Reference manual of the Bouclettes parallelizer

Pierre Boulet, Michèle Dion, Eric Lequiniou, Tanguy Risset

▶ **To cite this version:**

Pierre Boulet, Michèle Dion, Eric Lequiniou, Tanguy Risset. Reference manual of the Bouclettes parallelizer. [Research Report] LIP TR-94-04, Laboratoire de l'informatique du parallélisme. 1994, 2+28p. hal-02102691

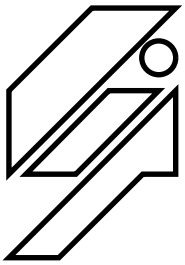
HAL Id: hal-02102691

https://hal-lara.archives-ouvertes.fr/hal-02102691v1

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Laboratoire de l'Informatique du Parallélisme

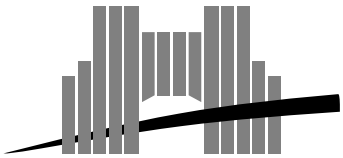
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Reference manual of the Bouclettes *parallelizer*

Pierre Boulet
Michèle Dion
Éric Lequiniou
Tanguy Risset

October 1994

Technical Report N^{O} 94-04



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Reference manual of the *Bouclettes* parallelizer

Pierre Boulet
Michèle Dion
Éric Lequiniou
Tanguy Risset

October 1994

Abstract

This documents presents the first version of the *Bouclettes* automatic parallelizer developed at LIP. It gives a detailed description of the functionalities and internal mechanics of the parallelizer, from the graphical interface to the syntactical analysis, dependence analysis, scheduling, allocation and rewriting modules as well as the tools that are used.

Keywords: parallelizer, nested loops, compiler, dependence analysis, scheduling, allocation, loop rewriting, PIP

Résumé

Ce document présente la première version du paralléliseur automatique *Bouclettes* développé au sein du LIP. Il donne une description détaillée des fonctionnalités et des mécanismes internes du paralléliseur, de l'interface graphique aux modules d'analyse syntaxique, d'analyse de dépendances, d'ordonnancement, d'allocation et de réécriture, ainsi que les outils que ces modules utilisent.

Mots-clés: paralléliseur, nids de boucles, compilateur, analyse de dépendances, ordonnancement, allocation, réécriture de boucles, PIP

Contents

1 Introduction

This document gives a detailed overview of the functionalities and internal mechanics of the *Bouclettes* parallelizer developed in the "Paradigme" group at the LIP, ENS-Lyon. Here is described the first basic version of this parallelizer. This program is intended as a working basis for future developments.

The user can enter a simple loop nest (uniform and perfect) and transform it step by step (dependence analysis, scheduling, allocation and rewriting) into a parallel loop nest where the outer loop is sequential and the inner loops are parallel.

We first describe the graphical user interface and its implementation and then each module constituting the program. In the appendix we provide the reader with an explanation of PIP's use and with an example of rewriting.

2 The graphical interface

The goal of this section is to briefly describe the interface module. It has been developed so as to propose a common interface to the different parallelization tools under development.
This interface works in the OpenWindows 3.0 and SunOs 4.1.3 environment. It has been developed in C using the XView library. This interface has the same look and feel than the other OpenWindows applications. To fulfill this goal, the developments have been done using the Guide interface generator.
The interface calls binary executables and the communications between the modules is done via files.

2.1 Description of the interface

The different functionalities of the interface are shown below.

2.1.1 File Menu

This menu allows the user to load a new file (Load button) or to quit the application (Quit button). Once a file is loaded the user can edit it in the edition window and by pressing the right mouse button, the traditional "textedit" menu appears.

2.1.2 Analysis Button

This button launches the syntactical analysis of the loaded file. The syntax of the file is verified and the resulting internal structure is rewritten in a new window.

2.1.3 Dependences Button

This button launches the dependences analysis. It prints the found dependences in a new window.

2.1.4 Environment Button

This button prints the environment in a new window (arrays and their size, loop indices and their depth and parameters of the problem).

2.1.5 Scheduling Menu

This button launches the scheduling. This menu allows to choose between different scheduling techniques but at the time being, only the linear scheduling is implemented.

2.1.6 Rewriting Button

This button starts the rewriting stage following the chosen schedule and allocation.

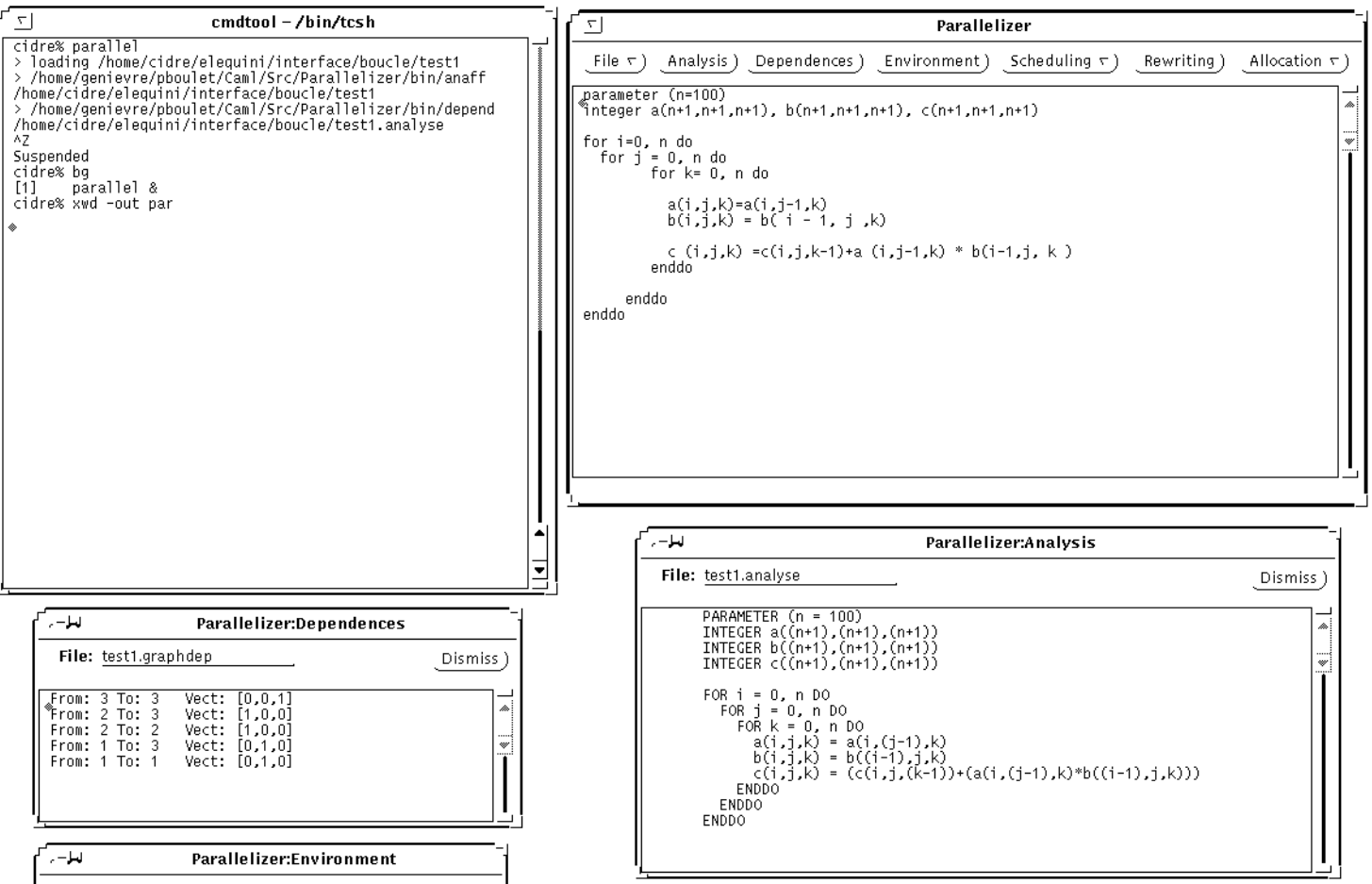


Figure 1: Global view of the interface

2.1.7 Allocation Menu

This menu proposes different allocation techniques. For the moment the chosen technique is the unimodular completion of the scheduling vector. There is no use of this button for the moment as the completion is done during the rewriting.

2.2 The files

This section describes the different files used by the interface module. These files are in directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/interface

- **parallel.G**: this file describes the interface. It is written in a specific language and is automatically generated by Guide.
- **parallel_ui.h**: include file used by parallel_ui.c, generated from parallel.G.
- **parallel_ui.c**: this file describes the different objects of the interface. It is generated from parallel.G.
- **parallel_stubs.c**: this file contains the main program that activates the widgets and controls the events. This file contains particularly all the call-backs to the procedures associated with the events. This file is rebuilt at each modification of the interface. It can also be edited to directly modify the program.
- **parallel**: executable file

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/interface/DOC

2.3 Communication between the modules and the interface

As stated in the introduction, the communication between the different modules and between the modules and the interface is done by files. There are two kinds of files: the caml objects files and the ASCII ones. The first ones can only be used by caml programs because the contain caml data structures. The second ones are human readable and are generated by the called caml programs and then used by the interface, they are the display.

Remark: The location of the executable files corresponding to the different modules is given in the file parallel_stubs.c by the constant PARALLEL_HOME. The current path is "/home/genievre/pboulet/Caml/Src/Parallelizer/bi

Here is the list of the files currently created and used for the communication:

- **file.ana**: caml file, output of the analyse of the input file, produced by `analyse`.
- **file.analyse**: ASCII output of `analyse`.
- **file.env**: caml file corresponding to the environment, produced by `analyse`.
- **file.environment**: ASCII output of the environment.
- **file.gd**: caml file corresponding to the dependence analysis, produced by `depend`.
- **file.graphdep**: ASCII output of `depend`.
- **file.sch**: caml file, output of the scheduling phase, produced by `schedule`.
- **file.schedule**: ASCII output of `schedule`.
- **file.par**: caml file, output of the rewriting module, produced by `rewrite`.
- **file.parallel**:ASCII output of `rewrite`.

2.4 Modifications

This section explains how to modify the interface module.

2.4.1 Modification of the components of the interface

Here is explained the method to modify the graphical interface, like to add a new button or a new window.

1. Start Guide: type guide
   (the executable is under /usr/local/guide/bin/guide).
2. Load the file parallel.G form the interface of Guide (File menu, Load option).
3. Modify the interface using guide's possibilities. One can add new windows, buttons, menus, etc. New functions can also be attached to events.
4. Save the changes (File menu, Save option).
5. Type `make` to update the files which depend on parallel.G (execution of the command `gxv parallel.G` which updates the source files and the eventual recompilation of these files).

2.4.2 Modification of the code

The actions relative to the different events have to be written by the programmer. The method to follow is summarized below:

1. Attach some functions to the events of the interface using guide (see above).
2. Either the function is simple, then its code can be entered directly with guide, or the function is more complex, only its name is given. This function has now to be coded. It is done in the file parallel_stubs.c where the body of the function has to be completed.
3. Do not forget to rerun `make` for the compilation of the executables.

3 The tools module

This module contains several tools that are useful for nearly all other modules of the parallelizer *Bouclettes*. These tools are mainly a module of calculus over rational numbers and another module of calculus over (affine) expressions. A third module `toolbox.ml(i)` contains a few general use functions that don't fit anywhere else.

3.1 The files of the tools module

The files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/tools

- **rational.ml (.mli)**: calculus using rational numbers.
- **expressions.ml (.mli)**: affine and non affine expression manipulation.
- **toolbox.ml (.mli)**: miscellaneous functions.

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/tools/DOC

3.2 The rational calculus

The type `rational` is defined by:

```
type rational == int * int;;
```

The file `rational.mli` defines the functions that operate on this type:

```
value pgcd : int -> int -> int
and pgcd_liste : int list -> int
and ppcm : int -> int -> int
and ppcm_liste : int list -> int
and simplr : rational -> rational
and addr : rational -> rational -> rational
and subr : rational -> rational -> rational
and multr : rational -> rational -> rational
and divr : rational -> rational -> rational
and invr : rational -> rational
and floorr : rational -> rational
and ceilr : rational -> rational
and negr : rational -> rational
and equalr : rational -> rational -> bool
and eqr : rational -> rational -> bool
and greater_thanr : rational -> rational -> bool
and gtr : rational -> rational -> bool
and int_of_rational : rational -> int
and rational_of_int : int -> rational
and outputr : out_channel -> rational -> unit
and printr : rational -> unit;;
```

All these functions return simplified rational expressions given by function `simplr`. The functions `eqr` and `gtr` are declared infixed by:

```
#infix "eqr";;
#infix "gtr";;
```

3.3 The expressions

3.3.1 The expression type

The expression type is `expr` and is defined by:

```
type ident == string;;
type array = {Id:ident; Indices:expr list}
  and expr =
      MIN of expr list
    | MAX of expr list
    | PLUS of expr*expr
    | MINUS of expr*expr
    | MULT of expr*expr
    | DIV of expr*expr
    | MIN_UN of expr
    | FLOOR of expr
    | CEIL of expr
    | INT of int
    | RATIO of rational
    | VAR of ident
```

```
  | BOOL of bool
  | AREF of array;;
```

It represents an expression tree.

3.3.2 The normalized affine form

Most of the functions defined here deal with affine expressions which are defined as sums of numbers and factors of numbers and an identifier. The major function of this module is `normalizeaf` that takes as input an expression and returns the exception `non_affine` is the input expression is not affine or it returns a normalized form of the input expression if it is affine. This normalized form is defined as follows:

- it is a comb of `PLUS`
- each leaf is a `MULT (RATIO r, VAR ident)` except the last one that is a `RATIO r`
- each identifier appears only once in the expression

The normalized form is built in several successive stages:

- All the `MINUS` and `MIN_UN` nodes are removed so that the only remaining additive nodes are `PLUS` nodes. This is done by the function `simpl_minus`.
- The second stage is a simplification (by function `simpl_nb`:
  - integers are transformed into rationals
  - factors of a number and an identifier are transformed into `MULT (RATIO r, VAR id)`
  - quotients are computed or replaced by multiplications when it is possible
- The third stage is a distribution of the constants that are factors of a subexpression (by function `distribute`).
- The fourth stage is the transformation of the tree of `PLUS`'s into a *comb* of `PLUS`'s.
- We then factorize the variables and the constants (function `factorize`) and remove the zero branches that may result from the previous operation (function `prune`).

3.3.3 The functions

- `normalizeaf : expr -> expr`
  normalizes its input
- `is_affine : expr -> bool`
  indicates if its input is an affine expression or not
- `addaf : expr -> expr -> expr`
  computes the normalized form of the sum of two affine expressions
- `subaf : expr -> expr -> expr`
  computes the normalized form of the difference of two affine expressions
- `mult_int_af : int -> expr -> expr`
  computes the normalized form of the product of an affine expression by an integer
- `outpute : out_channel -> expr -> unit`
  `and printe : expr -> unit`
  outputs an expression on a specified output channel or on the standard output

- `outputaf : out_channel -> expr -> unit`
  `and printaf : expr -> unit`
  outputs an affine expression on a specified output channel or on the standard output

- `neg_expr : expr -> expr`
  negates an expression

3.4 The toolbox

The file **toolbox.mli** says it all:

```
(* outputs a list of elements outputed by "output_fun" on channel "ch" *)
value output_list : out_channel -> (out_channel -> 'a -> unit) -> 'a list -> unit;;

(* outputs an int on channel ch *)
value output_int : out_channel -> int -> unit;;

(* executes a command "c" with arguments "a" ("a" is a string vect)*)
value exe : string -> string vect -> unit;;

(* gives the name of a file without the extension *)
value base_filename : string -> string;;
```

4 The matrices module

In this section we describe the functions that have been written to do matrix computations. In the directory **matrices** can be found functions to compute the matrix product, the inverse of a matrix, elementary row and column operations and the Hermite form computation that leads to the unimodular completion of an integer vector.

4.1 The files of matrices

The files can be found in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/matrices

- **det_inv.ml (.mli)**: output of a rational matrix, rational matrices product and rational matrix inverse computation

- **mat_elem.ml (.mli)**: elementary operations on integer matrices (row, column management, transpose, matrix product and output)

- **hermite.ml (.mli)**: computation of the Hermite form of an integer matrix and unimodular completion of an integer vector using the Hermite form

- **complete.ml**: the source code of the executable **complete** that completes an integer vector into a unimodular matrix

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/matrices/DOC

4.2 The functions available

4.2.1 det_inv

This module contains functions operating on rational matrices.

- `identityr : int -> int -> (int * int) vect vect`
  `identity n p` generates a $n \times p$ identity matrix
- `randomatr : int -> int -> (int * int) vect vect`
  `randomatr n p` generates a $n \times p$ "random" matrix
- `outputvectr : out_channel -> (int * int) vect -> unit`
  outputs a rational vector on the given channel
- `outputmatr : out_channel -> (int * int) vect vect -> unit`
  `printmatr : (int * int) vect vect -> unit`
  output a rational matrix on the given channel or the standard output
- `prodmatr : (int * int) vect vect -> (int * int) vect vect -> (int * int) vect vect`
  computes the product of its two matrix arguments
- `inv_matr : (int * int) vect vect -> (int * int) vect vect`
  computes the inverse of its argument

4.2.2 mat_elem

This module implements elementary operations on integer matrices.

- `mat_copy : int vect vect -> int vect vect`
  `mat_copy a` makes a copy of matrix `a`
- `minus_row : int vect vect -> int -> int vect vect`
  `minus_row a i` changes the sign of row `i` of `a`, *warning*: this function modifies its argument
- `minus_column : int vect vect -> int -> int vect vect`
  `minus_column a j` changes the sign of column `j` of `a`, *warning*: this function modifies its argument
- `exchange_rows : int vect vect -> int -> int -> int vect vect`
  `exchange_rows a i j` exchanges rows `i` and `j` of matrix `a`, *warning*: this function modifies its argument
- `exchange_columns : int vect vect -> int -> int -> int vect vect`
  `exchange_columns a i j` exchanges columns `i` and `j` of matrix `a`, *warning*: this function modifies its argument
- `add_row : int vect vect -> int -> int -> int -> int vect vect`
  `add_row a i j x`: row `i` ¡- row `i` + `x` $*$ row `j` *warning*: this function modifies its argument
- `add_column : int vect vect -> int -> int -> int -> int vect vect`
  `add_column a i j x`: column `i` ¡- column `i` + `x` $*$ `column j`, *warning*: this function modifies its argument
- `small_row : int vect vect -> int -> int`
  `small_row a q` finds the column number of the smallest non zero element of row `q` of matrix `a`, whose column index is greater than `q` and returns 0 if all elements are zero unless the $\mathtt{q}^{th}$
- `small_column : int vect vect -> int -> int`
  `small_column a q` finds the row number of the smallest non zero element of column `q` of matrix `a`, whose row index is greater than `q` and returns 0 if all elements are zero unless the $\mathtt{q}^{th}$
- `identity : int -> int -> int vect vect`
  `identity n p` returns an identity (`n` by `p`) matrix
- `randomat : int -> int -> int vect vect`
  `randomat n p` returns a random (`n` by `p`) matrix

- `outputvect : out_channel -> int vect -> unit`
  `printvect : int vect -> unit`
  prints a vector

- `outputmat : out_channel -> int vect vect -> unit`
  `printmat : int vect vect -> unit`
  prints a matrix

- `transpose : int vect vect -> int -> int vect vect`
  `transpose a q` transposes the sub-matrix of `a` which indices are greater or equal to `q`, *warning*: this function modifies its argument

- `dotij : int vect vect -> int -> int vect vect -> int -> int`
  `dotij a i b j` computes the dot product of the $\texttt{i}^{th}$ row of `a` by the $\texttt{j}^{th}$ column of `b`

- `prodmat : int vect vect -> int vect vect -> int vect vect`
  computes the matrix product

All these functions are fairly simple `vect` manipulations.

4.2.3 hermite

The function `basis` defined by

```
value basis : int vect vect -> int vect vect * int vect vect * int vect vect;;
```

computes the hermite form [Dar93] of its argument. For all matrix A of Z_n, there exists a unimodular matrix Q and a matrix H such that:

- H is upper triangular with greater or equal to 0 coefficients.
- each non diagonal coefficient is less than the diagonal coefficient of the same column (expect when the diagonal coefficient is null).
- $A = QH$.

The three matrices that `basis` returns are respectively H, Q and Q^{-1}.

Function `complete` defined by

```
value complete : int vect -> int vect vect;;
```

completes its argument (a vector) into a unimodular matrix [Dar93].

5 The analysis module

This section presents the internal structure of a program and the analyzer that recognizes it. This module, developed in the directory `analysis` contains the definition of the internal representation of a program and a few elementary operations on such a structure.

5.1 The files of analysis

The files are in the directory **/home/genievre/pboulet/Caml/Src/Parallelizer/analysis**

- **struct.ml (.mli)**: exceptions and types defined in this module and two extraction functions
- **analysis.ml (.mli)**: lexical and syntactical analyzer of a program and printing functions
- **anaff.ml**: defines the executable analyzer

The documentation files are in **/home/genievre/pboulet/Caml/Src/Parallelizer/analysis/DOC**

5.2 The type definitions

The internal structure of a program is defined in `struct.mli`:

```
type index = {Index:ident; Lower_bound:expr; Upper_bound:expr; Stride:expr};;
type loop = {Ind_do:index; Body_do:inst list}
  and cond = {Test:expr; Then:inst list; Else: inst list}
  and affect = {Lexpr:array; Rexpr:expr}
  and forall = {Ind_forall:index list; Body_forall:inst list}
  and inst = LOOP of loop
    | COND of cond
    | AFFECT of affect
    | FORALL of forall;;
type prog = {Declarations:ident list; Instructions:inst list};;
```

A `prog` consists in a list of declarations (type `ident` is defined in module **expressions**) and a list of instructions. An instruction is a loop, a conditional statement, a forall statement or an affectation.

The environment is handled by a hash table:

```
type fortran_type = INTEGER | REAL | LOGICAL;;
type quality = PARAM of expr (*value*)
  | VARIABLE
  | INDEX of int (*depth in the loop nest*)
  | ARRAY of int*(expr list) (*dimension, bounds*);;
type attribute = {Type:fortran_type; Quality:quality};;
```

The environment variable `env` is of type `(string, attribute) hashtbl_t`, which means that it is a hash-table that associate an `attribute` to each `string` entry it contains. This attribute indicates which is the type of the object associated with the string and what is the object: a parameter, a variable, a loop index or an array.

5.3 The exceptions of the analysis module

The file **struct.mli** contains the definitions of two exceptions:

- `exception non_uniform of string;;`
  Which is used by a function that expects a program with uniform dependences and finds a non uniform dependence. The `string` argument of this exception is used to indicate which function has raised it.
- `exception non_perfect of string;;`
  Which is used by a function that expects a perfectly nested loop nest and finds one that is not perfectly nested.The `string` argument of this exception is used to indicate which function has raised it.

5.4 The functions of the analysis module

analysis.mli contains the definitions of:

- `analysis`:

  ```
  value analysis : string -> prog * (string, attribute) t;;
  ```

  This function takes a filename as argument and analyzes its content to return a pair: the internal representation of the program whose text is in the argument file and the environment of this program.
- `outputprog`, `printprog`, `outputenv`, `printenv`:

```
value outputprog : out_channel -> (string, attribute) t -> prog -> unit
and printprog : (string, attribute) t -> prog -> unit
and outputenv : out_channel -> (string, attribute) t -> unit
and printenv : (string, attribute) t -> unit;;
```

These functions allow to print in a human readable form a `prog` and a `(string, attribute) t`, either on a specified channel (`output...`) or on the standard output channel (`print...`).

struct.mli contains the definitions of:

- `extr_loop`:

  ```
  value extr_loop : loop -> index list * inst list;;
  ```

  This function extracts the indices of a loop nest and its body.

- `extr_perf_prog`:

  ```
  value extr_perf_prog : prog -> index list * inst list;;
  ```

  This function extracts the indices and the body of a perfect loop nest program.

5.5 How does it work?

Basically, all the functions except the analyzer are basic manipulations of the `prog` type.
The analyzer is built in two phases:

1. the lexical analyzer `analex` that takes a `char stream` and returns a `token stream`, where a `token` is defined by:

   ```
   type token = T_FOR | T_TO | T_DO | T_EGAL | T_ENDDO | T_PARG | T_PARD
     | T_PLUS | T_MINUS | T_MULT | T_DIV | T_1OP of string | T_2OP of string
     | T_SEP | T_NL | T_EXP of string | T_MAX | T_MIN | T_INT | T_REAL
     | T_LOGICAL | T_PARAM;;
   ```

2. the syntactical analyzer `analysis` that takes a `token stream` and returns the `prog` structure and the environment hash-table.

6 The dependences module

This section deals with the dependence analysis of a uniform perfect loop nest. We first build the communication graph described in [Dar93] and then use it to build the reduced dependence graph.

6.1 The files of dependences

The files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/dependences

- **graph_com.ml (.mli)** extraction of the communication graph from a `prog`
- **graph_dep.ml (.mli)** construction of the dependence graph from a communication graph
- **depend.ml** the source of the executable that does the dependence analysis

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/dependences/DOC

6.2 The communication graph

We use this structure as an intermediate structure to compute the dependence graph (see section 6.3).

6.2.1 Definition and type

The communication graph represents the communications that are needed given an allocation of the data and the computations. The vertices are the statements of the body of the loop nest and the data arrays. The edges are the difference between the indices of the two vertices they join. This graph is represented by the type:

```
type depend  ={nomd:ident; vect: expr list};;
type instr_dep = {nomi:ident; suci:depend};;
type var_dep = {nomv:ident; sucv:depend list};;

type graph_com = {instr_list: instr_dep list;
                  var_list:var_dep list};;
```

6.2.2 The function and its implementation

The communication graph is computed by the function:

```
value  gcom : prog -> graph_com;;
```

The computation of this graph is done in three phases:

1. We first build the index of an instruction and the list of instructions (function `gcom_1`).
2. We then build an intermediate structure defined by a `list` of `instr_inter`:

   ```
   type instr_inter = {nomint:ident; ecrite:depend; lues:depend list};;
   ```

   This is done by function `gcom_2`.
3. We then convert this intermediate structure to a `graph_com` via `gcom_3`.

6.3 The dependence graph

6.3.1 Definition and type

The dependences carry the constraints on the order of evaluation of the instructions. To respect the semantic of the sequential program, we have to respect the dependences. Let us note $S_i(I)$ the instance of instruction i corresponding to iteration vector I. There is a dependence between two iterations $S_i(I)$ and $S_j(J)$ if:

- $S_i(I)$ is executed before $S_j(J)$
- $S_i(I)$ and $S_j(J)$ both reference a same memory location and at least one of these references is a write access.

See [Dar93] for more details.

As here all the dependences are uniform, the dependence graph can be represented by a reduced dependence graph whose vertices are the different statements of the body of the loop nest and whose vertices are labeled by the dependence vectors (the difference between the iteration vectors of the two instructions causing the dependence). This graph is represented internally by the type `graph_dep` defined by:

```
type v_d = {origin:string; dest:string; vd:int list};;
type graph_dep == v_d list;;
```

6.3.2 The function and its implementation

As we need all the dependences positive to compute the scheduling, the function

```
value gdep : graph_com -> graph_dep;;
```

computes the dependence graph with all the dependences positive.

This computation is done in three phases:

1. Starting with the communication graph, we first compute the flow dependences and the anti dependences.
2. We then compute the output dependences.
3. And finally we make the dependences positive.

The files **graph_dep.ml (.mli)** also contain the implementation of two functions that print a `graph_dep` on a given channel or on the standard output:

```
value outputgd : out_channel -> graph_dep -> unit
  and printgd : graph_dep -> unit;;
```

7 The Scheduling Module

This section explains how is implemented the **scheduling** in the parallelizer *bouclettes*. This module, developed in the directory `scheduling`, calculates the best linear schedule associated to a nested loop. The method used is described in [Dar93].

The module `scheduling` takes in input the results of the modules `analyse` and `dependences`. It uses the software `PIP` and its interface implemented in the module `interface_PIP` to calculate the best scheduling vector.

7.1 Files

The files are in the directory
/home/genievre/pboulet/Caml/Src/Parallelizer/scheduling:

- **loop_to_lp.ml (.mli)**: going from a nested loop to a linear program,
- **scheduling.ml (.mli)**: going from a linear program to the scheduling using the module `interface_PIP`,
- **schedule.ml**: generates the executable associated to the module `scheduling`.

The documentation files are in the directory
/home/genievre/pboulet/Caml/Src/Parallelizer/scheduling/DOC.

7.2 Type

The type "scheduling" is defined in the file `scheduling.mli`.

```
   type scheduling =
S_IF_THEN_ELSE of s_if_then_else
|SCHEDULING of expr vect
|S_BOT
and s_if_then_else = S_IF: expr; S_THEN: scheduling; S_ELSE: scheduling;;
```

The scheduling vector is represented by an expression vector which can contain parameters and conditions on parameters.

7.3 Functions

The main functions defined in the module scheduling are:

- `loop_to_lp :`
  `value v_d list -> prog -> prog_lin;;`
  The function `loop_to_lp` transforms the nested loop in a linear program. It takes in argument two objects:
  - an object of type `prog` calculated by the `module analysis`, it describes the nested loop,
  - the list of dependence vectors calculated by the module `dependences`.
- `schedule :`
  `value schedule : prog_lin -> scheduling;;`
  The function `schedule` takes in input a linear program and returns the best scheduling vector. It uses the module `interface_PIP` and more precisely `solve_lp` which returns the solution of a linear program.

7.4 How it works

The search for the best scheduling vector Given a uniform loop nest, the total execution time for a linear schedule π is given by:

$$T_{\sigma_\pi} = 1 + \max(\lfloor \pi p \rfloor, p \in Dom) - \min(\lfloor \pi q \rfloor, q \in Dom)$$

The best linear schedule is the one that minimizes T_{σ_π} over all rational vectors π such that $\pi D \geq 1$.

In [Dar93], Darte proposes a method to find the optimal scheduling vector which consists in solving only a single linear program. Finding the optimal scheduling is solving the problem:

$$\min_{XD \geq 1} \max_{(Ap \leq b, Aq \leq b)} X(p-q)$$

And, by the duality theorem of linear programming ([Sch86]), the optimal scheduling is obtained by solving the following linear problem:

$$\begin{cases} XD \geq 1 \\ X_1 A = X \\ X_2 A = -X \\ X_1 \geq 0 \\ X_2 \geq 0 \\ min(X_1 + X_2)b \end{cases}$$

For more details, see [Dar93].

Remark that this problem is linear in b. Thus, the search of the best scheduling vector for the family of domains $Ax \leq Nb$ where N is a parameter is reduced to the search on the domain $Ax \leq b$ which can be done without knowing N at compile-time.

Back to the `scheduling` module For the moment, the `scheduling` module is only implemented for perfect uniform nested loop and for the family of domains $Ax \leq Nb$.

The function `graph_dep_to_matrix` implemented in `loop_to_lp.ml` calculates the dependence matrix D. The function `get_domain` also implemented in `loop_to_lp.ml` calculates the matrix A, the vector b and eventually a parameter N. From A, b, N and D, the linear program can be generated and solved with the function `solve_lp` implemented in the module `interface_PIP`.

8 The interface_PIP module

This section explains how is implemented the interface between the parallelizer *bouclettes* and the PIP software. This module, developed in the directory `interface_PIP`, allows to solve a parameterized linear programming problem by using PIP. A linear programming problem (LP) is of type `prog_lin` (described hereafter) and the solution is of type `sol_prog_lin`. A brief recall of the way PIP works is done in section A.

8.1 Files of interface_PIP

The files are in the directory
/home/genievre/pboulet/Caml/Src/Parallelizer/interface_PIP

- **type_lin_prog.ml (.mli)**: exceptions and types defined in this module.
- **aff_lp.ml (.mli)**: all the output functions needed.
- **lp_to_pip.ml (.mli)**: going from a LP (in the `prog_lin` type) to a problem that can be solved by PIP.
- **exec_pip.ml (.mli)**: executing PIP (via Unix).
- **pip_to_sol.ml (.mli)**: going from a solution given by PIP to a solution in the `sol_prog_lin` type.

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/interface_PIP/DOC

8.2 Types of interface_PIP

The types "prog_lin" and "sol_prog_lin" are defined in the file `type_prog_lin.mli`.

- **prog_lin**:

```
type maxormin= max | min ;;
type prog_lin =
{INTORNOT: bool;
MAXORMIN:maxormin;
NBPAR: int;
NAMEPAR: string list;
POSPAR:string list;
NBVAR: int;
NAMEVAR: string list;
POSVAR:string list;
CONTEXT: expr list;
LISTINEQ: expr list;
COST: expr;
COMMENTS: string
};;
```

  - INTORNOT indicates if the LP is to be solve in integral or rational mode. For the moment the integral mode is not implemented thus it should be always false.
  - MAXORMIN indicates if the cost function must be minimized or maximized.
  - NBPAR indicates the number of parameters of the LP.
  - NAMEPAR indicates the names of the parameters of the LP

- POSPAR indicates which parameters are positive. WARNING: this information should be also present in the field CONTEXT, if the inequality indicating that a parameter is positive is not in the context, then the parameter will not be taken as positive, even if the name of the parameter is indicated in this field.
- NAMEVAR indicates the names of the variable of the LP
- POSVAR indicates which variable is positive. As for the parameters, this information should be also present in the field LISTINEQ.
- CONTEXT: the list of inequalities of the LP that involve only parameters. Each inequality is an expression that must be positive. This expression must be affine.
- LISTINEQ: the list of inequalities (affine expression positive) involving variables and parameters (possibly).
- COST: the cost function of the LP.
- COMMENTS: the comments set by the user (are output when outputting the LP).

- **sol_prog_lin**:

```
type quast=
QUAST of if_then_else
|SOL of expr vect
| BOT
and if_then_else={IF: expr; THEN: quast ; ELSE: quast};;
```

The quast is the basic structure of the solution of a parameterized LP. The leaves of the quast (which are the possible solutions) are vectors, the first coordinate of the solution is the value of the cost function. The other coordinates are the values of the variables that reach this value for the cost function (in the order specified in the variable list). For the moment, the cut of the leaf of the quast described at the end of section A is not implemented

```
type sol_prog_lin =
{S_INTORNOT: bool;
S_MAXORMIN:maxormin;
S_NBPAR: int;
S_NAMEPAR: string list;
S_NBNEWPAR:int;
S_NAMENEWPAR:string list;
S_NBVAR:int;
S_NAMEVAR:string list;
S_SOL: quast;
S_COMMENTS: string
};;
```

The fields of `sol_prog_lin` are very close to the ones of `prog_lin`. Two more appear: S_NBNEWPAR and S_NAMENEWPAR which are now useless.

8.3 Function of interface_PIP

The only function that should be used by users is:

- **solve_lp**:
  `value solve_lp:  prog_lin -> sol_prog_lin ;;`
  This function solves an LP in rational mode.

8.4 Exceptions of interface_PIP

There are several types of exceptions that can be raised but that should not. These exceptions are almost all linked to the fact that the expression in LISTINEQ and CONTEXT must be affine. These exceptions are defined in `type_lin_prog.ml` and the name of the function where they have been raised appears at the beginning (for instance, exception `get_ppcm_non_affine_in_lp` has been raised by the `get_ppcm` function).

Some problems may appear when you try to minimize (or maximize) a cost function that is not bounded. In this case we have chosen to print a Warning message on the standard output. This can be easily change into an exception.

- **unbounded solution**
  The message appearing in the non-bounded solution case is like the following:
  ```
  Warning !!  Unbounded solution
  0
  ```
  It means that the program was looking for a solution with some Big Parameters (see section A) and could not find them. This warning is printed on the screen and the program goes on. Of course, the solution given is false (it should be unbounded).

- **false solution**
  Another type of warning may appear. As explained in the section A, Some of the branches of the Quast solution may be cut, when the condition of the quast is always verified for example. In this case, we should cut the "dead leaves" but it is not implemented yet (because in my opinion it will never happen). Thus, a warning is raised if this situation happens, with the following message.
  ```
  Warning, coefficient of GP2 does not correspond to
  the sum of the coefficient of the other parameter in:
  n+p+0
  ```
  Be careful, if this warning appears, the solution found could be false. Please, inform me if it happens.

8.5 Algorithm of interface_PIP

The treatment proceeds in the following steps:

- translate the LP
- execute PIP.
- read and transform the solution

8.5.1 translating the LP

The original LP (type `prog_lin`) is transformed into an internal structure that looks like the files for PIP (type `prog_lin_for_pip` defined in `type_lin_prog`). This phase must add adequate Big Parameters in order to obtain an equivalent LP where variables are non-negative and where the goal is to minimize the first variable. This work is done with the program defined in the file `lp_to_pip.ml`.

First, we get the parameter that are declared (in `lp.POSPAR`) to be positive and we perform a variable change on the other parameters: $n \rightarrow n + GP2$ (function `get_pos_para`). We get a new LP where all the parameters are positive (with one more parameter: GP2) and a list that recalls the parameters that have been transformed. Then, a new variable "XCOST" is introduced to represent the cost function, and we introduce the first inequation: $XCOST - LP.COST \geq 0$ (resp $\leq$ if we should maximize).

Then we "format" the LP, it means that we perform the variable changes explained in section A: with a Big Parameter GP1 to ensure that the variables will be positive and we produce a pip-like LP (function `format_caml_to_pip_lp`). The result of this last function is given in a form which is very close to the PIP format (`Prog_lin_for_pip`). The coefficients of the PIP vectors (#[1 0 ...]) are in the following order:
coefficient of the cost function (only appears in the first inequality)
variables coefficient (in the order of the variable list)
constant coefficient

coefficient of GP1
coefficient of GP2
coefficient of the other parameters (in the order of the list)

8.5.2 execute PIP

This part is done in file `exec_pip.ml`. The internal structure is written in the file `#LP_PIP.p` and the Unix command :
`pip #LP_PIP.p #LP_PIP.res`
is executed. Thus the result is in the file `#LP_PIP.res`.

8.5.3 read and transform the solution

The file `#LP_PIP.res` is analyzed and stored in an primary solution (type `Sol_prog_lin`). This work is done in file `pip_to_sol.ml`.

First, in `get_sol_pip`, we analyze the output file of PIP (`#LP_PIP.res`), we get the solution in a Quast, and with the variable list and the parameter list, we put the right parameter names in the condition and the leaves of the quast.

In `simplify_quast_sol`, we remove the occurrence of the Big Parameters GP1 and GP2. If something wrong is detected, a Warning is printed (no exception raised). Then we get the solution Quast.

9 The rewriting module

This section explains how is implemented the rewriting module. This module, developed in the directory `rewriting`, allows to rewrite a loop after an integral transformation over the indices which is unimodular or lower triangular. The unimodulary transformation is usually indicated by the scheduling and allocation modules. From these modules we get a matrix and we obtain the new indices by applying this matrix to the indices of the original loop nest. The work described here allows to enumerate all the computations of the original loop nest with another loop nest of which the indices are the new computed indices. With the usual basic modules, this module uses the `scheduling` module, the `matrices` module and the `Interface_PIP` module. For the moment the transformation to apply to the nest is obtained by completing the scheduling vector into a unimodulary matrix with the Hermite algorithm. The rewriting after a non-unimodular transformation is not used in the interface with the compiler *bouclettes*.

9.1 Files of rewriting

The files are in the directory
/home/genievre/pboulet/Caml/Src/Parallelizer/rewriting

- **rewriting.ml (.mli)**: all the unimodular rewriting treatment and only the rewriting treatment.
- **rewrite.ml** : the executable program called from the `rewriting` button. Here the unimodulary completion of the scheduling vector is called.
- **triang_rewrite.ml (.mli)**: The rewriting with a lower triangular integral matrix

The documentation files are in the directory:
/home/genievre/pboulet/Caml/Src/Parallelizer/rewriting/DOC

9.2 Function of rewriting

There are two functions that should be used by users:

- **rewrite_nest**:
  `value rewrite_nest : int -> prog -> int vect vect -> prog;;`
  This function takes a nest and an integral unimodulary matrix and gives the nest which is the transformation of the nest by the matrix. The first parameter is the identification number of the transformation. Indeed, as the new index names are not specified by the user, we have chosen to name them: `NINDx_y` where `x` is the identification number of the transformation (for the moment it is always 1) and `y` is the depth of the new index (starting a 0).

- **tri_rewrite_nest**:
  `value tri_rewrite_nest : int -> prog -> int vect vect -> prog;;`
  This function takes a nest and an integral lower triangular matrix and gives the nest which is the transformation of the nest by the matrix. Be careful, the upper triangle of the matrix is not taken into account.

In practice, every integral transformation can be decomposed into the product of a unimodular and a lower triangular transformation. With the two functions described here, one can rewrite a nest after any integral transformation.

9.3 Exception of rewriting

Some classical exceptions can be raised like `non_affine` or `non_perfect`. The exceptions defined here are:

- **A_ref_in_bound** when there is an array reference in a loop bound. This exception should not happen if the analysis is done.

- **index_not_found** and **not_enough_depth** are internal exceptions that should not be raised. If they do, a string indicates in which function they are raised.

9.4 Algorithm for rewriting

The rewriting of the nest uses the technique described in [CFR93]. We find the new bounds from the outermost index to the innermost index. When working on an index at depth i, the outer-more indices (at depth $j < i$) are considered as parameters while the inner-more (at depth $j > i$) are still indices (or variables). The domain of this nest represents a parameterized convex polyhedron of dimension $n - i + 1$ if n is the depth of the original nest. We look for the extrema (minimum and maximum) of the first coordinate upon this polyhedron. The interface_PIP module allows to perform this search. For each index and each problem (min or max) a linear programming problem is written, the constraints are the inequation defining the domain and the cost function is the index to minimize. This problem is solved by PIP and we get the new bounds.

The function `rewrite_nest` gets useful information (like the list of the names of the indices of the nest) and calls the recursive function `rewrite_loop` that successively computes the bounds on the new indices. The algorithm of `rewrite_loop` is:

- if it is a real loop
  - writes the LP for the lower bound
  - solves the LP and get the lower bound
  - writes the LP for the upper bound
  - solves the LP and get the upper bound
  - recursive call for the loop body
- else modify the list of instruction with the new indices.

Modifying the list of instructions is just technical, one has to perform the variable change $(i_1, \ldots, i_n) \rightarrow U^{-1}(i_1, \ldots, i_n)$ (where U is the transformation to apply) in all the array references of the nest.

Writing the LP for a depth i is a little more complicated as you have to transfer all the inequalities defining the domain in term of the new indices ($U^{-1}(i_1, \ldots, i_n)$), to separate the context (inequalities involving only parameters) and the domain (other inequalities). Remember that the parameters are the original parameters plus the surrounding new indices.

Getting the solution means just to replace as indicated in [CFR93] the quast given by PIP by a maximum (resp minimum)upon its leaves if it is a lower bound (resp upper bound)

the function `triang_rewrite` is a little more simple, it just consist in algebraic manipulation but it is quite technical. The transformation is precisely described in [Ris94] p83.

A How does PIP work

This is a very brief explanation on one example, for more details please refer to [FT90] or [Fea88].

Let $D(m,n,k)$ be a parameterized polyhedron composed of (i,j) such that:

$0 \leq i \leq n$
$0 \leq j \leq m$
$k \leq i+j$

under the constraint $k \leq n+m$.

Suppose that we want to minimize th first variable (i) over $D(m,n,k)$. This is the type of problem solved by PIP. PIP can solve this problem and give a rational or an integral solution. Here we will only deal with rational solution as the integral part of PIP software is not interfaced yet with *bouclettes*. First of all we must be aware of three characteristics of the PIP algorithm

- it computes the **lexicographic** minimum of the vector of variables (in particular it implies that it minimizes the first variable),
- all the variables and parameters are supposed to be non-negative,
- the parameters are supposed to be integral (for instance, if n is a parameter, $n > 7$ will be written $n \geq 8$).

A.1 Input file

To code this problem for PIP we write the following file (ex1.p):

```
( (
        (variables i j)
        (parameters k m n)
        m ≥ j
        n ≥ i
        i + j ≥ k
        m + n ≥ k
        here comments
)
2 3 3 1 -1 0
(
#[0 -1 0 0 1 0 ]
#[-1 0 0 0 0 1 ]
#[1 1 0 -1 0 0 ]
)
(
#[-1 1 1 0]
)
)
```

signification of the integer row:
2 − − − > two variables
3 − − − > three parameters
3 − − − > three equations on variables
1 − − − > one equation on parameters
-1 − − − > No Big parameter (see after)
0 − − − > rational solution

signification of the vectors (#[0 1 ...):
for example, the first one #[0 -1 0 0 1 0] represents the inequality $m - j \geq 0$. The first coefficients are those

of the variables i and j (here 0 for i and -1 for j. Then comes the coefficient of the constant term (here 0), and the coefficient of the parameters k,m and n (here 1 for m, 0 for others).

Be careful, variables and parameters are supposed to be non negative. Thus, inequalities like $i \geq 0$ are not written explicitly.

A.2 Unix command

To get the solution with the pip software, we execute:
pip ex1.p ex1.res

A.3 output file

The file ex1.res contains:

```
( (
        (variables i j)
        (parameters k m n)
        m ≥ j
        n ≥ i
        i + j ≥ k
        m + n ≥ k
        here comments
)
(if #[ -1 1 0 0]
        (list #[ 0 0 0 0]
            #[ 1 0 0 0]
        )
        (list #[ 1 -1 0 0]
            #[ 0 1 0 0]
        )
)
)
```

which can be read as:

if $m \geq k$
    $(i, j) = (0, k)$
else
$(i, j) = (k - m, m)$

The vectors (#[-1 1 ...) are read the same way, but they only concern parameters and constant term. The first coefficients are for the parameters and the last is for the constant term.

A.4 Getting the maximum instead of the minimum

To compute the maximum over a domain $D(z)$, we compute the minimum of $GP1 - D(z)$, $GP1$ being a parameter as big as we want (Big parameter). This ensure that $GP1 - D(z)$ will have positive coefficient. The rule for internal PIP computations is the following: when PIP has to decide the sign of an expression, if the coefficient of the Big Parameter is not null, then it gives the sign of the whole expression. Be careful, we cannot use two Big Parameters otherwise this last rule would become false. In particular we cannot set inequalities like $n \geq GP1$ in the context.

Three stages:

- we introduce a new parameter $GP1$ and we perform a variable change upon all the **variables** (without touching the parameters: $(i'_1, i'_2, ..., i'_n) = (GP1 - i_1, GP1 - i_2, ..., GP1 - i_n)$,
- we compute the lexicographic minimum of the new polyhedron and we get the solution as a function of $(p_1, ..., p_m)$ (the parameters) and $GP1$,
- we perform the variable change the other way around: $(i_1, i_2, ..., i_n) = (GP1 - i'_1, GP1 - i'_2, ..., GP1 - i'_n)$ and we get the lexicographic maximum of our original polyhedron. If the solution depends upon $GP1$, then the polyhedron was not bounded.

In practice:

- we add one parameter "GP1" that we force to be a Big parameter. (the fifth integer in the integer row of the input file indicates its rank in the (variable+constant+parameters) list (starting at 0).
- we change the signs of the coefficients corresponding to **variables** and we set the coefficient of the new parameter to the sum of these coefficients (before having changed their sign)
- when we read the solution, we take the opposite and we ignore the coefficient of the Big Parameter (or we check that it is one if we are not sure that the polyhedron is bounded).

Example: the lexicographic maximum upon the previous polyhedron.

```
( (
        (variables i j)
        (parameters k m n GP1 (Big par))
        m ≥ j
        n ≥ i
        i + j ≥ k
        m + n ≥ k
        here comments
)
2 4 3 1 6 0
(
#[0 1 0 0 1 0 -1 ]          m ≥ GP1 − j
#[1 0 0 0 0 1 -1]           n ≥ GP1 − i
#[-1 -1 0 -1 0 0 2]         2GP1 − i − j ≥ k
)
(
#[-1 1 1 0 0]           m + n ≥ k
) )
```

and the result:

```
( (
        (variables i j)
        (parameters k m n GP1 (Big par))
        m ≥ j
        n ≥ i
        i + j ≥ k
        m + n ≥ k
        here comments
)
(list #[ 0 0 -1 1 0]        (GP1 − i, GP1 − j) = (GP1 − n, GP1 − m)
  #[ 0 -1 0 1 0]            thus: (i, j) = (n, m)
) )
```

A.5 Dealing with non negative variables

If you look carefully at the transformation that we have performed in the last section, you will realize that the variables of the original LP are not any more supposed to be positive. As we have performed the variable change, $G - i_1$ is positive whatever the sign of i_1 is. Thus we are able to find maxima of problems where variables are of any sign.

Suppose now that we want to compute the minimum of a polyhedron which variables are not positive. We can perform a diagonal shift of all the domain. This shift must be large enough to bring the whole domain, in the positive quadrant. Thus we can use the big parameter and perform the variable change: $(i'_1, i'_2, ..., i'_n) = (GP1 + i_1, GP1 + i_2, ..., GP1 + i_n)$.

The new variables are positive (because $GP1$ can be as big as possible), and the lexicographic minima of the two domains are the same, except that one is shifted by $(GP1, GP1, \ldots, GP1)$.

Thus we compute the minimum upon $(i'_1, \ldots, i'_n)$ and we subtract $(GP1, GP1, \ldots, GP1)$ to the solution. If there remains some G is the result, then the original polyhedron was not bounded.

The last problem to solve is about parameters. As we have mentioned before, we cannot allow two big parameters. Thus, the variable change that we just explained apply only to variables, not to parameters, and parameters are also supposed to be positive.

To allow parameter of any sign, we must perform the same kind of manipulation: a "change of parameter": $(n'_1, n'_2, ..., n'_n) = (GP2 + n_1, GP2 + n_2, ..., GP2 + n_n)$. Where $GP2$ is a new parameter which is as big as we want (but which has no special property for PIP). Then we compute the solution in term of the new parameters and we perform the parameter change the other way around. The only difference with the variable treatment is that we should cut impossible leaf ourselves.

Because of the way PIP computes the solution [CFR93], the final solution will not contain any $GP2$ unless it is not bounded but suppose for example that a condition of the resulting quast is something like

if $m - 2G \geq 0$
    $(i, j) = (0, k)$
else
$(i, j) = (k - m, m)$

Then we have to change that in $(i, j) = (k - m, m)$ because $m - 2G$ cannot be positive.

I don't think this situation can happen thus it is not implemented yet

B Example of rewriting

This section illustrates by an example, the use of the rewriting module functions. The example is taken in [Ris94] p86.

consider the following nest:

```
DO i = 0 , m
      DO j = 0 , n
            DO k = 0 , i+j
                  S(i,j,k)
            ENDDO
      ENDDO
ENDDO
```

and suppose that, for a very important purpose, you sincerely want to apply the following matrix T to the nest:

$$T = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 4 & 3 \\ 2 & 0 & 0 \end{pmatrix}$$

When working into interpreted camllight (execute `pboulet/bin/cl`), open all the modules by including `go.ml` for instance. The nest is stored in the file `rewriting/Ex/test_these`. The first thing to perform is the analysis:

```
#let (p1,q1) = analyse "test_these";;
```

The we construct the matrix T:

```
#let m0= make_matrix 3 3 0;;
#vect_assign m0.(0) 2 1;;
#vect_assign m0.(1) 1 4;;
#vect_assign m0.(1) 2 3;;
#vect_assign m0.(2) 0 2;;
```

Then we decompose `m0`. Be careful, we need the product of a lower triangular matrix and a unimodulary one thus we have to perform some transposition if we want to use the Hermite decomposition.

```
#open "mat_elem";;

let m1=transpose m0 0;;
let (m11,m12,m13)= basis m1;;
let m2=transpose m12 0;;
let m3=transpose m11 0;;
```

Now we have T=`m3*m2` with:

$$m3 = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 4 & 0 \\ 0 & 0 & 2 \end{pmatrix} \text{ and } m2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

We first apply m2 to the nest:

```
#let p2= rewrite_nest 1 p1 m2;;
rewriting the loop ...
p2 :  prog = {Declarations=[]; Instructions=[LOOP ...
#printprog q1 p2;;

FOR NIND1_0 = 0,((1*n)+((1*m)+0)) DO
   FOR NIND1_1 = max(0,((-1*m)+((1*NIND1_0)+0))), ((1*n)+0) DO
      FOR NIND1_2 = max(((1*NIND1_0)+((-1*NIND1_1)+0)),0), ((1*m)+0) DO
         a(((1*NIND1_2)+0),((1*NIND1_1)+0),((1*NIND1_0)+0)) =
            b(((1*NIND1_2)+0),((1*NIND1_1)+0),((1*NIND1_0)+-1))
      ENDDO
   ENDDO
ENDDO
```

Which is exactly the result obtained in [Ris94] p89. Apply now the lower triangular matrix `m3`:

```
#let p3= tri_rewrite_nest 2 p2 m3;;
p3 :  prog = {Declarations=[]; Instructions=[LOOP ...
#printprog q1 p3;;

FOR NIND2_0 = (0+(1*0)), (0+(1*((1*n)+((1*m)+0)))) DO
   FOR NIND2_1 = (((3*NIND2_0)+0)+(4*max(0,((-1*m)+((1*NIND2_0)+0))))),...
         ...(((3*NIND2_0)+0)+(4*((1*n)+0))), 4 DO
```

```
      FOR NIND2_2 =(0+(2*max(((7/4*NIND2_0)+((-1/4*NIND2_1)+0)),0))), ...
            ...  (0+(2*((1*m)+0))), 2 DO
         a(((1/2*NIND2_2)+0),((-3/4*NIND2_0)+((1/4*NIND2_1)+0)),((1*NIND2_0)+0)) =
            b(((1/2*NIND2_2)+0),((-3/4*NIND2_0)+((1/4*NIND2_1)+0)),((1*NIND2_0)+-1))
      ENDDO
   ENDDO
ENDDO
```

Which is exactly the nest obtained in [Ris94] p90. All the commands described here are written in the file `rewriting/Ex/test_ex.ml`.

References

[CFR93] J.F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. Technical Report 93-15, LIP, Lyon , France, 1993.

[Dar93] Alain Darte. *Techniques de parallélisation automatique de nids de boucle.* PhD thesis, Ecole Normale Supérieure de Lyon, 1993.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[FT90] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel pip. Technical Report 90-2, Institut Blaise Pascal, UPMC, Laboratoire MASI, January 1990.

[Ris94] Tanguy Risset. *Parallélisation automatique: du modèle systolique à la compilation de nids de boucles.* PhD thesis, ENS-Lyon, February 1994.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.