Clan

A Polyhedral Representation Extractor for High Level Programs
Edition 1.0, for Clan 0.8.0
June 3rd 2014

Cédric Bastoul

This manual is for Clan version 0.8.0, 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 **1**

2 Using the Clan Software **3**

2.1 A First Example 3
2.2 Writing The Input File 6
2.2.1 SCoP Pragmas 7
2.2.2 Semantical Restrictions 7
2.2.3 Syntactical Restrictions 7
2.2.3.1 Loop Initialization 8
2.2.3.2 Loop and `if` Condition 8
2.2.3.3 Loop Step 9
2.2.3.4 Array Subscript 9
2.3 Reading The Output File 9
2.4 Calling Clan 9
2.5 Clan Options 9
2.5.1 Output `-o <output>` 9
2.5.2 Autoscop `-autoscop` 10
2.5.3 Autopragma `-autopragma` 10
2.5.4 Autoinsert `-autoinsert` 10
2.5.5 Inputscop `-inputscop` 10
2.5.6 Precision `-precision <value>` 10
2.5.7 Bounded Context `-boundedctxt` 10
2.5.8 No Loop Context `-noloopctxt` 10
2.5.9 No Domain Simplify `-nosimplify` 10
2.5.10 Extbody `-extbody` 10
2.5.11 Help `--help` or `-h` 10
2.5.12 Version `--version` or `-v` 11

3 Using the Clan Library **13**

3.1 Clan Data Structures Description 13
3.1.1 clan_options_t 13
3.2 Clan Functions Description 13
3.2.1 clan_scop_extract 13
3.2.2 clan_scop_insert_pragmas 14
3.2.3 clan_scop_print 14
3.2.4 Allocation and Initialization Functions 14
3.2.5 Memory Deallocation Functions 14
3.2.6 Printing Functions 14
3.3 Example of Library Use 14

4 Installing Clan **17**

4.1 License 17
4.2 Requirements 17
4.2.1 OpenScop Library (bundled) 17
4.3 Clan Basic Installation 17
4.4 Optional Features 18
4.5 Uninstallation 18

5 Documentation **19**

6 Development **21**

6.1 Copyright Issue 21
6.2 Repository 21
6.3 Coding Style 21

7 References **23**

1 Introduction

Clan is a free software and library which translates some particular parts of high level programs written in C, C++, C# or Java into a polyhedral representation, namely OpenScop (see [Bas12], page 23). This representation may be manipulated by other tools to, e.g., achieve complex analyses or 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 23). It is designed to be the front-end of any source-to-source automatic optimizers and/or parallelizers. The OpenScop output format has been chosen to be polyhedral library independent, so Clan may integrate any polyhedral compilation framework easily. Clan has been successfuly integrated to PoCC (`http://pocc.sourceforge.net`) and Pluto (`http://pluto-compiler.sourceforge.net`) high-level compilers. The OpenScop format for polyhedral representation of programs, as well as an introduction to the polyhedral model for dummies, is presented in an external document, see [Bas12], page 23. If the reader is not familiar with such a representation, we highly recommend him/her to read that document first.

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, even if we do think so. A lot of reports are necessary to freeze the library API and it is much probable that Clan will accept larger and larger subsets of C over time (which should not introduce upward compatibility problem). You are very welcome and encouraged to send reports on bugs, wishes, critics, comments, suggestions or (please !) successful experiences to `clan-development@googlegroups.com`.

2 Using the Clan Software

2.1 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 close enough to C). It translates some parts of the program to a polyhedral, matrix-based, representation called OpenScop. We call such a program part a *static control part*, or SCoP fo short.

Clan can find automatically the program parts that it is able to translate. However, as it will be detailed momentarily, to be a real SCoP, those code parts must have some properties which are beyond the analysis power of Clan (like pointer alias or function side-effects analysis). More complex tools like the GRAPHITE framework of GCC (http://gcc.gnu.org/wiki/Graphite) or the Polly framework of LLVM (http://polly.grosser.es) are devoted to such a complex, highly technical problem. Using Clan, the user should rather specify thanks to pragmas where begin the SCoPs to process, and where they finish.

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 2.2 [Writing The Input File], page 6).

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 OpenScop polyhedral representation on the standard output:

```
# [File Generated by Clan 0.7.0]
<OpenScop>

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

# Context
CONTEXT
0 3 0 0 0 1

# Parameters are provided
1
<strings>
N
</strings>

# Number of statements
2

# =============================================== Statement 1
# Number of relations describing the statement:
3

# ----------------------------------------------  1.1 Domain
DOMAIN
4 5 2 0 0 1
# e/i| i   j | N | 1
   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
5 10 5 2 0 1
# e/i|s1  s2  s3  s4  s5 | i   j | N | 1
   0  -1   0   0   0   0   0   0   0   0    ## s1 == 0
   0   0  -1   0   0   0   1   0   0   0    ## s2 == i
   0   0   0  -1   0   0   0   0   0   0    ## s3 == 0
   0   0   0   0  -1   0   0   1   0   0    ## s4 == j
   0   0   0   0   0  -1   0   0   0   0    ## s5 == 0
```

```
# ---------------------------------------------- 1.3 Access
WRITE
3 8 3 2 0 1
# e/i|Arr [1] [2]| i   j | N | 1
   0  -1   0   0   0   0   0   1    ## Arr == c
   0   0  -1   0   1   0   0   0    ## [1] == i
   0   0   0  -1   0   1   0   0    ## [2] == j

# ---------------------------------------------- 1.4 Body
# Statement body is provided
1
<body>
# Number of original iterators
2
# List of original iterators
i j
# Statement body expression
c[i][j] = 0.0;
</body>

# =============================================== Statement 2
# Number of relations describing the statement:
6

# ---------------------------------------------- 2.1 Domain
DOMAIN
6 6 3 0 0 1
# e/i| i   j   k | N | 1
   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
7 13 7 3 0 1
# e/i|s1  s2  s3  s4  s5  s6  s7 | i   j   k | N | 1
   0  -1   0   0   0   0   0   0   0   0   0   0   0    ## s1 == 0
   0   0  -1   0   0   0   0   0   1   0   0   0   0    ## s2 == i
   0   0   0  -1   0   0   0   0   0   0   0   0   0    ## s3 == 0
   0   0   0   0  -1   0   0   0   0   1   0   0   0    ## s4 == j
   0   0   0   0   0  -1   0   0   0   0   0   0   1    ## s5 == 1
   0   0   0   0   0   0  -1   0   0   0   1   0   0    ## s6 == k
   0   0   0   0   0   0   0  -1   0   0   0   0   0    ## s7 == 0

# ---------------------------------------------- 2.3 Access
WRITE
3 9 3 3 0 1
# e/i|Arr [1] [2]| i   j   k | N | 1
```

```
     0  -1   0   0   0   0   0   0   1     ## Arr == c
     0   0  -1   0   1   0   0   0   0     ## [1] == i
     0   0   0  -1   0   1   0   0   0     ## [2] == j

  READ
  3 9 3 3 0 1
  # e/i|Arr  [1]  [2]| i    j    k  | N  | 1
     0  -1   0   0   0   0   0   0   1     ## Arr == c
     0   0  -1   0   1   0   0   0   0     ## [1] == i
     0   0   0  -1   0   1   0   0   0     ## [2] == j

  READ
  3 9 3 3 0 1
  # e/i|Arr  [1]  [2]| i    j    k  | N  | 1
     0  -1   0   0   0   0   0   0   2     ## Arr == a
     0   0  -1   0   1   0   0   0   0     ## [1] == i
     0   0   0  -1   0   0   1   0   0     ## [2] == k

  READ
  3 9 3 3 0 1
  # e/i|Arr  [1]  [2]| i    j    k  | N  | 1
     0  -1   0   0   0   0   0   0   3     ## Arr == b
     0   0  -1   0   0   0   1   0   0     ## [1] == k
     0   0   0  -1   0   1   0   0   0     ## [2] == j

  # ----------------------------------------------  2.4 Body
  # Statement body is provided
  1
  <body>
  # Number of original iterators
  3
  # List of original iterators
  i j k
  # Statement body expression
  c[i][j] = c[i][j] + a[i][k]*b[k][j];
  </body>

  </OpenScop>
```

We will not describe here the structure and the components of this output. This is done in depth in a separated document, along with a complete introduction to the polyhedral representation of programs for the dummies (see [Bas12], page 23). This file format, called `OpenScop` has been designed to be the possible input file format of most polyhedral tools. If you have the minimum knowledge about the polyhedral representation of programs, you should probably already feel familiar with this file format.

2.2 Writing The Input File

The input file of Clan is a source code 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, as long as the code parts to translate would be correct in C. Clan will only translate program parts which must respect syntactical and semantical restrictions over plain

C, and which are delimited with specific pragmas. Those pragmas and restrictions are detailed in the next sections.

2.2.1 SCoP Pragmas

The input file may contain several code parts to translate delimited **by the user** thanks to pragmas. Those pragmas indicate to Clan which program parts needs to be translated, and more important, which program parts can be translated to a polyhedral representation. We call them SCoPs (for Static Control Parts). Those parts must respect several syntactical and semantical properties which ensure an easy and safe translation to a polyhedral representation.

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

```
#pragma scop
```

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

```
#pragma endscop
```

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

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

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

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

Clan trusts the user: it will not check hardly whether a program part to translate is actually a SCoP or not. It will only try to translate the program part to a polyhedral representation. If there is a syntactical issue, it will complain and try to provide the user with some hints about the problem. If there is a semantical issue, it will not detect it. Hence, SCoP pragmas must be placed with caution.

2.2.2 Semantical Restrictions

There are a few but very important semantical restrictions a program part between SCoP pragmas must satisfy:

- All functions called within the SCoP are pure, i.e., they work only with their respective input parameters and local variables and have no side effects, e.g., they will not use a global variable or step outside array boundaries.
- No aliasing of array names is possible within the SCoP, e.g., if two arrays (or pointers) refer to the same memory location, they must have different names.
- Pointer references behave like variables or arrays, e.g., function pointers are not allowed.

Clan doesn't have (and will much probably never have) the analysis power to check for those properties, hence, it is the responsibility of the user to ensure they are satisfied. The action of setting the SCoP pragmas states explicitely that they are respected. If they are not, Clan's behavior is undefined.

2.2.3 Syntactical Restrictions

Clan only accepts a subset of C between SCoP pragmas. The big picture is, Clan is able to translate loop-based codes where loop bounds, `if` conditions and array subscripts are made of affine expressions involving only outer loop iterators, integer constants (a.k.a. parameters) and integer literals. The following general restrictions apply to all the code between SCoP pragmas:

- The only allowed control keywords are `for`, `while`, `if` and `else`, with restrictions as described below.
- Declarations are not allowed.
- Any C instruction without control keywords is accepted, with restrictions for array subscripts as described below.

More specific restrictions apply to the control statements and the array subscript. They refer to affine expressions and specific operators:

- Affine expressions are additive forms of loop iterators (e.g., `i`), parameters (e.g., `N`) and integers, with integer coefficients, e.g., `7*i + 13*N + 42`. Expressions which simplify to affine forms are accepted as valid affine expresions, e.g., `3*(i*2 + N)`, as long as the only operators in such expressions are `+`, `-` and `*`.
- Four particular operators may be used in some expressions: (1) the `max` operator returns the maximum between an affine expression or a `max` expression and another affine expression or `max` expression, e.g., `max(i+1,max(j,n))` is possible. (2) The `min` operator works like the `max` operator but for minimum. (3) The `ceild` operator gives the integer ceiling of an affine expression divided by an integer, e.g., `ceild(2*i+n, 3)` is a correct use or the operator. (4) The `floord` operator gives the integer floor of an affine expression divided by an integer, e.g., `floord(2*i+n, 3)` is a correct use of the operator.

Four specific zones in SCoP codes are restricted:

```
for (initialization; condition; step)
  if (condition)
    ... [subscript] ...
```

- the first part of `for` loop expressions, called loop initialization,
- the second part of `for` loop expressions, called loop condition, and the condition of `if` conditionals, share the same kind of restriction,
- the third part of `for` loop expressions, called loop step,
- the array subscripts.

Those restrictions are detailed in the next sections.

2.2.3.1 Loop Initialization

Each loop initialization must be an assignment of the loop counter such that the right hand side is one or several affine expressions aggregated with `max` (resp. `min`) operators if the loop step is positive (resp. negative). Optionally, the loop iterator can be declared (as an `int`) in the loop initialization part.

For instance:

- `j = 3*i + 2*N` is correct.
- `int j = 3*i + 2*N` is correct.
- `j = ceild(i + N, 10)` is correct if the `j`-loop step is positive.
- `j = max(i, ceild(N, 3))` is correct if the `j`-loop step is positive.
- `j = min(min(N,10), 7*i)` is correct if the `j`-loop step is negative.
- `j = min(max(i, 1), N)` is NOT correct: mixed `min` and `max`.

2.2.3.2 Loop and `if` Condition

Each loop or `if` condition must be a (composition of) constraint(s) on affine expressions only.

- Supported C operators are `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&&` and `||`.
- `min` and `max` operators can be used to aggregate expressions in `>`, `>=`, `<` and `<=` constraints. `min` (resp. `max`) expressions must be in the greater (resp. lower) side of the constraints.
- Constraints involving the modulo operator are possible in the following form only: let `a` be an affine expression and `x` and `y` two positive integers, then the condition `(a % x == y)` is accepted.
- Function calls alone can be used as valid `if` conditions.

For instance:

- `i + 2*j < N` is correct.
- `max(i, j) < floord(N, 7)` is correct.
- `N>i && !(j>0 || N!=1)` is correct.
- `((2*i+1)%3 == 1) && i>j` is correct.
- `func(A[i], b)` is correct in a `if` condition.
- `min(2*i, N) < 0` is NOT correct: `min` on the lower side.
- `i + 2` is NOT correct: use `(i + 2) != 0` instead.

2.2.3.3 Loop Step

Updating the loop iterator is only allowed in the loop step part. It must be done by adding an integer to the previous iterator value. Let `i` be a loop iterator and `x` an integer (a literal like `42`, not an integer variable), the following forms are accepted for the loop step part: `i++`, `++i`, `i--`, `--i`, `i += x`, `i -= x`, `i = i+x` and `i = i-x`. The iterator can optionally be enclosed in parentheses in this part: `++(i)`.

2.2.3.4 Array Subscript

Array subscripts must be affine expressions.

2.3 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 OpenScop, see [Bas12], page 23. 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 CLooG) while being polyhedral library independent. To guarantee an up-to-date information, the reader is invited to read the OpenScop document directly see [Bas12], page 23.

2.4 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 OpenScop 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 2.5 [Clan Options], page 9). `file` is the input file list (Clan may extract SCoPs from several input files). `stdin` is a special value: when used, input is the standard input and no other input file can be provided. 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).

2.5 Clan Options

2.5.1 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`.

2.5.2 Autoscop `-autoscop`

`-autoscop`: this option asks Clan to automatically find SCoPs within the input souce file. This options should be used with care because Clan will only ensure the syntactical restrictions are respected (see Section 2.2.2 [Semantical Restrictions], page 7), but not the semantical restrictions (see Section 2.2.2 [Semantical Restrictions], page 7). If the input file already contains SCoP pragmas, this option will leave them unmodified (as a consequence it will not be possible to include the user-SCoPs to larger automatically detected SCoPs).

2.5.3 Autopragma `-autopragma`

`-autopragma`: this option asks Clan to automatically generate a copy of the input file with SCoP pragmas inserted around automatically detected SCoPs. This options works in the same way as Autoscop (see Section 2.5.2 [Autoscop], page 10), but outputs a code instead of the OpenScop representation.

2.5.4 Autoinsert `-autoinsert`

`-autoinsert`: this option is equivalent to `-autopragma` but it inserts SCoP pragmas directly in the input file. Use with care: check with `-autopragma` first.

2.5.5 Inputscop `-inputscop`

`-inputscop`: this option asks Clan to use an OpenScop input instead of a source code (this is supposed to be a pass-through process).

2.5.6 Precision `-precision <value>`

`-precision`: this option asks Clan to work using a given precision for the various integer elements, `value` may be `32` for 32 bits, `64` for 64 bits (the default) or `0` fot GMP (the installed OpenScop library must have been compiled with GMP support for this option to apply).

2.5.7 Bounded Context `-boundedctxt`

`-boundedctxt`: this option asks Clan to generate context relations such that every parameter is considered to be >= -1.

2.5.8 No Loop Context `-noloopctxt`

`-noloopctxt`: this option asks Clan to avoid generating additional constraints to ensure the initial values of the loop iterators respect the loop conditions. This option allows to generate simpler iteration domains, however it should be used with care. For instance without this option, a loop such as `for (i = 0; i > 2; i++)` cannot be detected as an empty loop.

2.5.9 No Domain Simplify `-nosimplify`

`-nosimplify`: this option asks Clan to avoid trying to simplify the iteration domains. Clan is not linked to a polyhedral library, hence it is just able to remove trivially redundant constraints and union parts. However, for debugging or time saving reason, this step can be disabled using this option.

2.5.10 Extbody `-extbody`

`-extbody`: this option asks Clan to use the extbody extension (see OpenScop's documentation) instead of the basic body for each statement.

2.5.11 Help `--help` or `-h`

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

2.5.12 Version `--version` or `-v`

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

3 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 (an `osl_scop_t` structure from the OpenScop Library Clan depends on) which contains the polyhedral representation of the SCoP. Some other functions are provided for convenience reasons.

3.1 Clan Data Structures Description

In this section, we describe the data structures relevant for an user to use Clan as a library. As this is not a developer's guide, data structure devoted to internal use as well as internal use fields are not described here.

3.1.1 clan_options_t

```
struct clan_options {
  char * name ;   /* Name of the input file */
  int castle;     /* 1 to print the Clan Castle, 0 otherwise */
  int structure;  /* 1 to dump the SCoP structure, 0 otherwise */
  int autoscop;   /* 1 to extract SCoPs automatically, 0 otherwise */
  int autopragma; /* 1 to insert SCoP pragmas in the code, 0 otherwise */
  int inputscop;  /* 1 to read an OpenScop file, 0 for a source code */
  int precision;  /* 0 for GMP, 32 for 32 bits, 64 for 64 bits */
  int bounded_context; /* 1 to set all parameters to >= -1, 0 otherwise */
};
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 2.4 [Calling Clan], page 9). The default values for the various fields set by the `clan_options_malloc()` function (see Section 3.2.4 [Allocation and Initialization Functions], page 14) are the following:

- `name`: NULL.
- `castle`: 1 (do print the castle).
- `structure`: 0 (do not dump the internal data structure).
- `autoscop`: 0 (automatic SCoP extraction disabled).
- `autopragma`: 0 (automatic pragma insertion disabled).
- `inputscop`: 0 (read a source file by default).
- `precision`: 64 (use 64 bits precision by default).
- `bounded_context`: 0 (ubounded parameters by default).

3.2 Clan Functions Description

3.2.1 clan_scop_extract

```
osl_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 3.1.1 [clan_options_t], page 13). It returns a pointer to the extracted SCoP, translated into an `osl_scop_t` data structure (see [Bas12], page 23).

3.2.2 clan_scop_insert_pragmas

```
void clan_scop_insert_pragmas(osl_scop_p scop, char* file, int test);
```

The `clan_scop_insert_pragmas` function inserts SCoP pragmas (`#pragma scop` and `#pragma endscop`) in the file named `file` around the SCoPs described in the `scop` list, except if they are already surrounded with such pragmas. The boolean `test`, when set to 1, allows to only test the function and to generate a temporary file (the name of the file is in the `CLAN_AUTOPRAGMA` file) instead of modifying `file`.

3.2.3 clan_scop_print

```
void clan_scop_print (
  FILE * output,
  osl_scop_p scop,
  clan_options_p options
);
```

The function `clan_scop_print` is a pretty printer for `osl_scop_t` structures. It dumps the `scop` informations in OpenScop format (see [Bas12], page 23) 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 3.1.1 [clan_options_t], page 13).

3.2.4 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` has to be replaced by the name of the convenient structure (without '`clan`' prefix and '`_t`' suffix) for instance `clan_options_p clan_options_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.

3.2.5 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_options_free(clan_options_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.

3.2.6 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_options_print(FILE *, clan_options_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).

3.3 Example of Library Use

Here is a basic example showing how it is possible to use the Clan library, assuming that a standard installation has been done (note that this includes installing the OpenScop Library, `osl`, see Chapter 4 [Installing], page 17). 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 <osl/osl.h>
# include <clan/clan.h>

int main() {
  osl_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 OpenScop file. */
  osl_scop_print(stdout, scop);

  /* Save the planet. */
  clan_options_free(options);
  osl_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
```

4 Installing Clan

4.1 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 23 (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`

4.2 Requirements

Clan relies on the OpenScop Library, or `osl` for everything related to the OpenScop format. For convenience, an embedded version of osl is bundled along with Clan so it is not necessary for the user to do any additional step to install osl. However it is obviously possible to use an already installed version of osl as long as it is recent enough for Clan needs.

4.2.1 OpenScop Library (bundled)

OpenScop is a simple, polyhedral-library independent format to ease polyhedral representation transfers between the various elements of a polyhedral framework. The OpenScop Library, or `osl`, is an implementation of the OpenScop standard with utility functions. It can be freely downloaded from `http://www.lri.fr/~bastoul`. Since a convenient version of osl is bundled in Clan, it is not necessary to install it. However a user may wish to use a separate installation of osl. The user can compile it by typing the following commands on the osl root directory:

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

The osl 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 osl 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 osl library by specifying the convenient configure script options (see Section 4.4 [Optional Features], page 18).

4.3 Clan Basic Installation

Once downloaded and unpacked (e.g. using the '`tar -zxvf clan-0.8.0.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`.

4.4 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, `configure` will use the bundled OpenScop Library (osl). Using the `--with-osl` option of `configure` the user can specify that `no` osl, a previously installed (`system`) osl, a `bundled` osl, or a `build` osl should be used. In the latter case, the user should also specify the build location using `--with-osl-builddir=PATH`. In case of an installed osl, the installation location can be specified using the `--with-osl-prefix=PATH` and `--with-osl-exec-prefix=PATH` options of `configure`.

4.5 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`.

5 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 4 [Installing], page 17). 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 PDF format (by typing `texi2pdf clan.texi` or `make pdf`) or HTML format (by typing `makeinfo --html clan.texi`, using `--no-split` option to generate a single HTML file) or info format (by typing `makeinfo clan.texi`).

6 Development

6.1 Copyright Issue

Clan is an Open Source project and you should feel free to contribute by adding functionalities, correcting bugs or improving documentation. However, for painful administrative reasons, the copyright should not be impacted by your work. Hence, if you are doing a significant contribution to the main part, Clan's maintainer may ask you for an agreement about this copyright. If you plan to do such a significant contribution, it may be wise to discuss this issue with the maintainer first.

6.2 Repository

The main repository of Clan is `http://repo.or.cz/w/clan.git`. Developers may ask Clan's maintainer to open them a write access to this repository. Only the maintainer should ever change the `master` branch. Developers should work on their own branches. To avoid any problem developers should use the *fork* functionality of the repository.

6.3 Coding Style

Clan is written in C using an object oriented style. Each important data structure (e.g., `struct foo`) has its own header file (`include/clan/foo.h`) where lies the definition of the data structure, the two typedefs for the data structure (one for the structure, `clan_foo_t`, and one for a pointer to the structure, `clan_foo_p`), the prototypes of the various functions related to this data structure, all named using the prefix `"clan_foo_"`. The source code of the functions is provided in a separated C file (`source/foo.c`).

Utility functions independent from the main data structures may be placed in separate source files (e.g., definition in `include/clan/util.h` and code in `source/util.c`). Tool-wide preprocessor directives are placed in `include/clan/macros.h`, macros are prefixed with `"CLAN_"`.

The core code itself has to be written according to the Google C++ Coding Style: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml` (for what can apply to C), plus the naming conventions discussed above with highest priority.

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.
- [Bas12] C. Bastoul. A Specification and a Library for Data Exchange in Polyhedral Compilation Tools. Technical Report, Paris-Sud University, France, September 2012.