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$

Parallèle

$\mathcal{R}_e(t \,\&\, t', g, s) = \mathcal{R}_e(t, g', s')$ **avec :**

- $s'(x) = (x = \varepsilon)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$
- $g'(c, n) = \mathcal{R}_e(t', g'', s'')$ **avec :**
  - $s''(x) = (x = \varepsilon)?\ g(c, n)\ :\ s(x)$
  - $g''(c', n') = g(c \wedge c', n \,\&\, n')$

Choix pesant

La résolution de l'indéterminisme est "confiée" à l'environnement. Il faut tenir compte du poids nul qui équivaut à une interdiction.

$\mathcal{R}_e(t/w \mid t'/w', g, s) =$

- $r(\delta)$ **si** $(e : w = 0)$ **et** $(e : w' = 0)$**, sinon**
- $\mathcal{R}_e(t, g, s)$ **si** $(e : w' = 0)$**, sinon**
- $\mathcal{R}_e(t', g, s)$ **si** $(e : w = 0)$**, sinon**
- $\mathcal{R}_e((e : w \succ w')?\ t \succ t'\ :\ t' \succ t, g, s)$

Boucle contrainte

- Les fonctions de poids calculent selon le nombre d'itérations déjà effectuées i :
  - ⋆ le poids de "continuer" $\omega_c(i)$
  - ⋆ le poids de "stopper" $\omega_s(i)$
- Elles sont déterminées statiquement (nature et arguments du `loop`).
- Le nombre d'itérations révolues est syntaxiquement attaché à la boucle (0 partout dans la trace principale).
- La syntaxe abstraite d'une boucle est donc : $t_i^{(\omega_c,\omega_s)}$
- On a :

$$t_i^{(\omega_c,\omega_s)} \quad = \quad (t \setminus \varepsilon) \cdot t_{i+1}^{(\omega_c,\omega_s)} / \omega_c(i) \quad | \quad \varepsilon / \omega_s(i)$$

Le back-end du compilateur

Interprétation

Génération des contraintes et résolution sont des problèmes othogonaux :

- **La génération des contraintes correspond exactement à la sémantique qui a été présentée.**
- **Le solveur utilisé actuellement est celui de l'outil de test Lucky/Lurette : c'est un solveur mixte basé sur des BDDS et une librairie de polyèdres.**

Compilation en automate

- **Génère un automate dans le format utilisable par Lucky/Lurette.**
- **La génération d'automate suit (presque) à la ligne la sémantique opérationnelle :**
  - ⋆ **on calcule statiquement tous les choix possibles (les états étant les dérivations du programme initial) ;**
  - ⋆ **la terminaison est garantie car le nombre de dérivations est fini,**
  - ⋆ **de plus, les boucles instantanées sont "cassées" (pas de run-time error)**
  - ⋆ **le traitement du blocage est simplifié car la "machine cible" (Lucky) prend en charge son traitement de manière native.**

Quelle est la suite ?

- **La compilation en automate plat n'est pas très satisfaisante (produit d'automates = explosion). Une compilation en automates parallèle serait plus efficace, mais demande une évolution importante de la machine cible.**
- **Au niveau langage, la prochaine version doit permettre d'écrire des définition de *traces récursives terminales* ; intuitivement, il s'agit de proposer un style de programmation en automates "explicites".**