Clan

A Polyhedral Representation Extractor for High Level Programs
Edition 1.0, for Clan 1.0.0
May 4th 2008

Cédric Bastoul

This manual is for Clan version 1.0.0, a software which extracts the polyhedral representation of some parts of high level programs written in C, C++, C# or Java.

It would be quite kind to refer the following paper in any publication that results from the use of the Clan software or its library (the reason to cite it is, amongst many other interesting things, it defines what is a *SCoP*, or *static control part*):

```
@InProceedings{Bas03,
  author =    {C\'edric Bastoul and Albert Cohen and Sylvain Girbal and
               Saurabh Sharma and Olivier Temam},
  title =     {Putting Polyhedral Loop Transformations to Work},
  booktitle = {LCPC'16 International Workshop on Languages and
               Compilers for Parallel Computers, LNCS 2958},
  pages =     {209--225},
  month =     {october},
  year =      2003,
  address =   {College Station, Texas}
}
```

Table of Contents

1 Introduction

Clan is a free software and library that translates some particular parts of high level programs written in C, C++, C# or Java into a polyhedral representation. This representation may be manipulated by other tools to, e.g., achieve complex program restructurations (for optimization, parallelization or any other kind of manipulation). It has been created to avoid tedious and error-prone input file writing for polyhedral tools (such as CLooG, LeTSeE, Candl etc.). Using Clan, the user has to deal with source codes based on C grammar only (as C, C++, C# or Java).

Clan stands for *Chunky Loop ANalyzer*: it is a part of the Chunky project, a research tool for data locality improvement (see [Bas03a], page 35). It is designed to be the front-end of any source-to-source automatic optimizers and/or parallelizers.

Clan is a very basic tool since it is only a translator from a given program representation to another representation. Nevertheless the current version is still under evaluation, and there is no guarantee that the upward compatibility will be respected. A lot of reports are necessary to freeze the library API and the input/output file shapes. The current output file format has been designed after discussions between several compilation researchers from various institutions. Thus you are very welcome and encouraged to send reports on bugs, wishes, critics, comments, suggestions or (please !) successful experiences to cedric.bastoul@inria.fr.

2 Polyhedral Representation of Programs

If you are reading the Clan's user manual, you probably don't need any explanation about the Polyhedral Model. It's unlikely someone will read this manual by chance. However some vicious advisor may ask their poor engineers/interns/students to work for the very first time on this exciting topic. Most papers on Polyhedral Compilation are hard to read. Despite my efforts, mine are no exception according to some reviewers... Hence I give there a new try to provide a comprehensive explanation of the polyhedral model without the size and style limits of a classical research paper.

Be aware that to be able to understand the Polyhedral Model, there are few prerequisites. You should not read the following while you still ignore what is:

- a `for` loop construction in C programs,
- an *affine expression*,
- a *vector*,
- a *matrix*,
- a *matrix-vector multiply*.

2.1 Motivation: Program Transformations

A direct translation of high level programs written, e.g., in C to assembly then object code is likely to produce (very) inefficient applications. Architectures are quite complex, including several levels of cache memory, many cores, deep pipelines, various number of functionnal units, of registers etc. The list of such "architectural features" is growing with each new generation of processors. To achieve the best performance, the object program must do a smart use of these features. Programmers use high level languages for productivity and portability: typically they do not have to take care of the target architecture but to ensure they do write programs that produce the right output. Hence, the problem of mapping the program to the target architecture in the most efficient way is left to the compiler.

The compiler may see a high level program as a specification *of an output*. The program is a list of operations to be executed to produce the output. As long as the output is guaranteed to be as the programmer specified in his code, the compiler is free to modify the program. For instance, let us imagine we are working on an architecture with only three registers and we consider the following statements written by a programmer:

```
x = a + b;
y = c + d;
z = a * b;
```

It is easy to see that we can reorder the three statements in any way without modifying the semantics (no statement reads or writes a variable that another statement writes). Because of the lack of registers, the solutions such that the first and the third statements are one after the other are better because `a` and `b` will be put in the processor registers by one statement and can be reused directly by the other one without reading to memory (this is called a *data locality improving* transformation). Hence a better statement order is, e.g.:

```
x = a + b;
z = a * b; // a and b are still in processor registers
y = c + d;
```

We could also notice that it is possible to run the three statements in parallel and explicit this in the way the compiler and/or the architecture is able to understand. Here we use OpenMP to describe parallelism. It is supported in GCC since 4.2 version (this is called a *parallelizing* transformation):

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     x = a + b;
   }
   #pragma omp section
   {
     y = c + d;
   }
   #pragma omp section
   {
     z = a * b;
   }
}
```

However, the right way to optimize this program is probably a mix of these two techniques, especially if the target architecture have some limitations to run too many operations in parallel:

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     x = a + b;
     z = a * b;
   }
   #pragma omp section
   {
     y = c + d;
   }
}
```

Such transformations are quite trivial. The reason is the statements are executed only once. The real sport begins when we have to deal with loops as we will see momentarily. However, Clan users have to be conscious that we *need* to transform programs to achieve the best performance and that the best transformation that have to be discovered (with often many, many efforts) and performed may be quite complex. Hence the need of powerful model and tools (Who said Polyhedral Model and CLooG ? Right !).

2.2 Thinking in Polyhedra

Since the very first compilers, the internal representation of programs is the *Abstract Syntax Tree*, or AST. In such representation, each statement appears only once even if it is executed many times (e.g., when it is enclosed inside a loop).

This is a limitation for program analysis. For instance if a statement *depends* on another statement (i.e., they access the same memory location and at least one of these accesses is a write), we will consider both statements as unique entities while the dependence relation may involve only few statement executions.

This is a limitation for program transformations. Loop transformations operate on statement executions. For instance, because they consider all statement executions at the same time, present day production compilers are not able to achieve loop fusion (that tries to merge the loop bodies of two loops) if the loop bounds of the two loops do not match.

This is a limitation for program manipulation flexibility. Trees are very rigid data structures that are not easy to manipulate. Program transformation may require very complex transfor-

mations that will imply deep modifications of the control flow. Hence, for complex program restructuration, the need for a more precise, more flexible representation.

The Polyhedral Model is a convenient alternative representation which combines analysis power, expressiveness and high flexibility. The drawback is it breaks the classical structure of programs that every programmer is familiar with. It requires some (real) efforts to be smoothly manipulated, but it definitely worth it. It is based on three main concepts, *iteration domain*, *scattering function* and *access function* that are described in depth in the following sections.

A program part that can be represented using the Polyhedral Model is called a **Static Control Part** or **SCoP** for short.

2.2.1 Iteration Domain

The key aspect of the Polyhedral Model is to consider *statement instances*. A statement instance is *one* execution of a statement. A statement outside a loop has only one instance while those inside loops may have many. Let us consider the following code with two statements `S1` and `S2`:

```
pi = 3.14;                // S1
for (i = 0; i < 5; i++)
  A[i] = pi;              // S2
```

The list of statement instances is the following (we just have to fully unroll the loop):

```
pi = 3.14;
A[0] = x;
A[1] = x;
A[2] = x;
A[3] = x;
A[4] = x;
```

Each instance of a statement which is enclosed inside a loop may be referred thanks to its outer loop counters (or *iterators*). In the Polyhedral Model we consider statements as functions of the outer loop counters that may produce statement instances: instead of simply `"S2"`, we use preferably the notation `S2(i)`. For instance we denote the statement instance `A[3] = x;` of the previous example as `S2(3)`. This means *instance of statement* `S2` *for* `i = 3`. If a statement `S3` is enclosed inside two loops of iterators `i` (outermost loop) and `j` (innermost loop), we would denote it `S3(i,j)`, and so on with more enclosing loops.

The ordered list of iterators (ordered from the outermost iterator to the innermost iterator) is called the **iteration vector**. For instance the iteration vector for `S3` is `(i,j)`, for `S2` it is `(i)`, and for `S1` it is empty since it has no enclosing loop: `()`. A more precise reading at the notation `S2(3)` would show that it denotes the instance of statement `S2` for the iteration vector `(2)`.

Obviously, dealing with statement instances does not mean we have to unroll all loops. First because there would be probably too many instances to deal with, and second because we probably just don't know how many instances there are. For instance in the following loop it is not possible to know (at compile time) how many times the statement `S3` will be executed:

```
for (i = 2; i <= N; i++)
  for (j = 2; j <= N; j++)
    A[i] = pi;              // S3
```

Such a loop is said to be *parametric*: it depends on (at least) a value called a *parameter* which is not modified during the execution of the whole loop, but is unknown at compile time. Here the only parameter is `N`.

A compact way to represent all the instances of a given statement is to consider the set of all possible values of its iteration vector. This set is called the **iteration domain**. It can be conveniently described thanks to all the constraints on the various iterator the statement depends on. For instance, let us consider the statement `S3` of the previous program. The

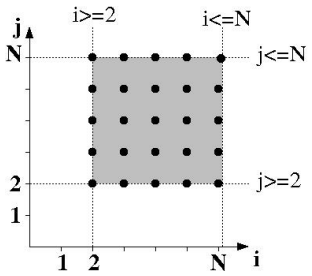
iteration domain is the set of iteration vectors `(i,j)`. Because of the parameter, we are not able to achieve a precise list of all possible values, it would look like this:

```
(2,2)  (2,3)  (2,4)  ...  (2,N)
(3,2)  (3,3)  (3,4)  ...  (3,N)
...    ...    ...    ...  ...
(N,2)  (N,3)  (N,4)  ...  (N,N)
```

A better way is to say it is the set of iteration vectors `(i,j)` such that `i` is an integer greater or equal than 2 and lower or equal than `N`, and `j` is an integer greater or equal than 2 and lower or equal than `N`. This may be written in the following mathematical form:

$$D_{S3} = \{(i,j) \in Z^2 \mid 2 \leq i \leq N \wedge 2 \leq j \leq N\}$$

It is easy to see that this iteration domain is a part of the 2-dimensional space Z^2. We often use in our research papers a graphical representation that gives a better view of this subspace:



Here the iteration domain is specified thanks to a set of constraints. When those constraints are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a *polyhedron* (more precisely this is a *Z-polyhedron*, but we use *polyhedron* for short). Hence the Polyhedral Model.

To facilitate the manipulation of the affine constraints, we use a matrix representation. To write it, we use the *homogeneous* iteration vector: it is simply the iteration vector with some additional dimensions to represent the parameters and the constant. For instance for the statement `S3`, the iteration vector in homogeneous coordinates is `(i,j,N,1)` (we will now call it *iteration vector* directly for short). Then we write all the constraints as affine inequalities of the form $p(i) \geq 0$. For instance for the statement `S3` the set of constraints is:

$$\begin{cases} i - 2 & \geq 0 \\ -i + N & \geq 0 \\ j - 2 & \geq 0 \\ -j + N & \geq 0 \end{cases}$$

Lastly, we translate the constraint system to the form **domain matrix** `*` *iteration vector* `>= 0` (please someone show me how to do this in TeX -not LaTeX- for the texinfo manual !):

```
[  1  0  0 -2 ]   [ i ]    [ 0 ]
[ -1  0  1  0 ]   [ j ]    [ 0 ]
[  0  1  0 -2 ] * [ N ] >= [ 0 ]
[  0 -1  1  0 ]   [ 1 ]    [ 0 ]
```

The domain matrix (along with the iteration vector which is most of the time an implicit information) will be used in all our tools to provide the informations on the iteration domain of a given statement.

2.2.2 Scattering Function

There is no ordering information inside the iteration domain: it only describes the set of statement instances but **not** the order in which they have to be executed relatively to each other. In the past the lexicographic order of the iteration domain was considered, this is no more true (especially when using CLooG). If we don't give any ordering information, this means that the statement instances may be executed in any order (this is useful, e.g., to specify parallelism), but some statement instances may depend on some others and it may be critical to enforce a given order (or non-order). Hence we need another information.

We call *scattering* any kind of ordering information in the Polyhedral Model. There exists many kind of ordering indeed, as *allocation*, *scheduling*, *chunking* etc. Nevertheless they are all expressed in the same way, using *logical stamps* that can have various semantics.

In the case of **scheduling**, the logical stamps are logical dates that express at which date a statement instance have to be executed. For instance, let us consider the following three statements:

```
x = a + b;   // S1
y = c + d;   // S2
z = a * b;   // S3
```

The scheduling of a statement `S` is typically denoted by θ_S. Let us consider the following logical dates for each statement:

$$\begin{aligned} \theta_{S1} &= 2 \\ \theta_{S2} &= 3 \\ \theta_{S3} &= 1 \end{aligned}$$

It means that statement `S3` have to be executed at logical date `1`, statement `S1` have to be executed at logical date `2` and statement `S2` have to be executed at logical date `3`. The target code have to respect this scheduling (the order of the logical dates), hence it would look like the following where the variable `t` denotes the time:

```
t = 1;
z = a * b;   // S3
t = 2;
x = a + b;   // S1
t = 3;
y = c + d;   // S2
```

When some statements share the same logical date, this means that, once the program reaches this logical date, the two statements can be executed in any order, or better, in parallel. For instance let us consider the following scheduling:

$$\begin{aligned} \theta_{S1} &= 1 \\ \theta_{S2} &= 2 \\ \theta_{S3} &= 1 \end{aligned}$$

Statements `S1` and `S3` have the same logical date, hence the target code would be:

```
t = 1;
#pragma omp parallel sections
{
   #pragma omp section
   {
     x = a + b;   // S1
   }
   #pragma omp section
   {
     z = a * b;   // S3
   }
}
t = 2;
y = c + d;        // S2
```

Logical dates may be multidimensional, as clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes and so on. For instance we can consider the following multidimensional schedules for our example:

$$\begin{aligned} \theta_{S1} &= (1,1) \\ \theta_{S2} &= (2,1) \\ \theta_{S3} &= (1,2) \end{aligned}$$

It is not very hard to decypher the meaning of such scheduling. Because of the first dimension, statements `S1` and `S3` will be executed before statement `S2` (`S1` and `S3` are executed at day 1, while `S2` is executed at day 2). The second dimension is not really useful there for `S2` because it is the only statement executed at day 2. Nevertheless it allows to order `S1` and `S3` relatively to each other since `S1` is executed at hour 1 of day 1 while `S3` is executed at hour 2 of day 1. The corresponding target code is the following, with some additional time variables for a better view of the ordering (`t1` corresponds to the first time dimension, `t2` to the second one):

```
t1 = 1;
t2 = 1;
x = a + b;   // S1
t2 = 2;
z = a * b;   // S3
t1 = 2;
t2 = 1;
y = c + d;   // S2
```

In the case of **allocation** (in the litterature we can find some papers that call it *placement*), the logical stamps are a processor number that expresses on which processor a statement instance has to be executed. Typically, allocations are written in the same way as scheduling (hence the general term of *scattering*), here we denote it P_S for a statement `S`. For instance, let us consider the following allocation:

$$\begin{aligned} P_{S1} &= 1 \\ P_{S2} &= 2 \\ P_{S3} &= 1 \end{aligned}$$

The corresponding target code have to take into account that both statements `S1` and `S3` have to be executed on the same processor (they have the same logical number 1) and that statement `S2` have to be executed on another processor (logical number 2). A possible target code is the following:

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     // Logical processor 1
     x = a + b;   // S1
     z = a * b;   // S3
   }
   #pragma omp section
   {
     // Logical processor 2
     y = c + d;   // S2
   }
}
```

We can note that no order have been specified for the statements `S1` and `S3` that are executed on the same processor. Hence any order is satisfying. For sake of flexibility, it is usual to build a scattering whose various dimensions do not have the same semantics. A typical construction is *space/time mapping* where the first `n` dimensions are devoted to allocation, then the last `m` dimensions are devoted to scheduling. Typically, space/time mapping is written in the same way as scheduling (hence again the general term of *scattering*), here we denote it for a statement `S` as M_S. For instance, let us consider the following space/time mapping for our example where one dimension is devoted for mapping and one dimension is devoted to scheduling:

$$\begin{array}{lcl} M_{S1} & = & (1,2) \\ M_{S2} & = & (2,1) \\ M_{S3} & = & (1,1) \end{array}$$

Here we have the same first dimension as the previous example, thus the allocation of the statements to processors is the same. The second dimension precises on a given processor at which logical date a statement instance has to be executed. Here, the statement `S1` is executed at day 2 on processor 1 while the statement `S3` is executed at day 1 onto the same processor. It follows this space/time mapping corresponds to the following target code (we added an additional variable to represent the local logical clocks):

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     // Logical processor 1
     t = 1;
     z = a * b;   // S3
     t = 2;
     x = a + b;   // S1
   }
   #pragma omp section
   {
     // Logical processor 2
     t = 1;
     y = c + d;   // S2
   }
}
```

For the same reason as discussed for iteration domains (see Section 2.2.1 [Iteration Domain], page 5), it is not possible to define a scattering for each statement instance, especially if the statement belongs to a (possibly parametric) loop. The iteration vector fully defines an instance

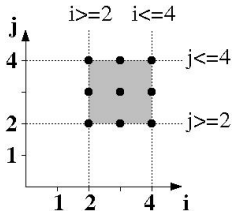
of a given statement. Thus, a practical way to provide a scattering for each instance of a given statement is to use a *function* that depends on the iteration vector. In this way the function may give for each iteration vector a different scattering. We call these functions **scattering functions**. Scattering functions are *affine* functions of the outer loop counter and the global parameters. For instance, let us consider the following source code:

```
for (i = 2; i <= 4; i++)
  for (j = 2; j <= 4; j++)
    P[i+j] += A[i] + B[j]; // S4
```

The iteration domain of the statement `S4` is:

$$D_{S4} = \{(i,j) \in Z^2 \mid 2 \leq i \leq 4 \wedge 2 \leq j \leq 4\}.$$

If you are still not comfortable with the mathematical notation, it corresponds to the following graphical representation:



The list of the statement instances of `S4` (the integral points of its iteration domain) corresponds to the following iteration vectors:

```
iteration vector
     (2,2)
     (2,3)
     (2,4)
     (3,2)
     (3,3)
     (3,4)
     (4,2)
     (4,3)
     (4,4)
```

Let us suppose we want to schedule the instances of the statement `S4` (the integral points of its iteration domain) using the following scheduling function:

$\theta_{S4}(i,j) \;=\; (j+2, 3*i+j)$

We only have to apply the function to each iteration vector to find the logical date of each instance:

```
iteration vector       logical date
     (2,2)       -->       (4,8)
     (2,3)       -->       (5,9)
     (2,4)       -->       (6,10)
     (3,2)       -->       (4,11)
     (3,3)       -->       (5,12)
     (3,4)       -->       (6,13)
     (4,2)       -->       (4,14)
     (4,3)       -->       (5,15)
     (4,4)       -->       (6,16)
```

Polyhedral Model users do not have to take care about the generation of a target code that respects the scattering: the CLooG tool is there to solve the problem quite easily

(`http://www.cloog.org`). For the previous example, the target code would be the following (`t1` and `t2` corresponds to the two dimensions of the logical date):

```
for (t1 = 4; t1 <= 6; t1++) {
  for (t2 = t1+4; t2 <= t1+10; t2++) {
    if ((-t1+t2+2)%3 == 0) {
      i = (-t1+t2+2)/3 ;
      j = t1-2 ;
      P[i+j] += A[i] + B[j]; // S4
    }
  }
}
```

Obviously with such a twisted scheduling, it is hard to see the "meaning" of the transformation. To name any kind of program transformation as a magic spell ("tile", "fuse", "skew"...) is an old bad habit that should be changed in the Polyhedral Model: a scheduling may be an arbitrary complex sequence of basic-old-good transformations. Nevertheless it is most of the time quite easy to translate well known transformations to schedules. For instance, let us consider this new scheduling function:

$$\theta_{S4}(i,j) \;=\; (j,i)$$

Using CLooG, we can generate the target code:

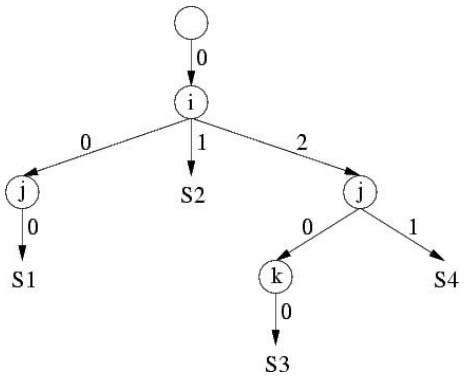
```
for (t1 = 2; t1 <= 4; t1++) {
  for (t2 = 2; t2 <= 4; t2++) {
    i = t2;
    j = t1;
    P[i+j] += A[i] + B[j]; // S4
  }
}
```

It is easy to see (and analyze) that it corresponds to a classical *loop interchange* transformation.

A very useful example of multi-dimensional scattering functions is the **scheduling of the original program**. The method to compute it is quite simple (see [Fea92], page 35). The idea is to build an abstract syntax tree of the program and to read the scheduling for each statement. For instance, let us consider the following implementation of a Cholesky factorization:

```
/* A Cholesky factorization kernel. */
for (i=1;i<=N;i++) {
  for (j=1;j<=i-1;j++) {
    a[i][i] -= a[i][j] ;           /* S1 */
  }
  a[i][i] = sqrt(a[i][i]) ;        /* S2 */
  for (j=i+1;j<=N;j++) {
    for (k=1;k<=i-1;k++) {
      a[j][i] -= a[j][k]*a[i][k] ; /* S3 */
    }
    a[j][i] /= a[i][i] ;           /* S4 */
    }
  }
}
```

The corresponding abstract syntax tree is given in the following figure. It directly gives the scattering functions (schedules) for all the statements of the program.



$$\begin{cases} \theta_{S1}(i,j) & = (0,i,0,j,0) \\ \theta_{S2}(i) & = (0,i,1) \\ \theta_{S3}(i,j,k) & = (0,i,2,j,0,k,0) \\ \theta_{S4}(i,j) & = (0,i,2,j,1) \end{cases}$$

These schedules depend on the iterators and give for each instance of each statement a unique execution date. Using such scattering functions allows CLooG to re-generate the input code.

To easily manipulate the scattering function of any statement `S`, we translate it to the matrix form: $\theta_S(\textit{iteration vector})$ **= scattering matrix** * *iteration vector*. For instance let us consider again our previous example $\theta_{S4}(i,j) = (j+2, 3*i+j)$. We write it in the following way (again please someone show me how to do this in TeX -not LaTeX- for the texinfo manual !):

```
     [ i ]    [  0  1  2 ]   [ i ]
T_S4([ j ]) = [  3  1  0 ] * [ j ]
     [ 1 ]                   [ 1 ]
```

The scattering matrix (along with the iteration vector which is most of the time an implicit information) will be used in all our tools to provide the informations on the scattering of a given statement.

2.2.3 Access Function

Before applying any transformation, it is essential to deeply analyze both the original program and the transformation to ensure the transformation does not imply any modification of the original program semantics. In the Polyhedral Model, we can reach a **total analysis power**: we are able to achieve an exact analysis when all the memory accesses are made through arrays (note that variables are a particular case of arrays since they are simply arrays with only one memory location) with affine subscripts that depend on outer loop counters and global parameters (note that *subscripts* are sometimes called *index* or *accesses* in the litterature).

For instance let us consider the array access `A[2*i+j][j][i+N]`. It has three dimensions, each subscript dimension is an affine form of some outer loop iterarors (`i` and `j`) and global parameters (`N`) hence it corresponds to an acceptable array access in the Polyhedral Model.

Each array access can target a different memory cell depending on the statement instance, i.e., depending on the iteration vector. Thus we use access functions (or subscript functions or index functions as you prefer) depending on the iteration vector to describe an array access. In our example, the access function would be written $F_A(i,j) = (2*i+j, j, i+N)$.

To easily manipulate the access function of any array `A`, we translate it to the matrix form: $F_A(\textit{iteration vector})$ **= access matrix** * *iteration vector*. For instance let us consider again our previous example: we write it in the following way (again please someone show me how to do this in TeX -not LaTeX- for the texinfo manual !):

```
     [ i ]    [  2  1  0  0 ]   [ i ]
F_A([ j ]) = [  0  1  0  0 ] * [ j ]
     [ N ]    [  1  0  1  0 ]   [ N ]
     [ 1 ]                      [ 1 ]
```

The access matrix (along with the iteration vector which is most of the time an implicit information) will be used in all our tools to provide the informations on the access of a given statement.

3 Using the Clan Software

3.1 A First Example

Clan takes as input a source code file than can be written in either C or C++ or C# or Java (or any other imperative language that is close enough to C). It is very simple as it only translates a part of a program that can be represented using the Polyhedral model (see Chapter 2 [Polyhedral Representation], page 3) in a matrix form. Clan does not find itself the program parts that could be represented using the Polyhedral Model. More complex tools like WRAP-IT for the ORC compiler (`http://www.lri.fr/~girbal/site_wrapit`)or the GRAPHITE branch of GCC (`http://gcc.gnu.org/wiki/Graphite`) are devoted to such a complex, highly technical problem. Using Clan, the user has to specify thanks to pragmas where begins the SCoP he is interested by, and where it ends.

For instance, let us consider the following source code in C of a matrix-matrix multiply program that reads two matrices, achieves the multiply then prints the result. Let us also consider that the user is only interested in the matrix-matrix multiply kernel:

```
/* matmul.c 128*128 matrix multiply */
#include <stdio.h>
#define N 128

int main() {
  int   i,j,k;
  float a[N][N], b[N][N], c[N][N];

  /* We read matrix a then matrix b */
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      scanf(" %f",&a[i][j]);
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      scanf(" %f",&b[i][j]);

  /* c = a * b */
#pragma scop
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      c[i][j] = 0.0;
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
#pragma endscop

  /* We print matrix c */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf("%6.2f ",c[i][j]);
    printf("\n");
  }

  return 0;
}
```

The tags to ask Clan to consider a given part of the code are provided thanks to the pragmas `#pragma scop` and `#pragma endscop`. It can have different forms depending on the input language. This is explained in a further section (see Section 3.2 [Writing The Input File], page 18).

This source code file may be called '`matmul.c`' (this example is provided in the Clan distribution as `test/matmul.c`) and we can ask Clan to process it and to generate the polyhedral representation by a simple call to Clan with this file as input: '`clan matmul.c`'. By default, Clan will print the polyhedral representation in the standard output:

```
# [File generated by Clan 1.0.0 64 bits]

SCoP

# ============================================== Global
# Language
C

# Context
0 3

# Parameter names are provided
1
# Parameter names
N

# Number of statements
2

# ============================================== Statement 1
# ----------------------------------------------  1.1 Domain
# Iteration domain
1
4 5
   1    1    0    0    0    ## i >= 0
   1   -1    0    1   -1    ## -i+N-1 >= 0
   1    0    1    0    0    ## j >= 0
   1    0   -1    1   -1    ## -j+N-1 >= 0

# ----------------------------------------------  1.2 Scattering
# Scattering function is provided
1
# Scattering function
5 5
   0    0    0    0    0    ## 0
   0    1    0    0    0    ## i
   0    0    0    0    0    ## 0
   0    0    1    0    0    ## j
   0    0    0    0    0    ## 0

# ----------------------------------------------  1.3 Access
# Access informations are provided
1
# Read access informations
```

```
0 5
# Write access informations
2 5
   1    1    0    0    0    ## c[i][j]
   0    0    1    0    0    ##

# ------------------------------------------------  1.4 Body
# Statement body is provided
1
# Original iterator names
i j
# Statement body
c[i][j]=0.0;


# ================================================ Statement 2
# ------------------------------------------------  2.1 Domain
# Iteration domain
1
6 6
   1    1    0    0    0    0    ## i >= 0
   1   -1    0    0    1   -1    ## -i+N-1 >= 0
   1    0    1    0    0    0    ## j >= 0
   1    0   -1    0    1   -1    ## -j+N-1 >= 0
   1    0    0    1    0    0    ## k >= 0
   1    0    0   -1    1   -1    ## -k+N-1 >= 0

# ------------------------------------------------  2.2 Scattering
# Scattering function is provided
1
# Scattering function
7 6
   0    0    0    0    0    0    ## 0
   0    1    0    0    0    0    ## i
   0    0    0    0    0    0    ## 0
   0    0    1    0    0    0    ## j
   0    0    0    0    0    1    ## 1
   0    0    0    1    0    0    ## k
   0    0    0    0    0    0    ## 0

# ------------------------------------------------  2.3 Access
# Access informations are provided
1
# Read access informations
6 6
   1    1    0    0    0    0    ## c[i][j]
   0    0    1    0    0    0    ##
   2    1    0    0    0    0    ## a[i][k]
   0    0    0    1    0    0    ##
   3    0    0    1    0    0    ## b[k][j]
   0    0    1    0    0    0    ##
# Write access informations
```

```
2 6
   1    1    0    0    0    0    ## c[i][j]
   0    0    1    0    0    0    ##

# ---------------------------------------------  2.4 Body
# Statement body is provided
1
# Original iterator names
i j k
# Statement body
c[i][j]=c[i][j]+a[i][k]*b[k][j];


# ================================================ Options
```

We will not describe here precisely the structure and the components of this output, this is described in depth in a further section (see Section 3.3 [Reading The Output File], page 18). This file format, called `.scop` has been designed to be the input file format of most of the polyhedral tools. If you read the description of the polyhedral representation of programs, you should already feel familiar with this file format (see Chapter 2 [Polyhedral Representation], page 3).

3.2 Writing The Input File

The input file of Clan is a source code file written in any language based on C for the `for` loop, the `if` and for the array accesses. C, C++, Java and C# are good examples that should work pretty well with Clan.

The input file may contain a static control part (i.e., a part of the program that can be represented using the Polyhedral Model as described in the corresponding chapter, see Chapter 2 [Polyhedral Representation], page 3) delimitated **by the user** thanks to pragmas. Clan trusts the user: it will not check hardly whether the program part is actually a SCoP or not. It will only try to translate the program part to a polyhedral representation and will fail with *syntax error* if it reads something wrong.

In C, C++ and C#, the pragma to tag the beginning of the SCoP is:

```
#pragma scop
```

and the pragma to tag the end of the SCoP is:

```
#pragma endscop
```

In Java, the pragma to tag the beginning of the SCoP is:

```
/*@ scop */
```

and the pragma to tag the end of the SCoP is:

```
/*@ end scop */
```

3.3 Reading The Output File

The output text file of Clan provides an explicit polyhedral representation of a static control part. The output file format is called *.scop* format. It has been designed by various researchers in polyhedral compilation from various institutions. It builds on previous popular polyhedral file formats like *.cloog* to provide a unique, extensible file format to every polyhedral compilation tools (including future versions of CLooG). This file is composed of two main parts. The first part is devoted to the polyhedral representation of a SCoP. It contains what is strictly necessary to enter a complete source-to-source framework in the polyhedral model and to output

a semantically equivalent code for the SCoP, from analysis to code generation. The second part of the file contains options, i.e. extensions to provide additional informations to some tools.

The following grammar describes the structure of the first part of the .scop file format where terminals are preceeded by "_". Each relevant part will be explained in more details momentarily. Its looks long but it has been artificially extended to be easily understood and it can be easily simplified:

```
File                ::= SCoP
SCoP                ::= "SCoP"   Context Statements
Context             ::= Language Domain  Parameters
Statements          ::= Nb_statements Statement_list
String_list         ::= _String   String_list    | (void)
Statement_list      ::= Statement Statement_list | (void)
Domain_list         ::= Domain    Domain_list    | (void)
Statement           ::= Iteration_domain Scattering Access Body
Iteration_domain    ::= Domain_union
Domain_union        ::= Nb_domains Domain_list
Scattering          ::= "0" | "1" Scattering_function
Access              ::= "0" | "1" Read_function Write_function
Parameters          ::= "0" | "1" Parameter_list
Body                ::= "0" | "1" Iterator_list Body_text
Language            ::= "C" | "C++" | "C#" | "Java" | "Toy"
Parameter_list      ::= String_list
Iterator_list       ::= String_list
Domain              ::= _Matrix
Scattering_function ::= _Matrix
Read_function       ::= _Matrix
Write_function      ::= _Matrix
Nb_statements       ::= _Integer
Nb_domains          ::= _Integer
Body_text           ::= _String
```

- '`Context`' represents the informations that are shared by all the statements. It consists on the language used (which can be '`C`', '`C++`', '`C#`' or '`Java`'), the global constraints on parameters and optionally the parameter names. The set of constraints on parameters is essential since it provides the number of parameters. The '`Domain`' encoding includes the number of unknown (here the number of parameters) (see Section 3.3.1 [Domain Representation], page 20). Even if there are no constraints, this number has to be correct. After the constraints, it is possible to provide the list of parameter names (the textual names in the original program). A '`0`' means we don't provide the list of parameter names, and a '`1`' means the list of parameter names is provided afterward. The original parameter names are necessary for the code generator to be able to generate a code that can replace directly the SCoP in the original program by copy/paste. In the case of a '`0`', parameter names will probably be generated by the code generator (this is the case when using CLooG) or will be extracted from another input source.
- '`Statements`' represents the informations on the statements. '`Nb_statements`' is the number of statements in the program, i.e. the number of '`Statement`' items in the '`Statement_list`'. '`Statement`' represents the informations on a given statement. To each statement is associated four informations, the first one is mandatory while the three others are optional. The statement iteration domain '`Iteration_domain`' is the required information, then one can provide optionally a scattering function, the access functions and the statement body, in this order. Each optional information is preceeded by a boolean that precises whether the optional information is provided or not. The iteration domain (see

Section 2.2.1 [Iteration Domain], page 5) is represented using a matrix (see Section 3.3.1 [Domain Representation], page 20). Next, the scattering function (see Section 2.2.2 [Scattering Function], page 7) is represented using a matrix as well (see Section 3.3.2 [Scattering Representation], page 21). The access functions (see Section 2.2.3 [Access Function], page 12) are represented using two matrices, one for read accesses and another one for write accesses (see Section 3.3.3 [Access Representation], page 21). The statement body is made of two parts: first, the list of surrounding loop counters in the original program and second, the text string of the statement. This representation allows to apply the substitution of the original iterators with new iterators in the target program.

The main terminal parts (domains, scattering and access functions) are detailed in the next subsections. Lastly, we will describe the option part (see Section 3.3.4 [Option Part], page 22).

3.3.1 Domain Representation

As shown by the grammar, the input file describes the various informations thanks to strings, integers and domains. Each domain is defined by a set of constraints in the PolyLib format (see [Wil93], page 35). They have the following syntax:

1. Some optional comment lines beginning with '`#`'.
2. The row and column numbers, possibly followed by comments.
3. The constraint rows. Each row corresponds to a constraint the domain have to satisfy. Each row must be on a single line and is possibly followed by comments. The constraint is an equality $p(x) = 0$ if the first element is 0, an inequality $p(x) \geq 0$ if the first element is 1. The next elements are the unknown coefficients, followed by the parameter coefficients. The last element is the constant factor.

For instance, assuming that '`i`', '`j`' and '`k`' are the loop iterators and '`m`' and '`n`' are the parameters, the domain defined by the following constraints :

$$\begin{cases} -i + m & \geq 0 \\ -j + n & \geq 0 \\ i + j - k & \geq 0 \end{cases}$$

can be written in the input file as follows :

```
# This is a domain
3 7                     # 3 lines and 7 columns
# eq/in i  j  k  m  n  1
    1  -1  0  0  1  0  0 #    -i + m >= 0
    1   0 -1  0  0  1  0 #    -j + n >= 0
    1   1  1 -1  0  0  0 # i + j - k >= 0
```

Each iteration domain '`Iteration_domain`' of a given statement is a *union* of polyhedra '`Domain_union`'. A union is defined by its number of elements '`Nb_domains`' and the elements themselves '`Domain_list`'. For instance, let us consider the following pseudo-code:

```
for (i = 1; i <= n; i++) {
  if ((i >= m) || (i <= 2*m))
    S1;
}
```

The iteration domain of '`S1`' can be divided into two polyhedra and written in the .scop file as follows:

```
2 # Number of polyhedra in the union
# First domain
3 5                # 3 lines and 5 columns
# eq/in i   m   n   1
    1   1   0   0 -1 #  i >= 1
    1  -1   0   1   0 #  i <= n
    1   1  -1   0   0 #  i >= m
# Second domain
3 5                # 3 lines and 5 columns
# eq/in i   m   n   1
    1   1   0   0 -1 #  i >= 1
    1  -1   0   1   0 #  i <= n
    1  -1   2   0   0 #  i <= 2*m
```

3.3.2 Scattering Representation

Scattering functions are depicted in the input file thanks a representation very close to the domain one. The difference is each row do not describe a constraint but a scattering function dimension (see Section 2.2.2 [Scattering Function], page 7). By convention, the first element of each row (the one that defines whether the constraint is an equality or an inequality for domains) *must be set to 0*. The next elements are the unknown coefficients, followed by the parameter coefficients. The last element is the constant factor. For instance, assuming that '`i`', '`j`' and '`k`' are the loop iterators and '`m`' and '`n`' are the parameters, the scattering function $\theta_S(i,j,k) = (j+2, 3*i+j, k+n+1)$ may be written in a .scop file in the following way:

```
# A scattering function
3 7                        # 3 dimensions and 7 columns
# 0  i  j  k  m  n  1
  0  0  1  0  0  0  2     # j+2
  0  3  1  0  0  0  0     # 3*i+j
  0  0  0  1  0  1  1     # k+n+1
```

Note that this representation is different from the .cloog format: the useless and error-prone identity matrix part disappeared.

The scattering function extracted by Clan is the scheduling of the original program as described in a previous section (see Section 2.2.2 [Scattering Function], page 7). It allows a code generator (like CLooG) to reconstruct directly the original program or a dependence analyzer (like Candl) to achieve its data dependence calculation.

3.3.3 Access Representation

Access functions are depicted in the input file thanks a representation very close to the domain one. The difference is each row do not describe a constraint but a access function dimension (see Section 2.2.3 [Access Function], page 12). Moreover, the matrix representation do not describes only one access function but a set of access functions. Each array accessed in the SCoP has a unique strictly positive identification number. The first element of each row (the one that defines whether the constraint is an equality or an inequality for domains) corresponds to the array identifier iff the row corresponds to the first dimension of the access function. If the first element is 0, this means the row corresponds to the next dimension of the access function, with respect to the previous row. The next elements are the unknown coefficients, followed by the parameter coefficients. The last element is the constant factor. For instance, assuming that '`i`', '`j`' and '`k`' are the loop iterators and '`m`' and '`n`' are the parameters, the set of array accesses `A[2*i+j][j][i+n]`, `B[i+j]` and `A[k][j][1]` (the identifier of `A` is 1 and the identifier of `B` is 2) may be written in a .scop file in the following way:

```
# A set of access functions
7 7                     # 7 rows and 7 columns
# id  i  j  k  m  n  1
   1  2  1  0  0  0  0   # A[2*i+j][j][i+n]
   0  0  1  0  0  0  0   #
   0  1  0  0  0  1  0   #
   2  1  1  0  0  0  0   # B[i+j]
   1  0  0  1  0  0  0   # A[k][j][1]
   0  0  1  0  0  0  0   #
   0  0  0  0  0  0  1   #
```

3.3.4 Option Part

The end of the .scop file is made of a succession of options delimited using XML-like tags. Each tool will take care of known options and will ignore the others. There is no specification for the option body as it is tool-dependent. Nevertheless, authors are invited to put the name of the tool inside the option name to avoid conflicts. A reserved option name is '`Comments`' that allows to put some comments in the second part of the .scop file. For instance, this could be a possible second part of a .scop file:

```
<Comments>
    Just a comment example.
<\Comments>

<CLooG foobar>
    This is supposed to provide CLooG some interesting
    additional informations.
<\CLooG foobar>
```

A second reserved name is '`arrays`', Clan can optionally print a table of the arrays referenced in the access functions. For instance, for a program referencing first the array '`FOO`', then the array '`BAR`':

```
<arrays>
# Number of referenced arrays
2
# First reference
1 FOO
# Second reference
2 BAR
</arrays>
```

3.4 Calling Clan

Clan is called by the following command:

```
clan [ options | file ]
```

The default behavior of Clan is to read the input source code from a file and to print the generated .scop file on the standard output. Clan's behavior and the output file are under the user control thanks to some options which will be detailed momentarily (see Section 3.5 [Clan Options], page 22). `file` is the input file. `stdin` is a special value: when used, input is standard input. For instance, we can call Clan to process the input file `basic.c` with default options by typing: `clan basic.c` or `more basic.c | clan stdin` (usual `more basic.c | clan -` works too).

3.5 Clan Options

3.5.1 Output `-o <output>`

`-o <output>`: this option sets the output file. `stdout` is a special value: when used, output is standard output. Default value is `stdout`.

3.5.2 Arrays Tag `-arraystag`

`-arraystag`: this option dumps the table of referenced arrays at the end of the .scop file, between the '`<arrays>`' and '`</arrays`' tags.

3.5.3 Help `--help` or `-h`

`--help` or `-h`: this option asks Clan to print a short help.

3.5.4 Version `--version` or `-v`

`--version` or `-v`: this option asks Clan to print some release and version informations.

4 Using the Clan Library

The Clan Library was implemented to allow the user to call Clan directly from his programs, without file accesses or system calls. The user only needs to link his programs with C libraries. The Clan library mainly provides one function (`clan_scop_extract`) which takes as input the source code file to process with some options, and returns the data structure corresponding to the SCoP (a `clan_scop_t` structure) which contains the polyhedral representation of the SCoP. The user can work with this data structure and/or use the printing function to output a .scop file. Some other functions are provided for convenience reasons. These functions as well as the data structures are described in this section.

4.1 Clan Data Structures Description

In this section, we describe the data structures used by the loop analyzer to represent and to process a SCoP. As this is not a developer's guide, data structure devoted to internal use as well as internal use fields are not described here.

4.1.1 clan_matrix_t

The `clan_matrix_t` structure is the same as the PolyLib `Matrix` data structure (see [Wil93], page 35) in order to be used directly by tools using this format with a simple cast.

```
struct clan_matrix
{
  unsigned NbRows;     /* The number of rows */
  unsigned NbColumns;  /* The number of columns */
  clan_int_t ** p;     /* An array of pointers to the matrix row */
  clan_int_t * p_Init; /* The matrix rows contiguously in memory */
  int p_Init_size;     /* Initial size of p_Init */
};
typedef struct clan_matrix   clan_matrix_t;
typedef struct clan_matrix * clan_matrix_p;
```

The whole matrix is stored in memory row after row at the `p_Init` address. `p` is an array of pointers where `p[i]` points to the first element of the i^{th} row. `NbRows` and `NbColumns` are respectively the number of rows and columns of the matrix.

To be able to provide different precision versions (Clan supports 32 bits, 64 bits and arbitrary precision through the GMP library), the `clan_int_t` type depends on the configuration options (it may be `long int` for 32 bits version, `long long int` for 64 bits version, and `mpz_t` for multiple precision version). The `p_Init_size` field is needed to free the memory allocated by `mpz_init` in the multiple precision version. Set this field to the initial size of the `p_Init` array (its size may vary during processing, for instance if you are using PolyLib).

4.1.2 clan_matrix_list_t

```
struct clan_matrix_list
{
  clan_matrix_p elt;             /* An element of the list. */
  struct clan_matrix_list* next; /* Pointer to the next element. */
};
typedef struct clan_matrix_list   clan_matrix_list_t;
typedef struct clan_matrix_list * clan_matrix_list_p;
```

The `clan_matrix_list_t` structure is a `NULL`-terminated linked list of `clan_matrix_t` data structures. It is used for instance to represent a union of convex domains (each matrix representing a convex part of the union). `elt` is a matrix element of the list and `next` is the pointer to the next element of the list.

4.1.3 clan_statement_t

```
struct clan_statement
{
  clan_matrix_list_p domain;    /* Iteration domain */
  clan_matrix_p scattering;     /* Scattering function */
  clan_matrix_p read;           /* Array read access informations */
  clan_matrix_p write;          /* Array write access informations */
  int nb_iterators;             /* Original depth of the statement */
  char ** iterators;            /* Array of iterator names */
  char * body;                  /* Original statement body */
  struct clan_statement * next; /* Next statement in the linked list */
};
typedef struct clan_statement   clan_statement_t;
typedef struct clan_statement * clan_statement_p;
```

The `clan_statement_t` structure represents a `NULL` terminated linked list of statements. The structure contains all elements of the polyhedral representation of the statement. It uses a `clan_matrix_list_t` data structure to represent a union for the iteration domain `domain` (it is simply a linked list of `clan_matrix_t` elements), and a `clan_matrix_t` data structure for the rest: the scattering function `scattering`, the set of array read accesses `read` and the set of array write accesses `write`. The particular representation of each element using the `clan_matrix_t` data structure is described with the output file format (see Section 3.3 [Reading The Output File], page 18). If an element is not present, the according pointer have to be `NULL`. The `nb_iterators` field gives the depth of the statement in the original program. It may be used with any matrix to compute the number of parameters (e.g. `nb_parameters = domain->NbColumns - nb_iterators - 2`) To represent the statement body, we use `iterators`, an array of `nb_iterators` strings for the surrounding loop counters names in the original program, and `body`, the statement body string in the original program.

4.1.4 clan_scop_t

```
struct clan_scop
{
  clan_matrix_p context;      /* Constraints on the SCoP parameters */
  int nb_parameters;          /* Number of parameters for the SCoP */
  char ** parameters;         /* Array of parameter names */
  int nb_arrays;              /* Number of arrays accessed in the SCoP */
  char ** arrays;             /* Array of array names */
  clan_statement_p statement; /* Statement list of the SCoP */
  char * optiontags;          /* The content (as a 0 terminated
                                 string) of the optional tags. */
  void * usr;                 /* A user-defined field,
                                 not touched by clan. */
};
typedef struct clan_scop   clan_scop_t;
typedef struct clan_scop * clan_scop_p;
```

`clan_scop_t` stores the useful informations of a static control part of a program to process it within a polyhedral framework. It contains the informations about the context (what is common to all statements in the SCoP) and the list of statements `statement`. The context is made of the constraints on the global parameters `context`. The representation of the context using the `clan_matrix_t` data structure is described with the output file format (this is a domain, see Section 3.3 [Reading The Output File], page 18). The list of parameter names is provided as an array of `nb_parameters` strings called `parameters` (`nb_parameters` is somewhat redundant

as it is supposed to be equal to `context->NbColumns - 2`). The list of array names is provided as an array of `nb_arrays` strings called `arrays`. Each accessed array in the SCoP has a unique identifier (a strictly positive number, see Section 3.3.3 [Access Representation], page 21), `arrays` provides the correspondance between an identifier and the real name of the accessed array. If an array has the identifier '`id`', then its real name is '`arrays[id - 1]`'. The `optiontags` field contains the remainder of the SCoP description file, as a `char*` string. Optional tags specified in the '`Options`' section are stored there. Finally, the `usr` field is a pointer for library users convenience. This field is not touched by Clan.

As an example, let us consider again the matrix-matrix multiply program (see Section 3.1 [A First Example], page 15). The next figure gives a possible representation in memory for this SCoP thanks to the Clan data structures (it has been actually printed by the `clan_scop_print` function, it is also possible to ask Clan to output the internal representation using the command line option `-structure`):

```
+-- clan_scop_t
|       |
|       +-- clan_matrix_t
|       |       0 3
|       |
|       +-- Original parameters strings: N
|       |
|       +-- Accessed array strings: c a b
|       |
|       +-- clan_statement_t (S1)
|       |       |
|       |       +-- clan_matrix_list_t
|       |       |       |
|       |       |       +-- clan_matrix_t
|       |       |       |       4 5
|       |       |       |       [    1    1    0    0    0 ]
|       |       |       |       [    1   -1    0    1   -1 ]
|       |       |       |       [    1    0    1    0    0 ]
|       |       |       |       [    1    0   -1    1   -1 ]
|       |       |       |
|       |       |
|       |       +-- clan_matrix_t
|       |       |       5 5
|       |       |       [    0    0    0    0    0 ]
|       |       |       [    0    1    0    0    0 ]
|       |       |       [    0    0    0    0    0 ]
|       |       |       [    0    0    1    0    0 ]
|       |       |       [    0    0    0    0    0 ]
|       |       |
|       |       +-- NULL matrix
|       |       |
|       |       +-- clan_matrix_t
|       |       |       2 5
|       |       |       [    1    1    0    0    0 ]
|       |       |       [    0    0    1    0    0 ]
|       |       |
|       |       +-- Original iterator strings: i j
|       |       |
|       |       +-- Original body: c[i][j]=0.0;
|       |       |
|       |       V
|       |   clan_statement_t (S2)
|       |       |
|       |       +-- clan_matrix_list_t
|       |       |       |
|       |       |       +-- clan_matrix_t
|       |       |       |       6 6
|       |       |       |       [    1    1    0    0    0    0 ]
```

```
|       |       |       |       [    1   -1    0    0    1   -1 ]
|       |       |       |       [    1    0    1    0    0    0 ]
|       |       |       |       [    1    0   -1    0    1   -1 ]
|       |       |       |       [    1    0    0    1    0    0 ]
|       |       |       | [    1    0    0   -1    1   -1 ]
|       |       |       |
|       |       |
|       |       +-- clan_matrix_t
|       |       |       7 6
|       |       |       [    0    0    0    0    0    0 ]
|       |       |       [    0    1    0    0    0    0 ]
|       |       |       [    0    0    0    0    0    0 ]
|       |       |       [    0    0    1    0    0    0 ]
|       |       |       [    0    0    0    0    0    1 ]
|       |       |       [    0    0    0    1    0    0 ]
|       |       |       [    0    0    0    0    0    0 ]
|       |       |
|       |       +-- clan_matrix_t
|       |       |       6 6
|       |       |       [    1    1    0    0    0    0 ]
|       |       |       [    0    0    1    0    0    0 ]
|       |       |       [    2    1    0    0    0    0 ]
|       |       |       [    0    0    0    1    0    0 ]
|       |       |       [    3    0    0    1    0    0 ]
|       |       |       [    0    0    1    0    0    0 ]
|       |       |
|       |       +-- clan_matrix_t
|       |       |       2 6
|       |       |       [    1    1    0    0    0    0 ]
|       |       |       [    0    0    1    0    0    0 ]
|       |       |
|       |       +-- Original iterator strings: i j k
|       |       |
|       |       +-- Original body: c[i][j]=c[i][j]+a[i][k]*b[k][j];
|       |       |
|       |
|
```

4.1.5 clan_options_t

```
struct clan_options
{
  char * name ;  /* Name of the input file. */
};
typedef struct clan_options   clan_options_t;
typedef struct clan_options * clan_options_p;
```

The `clan_options_t` structure contains all the possible options to rule Clan's behaviour (see Section 3.4 [Calling Clan], page 22). For the moment there are mainly internal options, but it's going to change in the future.

4.2 Clan Functions Description

4.2.1 clan_scop_extract

```
clan_scop_p clan_scop_extract(FILE * input, clan_options_p options);
```

The `clan_scop_extract` function extracts the polyhedral representation of a SCoP in the file provided thanks to the `input` pointer (the file, possibly `stdin`, has to be open for reading), according to some options provided thanks to the pointer `options` to a `clan_options_t` data structure (see Section 4.1.5 [clan_options_t], page 28). It returns a pointer to the extracted SCoP, translated into a `clan_scop_t` data structure (see Section 4.1.4 [clan_scop_t], page 26).

4.2.2 clan_scop_print_dot_scop

```
void clan_scop_print_dot_scop
(
  FILE * output,
  clan_scop_p scop,
  clan_options_p options
);
```

The function `clan_scop_print_dot_scop` is a pretty printer for `clan_scop_t` structures. It dumps the `scop` informations in .scop format (see Section 3.3 [Reading The Output File], page 18) in the file provided thanks to the pointer `output` (the file, possibly `stdout`, has to be open for writing), according to some options provided thanks to the pointer `options` to a `clan_options_t` data structure (see Section 4.1.5 [clan_options_t], page 28).

4.2.3 clan_scop_read

```
clan_scop_p clan_scop_read
(
  FILE * input,
  clan_options_p options
);
```

The function `clan_scop_read` reads a .scop file from the standard input, and returns a pointer on a freshly allocated `clan_scop_t` structure containing the SCoP information.

4.2.4 clan_scop_tag_content

```
char* clan_scop_tag_content
(
  clan_scop_p scop,
  char* from,
  char* to
);
```

The function `clan_scop_tag_content` reads the list of optional tags for the `clan_scop_t` (stored in the `optiontags` string), and returns a freshly allocated string of all characters between the two given strings `from` and `to`. If one or the other given strings are not found, or if there was no optional part specified in the .scop, `NULL` is returned.

4.2.5 Allocation and Initialization Functions

```
clan_structure_p clan_structure_malloc();
```

Each Clan data structure has an allocation and initialization function as shown above, where `structure` have to be replaced by the name of the convenient structure (without '`clan`' prefix and '`_t`' suffix) for instance `clan_scop_p clan_scop_malloc();`. These functions return pointers to an allocated structure with fields set to convenient default values. **Using those functions is mandatory** to support internal management fields and to avoid upward compatibility problems if new fields appear. An exception is `clan_matrix_malloc` since the `clan_matrix_t` needs two parameters: the number of rows and columns of the matrix we want to allocate:

```
clan_matrix_p clan_matrix_malloc(unsigned nbrows, unsigned nbcolumns);
```

4.2.6 Memory Deallocation Functions

```
void clan_structure_free(clan_structure_p);
```

Each Clan data structure has a deallocation function as shown above, where `structure` have to be replaced by the name of the convenient structure (without '`clan`' prefix and '`_t`' suffix) for instance `void clan_scop_free(clan_scop_p);`. These functions free the allocated memory for

the structure provided as input. They free memory recursively, i.e. they also free the allocated memory for the internal structures. **Using those functions is mandatory** to avoid memory leaks on internal management fields and to avoid upward compatibility problems if new fields appear.

4.2.7 Printing Functions

```
void clan_structure_print(FILE *, clan_structure_p) ;
```

Each Clan data structure has a printing function as shown above, where `structure` have to be replaced by the name of the convenient structure (without '`clan`' prefix and '`_t`' suffix) for instance `void clan_scop_print(FILE *, clan_scop_p);`. These functions print the pointed structure (and its fields recursively) to the file provided as input (the file, possibly `stdout`, has to be open for writing).

4.3 Example of Library Utilization

Here is a basic example showing how it is possible to use the Clan library, assuming that a standard installation has been done. The following C program reads a source code input file on the standard input, then prints the solution on the standard output. Options are preselected to the default values of the Clan software.

```
/* example.c */
# include <stdio.h>
# include <clan/clan.h>

int main()
{
  clan_scop_p scop;
  clan_options_p options;

  /* Default option setting. */
  options = clan_options_malloc() ;

  /* Extraction of the SCoP. */
  scop = clan_scop_extract(stdin, options);

  /* Output of the .scop file. */
  clan_scop_print_dot_scop(stdout, scop, options);

  /* Save the planet. */
  clan_options_free(options);
  clan_scop_free(scop);

  return 0;
}
```

The compilation command could be:

```
gcc example.c -lclan -o example
```

A calling command with the input file test.c could be:

```
more test.c | ./example
```

5 Installing Clan

5.1 License

First of all, it would be very kind to refer the following paper in any publication that result from the use of the Clan software or its library, see [Bas03], page 35 (a bibtex entry is provided behind the title page of this manual, along with copyright notice).

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. `http://www.gnu.org/copyleft/lgpl.html`

5.2 Requirements

Clan is a stand-alone tool and library. For a basic use, it does not need any additional tool or library. Anyway, to be able to work in conjunction with other tools that manipulate multiple precision numbers, the GNU GMP library can be used as an option.

5.2.1 GMP Library (optional)

To be able to deal with insanely large coefficient, the user will need to install the GNU Multiple Precision Library (GMP for short) version 4.2.2 or above. It can be freely downloaded from `http://www.swox.com/gmp`. The user can compile it by typing the following commands on the GMP root directory:

- `./configure`
- `make`
- And as root: `make install`

The GMP default installation is `/usr/local`. This directory may not be inside your library path. To fix the problem, the user should set

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

if your shell is, e.g., bash or

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/lib
```

if your shell is, e.g., tcsh. Add the line to your .bashrc or .tcshrc (or whatever convenient file) to make this change permanent. Another solution is to ask GMP to install in the standard path by using the prefix option of the configure script: '`./configure --prefix=/usr`'.

Clan has to be built using the GMP library by specifying the convenient configure script options to buid the GMP version (see Section 5.4 [Optional Features], page 32).

5.3 Clan Basic Installation

Once downloaded and unpacked (e.g. using the '`tar -zxvf clan-1.0.0.tar.gz`' command), you can compile Clan by typing the following commands on the Clan's root directory:

- `./configure`
- `make`
- And as root: `make install`

The program binaries and object files can be removed from the source code directory by typing `make clean`. To also remove the files that the `configure` script created (so you can compile the package for a different kind of computer) type `make distclean`.

5.4 Optional Features

The `configure` shell script attempts to guess correct values for various system-dependent variables and user options used during compilation. It uses those values to create the `Makefile`. Various user options are provided by the Clan's configure script. They are summarized in the following list and may be printed by typing `./configure --help` in the Clan top-level directory.

- By default, the installation directory is `/usr/local`: `make install` will install the package's files in `/usr/local/bin`, `/usr/local/lib` and `/usr/local/include`. The user can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`.
- By default, Clan is built in 64bits version. If the user give to `configure` the option `--enable-int-version`, the 32bits version of Clan will be compiled. In the same way, the option `--enable-mp-version` have to be used to build the multiple precision version.
- By default, `configure` will look for the GMP library (necessary to build the multiple precision version) in standard locations. If necessary, the user can specify the GMP path by giving `configure` the option `--with-gmp=PATH`.

5.5 Uninstallation

The user can easily remove the Clan software and library from his system by typing (as root if necessary) from the Clan top-level directory `make uninstall`.

6 Documentation

The Clan distribution provides several documentation sources. First, the source code itself is as documented as possible. The code comments use a Doxygen-compatible presentation (something similar to what JavaDoc does for JAVA). The user may install Doxygen (see `http://www.stack.nl/~dimitri/doxygen`) to automatically generate a technical documentation by typing `make doc` or `doxygen ./autoconf/Doxyfile` at the Clan top-level directory after running the configure script (see Chapter 5 [Installing], page 31). Doxygen will generate documentation sources (in HTML, LaTeX and man) in the `doc/source` directory of the Clan distribution.

The Texinfo sources of the present document are also provided in the `doc` directory. You can build it in either DVI format (by typing `texi2dvi cloog.texi`) or PDF format (by typing `texi2pdf cloog.texi`) or HTML format (by typing `makeinfo --html cloog.texi`, using `--no-split` option to generate a single HTML file) or info format (by typing `makeinfo cloog.texi`).

7 References

- [Bas03a] C. Bastoul, P. Feautrier. Improving data locality by chunking. CC'12 International Conference on Compiler Construction, LNCS 2622, pages 320-335, Warsaw, april 2003.
- [Bas03] C. Bastoul and A. Cohen and S. Girbal and S. Sharma and O. Temam. Putting Polyhedral Loop Transformations to Work, LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958, pages 209–225, College Station, Texas, october 2003.
- [Fea92] P. Feautrier Some efficient solutions to the affine scheduling problem, part II: multi-dimensional time. International Journal of Parallel Programming, 21(6):389–420, December 1992.
- [Gri04] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation Thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004. *http://www.infosun.fmi.uni-passau.de/cl/loopo/*
- [Wil93] Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993.