Filière MP : ENS de Cachan, Lyon, Rennes et Paris

Page de garde du rapport de TIPE - Session 2014

NOM : MAURAS Prénoms : Simon

Lycée : Clemenceau

Classe : MP*

Ville : Nantes

Concours auxquels vous êtes admissible dans la banque MP inter-ENS :

(les indiquer par une croix inscrite dans les cases ci-dessous)

ENS Cachan	MP - option MP		MP - option MPI	
	Informatique	X		
ENS Lyon	MP - option MP		MP - option MPI	
	Informatique - option M	X	Informatique - option P	
ENS Rennes	MP - option MP		MP - option MPI	
	Informatique	X		
ENS Paris	MP - option MP		MP - option MPI	
	Informatique	X		

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans l'une des cases ci-dessous) :

Informatique	X	Mathématiques		Physique	

Titre du TIPE : Relations entre flux en programmation synchrone

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	T	5	Illustrations	I	20	Bibliographie	B	-

Résumé (ou descriptif succinct) du TIPE (6 lignes, maximum) :

Lucid Synchrone est un langage de programmation synchrone dont les « flux de données » sont usuellement définis de manière fonctionnelle. En faisant un ajout en amont du compilateur, nous avons pu introduire une nouvelle structure permettant de définir de manière relationnelle les flux de booléens. Afin de transformer la relation spécifiée en une fonction bien déterministe, nous avons utilisé des Diagrammes de Décision Binaire permettant de manipuler les formules logiques.

A Nantes, le 2 juin 2014
Signature du (de la) candidat(e)

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

LYCÉE CLEMENCEAU
1, Rue G. Clemenceau
BP 74205
44042 NANTES Cedex 1
Tél. 02 51 81 86 10 Fax 02 51 81 96 98

Relations entre flux en programmation synchrone

Simon Mauras

2013 - 2014

Les langages de programmation synchrone ont été conçus pour programmer des systèmes critiques nécessitant le respect strict de contraintes temporelles (ex : nucléaire, aéronautique, ...). Le thème annuel (transferts, échanges, relations, flux) m'a orienté vers ces langages qui reposent sur la notion centrale de "flux de données" et utilisent des relations pour spécifier les propriétés des programmes.

Nous avons voulu expérimenter la possibilité de définir des parties de programme par les relations à satisfaire. Pour cela nous avons introduit une définition **relationnelle** des **flux** dans Lucid Synchrone. Ce langage basé sur OCaml, est un prototype de recherche dont les sources du compilateur sont disponibles, ce qui nous a permis d'ajouter une nouvelle construction au langage. Nous avons réalisé un précompilateur permettant de transformer des programmes relationnels utilisant cette nouvelle structure en programmes fonctionnels en Lucid Synchrone.

Table des matières

1 La programmation synchrone

1.1 Le langage Lucid Synchrone

Le langage Lucid Synchrone est basé sur OCaml et possède donc une syntaxe qui lui est proche. On peut voir les variables (flux de données) comme des suites indexées par un sous ensemble de $\mathbb{N}$ représentant un échantillonnage du temps. Un flux a donc un type (booléen, entier, flottant, ...) et une horloge. Nous nous intéresserons aux flux de booléens définis sur une horloge qui leur est commune. Dans Lucid Synchrone, un flux est défini par une fonction qui est soit combinatoire (la valeur d'un flux à un instant ne dépend que des valeurs présentes d'autres flux) soit séquentielle (la valeur d'un flux à un instant peut dépendre du passé).

x	0	0	1	0	1	1	0	1
y	1	0	1	0	0	1	1	0
$x \vee y$	1	0	1	0	1	1	1	1
pre x	-	0	0	1	0	1	1	0
$f = x \wedge \neg$ `pre` x	-	0	1	0	1	0	0	1
$z =$ `if` f `then` y `else pre` z	-	-	1	1	0	0	0	0

Les flux x et y sont donnés en entrée. Le flux $x \vee y$ est calculé de manière combinatoire. L'équation définissant f est séquentielle car elle utilise l'opérateur `pre` qui fait référence au passé. L'équation récurrente définissant z est également séquentielle et échantillonne les valeurs de y sur les fronts montants de x.

L'équation qui définit z doit, en Lucid Synchrone, être intégrée dans un noeud qui définit de manière fonctionnelle comment transformer des flux d'entrée en flux de sortie.

```
let node une_fonction x y =
  let rec z = false
            -> if x && not (pre x)
                 then y
                 else pre z
  in z;;
(* val une_fonction : bool -> bool => bool *)
(* val une_fonction :: 'a -> 'a -> 'a *)
```

On remarque l'importance d'initialiser le flux `z` (opération `->`). La signature comporte des informations sur les types de données et l'horloge associée. Pour plus de détails la documentation est disponible en [6].

1.2 Des relations pour exprimer des propriétés

Avec les récurrences sur les flux on pourrait définir un programme qui, à chaque fois que son entrée est vraie, inverse la valeur de sa sortie et sinon la mémorise. Avant de programmer ce traitement, on peut le spécifier en énonçant les propriétés à respecter :
$\mathcal{P}_1$: La valeur est basculée si `entree` est vrai : `entree` $\Rightarrow$ `sortie` $=$ $\neg$ `pre sortie`
$\mathcal{P}_2$: La valeur est conservée si `entree` est fausse : $\neg$`entree` $\Rightarrow$ `sortie` $=$ `pre sortie`

Ces propriétés peuvent, après écriture du programme, servir à vérifier sa correction.
Notre approche consiste à tenter de synthétiser le programme à partir de ses propriétés. Ceci n'est possible que si les propriétés permettent de définir un comportement réactif et déterministe.

2 Définition relationnelle des flux

2.1 La construction let rel such

On propose d'ajouter un nouveau type de déclaration de noeud qui, au lieu de définir chaque sortie par une équation, définit l'ensemble du comportement par une seule relation. La fonction caractéristique de cette relation sera exprimée par une formule logique.

On ajoute donc à la grammaire de Lucid Synchrone [6] (section 3.12) une règle définissant le nouveau noeud `let rel such` et une pour les relations logiques.

impl-phrase	::=	*...*		
	|	*rel-definition*		
rel-definition	::=	`let rel` *lowercase-ident var-list* = *var-list* `such` [*relation* `->`] *relation*		
var-list	::=	*lowercase-ident* { *lowercase-ident* }		
relation	::=	(*relation*)	| *lowercase-ident*	
	|	`pre` *lowercase-ident*	| ∼ *relation*	
	|	*relation* `&` *relation*	| *relation* `|` *relation*	
	|	*relation* `=>` *relation*	| *relation* `<=>` *relation*	

Les opérateurs ∼, `|`, `&`, `=>` et `<=>` désignent respectivement la négation, la disjonction, la conjonction, l'implication et l'équivalence logique. Les programmes écrits en utilisant cette nouvelle syntaxe devront pouvoir être traduits en Lucid Synchrone standard.

2.2 Principe de la traduction

On note $\mathbb{B} = \{0, 1\}$ muni des opérations classiques $\vee$ et $\wedge$. On assimile la relation entre les m entrées et les n sorties (les valeurs précédentes des sorties sont vues comme des variables d'entrée) à sa fonction caractéristique $\mathcal{R} : \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$.
On pose i dans $[\![1, n]\!]$ et on définit $\mathcal{R}_i$ de $\mathbb{B}^{m+1}$ dans $\mathbb{B}$:

$$\mathcal{R}_i : \quad (e_1, ..., e_m, s_i) \quad \mapsto \quad \exists (s_1, ..., s_{i-1}, s_{i+1}, ..., s_n) \in \mathbb{B}^{n-1},\ \mathcal{R}(e_1, ..., e_m, s_1, ..., s_n)$$

Par construction on montre que : $\forall e \in \mathbb{B}^m,\ \forall (s_1, ..., s_n),\ \mathcal{R}(e, (s_1, ..., s_n)) \Rightarrow R_i(e, s_i)$

La décomposition de Shannon donne : $\forall e \in \mathbb{B}^m,\ \forall s_i,\ \mathcal{R}_i(e, s_i) = (s_i \wedge \mathcal{R}_i(e, 1)) \vee (\neg s_i \wedge \mathcal{R}_i(e, 0))$

On peut maintenant, pour tout e dans $\mathbb{B}^m$, construire la valeur de la sortie d'indice i :

$$s_i = \begin{cases} 1 & \text{si } (R_i(e,0), R_i(e,1)) = (0,1) \\ 0 & \text{si } (R_i(e,0), R_i(e,1)) = (1,0) \\ \texttt{Over constrained} & \text{si } (R_i(e,0), R_i(e,1)) = (0,0) \\ \texttt{Non deterministic} & \text{si } (R_i(e,0), R_i(e,1)) = (1,1) \end{cases}$$

Pour être traductible en fonction il doit exister, pour toute entrée, une unique sortie qui satisfasse notre relation. Sans l'existence la relation contraint trop les sorties, sans l'unicité il n'y a plus déterminisme. Il nous faut maintenant une structure de données nous permettant de manipuler nos formules logiques pour générer les $\mathcal{R}_i$.

2.3 Exemple

Reprenons l'exemple des propriétés spécifiées en section 1.2. Avec notre nouvelle construction, nous pouvons écrire (l'opérateur "`->`" sépare la relation initiale de la relation invariante) :

```
let rel modifier entree = sortie
    such ~ sortie
      -> entree <=> (~ sortie <=> pre sortie)
```

L'objectif de notre traduction est d'obtenir le code suivant où le flux de sortie est défini par une fonction :

```
let node modifier entree =
  let rec sortie = false -> if pre sortie then not entree else entree
  in sortie
```

3 Réalisation du précompilateur Prelucy

3.1 Analyse lexicale et syntaxique

Dans le compilateur Lucid Synchrone, l'analyseur lexical et l'analyseur syntaxique sont générés par ocamllex et ocamlyacc. La documentation officielle est disponible en [3], un exemple détaillé est donné dans [4] et les sources du compilateur Lucid Synchrone peuvent être trouvées en [5]. Les ajouts faits sont disponibles dans la section B.2. Nous avons choisi les opérateurs `&`, `|`, `=>`, `<=>` et `~` de manière à ce qu'il n'y ait pas de collision avec ceux pré-existant et leur précédence. Le mot clé `rel` a été ajouté pour faire disparaître un conflit shift/reduce.

3.2 Traduction des relations en Diagramme de Décision Binaire

Nous avons choisi les diagrammes de décision binaire (bdd) pour gérer nos formules logiques. Le package [2] est une implémentation des bdd écrite en OCaml donc facilement intégrable à notre précompilateur. Nous avons ajouté une surcouche (section B.3 : bdds.ml) de manière à ce que les variables soient des chaînes de caractères. Avec les constructeurs disponibles, il nous suffit de parcourir l'arbre syntaxique de la relation pour construire le bdd associé. Pour obtenir la relation $\mathcal{R}_i$ (section 2.2) on abstrait une à une les variables par disjonction des cas (la fonction `exists` utilise le caractère ordonné du diagramme). On affiche ensuite la fonction définissant chaque sortie (la fonction `printdef` est basée sur le fait que le diagramme est ordonné et réduit).

3.3 Réécriture en Lucid Synchrone

Pour la réécriture en Lucid Synchrone, la fonction `Relations.pretty` gère deux cas différents. Pour chaque "phrase" de l'implémentation étant une structure déjà existante du langage nous avons choisi de recopier le code de l'utilisateur, du fichier source vers le fichier cible.
Une définition relationnelle de la forme :

```
let rel <fun_name> e_1 ... e_m = s_1 ... s_n
  such relation_init -> relation_rec
```

est transformée en Lucid Synchrone sous la forme :

```
let node <fun_name> e_1 ... e_m = let
  rec s_1 = ... -> ...
  ...
  and s_n = ... -> ...
  in (s_1, ..., s_n)
```

Les définitions des sorties s_i sont calculées selon le schéma vu dans la section précédente.

4 Conclusion

Les tests réalisés sur notre précompilateur ont été plutôt concluants. Cependant de nombreuses pistes d'amélioration sont envisageables. Les diagrammes de décision binaire ayant une complexité dans le pire des cas en $\mathcal{O}(2^n)$ (où n est le nombre de variables), ils sont difficilement utilisables lorsque n dépasse plusieurs dizaines. Remarquons toutefois que le nombre n est relatif à un noeud et non pas à tout le programme, décomposer en plusieurs fonctions plus petites est donc généralement suffisant.

De plus dans un bdd, l'ordre des variables est important. Même si nous avons fait un choix qui semble cohérent (les sorties dépendent en général des entrées), trouver l'ordre optimal est un problème NP-Difficile.

Au delà de ces considérations, plusieurs fonctionnalitées pourraient être ajoutées. Même si nous ne travaillons qu'avec des flux de booléens, il serait par exemple utile de pouvoir utiliser, le flux booléen `x < 4` à partir d'un flux d'entiers `x`.

Nous avons imposé dans les noeuds relationnels que tout les flux soient sur la même horloge. Une extension serait de permettre à notre structure de gérer les opérateurs `when` et `merge`.

Enfin notre compilateur ne gère pas pour l'instant tous les cas valides envisageables. Le dernier exemple dans la section B.5 possède des états dans lesquels les sorties ne sont pas définies. Cependant l'initialisation nous assure qu'aucun de ces états ne peut être atteint. Au prix d'un calcul d'accessibilité (parcours d'un graphe) le pré-compilateur pourrait accepter un tel programme.

Il nous faudrait maintenant tester notre prototype sur un projet de taille industrielle pour valider l'intérêt de cette nouvelle construction pour concevoir certaines parties critiques d'une application à partir de la spécification de leurs propriétés logiques. Une telle expérimentation nous permettrait de mieux comprendre les avantages et les limites de la définition relationnelle des flux.

Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2009.

[2] Jean-Christophe Filliâtre. Quick implementation of a BDD library for OCaml, 2010. [En ligne] `https://www.lri.fr/~filliatr/ftp/ocaml/bdd/` (consulté en mai 2014).

[3] Xavier Leroy. *The OCaml system : Documentation and user's manual*, Septembre 2013. `http://caml.inria.fr/pub/docs/manual-ocaml` (consulté en mai 2014).

[4] Luc Maranget. *Cours de compilation*, 2002 - 2004. [En ligne] `http://pauillac.inria.fr/~maranget/X/compil/poly/poly.ps` (consulté en mai 2014).

[5] Marc Pouzet. Sources du compilateur Lucid Synchrone. [En ligne] `http://www.di.ens.fr/~pouzet/lucy.tgz` (consulté en mai 2014).

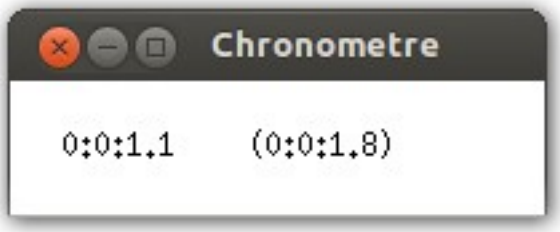
[6] Marc Pouzet. *Lucid Synchrone : Tutorial and Reference Manual*, Avril 2006. [En ligne] `http://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf` (consulté en mai 2014).

Contact

Christophe Declercq, Maître de conférence, Université de Nantes

A Chrono

Ce premier petit projet est un chronomètre, il m'a permis de découvrir le langage Lucid Synchrone et sa syntaxe. L'objectif était de comprendre le fonctionnement de plusieurs fonctionnalités (opérateur last, signaux) en vue d'ajouter une nouvelle structure au langage.



chrono.ls

```
1  open Graphics
2
3  let x = open_graph "";;
4  let x = set_window_title "Chronometre";;
5  let x = resize_window 200 50;;
6  let x = auto_synchronize false;;
7
8  type evenement = StartStop | PauseReset
9
10 let node sample n = top where
11     rec count = 0 -> if pre count <> 0 then pre count - 1 else n - 1
12     and top = count = 0
13
14 let node chronometre button top = (display, time) where
15     rec last stop = true
16     and last pause = false
17     and last display = 0
18     and last time = 0
19     and stop  = present
20                 | button(StartStop) -> not (last stop)
21                 | _ -> last stop
22                 end
23     and pause = present
24                 | button(PauseReset) & (last pause) -> false
25                 | button(PauseReset) & (not (last stop)) -> true
26                 | _ -> last pause
27                 end
28     and time  = present
29                 | top & (not stop) -> last time + 1
30                 | button(PauseReset)
31                   & (last stop)
32                   & (not (last pause)) -> 0
33                 | _ -> last time
34                 end
35     and display = if pause then last display else time
36
37 let affichage n =
```

```
    (string_of_int (n / 36000)) ^ ":" ^
    (string_of_int (n / 600 mod 60)) ^ ":" ^
    (string_of_int (n / 10 mod 60)) ^ "." ^
    (string_of_int (n mod 10));;


let input key = o where
    match key with
    | 'a' -> do emit o = StartStop done
    | 'b' -> do emit o = PauseReset done
    | _   -> do done
    end

let output (n, m) =
    clear_graph ();
    moveto 20 20;
    draw_string (affichage n);
    moveto 90 20;
    draw_string "(";
    draw_string (affichage m);
    draw_string ")";
    synchronize ();;

let node main () =
    let key = if key_pressed () then read_key () else ' ' in
    let button = input key in
    let top = sample 10 in
    let display, time = chronometre button top in
        output (display, time)
```

B PreLucy

Arborescence

Voici une partie de l'arborescence du projet. Le code des fichiers principaux est à la suite. Une archive de la dernière version de pré-compilateur est disponible à l'adresse suivante : `http://simon.mauras.org/TIPE`

```
.
+-- Makefile
+-- config
+-- main
|   +-- compiler.ml
|   +-- main.ml
+-- parsing
|   +-- lexer.mll
|   +-- par_aux.ml
|   +-- parser.mly
|   +-- parsing_errors.ml
|   +-- parsing_errors.mli
+-- relations
|   +-- bdd.ml
|   +-- bdd.mli
|   +-- bdds.ml
|   +-- relations.ml
|   +-- rel_check.ml
|   +-- rel_errors.ml
+-- global
    +-- debug.ml
    +-- location.ml
    +-- lucy.ml
    +-- misc.ml
    +-- ...
```

B.1 main

main/main.ml

Ce fichier permet de gérer la ligne de commande et les différents modes de compilation. Il a été simplifié pour ne contenir que la partie qui nous est ici utile.

```
(* Fichier adapte dans le cadre du TIPE *)

(***********************************************************************)
(*                                                                     *)
(*  Lucid Synchrone                                                    *)
(*                                                                     *)
(*  Author : Marc Pouzet                                               *)
(*  Organization : Demons, LRI, University of Paris-Sud, Orsay         *)
(*                                                                     *)
(***********************************************************************)

open Misc
open Modules
open Compiler

let compile file =
  if Filename.check_suffix file ".lsr"
  then
    let filename = Filename.chop_suffix file ".lsr" in
    let modname = Filename.basename filename in
    compile_implementation (String.capitalize modname) filename
  else
    raise (Arg.Bad ("don't know what to do with " ^ file))

(* show version *)
let show_version () =
  Printf.printf "The Lucid Synchrone pre-compiler, based on \
                 Lucid Synchrone compiler version %s (%s)\n"
    version date

(* the string of messages *)
let doc_version = "The version of the compiler"
and errmsg = "Options are:"

(* the main function: parse the command line *)
let main () =
  try
    Arg.parse
      ["-v", Arg.Unit show_version, doc_version;]
      compile
      errmsg;
  with
    Misc.Stop -> exit 0
  | Misc.Error -> exit 2
```

```
  | Misc.Internal_error s ->
      Printf.eprintf "Internal error: %s. \nPlease report it.\n" s;
      exit 100
;;

Printexc.catch main ();
exit 0;;
```

main/compiler.ml

Ce fichier contient les fonctions permettant de gérer la compilation. Il a été lui aussi simplifié et ne contient plus que l'appel à l'analyseur lexical et synthaxique puis aux fonctions de back-end et front-end (Relations.pretty)

```
(* Fichier adapte dans le cadre du TIPE *)

(***********************************************************************)
(*                                                                     *)
(*  Lucid Synchrone                                                    *)
(*                                                                     *)
(*  Author : Marc Pouzet                                               *)
(*  Organization : Demons, LRI, University of Paris-Sud, Orsay         *)
(*                                                                     *)
(***********************************************************************)

(* pre-compiling a file : parsing and ... *)

open Misc
open Location
open Lexer
open Parser
open Parsing_errors
open Debug
open Relations

(* parsing *)
let parse parsing_fun lexing_fun lexbuf =
  try
    parsing_fun lexing_fun lexbuf
  with
    Parsing.Parse_error ->
      let pos1 = Lexing.lexeme_start lexbuf in
      let pos2 = Lexing.lexeme_end lexbuf in
      let l = Loc(pos1,pos2) in
      syntax_error l;
      raise Error
  | Lexer.Lexical_error(errcode,pos1,pos2) ->
      let l = Loc(pos1,pos2) in
      lexical_error errcode l;
      raise Error
```

```
37
38  let parse_implementation lexbuf =
39    parse Parser.implementation_list Lexer.main lexbuf
40
41  (* the main functions *)
42  let compile_implementation module_name filename =
43    (* input and output files *)
44    let source_name = filename ^ ".lsr" in
45    let obj_name = filename ^ ".ls" in
46
47    let ic = open_in source_name in
48    let cc = open_out_bin obj_name in
49
50    try
51      initialise_location source_name ic;
52
53      (* parsing of the file *)
54      let lexbuf = Lexing.from_channel ic in
55      let decl_list = parse_implementation lexbuf in
56
57      (* debug : print decl_list on stdin *)
58      (* Debug.print_impl_phrase_list decl_list; *)
59
60      (* front-end & back-end *)
61      let out_formatter = Format.formatter_of_out_channel cc in
62      Relations.pretty out_formatter decl_list;
63
64      close_in ic;
65      close_out cc
66    with
67      x ->
68        close_in ic;
69        close_out cc;
70        raise x;;
```

B.2 parsing

parsing/lexer.mll

Le fichier en question fait une taille conséquente et ne contient qu'une modification mineure de ma part : l'ajout de quelques mots clés utiles à notre nouvelle structure.

```
(* ... *)

(* Ajouts a la liste des mots cles *)
["rel", REL; "such", SUCH]

(* ... *)

(* Ajout de deux operateurs dans rule main *)
  | "<=>" { SMALLEREQUALGREATER }
  | "~" { TILDE }

(* ... *)
```

parsing/parser.mly

Le fichier en question fait une taille conséquente et ne contient qu'une modification mineure de ma part : l'ajout d'une règle de grammaire permettant de gérer notre nouvelle structure.

```
/* ... */

/* Ajout de quelques token */
%token REL                    /* "rel"  mot cle */
%token SUCH                   /* "such" mot cle */
%token SMALLEREQUALGREATER    /* "<=>"  equivalence */
%token TILDE                  /* "~"    non */

/* ... */

/* Avec leur associativite et precedence */
%right SMALLEREQUALGREATER
%nonassoc TILDE

/* ... */

/* Puis les regles de grammaire */
implementation :
  | LET binding_list optional_semisemi %prec prec_let
      { make_impl(Lletdef(false, List.rev $2)) }
  | LET REC binding_list optional_semisemi  %prec prec_let
      { make_impl(Lletdef(true, List.rev $3)) }
  | LET REL IDENT rel_var_list EQUAL rel_var_list
    SUCH rel_optional_init_relation optional_semisemi %prec prec_let
      { make_impl(Lletsuch($3, $4, $6, fst $8, snd $8)) }
  | LET CLOCK IDENT EQUAL expression
```

```
      { make_impl(Lletclock($3, $5)) }
  | TYPE type_decl optional_semisemi
      { make_impl(Ltypedef $2) }
  | OPEN CONSTRUCTOR optional_semisemi
      { make_impl(Lopen $2) }
;

rel_var_list :
  |  IDENT rel_var_list { $1 :: $2 }
  |                     { [] }
;

rel_optional_init_relation :
  | rel_relation MINUSGREATER rel_relation  { ($1, $3) }
  | rel_relation                            { ($1, $1) }
;

rel_relation :
  | LPAREN rel_relation RPAREN
      { $2 }
  | IDENT
      { make_relation (Lrelvar($1)) }
  | PRE IDENT
      { make_relation (Lrelpre($2)) }
  | TILDE rel_relation
      { make_relation (Lrelnot($2)) }
  | rel_relation AMPERSAND rel_relation
      { make_relation (Lreland($1, $3)) }
  | rel_relation BAR rel_relation
      { make_relation (Lrelor($1, $3)) }
  | rel_relation EQUALGREATER rel_relation
      { make_relation (Lrelimpl($1, $3)) }
  | rel_relation SMALLEREQUALGREATER rel_relation
      { make_relation (Lrelequ($1, $3)) }
;

/* ... */
```

parsing/par_aux.ml

Ce fichier contient les fonctions permettant de construire l'arbre de la syntaxe abstraite. Nous avons ajouté un seul constructeur :

```
let make_relation desc =
  {r_desc = desc; r_loc = get_current_location()}
```

B.3 relations

relations/relations.ml

```
(* Fichier cree dans le cadre du TIPE *)

open Misc
open Lucy
open Location
open Rel_check
open Rel_errors
open Format

module Varstring = struct
  type var = string
  let to_string s :string = s
  let compare x y =
    if x < y
      then -1
    else if x = y
    then 0
    else 1
end

module Bdd = Bdds.Make(Varstring)

let rec bdd_of_relation rel = match rel.r_desc with
  | Lrelvar(s) -> Bdd.of_var s
  | Lrelpre(s) -> Bdd.of_var ("pre " ^ s)
  | Lrelnot(a)     -> Bdd.do_not (bdd_of_relation a)
  | Lreland(a, b)  -> Bdd.do_and (bdd_of_relation a)
                                 (bdd_of_relation b)
  | Lrelor(a, b)   -> Bdd.do_or (bdd_of_relation a)
                                (bdd_of_relation b)
  | Lrelimpl(a, b) -> Bdd.do_imply (bdd_of_relation a)
                                   (bdd_of_relation b)
  | Lrelequ(a, b)  -> Bdd.do_equal (bdd_of_relation a)
                                   (bdd_of_relation b);;

let copy_location out_formatter (Loc(pos1, pos2)) =
  seek_in !input_chan pos1;
  try
    for i = pos1 to pos2 - 1 do
      pp_print_char out_formatter (input_char !input_chan)
    done
  with End_of_file -> fprintf out_formatter "<EOF>";;

let pretty_decl out_formatter decl = match decl.i_desc with
  | Lletsuch(name, input, output, init_rel, rel) ->
     (try
```

```
47          fprintf out_formatter "(** Pre-compiler : %s **)" name;
48          pp_print_newline out_formatter ();
49
50          (* Checking properties *)
51          check decl.i_loc name input output init_rel rel;
52          let var_list = (List.map (fun s -> "pre " ^ s)
53                                   (get_pre_var_list rel))
54                      @ input
55                      @ output in
56
57          (*List.iter (fun s -> eprintf "%s, " s) var_list;
58          eprintf "\n";*)
59
60          Bdd.set_vars var_list;
61
62          let init_bdd = bdd_of_relation init_rel in
63          let bdd = bdd_of_relation rel in
64          (*Bdd.pretty err_formatter bdd;
65          pp_print_newline err_formatter ();
66          pp_print_newline err_formatter ();*)
67
68          pp_open_vbox out_formatter 2;
69          fprintf out_formatter "let node %s " name;
70          if input = []
71            then fprintf out_formatter "() "
72            else List.iter (fprintf out_formatter "%s ") input;
73          fprintf out_formatter "= let";
74          pp_print_cut out_formatter ();
75
76          let def_keyword = ref "rec" in
77          List.iter (fun s ->
78              fprintf out_formatter "%s %s" !def_keyword s;
79              def_keyword := "and";
80              pp_open_vbox out_formatter 0;
81              fprintf out_formatter " = ";
82              let other_var = List.filter (fun a -> not(a=s))
83                                          output in
84              Bdd.printdef out_formatter s
85                           (Bdd.exists other_var init_bdd);
86              pp_print_cut out_formatter ();
87              fprintf out_formatter "-> ";
88              Bdd.printdef out_formatter s
89                           (Bdd.exists other_var bdd);
90              pp_close_box out_formatter ();
91              pp_print_cut out_formatter ()) output;
92
93          fprintf out_formatter "in (%s" (List.hd output);
94          List.iter (fprintf out_formatter ", %s") (List.tl output);
95          fprintf out_formatter ")";
```

```
96          pp_close_box out_formatter ();
97          fprintf out_formatter "\n\n";
98
99        with Relations_error(err_code, l) ->
100         fprintf out_formatter "(* An error occurred... *)\n\n";
101         pp_print_flush out_formatter ();
102         print_relations_error err_code l;)
103   | _ ->
104      copy_location out_formatter decl.i_loc;
105      fprintf out_formatter  "\n\n";;
106
107 let pretty out_formatter decl_list =
108   List.iter (pretty_decl out_formatter) decl_list
```

relations/rel_check.ml

```
1  (* Fichier cree dans le cadre du TIPE *)
2
3  open Lucy
4  open Rel_errors
5
6  let check pos name input output init_rel rel =
7    let rec parcours disallowed_pre r = match r.r_desc with
8      | Lrelvar(s) ->
9           if not (List.mem s output) then
10          if not (List.mem s input) then
11            raise (Relations_error (Bad_var, r.r_loc));
12     | Lrelpre(s) ->
13          if disallowed_pre then
14            raise (Relations_error (Pre_disallowed, r.r_loc));
15          if not (List.mem s output) then
16          if not (List.mem s input) then
17            raise (Relations_error (Bad_var, r.r_loc))
18     | Lrelnot(a)     -> parcours disallowed_pre a
19     | Lreland(a, b)  -> parcours disallowed_pre a;
20                         parcours disallowed_pre b;
21     | Lrelor(a, b)   -> parcours disallowed_pre a;
22                         parcours disallowed_pre b;
23     | Lrelimpl(a, b) -> parcours disallowed_pre a;
24                         parcours disallowed_pre b;
25     | Lrelequ(a, b)  -> parcours disallowed_pre a;
26                         parcours disallowed_pre b in
27
28   if output = [] then
29     raise (Relations_error (Empty_outputs, pos));
30   if List.exists (fun a -> List.mem a output) input then
31     raise (Relations_error (Inout_var, pos));
32   if (List.mem name input) or (List.mem name output) then
33     raise (Relations_error (Bad_name, pos));
```

```
  parcours true init_rel;
  parcours false rel;;

let get_pre_var_list rel =
  let rec get_pre_var_list l r = match r.r_desc with
    | Lrelvar(s) -> l
    | Lrelpre(s) -> if List.mem s l
                    then l
                    else s::l
    | Lrelnot(a)     -> get_pre_var_list l a
    | Lreland(a, b)  -> get_pre_var_list (get_pre_var_list l b) a
    | Lrelor(a, b)   -> get_pre_var_list (get_pre_var_list l b) a
    | Lrelimpl(a, b) -> get_pre_var_list (get_pre_var_list l b) a
    | Lrelequ(a, b)  -> get_pre_var_list (get_pre_var_list l b) a in
  get_pre_var_list [] rel
```

relations/rel_errors.ml

```
(* Fichier cree dans le cadre du TIPE *)

open Misc
open Location

type relations_error =
   | Empty_outputs
   | Bad_name
   | Inout_var
   | Bad_var
   | Pre_disallowed;;

exception Relations_error of relations_error * location;;

let print_relations_error err_code l = match err_code with
  | Empty_outputs ->
      Printf.eprintf
         "%aThere's no output for this relation.\n\n"
         output_location l;
  | Bad_name ->
      Printf.eprintf
         "%aRelation's name is used as a variable.\n\n"
         output_location l;
  | Inout_var ->
      Printf.eprintf
         "%aVariable cannot be both input and output.\n\n"
         output_location l;
  | Bad_var ->
      Printf.eprintf
         "%aThis variable isn't defined.\n\n"
         output_location l;
```

```
  | Pre_disallowed ->
      Printf.eprintf
         "%aThe use of Pre here is disallowed.\n\n"
         output_location l;
```

relations/bdds.ml

```
(* Fichier cree dans le cadre du TIPE *)

module type Variable =
sig
  type var
  val to_string : var -> string
  val compare : var -> var -> int
end

module type B =
   sig
      type var
      type bdd
      val stats : unit -> (int * int * int * int * int * int) array
      val get_vars : unit -> var list
      val set_vars : var list -> unit
      val of_var : var -> bdd
      val if_then_else : var -> bdd -> bdd -> bdd
      val t : unit -> bdd
      val f : unit -> bdd
      val (+) : bdd -> bdd -> bdd
      val do_or : bdd -> bdd -> bdd
      val ( * ) : bdd -> bdd -> bdd
      val do_and : bdd -> bdd -> bdd
      val do_not : bdd -> bdd
      val exists : var list -> bdd -> bdd
      val forall : var list -> bdd -> bdd
      val (<=>) : bdd -> bdd -> bdd
      val do_equal : bdd -> bdd -> bdd
      val (<>) : bdd -> bdd -> bdd
      val do_diff : bdd -> bdd -> bdd
      val do_imply : bdd -> bdd -> bdd
      val (=>) : bdd -> bdd -> bdd
      val to_then : bdd -> bdd
      val to_else : bdd -> bdd
      val to_var : bdd -> var
      val equal : bdd -> bdd -> bool
      val tautology : bdd -> bool
      val imply : bdd -> bdd -> bool
      val pretty : Format.formatter -> bdd -> unit
      val printdef : Format.formatter -> var -> bdd -> unit
    end
```

```
43
44  module Make (V : Variable): (B with type var = V.var) =
45  struct
46    type var = V.var
47
48    type bdd = Bdd.t
49
50    let stats () = Bdd.stats()
51
52    let les_vars = ref ([] : (var * Bdd.variable) list)
53
54    let get_vars () = List.map fst !les_vars
55
56    let numerote l =
57      let rec numerote n = function
58        | [] -> []
59        | e::l -> (e,n)::(numerote (n+1) l) in
60      numerote 1 l
61
62    let set_vars lv = (Bdd.set_max_var (List.length(lv)) ;
63                       les_vars := numerote lv)
64
65    let variable_of_var v =
66      if List.mem_assoc v !les_vars
67        then List.assoc v !les_vars
68        else 0
69
70    let var_of_variable v = fst (List.nth !les_vars (v - 1))
71
72    let of_var v = Bdd.mk_var (variable_of_var v)
73
74    let if_then_else v t1 t0 = Bdd.make (variable_of_var v) t0 t1
75
76    let t () = Bdd.one
77    let f () = Bdd.zero
78
79    let do_or = Bdd.mk_or
80    let (+) x y = do_or x y
81
82    let do_and = Bdd.mk_and
83    let ( * ) x y = do_and x y
84
85    let do_not = Bdd.mk_not
86
87    let do_imply = Bdd.mk_imp
88    let (=>) x y = do_imply x y
89
90    let do_diff x y = do_or (do_and x (do_not y)) (do_and (do_not x) y)
91    let ( <> ) x y = do_diff x y
```

```
92
93  let do_equal x y = do_and (do_imply x y)(do_imply y x)
94  let (<=>) x y = do_equal x y
95
96  let to_then = Bdd.high
97
98  let to_else = Bdd.low
99
100 let to_var x = var_of_variable (Bdd.var x)
101
102 let tautology = Bdd.tautology
103
104 let equal x y = tautology (do_equal x y)
105
106 let imply x y = tautology (do_imply x y)
107
108 open Format
109
110 let pretty out_formatter t =
111   let parouv p n = if p > n then fprintf out_formatter "(" in
112   let parfer p n = if p > n then fprintf out_formatter ")" in
113   let rec pretty p t = match Bdd.view(t) with
114     | Bdd.One ->  fprintf out_formatter "true"
115     | Bdd.Zero ->  fprintf out_formatter "false"
116     | Bdd.Node(v, t1, t2) -> match (Bdd.view t1, Bdd.view t2) with
117         | (Bdd.Zero, Bdd.One) ->
118             fprintf out_formatter "%s"
119                     (V.to_string (var_of_variable v))
120         | (Bdd.One, Bdd.Zero) ->
121             fprintf out_formatter "not %s"
122                     (V.to_string (var_of_variable v))
123         | (b, Bdd.One) ->
124             parouv p 2;
125             fprintf out_formatter "%s or"
126                     (V.to_string (var_of_variable v));
127             pp_print_space out_formatter ();
128             pretty 2 t1; parfer p 2
129         | (Bdd.One, b) ->
130             parouv p 1;
131             fprintf out_formatter "%s =>"
132                     (V.to_string (var_of_variable v));
133             pp_print_space out_formatter ();
134             pretty 1 t2;
135             parfer p 1
136         | (b, Bdd.Zero) ->
137             fprintf out_formatter "not %s and"
138                     (V.to_string (var_of_variable v));
139             pp_print_space out_formatter ();
140             pretty 3 t1
```

```
141             | (Bdd.Zero, b) ->
142                 fprintf out_formatter "%s and"
143                         (V.to_string (var_of_variable v));
144                 pp_print_space out_formatter ();
145                 pretty 3 t2
146             | (b1,b2) ->
147                 pp_open_tbox out_formatter ();
148                 pp_set_tab out_formatter ();
149                 pp_open_hbox out_formatter () ;
150                 fprintf out_formatter "if %s"
151                         (V.to_string (var_of_variable v));
152                 pp_close_box out_formatter ();
153                 pp_print_tbreak out_formatter 0 0;
154                 pp_open_hovbox out_formatter 5 ;
155                 fprintf out_formatter "then "; pretty 0 t2;
156                 pp_close_box out_formatter ();
157                 pp_print_tbreak out_formatter 0 0;
158                 pp_open_hovbox out_formatter 5 ;
159                 fprintf out_formatter "else "; pretty 0 t1;
160                 pp_close_box out_formatter ();
161                 pp_close_tbox out_formatter () in
162       pp_open_hvbox out_formatter 0;
163       pretty 0 t;
164       pp_close_box out_formatter ()
165
166     let rec exists lv t = match Bdd.view t with
167       | Bdd.One -> t
168       | Bdd.Zero -> t
169       | Bdd.Node(x,t1,t2) ->
170          if lv = []
171            then t
172          else if x > variable_of_var (List.hd lv)
173            then exists (List.tl lv) t
174          else if x = variable_of_var (List.hd lv)
175            then do_or (exists (List.tl lv) t1)
176                       (exists (List.tl lv) t2)
177            else Bdd.make x (exists lv t1)(exists lv t2)
178
179     let rec forall lv t = match Bdd.view t with
180       | Bdd.One -> t
181       | Bdd.Zero -> t
182       | Bdd.Node(x,t1,t2) ->
183          if lv = []
184            then t
185          else if x > variable_of_var (List.hd lv)
186            then forall (List.tl lv) t
187          else if x = variable_of_var (List.hd lv)
188            then do_and (forall (List.tl lv) t1)
189                        (forall (List.tl lv) t2)
```

```
190           else Bdd.make x (forall lv t1)(forall lv t2)
191
192   let printdef out_formatter v t =
193     let rec pretty v0 p t = match Bdd.view(t) with
194       | Bdd.One ->  fprintf out_formatter "Non_deterministic"
195       | Bdd.Zero -> fprintf out_formatter "Over_constrained"
196       | Bdd.Node(v, t1, t2) ->
197          if not ((var_of_variable v) = v0) then begin
198            pp_open_tbox out_formatter ();
199            pp_set_tab out_formatter ();
200            pp_open_hbox out_formatter ();
201            fprintf out_formatter "if %s"
202                    (V.to_string (var_of_variable v));
203            pp_close_box out_formatter ();
204            pp_print_tbreak out_formatter 0 0;
205            pp_open_hovbox out_formatter 5;
206            fprintf out_formatter "then ";
207            pretty v0 0 t2;
208            pp_close_box out_formatter ();
209            pp_print_tbreak out_formatter 0 0;
210            pp_open_hovbox out_formatter 5;
211            fprintf out_formatter "else ";
212            pretty v0 0 t1;
213            pp_close_box out_formatter ();
214            pp_close_tbox out_formatter ();
215          end else (match (Bdd.view t1, Bdd.view t2) with
216            | (Bdd.Zero, Bdd.One) ->  fprintf out_formatter ("true")
217            | (Bdd.One, Bdd.Zero) ->  fprintf out_formatter ("false")
218            | (_, _) -> fprintf out_formatter "Error") in
219     pp_open_hvbox out_formatter 0;
220     pretty v 0 t;
221     pp_close_box out_formatter ()
222
223 end;;
```

B.4 global

global/debug.ml

Ce fichier ne contient que des fonctions basiques nous permettant d'afficher les différentes structures arborescententes pour déboguage.

global/location.ml

Ce fichier n'a pas été modifié dans le cadre du TIPE, il est cependant utile pour naviguer dans le fichier source (.lsr) et afficher certaines lignes en cas d'erreur syntaxique.

global/lucy.ml

Ce fichier contient la définition de l'arbre représentant la syntaxe abstraite. Nous avons ajouté deux types de noeud nécessaires à notre nouvelle structure.

```
(* ... *)

type impl_phrase =
  { i_desc: impl_desc;
    i_loc: location }

and impl_desc =
    Lletdef of is_rec * definition list
  | Lletclock of string * expression
  | Ltypedef of (string * string list * type_decl) list
  | Lopen of string
  | Lletsuch of string * string list * string list * relation * relation

and relation =
  { r_desc: relation_desc;
    r_loc: location }

and relation_desc =
    Lrelvar of string
  | Lrelpre of string
  | Lrelnot of relation
  | Lreland of relation * relation
  | Lrelor of relation * relation
  | Lrelimpl of relation * relation
  | Lrelequ of relation * relation

(* ... *)
```

global/misc.ml

Ce fichier n'a pas été modifié mais contient une compilation de fonctions utiles au compilateur Lucid Synchrone.

B.5 test

test/test.lsr

Ce fichier contient du code Lucid Synchrone relationnel permettant de tester le précompilateur.

```
(* Fichier de test créé dans le cadre du TIPE *)

let negation x = not x;;

let rel un_sur_deux = sortie
  such sortie -> (~sortie) <=> (pre sortie)

let rel negation entree = sortie such (~ sortie) <=> (entree)

let rel modifier entree = sortie
    such ~ sortie
      -> entree <=> (~ sortie <=> pre sortie)

let rel changer_mode bouton = mode1 mode2 sortie
  such mode1 & ~ mode2 & sortie
    -> (bouton <=> (~ mode1 <=> pre mode1))
     & (bouton <=> (~ mode2 <=> pre mode2))
     & (mode1 => sortie)
     & (mode2 => ~ sortie)


(* Test des messages d'erreur...
let rel no_output input = such input;;
let rel undefined_var input = output such undefined;;
let rel in_and_out variable = variable such variable <=> variable;;
let rel disallowed_pre input = output such pre output;;
let rel name_reused = name_reused such name_reused;;
*)
```

test/test.ls

Ce fichier contient le code Lucid Synchrone généré par le précompilateur.

```
let negation x = not x;;

(** Pre-compiler : un_sur_deux **)
let node un_sur_deux () = let
  rec sortie = true
             -> if pre sortie
                then false
                else true
  in (sortie)

(** Pre-compiler : negation **)
```

```
let node negation entree = let
  rec sortie = if entree
               then false
               else true
            -> if entree
               then false
               else true
  in (sortie)

(** Pre-compiler : modifier **)
let node modifier entree = let
  rec sortie = false
            -> if pre sortie
               then if entree
                    then false
                    else true
               else if entree
                    then true
                    else false
  in (sortie)

(** Pre-compiler : changer_mode **)
let node changer_mode bouton = let
  rec mode1 = true
           -> if pre mode1
              then if pre mode2
                   then if bouton
                        then false
                        else Over_constrained
                   else if bouton
                        then false
                        else true
              else if pre mode2
                   then if bouton
                        then true
                        else false
                   else if bouton
                        then Over_constrained
                        else false
  and mode2 = false
           -> if pre mode1
              then if pre mode2
                   then if bouton
                        then false
                        else Over_constrained
                   else if bouton
                        then true
                        else false
              else if pre mode2
```

```
                    then if bouton
                         then false
                         else true
                    else if bouton
                         then Over_constrained
                         else false
  and sortie = true
             -> if pre mode1
                then if pre mode2
                     then if bouton
                          then Non_deterministic
                          else Over_constrained
                     else if bouton
                          then false
                          else true
                else if pre mode2
                     then if bouton
                          then true
                          else false
                     else if bouton
                          then Over_constrained
                          else Non_deterministic
  in (mode1, mode2, sortie)
```