Librairie pour le calcul GSL
Projet

Samuel Colin

juin 2000

Table des matières

1 Introduction **3**

2 Préliminaires et rappels **4**
- 2.1 Calcul des prédicats 4
- 2.2 Démonstration automatique et substitutions 4

3 Les substitutions généralisées **6**
- 3.1 Origine de la définition des GSL 6
- 3.2 Syntaxe des GSL et sémantique associée 7

4 Fonctions de manipulation des GSL **8**
- 4.1 Terminaison d'une substitution 8
- 4.2 Faisabilité d'une substitution 9
- 4.3 Remarque intermédiaire 10
- 4.4 Prédicat avant-après 10

5 Outils utilisés **12**
- 5.1 Objective Caml 12
- 5.2 LaTeX 12
- 5.3 *HeVeA* 13
- 5.4 OCamlWeb 13

A Grammaire des GSL sous forme BNF **14**

B Exemple d'exécution **19**

Bibliographie **24**

Chapitre 1

Introduction

L'unité ESTAS [1] de L'INRETS [2] a initié depuis déjà quelques années le développement d'outils pour permettre aux utilisateurs de la méthode B (cf [Mariano97]) pour une description des méthodes formelles en général et de la méthode B en particulier) de couvrir la plupart de leurs besoins logiciels pour la bonne conduite de projets nécessitant cette méthode. Il est à noter également l'existence d'un autre projet de cette nature, [Atelier B] , de nature plus commerciale.

C'est dans ce contexte, dynamique s'il en est, que s'inscrit ce projet : il vise de façon relativement modeste, à apporter aux développements B de l'INRETS des outils de manipulation relatifs à l'un des piliers de la méthode B, les preuves de programme.

Ce projet a pour but d'étudier et mettre en lumière les propriétés d'un sous-ensemble du langage B (plus précisément de la notation en machines abstraites), les substitutions généralisées, utilisées lors de la spécification de l'une de ces machines. [3]

C'est dans cette optique que nous définirons une grammaire pour ces substitutions généralisées, ainsi que quelques fonctions de manipulation dont nous préciserons le sens au moment opportun.

[1] Evaluation des Systèmes de Transports Automatisés et leur sécurité

[2] INRETS :Institut National de Recherche sur les transports et leur sécurité

[3] Nous n'utiliserons dans la suite que le terme GSL (Generalized Substitutions Language) pour désigner les substitutions généralisées, pour des raisons purement pratiques.

Chapitre 2

Préliminaires et rappels

Dans ce chapitre, nous mentionnerons, sans entrer dans les détails, les supports théoriques nécessaires à la compréhension de ce qui sera expliqué lors des chapitres suivants.

2.1 Calcul des prédicats

Le calcul des prédicats est défini dans un domaine appelé «système formel». Celui-ci est composé d'ensembles définis comme suit :

- Un ensemble V de variables
- Un ensemble C de constantes
- Un ensemble F de fonctions qui à une ou plusieurs variables associent une valeur fonction de ces variables
- Un ensembles de relations appelées *prédicats*, qui à une ou plusieurs variables associent une valeur booléenne.

Par exemple, la relation d'infériorité $<$ est un prédicat reliant deux variables x et y (pouvant être des entiers, des réels, des chaînes de caractères, ...) et renvoyant la valeur *vrai* si x est inférieur à y , *faux* sinon.

2.2 Démonstration automatique et substitutions

L'utilité du calcul des prédicats est de fournir une base pour pouvoir développer des méthodes permettant de savoir si une formule donnée du calcul des prédicats est toujours vraie (on dit alors que c'est un *théorème*).

C'est ici qu'interviennent les substitutions : elles ont été définies la première fois pour l'une de ces méthodes, appelée méthode de résolution par unification.

On utilisait un ensemble appelé *ensemble des Termes*.Celui-ci contenait toutes les compositions possibles sur les ensembles V, C et F associés à la formule que l'on souhaitait démontrer (voir 2.1) , c'est-à-dire des fonctions de fonctions de variables, des fonctions de constantes, etc.

La formule était mise sous forme d'un ensemble de sous-formules, pour chacune desquelles on pouvait définir une ou plusieurs substitutions selon certains critères, de façon à démontrer si la formule entière était ou non un théorème. Qu'est-ce alors qu'une substitution ? Tout simplement le remplacement d'une variable par une valeur qu'elle *peut* prendre. Le lecteur intéressé pourra se référer à tout bon livre de logique, ou par exemple à [Andreas Herzig et al.].

Chapitre 3

Les substitutions généralisées

3.1 Origine de la définition des GSL

Devant les besoins de plus en plus pressants des entreprises en logiciels dits «critiques» ,furent développées des méthodes de conception et de validation des logiciels. L'une d'elles, très récente, est la *méthode B* (cf [Abrial 96]). L'un des points importants de telles méthodes est bien évidemment de pouvoir faire la preuve de la correction du programme en cours de conception. La méthode B se base sur la notation en machines abstraites (AMN) : ces machines sont de «petits bouts de programme» définis par leurs fonctionnalités, les variables qu'ils utilisent, les valeurs qu'ils renvoient, ainsi que les propriétés associées...

En résumé, une machine abstraite est une entité indépendante qui réalise un certain nombre d'actions, et dont les propriétés des variables manipulées sont bien définies. Ceci permet une forte corrélation entre le code d'un programme et sa correction, ce qui est recherché pour la conception de logiciels critiques.

Que peuvent bien avoir les GSL avec les machines abstraites, telle est la question que peut légitimement se poser le lecteur.La réponse vient juste après : l'AMN est basé sur les GSL.Plus précisément, l'AMN est une forme «étendue» de la notation des GSL : on dit plus volontiers que l'AMN est une notation GSL avec plus de *sucre syntaxique* [1].

Ici encore , le lecteur peut se demander alors quel est le rapport entre les GSL et les substitutions du calcul des prédicats.Celles-ci n'ont finalement en commun que leur sens mathématique : le remplacement d'une variable par une expression, et leur utilisation pour le calcul des prédicats (car les GSL sont utilisées pour pouvoir opérer une preuve du programme facilement). Nous verrons en effet plus loin que les prédicats sont intimement liés aux propriétés des opérations exprimées sous forme de GSL.

[1] C'est du moins le jargon utilisé

3.2 Syntaxe des GSL et sémantique associée

Soient x et z des variables, E une expression, P un prédicat, S et T des substitutions généralisées.

Une substitution appliquée à un prédicat s'écrit sous la forme :

$[S]P$, ce qui signifie que l'on applique au prédicat P les opérations spécifiées par S.

Voici les 6 expressions GSL de base qui permettent de construire n'importe quelle machine abstraite (ces expressions sont censées évidemment s'appliquer à un prédicat) :

1. $x := E$
2. $skip$
3. $P|S$
4. $S[]T$
5. $P \Longrightarrow S$
6. $@z.S$

Et le sens qui correspond à chacun de ces types de substitutions :

1. x est remplacé par l'expression E dans le prédicat auquel la substitution s'applique.
2. il ne se passe rien (on peut dire que $skip$ est l'élément neutre des GSL).
3. si le prédicat P est vrai alors on peut appliquer la substitution S (P est une précondition pour la substitution S).
4. une des deux substitutions S ou T doit être appliquée (mais lors de la preuve de programme il faudra tester l'application des 2 séparément).
5. si le prédicat P est vérifié, alors appliquer la substitution S , sinon arrêter (l'exécution du programme).

   Cette substitution s'appelle la *garde*.La distinction avec la précondition est subtile : la précondition est une assertion servant à la preuve de programme(donc utilisée à la compilation), et la garde serait plutôt un «garde-fou» à l'exécution.Elle est utilisée dans la notation AMN comme une sélection au cas par cas (ex. : selon que x vaut 1, faire telle opération, x vaut 2 faire telle opération, x vaut une autre valeur faire telle autre opération).
6. pour chaque valeur que peut prendre la variable z, appliquer la substitution S, moyennant que z ne soit pas une variable libre dans le prédicat auquel s'applique la substitution.

Une grammaire des GSL exprimée sous forme de Backus Naur se trouve page 14.
Il est à noter que la forme GSL exprimant l'universalité est annotée comme ne possédant qu'une seule variable. Si l'on souhaite appliquer une substitution généralisée à plusieurs variables en même temps , il suffira d'utiliser la récursivité de la grammaire pour cette règle.

Chapitre 4

Fonctions de manipulation des GSL

Nous décrirons dans ce chapitre comment déterminer les propriétés d'une expression GSL, c'est-à-dire les prédicats dont la résolution permettra d'indiquer sa terminaison, sa faisabilité ainsi que sa transition avant-après.

Etant donné les explications spartiates qui sont actuellement données dans les quelques références traitant de ces fonctions, nous nous efforcerons de donner un sens plus "concret" à celles-ci lors de leur définition.

Nous reprendrons ici les notations et variables définies dans le chapitre précédent.

4.1 Terminaison d'une substitution

Pour comprendre sans ambiguïté ce qui va suivre, il faut se rappeler qu'une substitution en notation AMN ,est également une suite d'opérations appliquées à une ou plusieurs variables.

La terminaison, pour parler français, permet d'indiquer si une substitution peut se "terminer", c'est-à-dire si son exécution *permet* d'obtenir un résultat.On ne cherche pas à démontrer que la substitution renvoie *tel* résultat, mais simplement qu'elle *renvoie effectivement* un résultat.

La terminaison peut être également définie en termes ensemblistes : la terminaison d'une substitution S est l'ensemble des états (des variables contribuant à S) qui permettent à S d'aboutir (en fait l'ensemble des états répondant aux critères définis ci-avant).

Ces deux façons de définir la terminaison sont bien entendu équivalentes.

Dans la littérature, il existe deux définitions plus mathématiques de cette fonction toutes deux ayant bien entendu le même sens :

1. $trm(S) \equiv [S]true$
2. $trm(S) \equiv [S](x = x)$

Pour corroborer la définition «informelle» donnée plus haut, on voit ici que S se termine si elle *permet* d'établir un prédicat toujours vrai (ce qui est en soi cohérent).Si $trm(S)$ était faux, cela signifierait que l'exécution de S n'aboutit pas, même si ce qui est requis pour S est un prédicat qui n'a aucun effet (si on requiert ,dans un programme AMN, quelque chose de toujours vrai, cela signifie que toute latitude est donnée à l'exécution de S). Donc la définition informelle correspond bien à la définition mathématique.

Maintenant, le lecteur rigoureux aura remarqué l'étrangeté de la première définition formelle : $true$.En effet, ici l'accent est donné au *sens* de la terminaison. Le logicien contestera que $true$ n'est pas un prédicat , et c'est la raison pour laquelle est apparue la seconde forme de la définition formelle, plus rigoureuse.

Là encore, on pourra protester car la variable x n'est pas forcée d'intervenir dans la substitution S. Qu'à cela ne tienne, définissons une forme qui pourra mettre tout le monde d'accord : soit $Vrai()$ le prédicat d'arité 0 dont la valeur booléenne est toujours «vrai», alors la définition de la terminaison devient : $trm(S) \equiv [S]Vrai()$

Les définitions (récursives) de la terminaison pour les GSL sont les suivantes :

$trm(x := E)$	$Vrai()$
$trm(skip)$	$Vrai()$
$trm(P\|S)$	$P \wedge trm(S)$
$trm(S[]T)$	$trm(S) \wedge trm(T)$
$trm(P \Longrightarrow S)$	$P \Rightarrow trm(S)$
$trm(@z.S)$	$\forall z.trm(S)$

4.2 Faisabilité d'une substitution

Maintenant que certaines aberrations syntaxiques et «flous sémantiques» ont été levés, la compréhension de cette section ne devrait pas poser trop de problèmes.

Informellement, la faisabilité d'une substitution (couramment notée fis) dénote sa capacité à ne pas renvoyer de résultat incohérent.

La définition de cette propriété est plus «parlante» en termes ensemblistes : la faisabilité d'une substitution S est l'ensemble des états (des variables participant à S) pour lesquels S n'*avorte pas*.

Ce qui, accordons-nous bien, n'est pas tout à fait la même chose que la terminaison.La section suivante traitera rapidement du pourquoi et du comment de cette curiosité sémantique.

Voici le petit tableau indiquant la définition récursive de la faisabilité :

$fis(x := E)$	$Vrai()$
$fis(skip)$	$Vrai()$
$fis(P\|S)$	$P \Rightarrow fis(S)$
$fis(S[]T)$	$fis(S) \vee fis(T)$
$fis(P \Longrightarrow S)$	$P \wedge fis(S)$
$fis(@z.S)$	$\exists z.fis(S)$

4.3 Remarque intermédiaire

Les deux fonctions précédentes semblent identiques de par leurs définitions. La raison de leur différence tient principalement à l'existence de la garde, un prédicat qui n'est vérifiable qu'à l'exécution. Un petit exemple valant mieux qu'un long discours(la signification de $false()$ est triviale) :

1. $false() \Longrightarrow skip$
2. $false()|skip$

Et l'interprétation :

1. La substitution se termine, mais est infaisable : l'exécution ne «passera» jamais par le prédicat $false()$.
2. La substitution est faisable, mais ne se termine pas : la précondition $false()$ n'est jamais vérifiée.

En résumé, la terminaison indique si la façon dont se construit une solution est bonne (vraie), et la faisabilité indique quel genre de solutions permettent de terminer la substitution (ou de l'exécuter).

4.4 Prédicat avant-après

Cette fonction est un peu différente des deux précédentes.En effet, alors que les terminaison et faisabilité sont des fonctions démonstratives , le prédicat avant-après se veut descriptif.

Le prédicat avant-après sera noté *prd* dans la suite, pour des raisons purement pratiques.

Possédant la donnée de la variable (ou des variables) pour lesquelles on souhaite connaître le *prd*, ainsi qu'une substitution S, le *prd* donne le prédicat indiquant toutes les valeurs possibles de la variable après exécution de S (ce qui, en passant, explique sa forte ressemblance avec fis). Il est défini comme suit :

$$\begin{array}{ll} \mathrm{prd}_x(S) \equiv \neg[S](x' \neq x) & \\ \mathrm{prd}_{x,y}(S) \equiv \neg[S](x', y' \neq x, y) & \text{s'il y a plusieurs variables} \end{array}$$

La valeur *«après»* d'une variable est usuellement adjointe d'un *«'»*. Petit exemple :

$$\begin{array}{ll} \mathrm{prd}_x(x := x+1) & \\ \quad\Leftrightarrow & \neg(x' \neq x+1) \\ \quad\Leftrightarrow & x' = x+1 \end{array}$$

A noter que la dernière transformation est esthétique et non nécessaire. Une autre de façon de voir les choses serait de dire que *prd* indique toutes les transformations possibles de la variable à laquelle il s'applique, pour que l'exécution n'avorte pas.

Le même exemple s'appliquant à plusieurs variables(x et y) :

$$\begin{array}{ll} \mathrm{prd}_{x,y}(x := x+1) & \\ \quad\Leftrightarrow & \neg[x := x+1](x', y' \neq x, y) \\ \quad\Leftrightarrow & \neg(x', y' \neq x+1, y) \\ \quad\Leftrightarrow & (x', y' = x+1, y) \\ \quad\Leftrightarrow & ((x' = x+1) \wedge (y' = y)) \end{array}$$

Ici également, noter que les deux dernières transformations sont purement esthétiques. Enfin, le tableau de définition récursive du prédicat avant-après :

$prd_x(x := E)$	$x' = E$
$prd_{x,y}(x := E)$	$x', y' = E, y$
$prd_x(skip)$	$x' = x$
$prd_x(P\|S)$	$P \Rightarrow prd_x(S)$
$prd_x(S[]T)$	$prd_x(S) \vee prd_x(T)$
$prd_x(P \Longrightarrow S)$	$P \wedge prd_x(S)$
$prd_x(@z.S)$	$\exists z.prd_x(S)$ si z \ x'

le «\» signifie que la variable z n'est pas libre dans x'.Autrement dit, que z fait partie de l'expression permettant de calculer x'.

Chapitre 5

Outils utilisés

5.1 Objective Caml

Habituellement désigné sous l'acronyme plus court d'OCaml, il s'agit du langage ayant permis de réaliser la partie implémentation du projet.

Outre ses capacités en tant que langage fonctionnel, c'est un langage facilitant grandement l'écriture d'analyseurs (lexicaux et syntaxiques), et un outil idéal pour manipuler des structures de données récursives (telles que les listes, les arbres, etc).

Il implémente également tout ce que tout programmeur peut être en droit d'attendre aujourd'hui : programmation impérative, concurrence et parallélisme, ainsi que calcul distribué (en cours de développement).

Plus de renseignements et de précisions pourront être trouvés dans [Leroy 99].

5.2 LaTeX

Sans entrer dans les détails ici, LaTeX est un outil de formatage de texte : plutôt que préparer l'apparence du texte final par visualisation directe ("what you see is what you get"), on insère les commandes de mise en forme dans le texte lui-même. La plus adéquate définition que j'aie pu lire à propos de LaTeX est la suivante : "LaTeX permet de mettre en forme un texte comme on le pense plutôt que comme on le voit" (c'est une traduction un peu bancale, certes, mais ô combien exacte).

Dans la pratique, si LaTeX rebutera ceux qui ne veulent même pas savoir comment fonctionne un ordinateur, il fera en revanche le bonheur des programmeurs chevronnés de tous poils qui n'ont pas envie de se casser la tête pour savoir quel sous-menu il faut sélectionner pour insérer un tableau (ce n'est bien entendu qu'un exemple...). Quelques références pourront satisfaire la curiosité du lecteur interpellé : [Leslie 94] et [LaTeX Companion].

5.3 *HeVeA*

HeVeA est un outil prenant en entrée un fichier écrit avec LaTeX et écrit en résultat un fichier HTML, visualisable directement par n'importe quel butineur web («browser» en anglais). Rien d'étonnant à cela, puisque le format HTML est déjà en soi un langage de formatage de texte.

Le (ou les) fichier(s) HTML résultant est hiérarchisé selon les chapitres , les sections, etc et les références (bibliographiques, de bas de page ,...) sont représentées sous forme de liens hypertextes.

L'avantage évident de cet outil est de pouvoir publier facilement un rapport, un article écrit en LaTeX sur la toile mondiale, ainsi que de cliquer au lieu de devoir feuilleter plusieurs pages avant de trouver ce que l'on cherche (ce qui est éminemment moins fatiguant... mais nécessite un ordinateur).

Voir [*HeVeA*] pour plus d'informations

5.4 OCamlWeb

Cet outil est ce que les anglo-saxons appelleraient un outil de *literate programming*. Ceci se traduit difficilement de façon aussi courte : il s'agit de programmer et de documenter son code en même temps , pour réunir le programme et sa signification en un même emplacement.

A cela, deux avantages :

1. Un gain de temps, parfois précieux, pour produire et le programme et son explication / but / fonctionnement.
2. Une obligation pour le programmeur (qui, de manière générale, n'aime pas documenter) d'être plus rigoureux dans son codage, voire même plus clair, toutes bonnes habitudes utiles par la suite.

Ocamlweb demande en entrée un fichier CAML, dont les commentaires ont un format un peu différent du format CAML habituel, et présente en sortie un fichier LaTeX où les commentaires correspondront à des sections entières du programme, séparées selon le bon sens du programmeur.Un exemple en est donné après l'annexe A.

Annexe A

Grammaire des GSL sous forme BNF

```
gsl_substitution    ::=  <gsl_affect>
                     |   <gsl_skip>
                     |   <gsl_precondition>
                     |   <gsl_bounded_choice>
                     |   <gsl_guard>
                     |   <gsl_forall>

gsl_affect          ::=  <variable> " :=" <expression>

gsl_skip            ::=  "skip"

gsl_precondition    ::=  <predicate> "|" <gsl_substitution>

gsl_bounded_choice  ::=  <gsl_substitution> "[]" <gsl_substitution>

gsl_guard           ::=  <predicate> "⟹" <gsl_substitution>

gsl_forall          ::=  "@" <variable> "." <gsl_substitution>

variable    ::=  <identifier>

expression  ::=  <original language-dependent expression>

predicate   ::=  <expression from the predicates' calculus>
```

Module GSL

1. module $GSL =$

struct

Les types var et expr sont définis de cette façon car le but ici est de montrer une implémentation des GSL , sans pour autant devoir se compliquer la vie... Le type prédicat est fortement inspiré du typage de l'arbre abstrait de BLAST (B language abstract syntax tree) , bien que le prédicat *true()* n'y figure pas

```
type var =string

type expr =string

type pred =
  | Pred_equal of pred × pred
  | Pred_notequal of pred × pred
  | Pred_var of var
  | Pred_expr of expr
  | Pred_forall of var × pred
  | Pred_exists of var × pred
  | Pred_and of pred × pred
  | Pred_or of pred × pred
  | Pred_not of pred
  | Pred_implies of pred × pred
  | Pred_equiv of pred × pred
  | Pred_true
```

2. Les trois fonctions qui suivent permettent de faire une décompilation "propre" permettant de faire presque immédiatement la correspondance entre une GSL , et le prédicat qui résulte de l'application de l'une des fonctions

```
let rec dc_var variable = variable

let rec dc_expr expression = expression

let rec dc_pred predicate =
  match predicate with
  | Pred_equal(p1,p2) →
      "(" ^ dc_pred p1 ^ "=" ^ dc_pred p2 ^ ")"
  | Pred_notequal(p1,p2) →
      "(" ^ dc_pred p1 ^ "=/=" ^ dc_pred p2 ^ ")"
  | Pred_var(x) →
      dc_var x
  | Pred_expr(e) →
      dc_expr e
  | Pred_forall(x , p) →
```

```
      "A !" ^ dc_var x ^ "/" ^ dc_pred p
  | Pred_exists(x , p)  →
      "E !" ^ dc_var x ^ "/" ^ dc_pred p
  | Pred_and(p1,p2)  →
      "(" ^ dc_pred p1 ^ "␣AND␣" ^ dc_pred p2 ^ ")"
  | Pred_or(p1,p2)  →
      "(" ^ dc_pred p1 ^ "␣OR␣" ^ dc_pred p2 ^ ")"
  | Pred_not(p)  →
      "NOT(" ^ dc_pred p ^ ")"
  | Pred_implies(p1,p2)  →
      "(" ^ dc_pred p1 ^ "␣=>␣" ^ dc_pred p2 ^ ")"
  | Pred_equiv(p1,p2)  →
      "(" ^ dc_pred p1 ^ "␣<=>␣" ^ dc_pred p2 ^ ")"
  | Pred_true  →
      "TRUE"
```

3. La définition du *type gsl* est extremement simple , mais malheureusement est très dépendante du langage dont elle dérive(B en l'occurrence) . Une première solution se voulait indépendante des types prédicats définis plus haut , mais cela revenait quasiment à utiliser un type absolu , car il fallait faire des suppositions sur les composantes du *type prédicat* (il contient *Pred_and* , et non *PredAnd* ,...)

```
type gsl =
  | Gsl_affect of var × expr
  | Gsl_skip
  | Gsl_precondition of pred × gsl
  | Gsl_bounded_choice of gsl × gsl
  | Gsl_guard of pred × gsl
  | Gsl_forall of var × gsl
```

4. Les fonctions de manipulation du type GSL : *trm* et *fis* . Ce sont les clones de leur définition , simplement elles sont écrites en OCaml (on voit ici clairement la puissance de ce langage à ce niveau d'abstraction)

```
let rec trm substitution =
  match substitution with
  | Gsl_affect(x,e)  →  Pred_true
  | Gsl_skip  →  Pred_true
  | Gsl_precondition(p,s)  →  Pred_and( p , trm s )
  | Gsl_bounded_choice(s1,s2)  →  Pred_and( trm s1 , trm s2 )
  | Gsl_guard(p,s) →  Pred_implies( p , trm s )
  | Gsl_forall(z , s)  →  Pred_forall(z , trm s )

let rec fis substitution =
  match substitution with
  | Gsl_affect(x,e)  →  Pred_true
```

```
| Gsl_skip → Pred_true
| Gsl_precondition(p,s) → Pred_implies( p , fis s )
| Gsl_bounded_choice(s1,s2) → Pred_or( fis s1 , fis s2 )
| Gsl_guard(p,s) → Pred_and( p , fis s )
| Gsl_forall(z , s) → Pred_exists(z , fis s )
```

5. Ces fonctions , comme on le devine plus loin , permettent l'utilisation de prd avec plusieurs variables . *Apostrophe* , qui construit le nom "après" d'une variable est fortement dépendante du type selon lequel est défini un identificateur , ce qui a aussi motivé l'abandon de l'indépendance des GSL

```
let rec build_sequence a_liste =
  match a_liste with
  | h :: t → if t = [] then h else h ^ "," ^ build_sequence t
  | [] → ""

let apostrophe variable = variable ^ "'"

let rec apply_subst v e var_list =
  match var_list with
  | hvar :: vart → if hvar = v then e :: vart else hvar :: (apply_subst v e vart )
  | [] → []

let create_parallel_equal x e var_list =
  let (vars,exprs) = ( List.map apostrophe var_list , apply_subst x e var_list ) in
  (build_sequence vars , build_sequence exprs)

let create_prd_skip var_list =
  let (vars,exprs) = ( List.map apostrophe var_list , var_list ) in
  (build_sequence vars , build_sequence exprs)
```

6. Les fonctions définies ci-avant permettent ici une quasi-équivalence de la fonction avec sa définition .Une remarque intéressante , est que des trois fonctions sur les GSL , *prd* est sûrement la moins indépendante des types sur lesquels est construit le type GSL .

```
let rec prd var_list substitution =
  match substitution with
  | Gsl_affect(x,e) →
      let (var_seq,expr_seq) = create_parallel_equal x e var_list in
      Pred_equal ( Pred_var(var_seq),Pred_expr(expr_seq) )
  | Gsl_skip →
      let (var_seq,expr_seq) = create_prd_skip var_list in
      Pred_equal ( Pred_var(var_seq),Pred_expr(expr_seq) )
  | Gsl_precondition(p,s) →
      Pred_implies( p , prd var_list s )
  | Gsl_bounded_choice(s1,s2) →
      Pred_or( prd var_list s1 , prd var_list s2 )
```

$\mid Gsl\_guard(p,s) \rightarrow$
    $Pred\_and(\ p\ ,\ prd\ var\_list\ s\ )$
$\mid Gsl\_forall(z\ ,\ s)\ \rightarrow$
    $Pred\_exists(z\ ,\ prd\ var\_list\ s\ )$

end

Annexe B

Exemple d'exécution

Cet exemple peut sembler curieux pour qui ne connaît pas Ocaml : comme c'est un langage qui peut être compilé, mais également interprété, c'est très pratique pour tester rapidement un bout de programme que l'on vient d'écrire...

```
        Objective Caml version 2.04

# #use "GSL.ml";;
module GSL :
  sig
    type var = string
    and expr = string
    and pred =
      | Pred_equal of pred * pred
      | Pred_notequal of pred * pred
      | Pred_var of var
      | Pred_expr of expr
      | Pred_forall of var * pred
      | Pred_exists of var * pred
      | Pred_and of pred * pred
      | Pred_or of pred * pred
      | Pred_not of pred
      | Pred_implies of pred * pred
      | Pred_equiv of pred * pred
      | Pred_true
    val dc_var : 'a -> 'a
    val dc_expr : 'a -> 'a
    val dc_pred : pred -> var
    type gsl =
      | Gsl_affect of var * expr
      | Gsl_skip
      | Gsl_precondition of pred * gsl
```

```
      | Gsl_bounded_choice of gsl * gsl
      | Gsl_guard of pred * gsl
      | Gsl_forall of var * gsl
    val trm : gsl -> pred
    val fis : gsl -> pred
    val build_sequence : string list -> string
    val apostrophe : string -> string
    val apply_subst : 'a -> 'a -> 'a list -> 'a list
    val create_parallel_equal :
      string -> string -> string list -> string * string
    val create_prd_skip : string list -> string * string
    val prd : var list -> gsl -> pred
  end
# let s1=GSL.Gsl_affect("x" , "x+1");;
val s1 : GSL.gsl = GSL.Gsl_affect ("x", "x+1")
# let s2=GSL.Gsl_skip;;
val s2 : GSL.gsl = GSL.Gsl_skip
# let s3=GSL.Gsl_bounded_choice(s1,s2);;
val s3 : GSL.gsl =
  GSL.Gsl_bounded_choice (GSL.Gsl_affect ("x", "x+1"), GSL.Gsl_skip)
# let x=GSL.Pred_var("x");;
val x : GSL.pred = GSL.Pred_var "x"
# let e=GSL.Pred_expr("0");;
val e : GSL.pred = GSL.Pred_expr "0"
# let p1=GSL.Pred_equal(x,e);;
val p1 : GSL.pred = GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0")
# let e2=GSL.Pred_expr("1");;
val e2 : GSL.pred = GSL.Pred_expr "1"
# let p2=GSL.Pred_notequal(x,e2);;
val p2 : GSL.pred = GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1")
# let s4=GSL.Gsl_guard(p1,s3);;
val s4 : GSL.gsl =
  GSL.Gsl_guard
   (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
    GSL.Gsl_bounded_choice (GSL.Gsl_affect ("x", "x+1"), GSL.Gsl_skip))
# let s5=GSL.Gsl_precondition(p2,s4);;
val s5 : GSL.gsl =
  GSL.Gsl_precondition
   (GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1"),
    GSL.Gsl_guard
     (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
      GSL.Gsl_bounded_choice (GSL.Gsl_affect ("x", "x+1"), GSL.Gsl_skip)))
# let trm1=GSL.trm s5;;
val trm1 : GSL.pred =
```

```
  GSL.Pred_and
   (GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1"),
    GSL.Pred_implies
     (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
      GSL.Pred_and (GSL.Pred_true, GSL.Pred_true)))
# let fis1=GSL.fis s5;;
val fis1 : GSL.pred =
  GSL.Pred_implies
   (GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1"),
    GSL.Pred_and
     (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
      GSL.Pred_or (GSL.Pred_true, GSL.Pred_true)))
# let prd1=GSL.prd ["x"] s5;;
val prd1 : GSL.pred =
  GSL.Pred_implies
   (GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1"),
    GSL.Pred_and
     (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
      GSL.Pred_or
       (GSL.Pred_equal (GSL.Pred_var "x'", GSL.Pred_expr "x+1"),
        GSL.Pred_equal (GSL.Pred_var "x'", GSL.Pred_expr "x"))))
# let prd2=GSL.prd ["y";"x";"z"] s5;;
val prd2 : GSL.pred =
  GSL.Pred_implies
   (GSL.Pred_notequal (GSL.Pred_var "x", GSL.Pred_expr "1"),
    GSL.Pred_and
     (GSL.Pred_equal (GSL.Pred_var "x", GSL.Pred_expr "0"),
      GSL.Pred_or
       (GSL.Pred_equal (GSL.Pred_var "y',x',z'", GSL.Pred_expr "y,x+1,z"),
        GSL.Pred_equal (GSL.Pred_var "y',x',z'", GSL.Pred_expr "y,x,z"))))
# GSL.dc_pred trm1;;
- : GSL.var = "((x=/=1) AND ((x=0) => (TRUE AND TRUE)))"
# GSL.dc_pred fis1;;
- : GSL.var = "((x=/=1) => ((x=0) AND (TRUE OR TRUE)))"
# GSL.dc_pred prd1;;
- : GSL.var = "((x=/=1) => ((x=0) AND ((x'=x+1) OR (x'=x))))"
# GSL.dc_pred prd2;;
- : GSL.var =
"((x=/=1) => ((x=0) AND ((y',x',z'=y,x+1,z) OR (y',x',z'=y,x,z))))"
# let s6=GSL.Gsl_affect("y","z");;
val s6 : GSL.gsl = GSL.Gsl_affect ("y", "z")
# let z=GSL.Pred_var("z");;
val z : GSL.pred = GSL.Pred_var "z"
# let e3=GSL.Pred_expr("Macro$oft");;
```

```
val e3 : GSL.pred = GSL.Pred_expr "Macro$oft"
# let p3=GSL.Pred_notequal(z,e3);;
val p3 : GSL.pred =
  GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft")
# let s7=GSL.Gsl_guard(p3,s6);;
val s7 : GSL.gsl =
  GSL.Gsl_guard
   (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
    GSL.Gsl_affect ("y", "z"))
# let s8=GSL.Gsl_forall("z",s7);;
val s8 : GSL.gsl =
  GSL.Gsl_forall
   ("z",
    GSL.Gsl_guard
     (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
      GSL.Gsl_affect ("y", "z")))
# let trm2=GSL.trm s8;;
val trm2 : GSL.pred =
  GSL.Pred_forall
   ("z",
    GSL.Pred_implies
     (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
      GSL.Pred_true))
# let fis2=GSL.fis s8;;
val fis2 : GSL.pred =
  GSL.Pred_exists
   ("z",
    GSL.Pred_and
     (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
      GSL.Pred_true))
# let prd3=GSL.prd ["y"] s8;;
val prd3 : GSL.pred =
  GSL.Pred_exists
   ("z",
    GSL.Pred_and
     (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
      GSL.Pred_equal (GSL.Pred_var "y'", GSL.Pred_expr "z")))
# let prd4=GSL.prd ["y";"x"] s8;;
val prd4 : GSL.pred =
  GSL.Pred_exists
   ("z",
    GSL.Pred_and
     (GSL.Pred_notequal (GSL.Pred_var "z", GSL.Pred_expr "Macro$oft"),
      GSL.Pred_equal (GSL.Pred_var "y',x'", GSL.Pred_expr "z,x")))
```

```
# GSL.dc_pred trm2;;
- : GSL.var = "A!z/((z=/=Macro$oft) => TRUE)"
# GSL.dc_pred fis2;;
- : GSL.var = "E!z/((z=/=Macro$oft) AND TRUE)"
# GSL.dc_pred prd3;;
- : GSL.var = "E!z/((z=/=Macro$oft) AND (y'=z))"
# GSL.dc_pred prd4;;
- : GSL.var = "E!z/((z=/=Macro$oft) AND (y',x'=z,x))"
# #quit;;
```

Bibliographie

[Mariano97] Georges Mariano, *évaluation de logiciels critiques développés par la méthode B, une approche quantitative*,
thèse,
université de Valenciennes et du Hainaut Cambrésis,
1997

[Atelier B] `http://www.atelierb.societe.com`

[Andreas Herzig et al.] Andreas Herzig, Gabriella Crocco, Olivier Gasquet et Bruno Gaume ,

`http://www.irit.fr/~Andreas.Herzig et`
`http://www.irit.fr/SSI/ACTIVITES/EQ_ALG/Herzig/C`

[Abrial 96] Jean-Raymond Abrial
The B Book — Assigning programs to meaning
Cambridge University Press, août 1996

[Leroy 99] Xavier Leroy ,
The Objective Caml system release 2.04 ,
documentation and user's manual ,
INRIA 1999 ,

`http://caml.inria.fr`

[Leslie 94] Leslie Lamport
LaTeX : a document preparation system ,
Addison-Wesley ,
Reading, Massachusetts ,deuxième édition, 1994

[LaTeX Companion] Michel Goossens, Frank Mittelbach et Alexander Samarin ,
The LaTeX Companion
Addison-Wesley ,
Reading, Massachusetts ,deuxième édition, 1994

[*HeVeA*] Luc Maranget ,

`http://para.inria.fr/~maranget/hevea`