PIPS High-Level Software Interface
Pipsmake Configuration

Rémi Triolet, François Irigoin
and many other contributors
MINES ParisTech
Mathématiques et Systèmes
Centre de Recherche en Informatique
77305 Fontainebleau Cedex
France

Id: pipsmake-rc.tex 23481 2018-08-22 19:35:27Z ancourt

You can get a printable version of this document on
`http://www.cri.ensmp.fr/pips/pipsmake-rc.htdoc/pipsmake-rc.pdf` and a HTML version on
`http://www.cri.ensmp.fr/pips/pipsmake-rc.htdoc`.

Chapter 1

Introduction

This paper describes high-level objects and functions that are potentially user-visible in a *PIPS*[1] [33] interactive environment. It defines the internal *software* interface between a user interface and program analyses and transformations. This is clearly not a user guide but can be used as a reference guide, the best one before source code because `PIPS` user interfaces are very closely mapped on this document: some of their features are automatically derived from it.

Objects can be viewed and functions activated by one of `PIPS` existing user interfaces: `tpips`[2], the tty style interface which is currently recommended, `pips`[3] [11], the old batch interface, improved by many shell scripts[4], *wpips* and `epips`, the X-Window System interfaces. The `epips` interface is an extension of *wpips* which uses `Emacs` to display more information in a more convenient way. Unfortunately, right now these window-based interfaces are no longer working and have been replaced by `gpips`. It is also possible to use `PIPS` through a Python API, pyps.

From a theoretical point of view, the object types and functions available in `PIPS` define an heterogeneous algebra with constructors (e.g. parser), extractors (e.g. prettyprinter) and operators (e.g. loop unrolling). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and experiences users as well, and it was deemed necessary to assist them by providing default computation rules and automatic consistency management similar to `make`. The rule interpretor is called *pipsmake*[6] and described in [10]. Its key concepts are the *phase*, which correspond to a `PIPS` function made user-visible, for instance, a parser, the *resources*, which correspond to objects used or defined by the phases, for instance, a source file or an AST (parsed code), and the virtual *rules*, which define the set of input resources used by a phase and the set of output resources defined by the phase. Since `PIPS` is an interprocedural tool, some real inpu resources are not known until execution. Some variables such as `CALLERS` or `CALLEES` can be used in virtual rules. They are expanded at execution to obtain an effective rule with the precise resources needed.

[1] `http://www.cri.ensmp.fr/pips`
[2] `http://www.cri.ensmp.fr/pips/line-interface.html`
[3] `http://www.cri.ensmp.fr/pips/batch-interface.html`
[4] Manual pages are available for *Init*, *Select*, *Perform*, *Display*, and *Delete*, and `pips`[5].
[6] `http://www.cri.ensmp.fr/pips/pipsmake.html`

For debugging purposes and for advanced users, the precise choice and tuning of an algorithm can be made using *properties*. Default properties are installed with `PIPS` but they can be redefined, partly or entirely, by a `properties.rc` file located in the current directory. Properties can also be redefined from the user interfaces, for example with the command `setproperty` when the *tpips* interface is used.

As far as their static structures are concerned, most object types are described in more details in *PIPS Internal Representation of Fortran and C code*[7]. A dynamic view is given here. In which order should functions be applied? Which object do they produce and vice-versa which function does produce such and such objects? How does `PIPS` cope with bottom-up and top-down interproceedurality?

Resources produced by several rules and their associated rule must be given alias names when they should be explicitly computed or activated by an interactive interface. This is otherwise not relevant. The alias names are used to generate automatically header files and/or test files used by PIPS interfaces.

FI: I do not understand.

No more than one resource should be produced per line of rule because different files are automatically extracted from this one[8]. Another caveat is that *all* resources whose names are suffixed with `_file` are considered printable or displayable, and the others are considered binary data, even though they may be ASCII strings.

This LaTeX file is used by several procedures to derive some pieces of C code and ASCII files. The useful information is located in the *PipsMake* areas, a very simple literate programming environment... For instance `alias` information is used to generate automatically menus for window-based interfaces such as `wpips` or `gpips`. Object (a.k.a resource) types and functions are renamed using the alias declaration. The name space of aliases is global. All aliases must have different names. Function declarations are used to build a mapping table between function names and pointer to C functions, `phases.h`. Object suffixes are used to derive a header file, `resources.h`, with all resource names. Parts of this file are also extracted to generate on-line information for `wpips` and automatic completion for `tpips`.

The behavior of `PIPS` can be slightly tuned by using `properties` and some environment variables. Most properties are linked to a particular phase, for instance to prettyprint, but some are linked to `PIPS` infrastructure and are presented in Chapter 2.

1.1 Informal Pipsmake Syntax

To understand and to be able to write new rules for `pipsmake`, a few things need to be known.

1.1.1 Example

The rule:

```
proper_references       > MODULE.proper_references
```

[7] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

[8] See the local Makefile: pipsmake-rc, and alias file: wpips-rc.

```
< PROGRAM.entities
< MODULE.code
< CALLEES.summary_effects
```

means that the method `proper_references` is used to generate the `proper_references` resource of a given `MODULE`. But to generate this resource, the method needs access to the resource holding the symbole table, `entities`, of the `PROGRAM` currently analyzed, the `code` resource (the instructions) of the given `MODULE` and the `summary_effects` 6.2.4 resource (the side effects on the memory) of the functions and procedures called by the given `MODULE`, the `CALLEES`.

Properties are also declared in this file. For instance

```
ABORT_ON_USER_ERROR FALSE
```

declares a property to stop interpreting user commands when an error is made and sets its default value to false, which makes sense most of the time for interactive uses of `PIPS`. But for non-regression tests, it may be better to turn on this property.

1.1.2 Pipsmake variables

The following variables are defined to handle interprocedurality:

`PROGRAM`: the whole application currently analyzed;

`MODULE`: the current `MODULE` (a procedure or function);

`ALL`: all the `MODULE`s of the current `PROGRAM`, functions and compilation units;

`ALLFUNC`: all the `MODULE`s of the current `PROGRAM` that are functions;

`CALLEES`: all the `MODULE`s called in the given `MODULE`;

`CALLERS`: all the `MODULE`s that call the given `MODULE`.

These variables are used in the rule definitions and instantiated before `pipsmake` infers which resources are pre-requisites for a rule.

The environment variable `PIPS_IGNORE_FUNCTION_RX` is taken as a regular expression to filter out unwanted functions, such as static functions, inlined or not, which arise in some standard header files from time to time. For instance, with gcc 4.8, you should define

```
export PIPS_IGNORE_FUNCTION_RX='!__bswap_'
```

1.2 Properties and Environment Variables

This paper also defines and describes global variables used to modify or fine tune PIPS behavior. Since global variables are useful for some purposes, but always dangerous, PIPS programmers are required to avoid them or to declare them explicitly as *properties*. Properties have an ASCII name and can have boolean, integer or string values.

Casual users should not use them. Some properties are modified for them by the user interface and/or the high-level functions. Some property combinations

may be meaningless. More experienced users can set their values, using their names and a user interface.

Experienced users can also modify properties by inserting a file called `properties.rc` in their local directory. Of course, they cannot declare new properties, since they would not be recognized by the `PIPS` system. The local property file is read *after* the default property file, `$PIPS_ROOT/etc/properties.rc`. Some user-specified property values may be ignored because they are modified by a PIPS function before it had a chance to have any effect. Unfortunately, there is no explicit indication of usefulness for the properties in this report.

The default property file can be used to generate a custom version of `properties.rc`. It is derived automatically from this documentation, `Documentation/pipsmake-rc.tex`.

PIPS behavior can also be altered by Shell environment variables. Their generic names is `XXXX_DEBUG_LEVEL`, where `XXXX` is a library or a phase or an interface name (of course, there are exceptions). Theoretically these environment variables are also declared as properties, but this is generally forgotten by programmers. A debug level of 0 is equivalent to no tracing. The amount of tracing increases with the debug level. The maximum useful value is 9.

Another Shell environment variable, `NEWGEN_MAX_TABULATED_ELEMENTS`, is useful to analyze large programs. Its default value is 12,000 but it is not uncommon to have to set it up to 200,000.

Properties and environement variables are listed below on a source library basis. Properties used in more than one library or used by `PIPS` infrastructure are presented first. Section 2.3 contains information about properties related to infrastructure, external and user interface libraries. Properties for analyses are grouped in Chapter 6. Properties for program transformations, parallelization and distribution phases are listed in the next section in Chapters 9 and 8. User output produced by different kinds of prettyprinters are presented in Chapter 10. Chaper 11 is dedicated to properties of the libraries added by CEA to implement Feautrier's method.

1.3 Outline

Rule and object declaration are grouped in chapters: input files (Chapter 3), syntax analysis and abstract syntax tree (Chapter 4), analyses (Chapter 6), parallelizations (Chapter 8), program transformations (Chapter 9) and prettyprinters of output files (Chapter 10). Chapter 11 describes several analyses defined by Paul Feautrier. Chapter 12 contains a set of menu declarations for the window-based interfaces.

Virtually every PIPS programmer contributed some lines in this report. Inconsistencies are likely. Please report them to the PIPS team[9]!

[9] pips-support@cri.ensmp.fr

Contents

Chapter 2

Global Options

Options are called *properties* in PIPS. Most of them are related to a specific *phase*, for instance the dependence graph computation. They are declared next to the corresponding phase declaration. But some are related to one library or even to several libraries and they are declared in this chapter.

Skip this chapter on first reading. Also skip this chapter on second reading because you are unlikely to need these properties until you develop in PIPS.

2.1 Fortran Loops

Are DO loops bodies executed at least once (F-66 style), or not (Fortran 77)?

```
ONE_TRIP_DO FALSE
```

is useful for use/def and semantics analysis but is not used for region analyses. This dangerous property should be set to FALSE. It is not consistently checked by PIPS phases, because nobody seems to use this obsolete Fortran feature anymore.

2.2 Logging

With

```
LOG_TIMINGS FALSE
```

it is possible to display the amount of real, cpu and system times directly spent in each phase as well as the times spent reading/writing data structures from/to PIPS database. The computation of total time used to complete a `pipsmake` request is broken down into global times, a set of phase times which is the accumulation of the times spent in each phase, and a set of IO times, also accumulated through phases.

Note that the IO times are included in the phase times.

With

```
LOG_MEMORY_USAGE FALSE
```

it is possible to log the amount of memory used by each phase and by each request. This is mainly useful to check if a computation can be performed on a given machine. This memory log can also be used to track memory leaks. `Valgrind` may be more useful to track memory leaks.

2.3 PIPS Infrastructure

`PIPS` infrastructure is based on a few external libraries, Newgen and Linear, and on three key *PIPS*[1] libraries:

- `pipsdbm` which manages resources such as `code` produced by PIPS and ensures persistance,
- `pipsmake` which ensures consistency within a `workspace` with respect to the producer-consumer rules declared in this file,
- and `top-level` which defines a common API for all `PIPS` user interfaces, whether human or API.

2.3.1 Newgen

Newgen offers some debugging support to check object consistency (`gen_consistent_p` and `gen_defined_p`), and for dynamic type checking. See Newgen documentation[50][51].

2.3.2 C3 Linear Library

This library is external and offers an independent debugging system.

The following properties specify how null (

```
SYSTEM_NULL "<null␣system>"
```

), undefined

```
SYSTEM_UNDEFINED "<undefined␣system>"
```

) or non feasible systems

```
SYSTEM_NOT_FEASIBLE "{0==-1}"
```

are prettyprinted by `PIPS`.

2.3.3 PipsMake Library

With

```
CHECK_RESOURCE_USAGE FALSE
```

it is possible to log and report differences between the set of resources actually read and written by the procedures called by pipsmake and the set of resources declared as read or written in pipsmake.rc file.

```
ACTIVATE_DEL_DERIVED_RES TRUE
```

[1] `http://www.cri.ensmp.fr/pips`

controls the rule activation process that may delete from the database all the derived resources from the newly activated rule to make sure that non-consistent resources cannot be used by accident.

```
PIPSMAKE_CHECKPOINTS 0
```

controls how often resources should be saved and freed. 0 means never, and a positive value means every n applications of a rule. This feature was added to allow long big automatic tpips scripts that can coredump and be restarted latter on close to the state before the core. As another side effect, it allows to free the memory and to keep memory consumption as moderate as possible, as opposed to usual tpips runs which keep all memory allocated. Note that it should not be too often saved, because it may last a long time, especially when entities are considered on big workspaces. The frequency may be adapted in a script, rarely at the beginning to more often latter.

2.3.4 PipsDBM Library

Shell environment variables `PIPSDBM_DEBUG_LEVEL` can be set to ? to check object consistency when they are stored in the database, and to ? to check object consistency when they are stored or retrieved (in case an intermediate phase has corrupted some data structure unwillingly).

You can control what is done when a workspace is closed and resources are saved. The

```
PIPSDBM_RESOURCES_TO_DELETE "obsolete"
```

property can be set to to "obsolete" or to "all".

Note that it is not managed from `pipsdbm` but from `pipsmake`, which knows what is obsolete or not.

2.3.5 Top Level Library

The `top-level` library is built on top of the `pipsmake` and `pipsdbm` libraries to factorize functions useful to build a `PIPS` user interface or API.

Property

```
USER_LOG_P TRUE
```

controls the logging of the session in the database of the current workspace. This log can be processed by `PIPS` utility `logfile2tpips` to generate automatically a *tpips* script which can be used to replay the current `PIPS` session, workspace by workspace, regardless of the `PIPS`user interface used.

Property

```
ABORT_ON_USER_ERROR FALSE
```

specifies how user errors impact execution once the error message is printed on `stderr`: return and go ahead, usually when `PIPS` is used interactively (default behavior), or abort and core dump for debugging purposes and for script executions, especially non-regression tests.

Property

```
CLOSE_WORKSPACE_AND_QUIT_ON_ERROR FALSE
```

specifies that user and internal errors must preserve as much as possible the workspace created by PIPS. This behavior stores on disk, as much as possible, all information available on the process that has just failed. This is useful when PIPS is called by another tool. This is not compatible with ABORT_ON_USER_ERROR 2.3.5, which seeks an immediate termination of the PIPS process.

Property

```
MAXIMUM_USER_ERROR 2
```

specifies the number of user error allowed before the programs brutally aborts.

Property

```
ACTIVE_PHASES "PRINT_SOURCE␣PRINT_CODE␣PRINT_PARALLELIZED77_CODE␣PRINT_CALL_GRAPH␣P
```

specifies which pipsmake phases should be used when several phases can be used to produce the same resource. This property is used when a workspace is created. A workspace is the database maintained by PIPS to contain all resources defined for a whole application or for the whole set of files used to create it.

Property

```
PIPSMAKE_WARNINGS TRUE
```

controls whether to show warning when reading and activating pipsmake rules. Turning it off is useful when validating with a specialized version of PIPS, as some undesirable warnings can be shown then.

Resources that create ambiguities for pipsmake are at least:

- parsed_printed_file
- printed_file
- callgraph_file
- icfg_file
- parsed_code, because several parsers are available
- transformers
- summary_precondition
- preconditions
- regions
- chains
- dg

This list must be updated according to new rules and new resources declared in this file. Note that no default parser is usually specified in this property, because it is selected automatically according to the source file suffixes when possible.

Until October 2009, the active phases were:

```
ACTIVE_PHASES "PRINT_SOURCE PRINT_CODE PRINT_PARALLELIZED77_CODE
               PRINT_CALL_GRAPH PRINT_ICFG TRANSFORMERS_INTRA_FAST
               INTRAPROCEDURAL_SUMMARY_PRECONDITION
               PRECONDITIONS_INTRA ATOMIC_CHAINS
               RICE_FAST_DEPENDENCE_GRAPH MAY_REGIONS"
```

They still are used for the old non-regression tests.

Property

```
CONSISTENCY_ENFORCED_P FALSE
```

specifies that properties cannot be set once a PIPS database has been created. Pipsmake does not know the impacts of properties on the resources. Setting a property can make a resource obsolete, but pipsmake is going to use it as consistent. To avoid the issue, set `CONSISTENCY_ENFORCED_P` 2.3.5 to true and `tpips`[2] will detect a user error if a property is possibly altered during a processing phase.

2.3.6 Warning Management

User warnings may be turned off. Definitely, this is not the default option! Most warnings *must* be read to understand surprising results. This property is used by library `misc`.

```
NO_USER_WARNING FALSE
```

By default, `PIPS` reports errors generated by system call `stat` which is used in library `pipsdbm` to check the time a resource has been written and hence its temporal consistency.

```
WARNING_ON_STAT_ERROR TRUE
```

Error messages are also copied in the Warnings file.

2.3.7 Option for C Code Generation

The syntactic constraints of C89 have been eased for declarations in C99, where it is possible to intersperse statement declarations within executable statements. This property is used to request C89 compatible code generation.

```
C89_CODE_GENERATION FALSE
```

So the default option is to generate C99 code, which may be changed because it is likely to make the code generated by PIPS unparsable by PIPS.

There is no guarantee that each code generation phase is going to comply with this property. It is up to each developper to decide if this global property is to be used or not in his/her local phase.

[2] `http://www.cri.ensmp.fr/pips/line-interface.html`

2.4 User and Programming Interfaces

2.4.1 Tpips Command Line Interface

tpips is one of PIPS user interfaces.

```
TPIPS_IS_A_SHELL FALSE
```

controls whether *tpips* should behave as an extended shell and consider any input command that is not a *tpips* command a Shell command.

2.4.2 Pyps API

This property is automatically set to TRUE when pyps is running.

```
PYPS FALSE
```

Chapter 3

Input Files

3.1 User File

An input program is a set of user Fortran 77, Fortran 90 or C source files and a name, called a *workspace*. The files are looked for in the current directory, then by using the colon-separated `PIPS_SRCPATH` variable for other directories where they might be found. The first occurrence of the file name in the ordered directories is chosen, which is consistent with `PATH` and `MANPATH` behaviour.

The source files are splitted by `PIPS` at the program initialization phase to produce one `PIPS`-private source file for each procedure, subroutine or function, and for each block data. A function like `fsplit` is used and the new files are stored in the workspace, which simply is a UNIX sub-directory of the current directory. These new files have names suffixed by `.f.orig`.

Since `PIPS` performs interprocedural analyses, it expects to find a source code file for each procedure or function called. Missing modules can be replaced by stubs, which can be made more or less precise with respect to their effects on formal parameters and global variables. A stub may be empty. Empty stubs can be automatically generated if the code is properly typed (see Section 3.3).

The *user* source files should not be edited by the user once `PIPS` has been started because these editions are not going to be taken into account unless a new workspace is created. But their preprocessed copies, the `PIPS` source files, safely can be edited while running `PIPS`. The automatic consistency mechanism makes sure that any information displayed to the user is consistent with the current state of the sources files in the workspace. These source files have names terminated by the standard suffix, `.f`.

New user source files should be automatically and completely re-built when the program is no longer under `PIPS` control, i.e. when the workspace is closed. An executable application can easily be regenerated after code transformations using the `tpips`[1] interface and requesting the `PRINTED_FILE` resources for all modules, including compilation units in C:

```
display PRINTED_FILE[%ALL]
```

Note that compilation units can be left out with:

```
display PRINTED_FILE[%ALLFUNC]
```

[1] `http://www.cri.ensmp.fr/pips/line-interface.html`

In both cases with C source code, the order of modules may be unsuitable for direct recompilation and compilation units should be included anyway, but this is what is done by explicitly requesting the code regeneration as described in § 3.4.

Note that PIPS expects proper ANSI Fortran 77 code. Its parser was not designed to locate syntax errors. It is highly recommended to check source files with a standard Fortran compiler (see Section 3.2) before submitting them to PIPS.

3.2 Preprocessing and Splitting

3.2.1 Fortran 77 Preprocessing and Splitting

The Fortran 77 files specified as input to PIPS by the user are preprocessed in various ways.

3.2.1.1 Fortran 77 Syntactic Verification

If the PIPS_CHECK_FORTRAN shell environment variable is defined to false or no or 0, the syntax of the source files is not checked by compiling it with a C compiler.If the PIPS_CHECK_FORTRAN shell environment variable is defined to true or yes or 1, the syntax of the file is checked by compiling it with a Fortran 77 compiler. If the PIPS_CHECK_FORTRAN shell environment variable is not defined, the check is performed according to CHECK_FORTRAN_SYNTAX_BEFORE_RUNNING_PIPS 3.2.1.1.

The Fortran compiler is defined by the PIPS_FLINT environment variable. If it is undefined, the default compiler is f77 -c -ansi).

In case of failure, a warning is displayed. Note that if the program cannot be compiled properly with a Fortran compiler, it is likely that many problems will be encountered within PIPS.

The next property also triggers this preliminary syntactic verification.

```
CHECK_FORTRAN_SYNTAX_BEFORE_RUNNING_PIPS TRUE
```

PIPS requires source code for all leaves in its visible call graph. By default, a user error is raised by Function initializer if a user request cannot be satisfied because some source code is missing. It also is possible to generate some synthetic code (also known as *stubs*) and to update the current module list but this is not a very satisfying option because all interprocedural analysis results are going to be wrong. The user should retrieve the generated .f files in the workspace, under the Tmp directory, and add some assignments (*def*) and *use*s to mimic the action of the real code to have a sufficient behavior from the point of view of the analysis or transformations you want to apply on the whole program. The user modified synthetic files should then be saved and used to generate a new workspace.

If PREPROCESSOR_MISSING_FILE_HANDLING 3.2.1.1 is set to "query", a script can optionally be set to handle the interactive request using PREPROCESSOR_MISSING_FILE_GENERATOR 3.2.1.1. This script is passed the function name and prints the filename on standard output. When empty, it uses an internal one.

Valid settings: error or generate or query.

```
PREPROCESSOR_MISSING_FILE_HANDLING "error"
```

```
PREPROCESSOR_MISSING_FILE_GENERATOR ""
```

The generated stub can have various default effect, say to prevent over-optimistic parallelization.

```
STUB_MEMORY_BARRIER FALSE
```

```
STUB_IO_BARRIER FALSE
```

3.2.1.2 Fortran 77 File Preprocessing

If the file suffix is `.F` then the file is preprocessed. By default PIPS uses `gfortran -E` for Fortran files. This preprocessor can be changed by setting the `PIPS_FPP` environment variable.

Moreover the default preprocessing options are `-P -D__PIPS__ -D__HPFC__` and they can be extended (not replaced...) with the `PIPS_FPP_FLAGS` environment variable.

3.2.1.3 Fortran 77 Split

The file is then split into one file per module using a PIPS specialized version of `fsplit`[2]. This preprocessing also handles

1. Hollerith constants by converting them to the quoted syntax[3];
2. unnamed modules by adding `MAIN000` or `PROGRAM MAIN000` or or `DATA000` or `BLOCK DATA DATA000` according to needs.

The output of this phase is a set of `.f_initial` files in per-module subdirectories. They constitute the resource `INITIAL_FILE`.

3.2.1.4 Fortran Syntactic Preprocessing

A second step of preprocessing is performed to produce `SOURCE_FILE` files with standard Fortran suffix `.f` from the `.f_initial` files. The two preprocessing steps are shown in Figure 3.1.

Each module source file is then processed by `top-level` to handle Fortran `include` and to comment out `IMPLICIT NONE` which are not managed by `PIPS`. Also this phase performs some transformations of complex constants to help the `PIPS` parser. Files referenced in Fortran `include` statements are looked for from the directory where the Fortran file is. The Shell variable `PIPS_CPP_FLAGS` is *not* used to locate these include files.

[2]The PIPS version of `fsplit` is derived from the BSD `fsplit` and several improvements have been performed.

[3]Hollerith constants are considered obsolete by the new Fortran standards and date back to 1889...

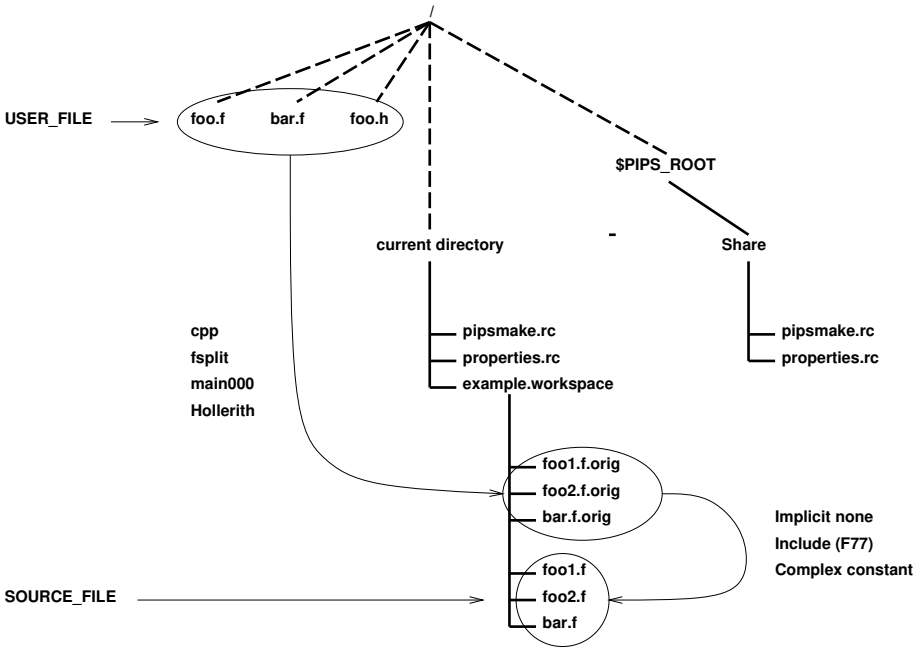

Figure 3.1: Preprocessing phases: from a user file to a source file

3.2.2 C Preprocessing and Splitting

The C preprocessor is applied before the splitting. By default PIPS uses `cpp -C` for C files. This preprocessor can be changed by setting the `PIPS_CPP` environment variable.

Moreover the `-D__PIPS__ -D__HPFC__ -U__GNUC__` preprocessing options are used and can be extended (not replaced) with the `PIPS_CPP_FLAGS` environment variable.

This `PIPS_CPP_FLAGS` variable can also be used to locate the include files. Directories to search are specified with the `-I`*`file`* option, as usual for the C preprocessor.

3.2.2.1 C Syntactic Verification

If the `PIPS_CHECK_C` shell environment variable is defined to `false` or `no` or `0`, the syntax of the source files is not checked by compiling it with a C compiler. If the `PIPS_CHECK_C` shell environment variable is defined to `true` or `yes` or `1`, the syntax of the file is checked by compiling it with a C compiler. If the `PIPS_CHECK_C` shell environment variable is not defined, the check is performed according to `CHECK_C_SYNTAX_BEFORE_RUNNING_PIPS` 3.2.2.1.

The environment variable `PIPS_CC` is used to define the C compiler available. If it is undefined, the compiler chosen is `gcc -c`).

In case of failure, a warning is displayed.

If the environement variable `PIPS_CPP_FLAGS` is defined, it should contain the options `-Wall` and `-Werror` for the check to be effective.

The next property also triggers this preliminary syntactic verification.

```
CHECK_C_SYNTAX_BEFORE_RUNNING_PIPS TRUE
```

Although its default value is `FALSE`, it is much safer to set it to true when dealing with new sources files. `PIPS` is not designed to process non-standard source code. Bugs in source files are not well explained or localized. They can result in weird behaviors and inexpected core dumps. Before complaining about `PIPS`, it is higly recommended to set this property to `TRUE`.

Note: the C and Fortran syntactic verifications could be controlled by a unique property.

3.2.3 Fortran 90 Preprocessing and Splitting

The Fortran 90 parser is a separate program, derived from gcc Fortran parser. It is activated directly when the workspace is created, and not by pipsmake.

3.2.4 Source File Hierarchy

The source files may be placed in different directories and have the same name, which makes resource management more difficult. The default option is to assume that no file name conflicts occur. This is the historical option and it leads to much simpler module names.

```
PREPROCESSOR_FILE_NAME_CONFLICT_HANDLING FALSE
```

3.3 Source Files

A `source_file` contains the code of exactly one module. Source files are created from user source files at program initialization by `fsplit` or a similar function if `fsplit` is not available (see Section 3.2). A source file may be updated by the user[4], but not by `PIPS`. Program transformations are performed on the internal representation (see 4) and visible in the prettyprinted output (see 10).

Source code splitting and preprocessing, e.g. `cpp`, are performed by the function `create_workspace()` from the `top-level` library, in collaboration with `db_create_workspace()` from library `pipsdbm` which creates the workspace directory. The user source files have names suffixed by `.f` or `.F` if `cpp` must be applied. They are split into *original* `user_files` with suffix `.f.orig`. These so-called original user files are in fact copies stored in the workspace. The syntactic PIPS preprocessor is applied to generate what is known as a `source_file` by PIPS. This process is fully automatized and not visible from PIPS user interfaces. However, the cpp preprocessor actions can be controlled using the Shell environment variable `PIPS_CPP_FLAGS`.

Function `initializer` is only called when the source code is not found. If the user code is properly typed, it is possible to force `initializer` to generate empty stubs by setting properties `PREPROCESSOR_MISSING_FILE_HANDLING` 3.2.1.1 and, to avoid inconsistency, `PARSER_TYPE_CHECK_CALL_SITES` 4.2.1.4. But remember that many Fortran codes use subroutines with variable numbers of

[4]The X-window interface, `wpips` has an `edit` entry in the transformation menu.

arguments and with polymorphic types. Fortran varargs mechanism can be achieved by using or not the second argument according to the first one. Polymorphism can be useful to design an IO package or generic array subroutine, e.g. a subroutine setting an array to zero or a subroutine to copy an array into another one.

The current default option is to generate a user error if some source code is missing. This decision was made for two reasons:

1. too many warnings about typing are generated as soon as polymorphism is used;
2. analysis results and code transformations are potentially wrong because no memory effects are synthesized; see Properties `MAXIMAL_PARAMETER_EFFECTS_FOR_UNKNOWN_FUNCTIONS` and `MAXIMAL_EFFECTS_FOR_UNKNOWN_FUNCTIONS` 6.2.7.7.

Sometimes, a function happen to be defined (and not only declared) inside a header file with the inline keyword. In that case PIPS can consider it as a regular module or just ignore it, as its presence may be system-dependant. Property `IGNORE_FUNCTION_IN_HEADER` 3.3 control this behavior and must be set *before* workspace creation.

```
IGNORE_FUNCTION_IN_HEADER TRUE
```

Modules can be flagged as "stubs", aka functions provided to PIPS but which shouldn't be inlined or modified. Property `PREPROCESSOR_INITIALIZER_FLAG_AS_STUB` 3.3 controls if the initializer should declare new files as stubs.

```
bootstrap_stubs > PROGRAM.stubs

flag_as_stub                    > PROGRAM.stubs
              < PROGRAM.stubs
```

```
PREPROCESSOR_INITIALIZER_FLAG_AS_STUB TRUE
```

```
initializer                     > MODULE.user_file
                                > MODULE.initial_file
```

Note: the generation of the resource `user_file` here above is mainly directed in having the resource concept here. More thought is needed to have the concept of user files managed by `pipsmake`.

MUST appear after initializer:

```
filter_file                     > MODULE.source_file
              < MODULE.initial_file
              < MODULE.user_file
```

In C, the initializer can generate directly a `c_source_file` and its compilation unit.

```
c_initializer                     > MODULE.c_source_file
                                  > COMPILATION_UNIT.c_source_file
                                  > MODULE.input_file_name
```

3.4 Regeneration of User Source Files

The `unsplit` 3.4 phase regenerates user files from available `printed_file`. The various modules that where initially stored in single file are appended together in a file with the same name. Not that just `fsplit` is reversed, not a preprocessing through `cpp`. Also the `include` file preprocessing is not reversed.

Regeneration of user files. The various modules that where initially stored in single file are appended together in a file with the same name.

```
alias unsplit 'User files Regeneration'

unsplit                          > PROGRAM.user_file
                < ALL.user_file
                < ALL.printed_file

unsplit_parsed                   > PROGRAM.user_file
                < ALL.user_file
                < ALL.parsed_printed_file
```

Chapter 4

Building the Internal Representation

The abstract syntax tree, a.k.a intermediate representation, a.k.a. internal representation, is presented in [34] and in *PIPS Internal Representation of Fortran and C code*[1].

4.1 Entities

Program entities are stored in `PIPS` unique symbol table[2], called `entities`. Fortran entities, like intrinsics and operators, are created by `bootstrap` at program initialization. The symbol table is updated with user local and global variables when modules are parsed or linked together. This side effect is not disclosed to `pipsmake`.

`bootstrap` > `PROGRAM.entities`

The entity data structure is described in *PIPS Internal Representation of Fortran and C code*[3].

The declaration of new intrinsics is not easy because it was assumed that there number was fixed and limited by the Fortran standard. In fact, Fortran extensions define new ones. To add a new intrinsic, C code in `bootstrap/bootstrap.c` and in `effects-generic/intrinsics.c` must be added to declare its name, type and Read/Write memory effects.

Information about entities generated by the parsers is printed out conditionally to property: `PARSER_DUMP_SYMBOL_TABLE` 4.2.1.4. which is set to false by default. Unless you are debugging the parser, do not set this property to TRUE but display the symbol table file. See Section 4.2.1.4 for Fortran and Section 4.2.3 for C.

[1] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

[2] FI: retrospectively, having a *unique* symbol table for all modules was a design mistake. The decision was made to have homogeneous accesses to local and global entities. It was also made to match NewGen *tabulated* type declaration.

[3] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

4.2 Parsed Code and Callees

Each module source code is parsed to produce an internal representation called `parsed_code` and a list of called module names, `callees`.

4.2.1 Fortran 77

Source code is assumed to be fully Fortran-77 compliant. The syntax should be checked by a standard Fortran compiler, e.g. `fort77` or at least `gfortran`, before the PIPS Fortran 77 parser is activated. On the first encountered error, the parser may be able to emit a useful message or the non-analyzed part of the source code is printed out.

`PIPS` input language is standard Fortran 77 with few extensions and some restrictions. The input character set includes underscore, `_`, and varying length variable names, i.e. they are not restricted to 6 characters are supported as well as dependent types for arrays.

4.2.1.1 Fortran 77 Restrictions

1. `ENTRY` statements are not recognized and a user error is generated. Very few cases of this obsolete feature were encountered in the codes initially used to benchmark `PIPS`. ENTRY statements have to be replaced manually by `SUBROUTINE` or `FUNCTION` and appropriate commons. If the parser bumps into a call to an ENTRY point, it may wrongly diagnose a missing source code for this entry, or even generate a useless but `pipsmake` satisfying stub if the corresponding property has been set (see Section 3.3).

2. Multiple returns are not in `PIPS` Fortran.

3. `ASSIGN` and assigned `GOTO` are not in `PIPS` Fortran.

4. Computed `GOTO`s are not in `PIPS` Fortran. They are automatically replaced by a `IF...ELSEIF...ENDIF` construct in the parser.

5. Functional formal parameters are not accepted. This is deeply exploited in `pipsmake`.

6. Integer PARAMETERs must be initialized with integer constant expressions because conversion functions are not implemented.

7. DO loop headers should have no label. Add a `CONTINUE` just before the loop when it happens. This can be performed automatically if the property `PARSER_SIMPLIFY_LABELLED_LOOPS` 4.2.1.4 is set to `TRUE`. This restriction is imposed by the parallelization phases, not by the parser.

8. Complex constants, e.g. `(0.,1.)`, are not directly recognized by the parser. They must be replaced by a call to intrinsic `CMPLX`. The `PIPS` preprocessing replaces them by a call to `COMPLX_`.

9. Function formulae are not recognized by the parser. An undeclared array and/or an unsupported macro is diagnosed. They may be substituted in an unsafe way by the preprocessor if the property

   `PARSER_EXPAND_STATEMENT_FUNCTIONS` 4.2.1.4

is set. If the substitution is considered possibly unsafe, a warning is displayed.

These parser restrictions were based on funding constraints. They are mostly alleviated by the preprocessing phase. PerfectClub and SPEC-CFP95 benchmarks are handled without manual editing, but for ENTRY statements which are obsoleted by the current Fortran standard.

4.2.1.2 Some Additional Remarks

- The PIPS preprocessing stage included in `fsplit()` is going to name unnamed modules `MAIN000` and unnamed blockdata `DATA000` to be consistent with the generated file name.
- Hollerith constants are converted to a more readable quoted form, and then output as such by the prettyprinter.

4.2.1.3 Some Unfriendly Features

1. Source code is read in columns 1-72 only. Lines ending in columns 73 and beyond usually generate incomprehensible errors. A warning is generated for lines ending after column 72.
2. Comments are carried by the *following* statement. Comments carried by `RETURN`, `ENDDO`, `GOTO` or `CONTINUE` statements are not always preserved because the internal representation transforms these statements or because the parallelization phase regenerates some of them. However, they are more likely to be hidden by the prettyprinter. There is a large range of prettyprinter *properties* to obtain less filtered view of the code.
3. Formats and character constants are not properly handled. Multi-line formats and constants are not always reprinted in a Fortran correct form.
4. Declarations are exploited on-the-fly. Thus type and dimension information must be available *before* common declaration. If not, wrong common offsets are computed at first and fixed later in Function `EndOfProcedure`). Also, formal arguments implicitly are declared using the default implicit rule. If it is necessary to declare them, this new declarations should occur *before* an `IMPLICIT` declaration. Users are surprised by the *type redefinition* errors displayed.

4.2.1.4 Declaration of the Standard Fortran 77 Parser

```
parser                          > MODULE.parsed_code
                                > MODULE.callees
        < PROGRAM.entities
        < MODULE.source_file
```

For parser debugging purposes, it is possible to print a summary of the symbol table, when enabling this property:

```
PARSER_DUMP_SYMBOL_TABLE FALSE
```

This should be avoided and the resource `symbol_table_file` be displayed instead.

The prettyprint of the symbol table for a Fortran or C module is generated with:

```
parsed_symbol_table        > MODULE.parsed_symbol_table_file
    < PROGRAM.entities
    < MODULE.parsed_code
```

Input Format

Some subtle errors occur because the PIPS parser uses a fixed format. Columns 73 to 80 are ignored, but the parser may emit a warning if some characters are encountered in this comment field.

```
PARSER_WARN_FOR_COLUMNS_73_80 TRUE
```

ANSI extension

PIPS has been initially developed to parse correct Fortran compliant programs only. Real applications use lots of ANSI extensions... and they are not always correct! To make sure that PIPS output is correct, the input code should be checked against ANSI extensions using property

CHECK_FORTRAN_SYNTAX_BEFORE_PIPS

(see Section 3.2) and the property below should be set to false.

```
PARSER_ACCEPT_ANSI_EXTENSIONS TRUE
```

Currently, this property is not used often enough in PIPS parser which let go many mistakes... as expected by real users!

Array Range Extension

PIPS has been developed to parse correct Fortran-77 compliant programs only. Array ranges are used to improve readability. They can be generated by PIPS prettyprinter. They are not parsed as correct input by default.

```
PARSER_ACCEPT_ARRAY_RANGE_EXTENSION FALSE
```

Type Checking

Each argument list at calls to a function or a subroutine is compared to the functional type of the callee. Turn this off if you need to support variable numbers of arguments or if you use overloading and do not want to hear about it. For instance, an IO routine can be used to write an array of integers or an array of reals or an array of complex if the length parameter is appropriate.

Since the functional typing is shaky, let's turn it off by default!

```
PARSER_TYPE_CHECK_CALL_SITES FALSE
```

Loop Header with Label

The PIPS implementation of Allen&Kennedy algorithm cannot cope with labeled DO loops because the loop, and hence its label, may be replicated if the loop is distributed. The parser can generate an extra `CONTINUE` statement to carry the label and produce a label-free loop. This is not the standard option because `PIPS` is designed to output code as close as possible to the user source code.

```
PARSER_SIMPLIFY_LABELLED_LOOPS FALSE
```

Most PIPS analyses work better if do loop bounds are affine. It is sometimes possible to improve results for non-affine bounds by assigning the bound to an integer variables and by using this variable as bound. But this is implemented for Fortran, but not for C.

```
PARSER_LINEARIZE_LOOP_BOUNDS FALSE
```

Entry

The entry construct can be seen as an early attempt at object-oriented programming. The same object can be processed by several function. The object is declared as a standard subroutine or function and entry points are placed in the executable code. The entry points have different sets of formal parameters, they may share some common pieces of code, they share the declared variables, especially the static ones.

The entry mechanism is dangerous because of the flow of control between entries. It is now obsolete and is not analyzed directly by PIPS. Instead each entry may be converted into a first class function or subroutine and static variables are gathered in a specific common. This is the default option. If the substitution is not acceptable, the property may be turned off and entries results in a parser error.

```
PARSER_SUBSTITUTE_ENTRIES TRUE
```

Alternate Return

Alternate returns are put among the obsolete Fortran features by the Fortran 90 standard. It is possible (1) to refuse them (option "NO"), or (2) to ignore them and to replace alternate returns by `STOP` (option "STOP"), or (3) to substitute them by a semantically equivalent code based on return code values (option "RC" or option "HRC"). Option (2) is useful if the alternate returns are used to propagate error conditions. Option (3) is useful to understand the impact of the alternate returns on the control flow graph and to maintain the code semantics. Option "RC" uses an additional parameter while option "HRC" uses a set of PIPS run-time functions to hide the set and get of the return code which make declaration regeneration less useful. By default, the first option is selected and alternate returns are refused.

To produce an executable code, the declarations must be regenerated: see property `PRETTYPRINT_ALL_DECLARATIONS` 10.2.22.6 in Section 10.2.22.6. This is not necessary with option "HRC". Fewer new declarations are needed if

variable PARSER_RETURN_CODE_VARIABLE 4.2.1.4 is implicitly integer because its first letter is in the I-N range.

With option (2), the code can still be executed if alternate returns are used only for errors and if no errors occur. It can also be analyzed to understand what the *normal* behavior is. For instance, OUT regions are more likely to be exact when exceptions and errors are ignored.

Formal and actual label variables are replaced by string variables to preserve the parameter ordre and as much source information as possible. See PARSER_FORMAL_LABEL_SUBSTITUTE_PREFIX 4.2.1.4 which is used to generate new variable names.

```
PARSER_SUBSTITUTE_ALTERNATE_RETURNS "NO"
```

```
PARSER_RETURN_CODE_VARIABLE "I_PIPS_RETURN_CODE_"
```

```
PARSER_FORMAL_LABEL_SUBSTITUTE_PREFIX "FORMAL_RETURN_LABEL_"
```

The internal representation can be hidden and the alternate returns can be prettyprinted at the call sites and modules declaration by turning on the following property:

```
PRETTYPRINT_REGENERATE_ALTERNATE_RETURNS FALSE
```

Using a mixed C / Fortran RI is troublesome for comments handling: sometimes the comment guard is stored in the comment, sometime not. Sometimes it is on purpose, sometimes it is not. When following property is set to true, *PIPS*[4] does its best to prettyprint comments correctly.

```
PRETTYPRINT_CHECK_COMMENTS TRUE
```

If all modules have been processed by PIPS, it is possible not to regenerate alternate returns and to use a code close to the internal representation. If they are regenerated in the call sites and module declaration, they are nevertheless not used by the code generated by PIPS which is consistent with the internal representation.

Here is a possible implementation of the two PIPS run-time subroutines required by the hidden return code ("HRC") option:

```
subroutine SET_I_PIPS_RETURN_CODE_(irc)
common /PIPS_RETURN_CODE_COMMON/irc_shared
irc_shared = irc
end
subroutine GET_I_PIPS_RETURN_CODE_(irc)
common /PIPS_RETURN_CODE_COMMON/irc_shared
irc = irc_shared
end
```

Note that the subroutine names depend on the PARSER_RETURN_CODE_VARIABLE 4.2.1.4 property. They are generated by prefixing it with SET_ and GET_. There implementation is free. The common name used should not conflict with application common names. The ENTRY mechanism is not used because it would be desugared by PIPS anyway.

[4] http://www.cri.ensmp.fr/pips

Assigned GO TO

By default, assigned GO TO and ASSIGN statements are not accepted. These constructs are obsolete and will not be part of future Fortran standards.

However, it is possible to replace them automatically in a way similar to computed GO TO. Each ASSIGN statement is replaced by a standard integer assignment. The label is converted to its numerical value. When an assigned GO TO with its optional list of labels is encountered, it is transformed into a sequence of logical IF statement with appropriate tests and GO TO's. According to Fortran 77 Standard, Section 11.3, Page 11-2, the control variable must be set to one of the labels in the optional list. Hence a STOP statement is generated to interrupt the execution in case this happens, but note that compilers such as SUN f77 and g77 do not check this condition at run-time (it is undecidable statically).

```
PARSER_SUBSTITUTE_ASSIGNED_GOTO FALSE
```

Assigned GO TO without the optional list of labels are not processed. In other words, PIPS make the optional list mandatory for substitution. It usually is quite easy to add manually the list of potential targets.

Also, ASSIGN statements cannot be used to define a FORMAT label. If the desugaring option is selected, an illegal program is produced by PIPS parser.

Statement Function

This property controls the processing of Fortran statement functions by text substitution in the parser. No other processing is available and the parser stops with an error message when a statement function declaration is encountered.

The default used to be not to perform this unchecked replacement, which might change the semantics of the program because type coercion is not enforced and actual parameters are not assigned to intermediate variables. However most statement functions do not require these extra-steps and it is legal to perform the textual substitution. For user convenience, the default option is textual substitution.

Note that the parser does not have enough information to check the validity of the transformation, but a warning is issued if legality is doubtful. If strange results are obtained when executing codes transformed with PIPS, his property should be set to false.

A better method would be to represent them somehow a local functions in the internal representation, but the implications for `pipsmake` and other issues are clearly not all foreseen... (Fabien Coelho).

```
PARSER_EXPAND_STATEMENT_FUNCTIONS TRUE
```

4.2.2 Declaration of HPFC Parser

This parser takes a different Fortran file but applies the same processing as the previous parser. The Fortran file is the result of the preprocessing by the `hpfc_filter` 8.3.2.1 phase of the original file in order to extract the directives and switch them to a Fortran 77 parsable form. As another side-effect, this parser hides some callees from `pipsmake`. This callees are temporary functions

used to encode HPF directives. Their call sites are removed from the code before requesting full analyses to PIPS. This parser is triggered automatically by the `hpfc_close` 8.3.2.5 phase when requested. It should never be selected or activated by hand.

```
hpfc_parser                          > MODULE.parsed_code
                                     > MODULE.callees
        < PROGRAM.entities
        < MODULE.hpfc_filtered_file
```

4.2.3 Declaration of the C Parsers

Three C parsers are used by *PIPS*[5]. The first one, called the C preprocessor parser, is used to break down a C file or a set of C files into multiple files, with one function per file and a global file with all external declarations, the compilation unit. This is performed when *PIPS*[6] is launched and its *workspace* is created.

The second one is called the C parser. It is designed to parse the function files. The last one is called the compilation unit parser and it deals with the compilation unit file.

4.2.3.1 Language parsed by the C Parsers

The C parsers are all based on the same initial set of lexical and syntactic rules designed for C77. They support some C99 extensions such as VLA, declarations in for loops...

The language parsed is larger than the language handled interprocedurally by PIPS:

1. recursion is not supported;
2. pointers to function are not supported;
3. internal functions, a gcc extension, are not supported.

4.2.3.2 Handling of C Code

A C file is seen in PIPS as a compilation unit, that contains all the objects declarations that are global to this file, and as many as module (function or procedure) definitions defined in this file.

Thus the compilation unit contains the file-global macros, the include statements, the local and global variable definitions, the type definitions, and the function declarations if any found in the C file.

When the PIPS workspace is created by PIPS preprocessor, each C file is preprocessed[7] using for instance `gcc -E`[8] and broken into a new which contains

[5] `http://www.cri.ensmp.fr/pips`

[6] `http://www.cri.ensmp.fr/pips`

[7] Macros are interpreted and include files are expanded. The result depends on the C preprocessor used,on its option and on the system environment (`/usr/include`,...).

[8] It can be redefined using `CPP_PIPS` and `PIPS_CPP_FLAGS` environment variables as explained in § 3.2.2.

only the file-global variable declarations, the function declarations and the type definitions, and one C file for each C function defined in the initial C file.

The new compilation units must be parsed before the new files, containing each one exactly one function definition, can be parsed. The new compilation units are named like the initial file names but with a bang extension.

For example, considering a C file `foo.c` with 2 function definitions:

```
enum { N = 2008 };
typedef float data_t;
data_t matrix[N][N];
extern int errno;

int calc(data_t a[N][N]) {
  [...]
}

int main(int argc, char *argv[]) {
  [..]
}
```

After preprocessing, it leads to a file `foo.cpp_processed.c` that is then split into a new `foo!.cpp_processed.c` compilation unit containing

```
enum { N = 2008 };
typedef float data_t;
data_t matrix[N][N];
extern int errno;

int calc(data_t a[N][N]);}

int main(int argc, char *argv[]);
```

and 2 module files containing the definitions of the 2 functions, a `calc.c`

```
int calc(data_t a[N][N]) {
  [...]
}
```

and a `main.c`

```
int main(int argc, char *argv[]) {
  [..]
}
```

Note that it is possible to have an empty compilation unit and no module file if the original file does not contain sensible C informations (such as an empty file containing only blank characters and so on).

4.2.3.3 Compilation Unit Parser

```
compilation_unit_parser         > COMPILATION_UNIT.declarations
        < COMPILATION_UNIT.c_source_file
        < PROGRAM.entities
```

The resource COMPILATION_UNIT.declarations produced by compilation_unit_parser is a special resource used to force the parsing of the new compilation unit before the parsing of its associated functions. It is in fact a hash table containing the file-global C keywords and typedef names defined in each compilation unit.

In fact phase compilation_unit_parser also produces parsed_code and callees resources for the compilation unit. This is done to work around the fact that rule c_parser was invoked on compilation units by later phases, in particular for the computation of initial preconditions, breaking the declarations of function prototypes. These two resources are not declared here because pipsmake gets confused between the different rules to compute parsed code : there is no simple way to distinguish between compilation units and modules at some times and handling them similarly at other times.

4.2.3.4 C Parser

```
c_parser                          > MODULE.parsed_code
                                  > MODULE.callees
        < PROGRAM.entities
        < MODULE.c_source_file
        < MODULE.input_file_name
        < COMPILATION_UNIT.declarations
```

If you want to parse some C code using tpips, it is possible to select the C parser with

```
activate C_PARSER
```

but this is not necessary as the parser is selected according to the source file extension. Some properties useful (have a look at *properties*) to deal with a C program are

```
PRETTYPRINT_C_CODE TRUE (obsolete, replaced by PRETTYPRINT_LANGUAGE ‘‘C’’)
PRETTYPRINT_STATEMENT_NUMBER FALSE
PRETTYPRINT_BLOCK_IF_ONLY TRUE
```

4.2.3.5 C Symbol Table

A prettyprint of the symbol table for a C module can be generated with passes parsed_symbol_table ?? and symbol_table ??.

The EXTENDED_VARIABLE_INFORMATION 4.2.3.5 property can be used to extend the information available for variables. By default the entity name, the offset and the size are printed. Using this property the type and the user name, which may be different from the internal name, are also displayed.

```
EXTENDED_VARIABLE_INFORMATION FALSE
```

4.2.3.6 Properties Used by the C Parsers

The C_PARSER_RETURN_SUBSTITUTION 4.2.3.6 property can be used to handle properly multiple returns within one function. The current default value is false, which preserves best the source aspect but modifies the control flow because the calls to return are assumed to flow in sequence. If the property is set to true, C

return statement are replaced, when necessary, either by a simple goto for void functions, or by an assignment of the returned value to a special variable and a goto. A unique return statement is placed at the syntactic end of the function. For functions with no return statement or with a unique return statement placed at the end of their bodies, this property is useless.

```
C_PARSER_RETURN_SUBSTITUTION FALSE
```

The C99 for-loop with a declaration such as **for**(**int** i = a ;...;...) can be represented in the RI with a naive representation such as:

```
{
  int i = a;
  for (;...;...)
}
```

This is done when the `C_PARSER_GENERATE_NAIVE_C99_FOR_LOOP_DECLARATION` 4.2.3.6 property is set to `TRUE`

```
C_PARSER_GENERATE_NAIVE_C99_FOR_LOOP_DECLARATION FALSE
```

Else, we can generate more or less other representation. For example, with some declaration splitting, we can generate a more representative version:

```
{
  int i;
  for(i = a;...;...)
}
```

if `C_PARSER_GENERATE_COMPACT_C99_FOR_LOOP_DECLARATION` 4.2.3.6 property set to `FALSE`.

```
C_PARSER_GENERATE_COMPACT_C99_FOR_LOOP_DECLARATION FALSE
```

Else, we can generate a more compact (but newer representation that can choke some parts of *PIPS*[9]...) like:

```
  statement with "int i;" declaration
    instruction for(i = a;...;...)
}
```

This representation is not yet implemented.

4.2.4 Fortran 90

The Fortran 90 parser is not integrated in pipsmake. It is activated earlier when the workspace is created.

4.3 Controlized Code (Hierarchical Control Flow Graph)

`PIPS` analyses and transformations take advantage of a hierarchical control flow graph (HCFG), which preserves structured part of code as such, and uses a

[9] `http://www.cri.ensmp.fr/pips`

control flow graph only when no syntactic representation is available (see [33]). The encoding of the relationship between structured and unstructured parts of code is explained elsewhere, mainly in the *PIPS Internal Representation of Fortran and C code*[10].

The `controlizer` 4.3 is the historical controlizer phase that removes `GOTO` statements in the parsed code and generates a similar representation with small CFGs. It was developped for Fortran 77 code.

The Fortran controlizer phase was too hacked and undocumented to be improved and debugged for C99 code so a new version has been developed, documented and is designed to be simpler and easier to understand. But, for comparison, the Fortran controlizer phase can still be used.

```
controlizer                        > MODULE.code
        < PROGRAM.entities
        < MODULE.parsed_code
```

For debugging and validation purpose, by setting *at most* one of the `PIPS_USE_OLD_CONTROLIZER` or `PIPS_USE_NEW_CONTROLIZER` environment variables, you can force the use of the specific version of the controlizer you want to use. This override the setting by activate. Ronan?

Note that the controlizer choice impacts the HCFG when Fortran entries are used. If you do not know what Fortran entries are, it is deprecated stuff anyway... ☺

The `new_controlizer` 4.3 removes `GOTO` statements in the parsed code and generates a similar representation with small CFGs. It is designed to work according to C and C99 standards. Sequences of sequence and variable declarations are handled properly. However, the prettyprinter is tuned for code generated by `controlizer` 4.3, which does not always minimize the number of goto statements regenerated.

The hierarchical control flow graph built by the `controlizer` 4.3 is pretty crude. The partial control flow graphs, called `unstructured` statements, are derived from syntactic constructs. The control scope of an unstructured is the smallest enclosing structured construct, whether a loop, a test or a sequence. Thus some statements, which might be seen as part of structured code, end up as nodes of an unstructured.

Note that sequences of statements are identified as such by `controlizer` 4.3. Each of them appears as a unique node.

Also, useless CONTINUE statements may be added as provisional landing pads and not removed. The exit node should never have successors but this may happen after some PIPS function calls. The exit node, as well as several other nodes, also may be unreachable. After clean up, there should be no unreachable node or the only unreachable node should be the exit node. Function `unspaghettify` 9.3.4.1 (see Section 9.3.4.1) is applied by default to clean up and to reduce the control flow graphs after `controlizer` 4.3.

The `GOTO` statements are transformed in arcs but also in `CONTINUE` statements to preserve as many user comments as possible.

The top statement of a module returned by the `controlizer` 4.3 used to contain always an unstructured instruction with only one node. Several phases

[10] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

in PIPS assumed that this always is the case, although other program transformations may well return any kind of top statement, most likely a `block`. This is no longer true. The top statement of a module may contain any kind of instruction.

Here is declared the C and C99 controlizer:

```
new_controlizer                         > MODULE.code
        < PROGRAM.entities
        < MODULE.parsed_code
```

Control restructuring eliminates empty sequences but as empty true or false branch of structured `IF`. This semantic property of *PIPS Internal Representation of Fortran and C code*[11] is enforced by libraries `effects`, `regions`, `hpfc`, `effects-generic`.

```
WARN_ABOUT_EMPTY_SEQUENCES FALSE
```

By unsetting this property `unspaghettify` 9.3.4.1 is not applied implicitly in the controlizer phase.

```
UNSPAGHETTIFY_IN_CONTROLIZER TRUE
```

The next property is used to convert C for loops into C while loops. The purpose is to speed up the re-use of Fortran analyses and transformation for C code. This property is set to false by default and should ultimately disappear. But for new user convenience, it is set to TRUE by `activate_language()` when the language is C.

```
FOR_TO_WHILE_LOOP_IN_CONTROLIZER FALSE
```

The next property is used to convert C for loops into C do loops when syntactically possible. The conversion is not safe because the effect of the loop body on the loop index is not checked. The purpose is to speed up the re-use of Fortran analyses and transformation for C code. This property is set to false by default and should disappear soon. But for new user convenience, it is set to TRUE by `activate_language()` when the language is C.

```
FOR_TO_DO_LOOP_IN_CONTROLIZER FALSE
```

This can also explicitly applied by calling the phase described in § 9.3.4.4.

FORMAT Restructuring

To able deeper code transformation, FORMATs can be gathered at the very beginning of the code or at the very end according to the following options in the unspaghettify or control restructuring phase.

```
GATHER_FORMATS_AT_BEGINNING FALSE
```

```
GATHER_FORMATS_AT_END FALSE
```

[11] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

4.3.1 Properties for Clean Up Sequences

To display the statistics about cleaning-up sequences and removing useless CONTINUE or empty statement.

```
CLEAN_UP_SEQUENCES_DISPLAY_STATISTICS FALSE
```

There is a trade-off between keeping the comments associated to labels and goto and the cleaning that can be do on the control graph.

By default, do not fuse empty control nodes that have labels or comments:

```
FUSE_CONTROL_NODES_WITH_COMMENTS_OR_LABEL FALSE
```

By default, do not fuse sequences with internal declarations. Turning this to `TRUE` results in variable renamings when the same variable name is used at several places in the analyzed module.

```
CLEAN_UP_SEQUENCES_WITH_DECLARATIONS FALSE
```

4.3.2 Symbol Table Related to a Module Code

The prettyprint of the symbol table for a Fortran or C module is generated with:

```
symbol_table        > MODULE.symbol_table_file
    < PROGRAM.entities
    < MODULE.code
```

4.4 Parallel Code

The internal representation includes special field to declare parallel constructs such as parallel loops. A parallel code internal representation does not differ fundamentally from a sequential code.

Chapter 5

Pedagogical Phases

Although this phases should be spread elsewhere in this manual, we have put some pedagogical phases useful to jump into PIPS first.

5.1 Using XML backend

A phase that displays, in debug mode, statements matching an XPath expression on the internal representation:

```
alias simple_xpath_test 'Output debug information about XPath matching'

simple_xpath_test > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

5.2 Operating of gen_multi_recurse

A phase that displays the explore path of function gen_multi_recurse.

```
alias gen_multi_recurse_explorer 'Output debug information about XPath matching'

gen_multi_recurse_explorer > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

5.3 Prepending a comment

Prepends a comment to the first statement of a module. Useful to apply post-processing after PIPS.

```
alias prepend_comment 'Prepend a comment to the first statement of a module'

prepend_comment > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

The comment to add is selected by this property:

```
PREPEND_COMMENT "/* This comment is added by PREPEND_COMMENT phase */"
```

5.4 Prepending a call

This phase inserts a call to function `MY_TRACK` just before the first statement of a module. Useful as a pedagogical example to explore the internal representation and Newgen. Not to be used for any pratical purpose as it is bugged. Debugging it is a pedagogical exercise.

```
alias prepend_call 'Insert a call to MY_TRACK just before the first statement of a module'

prepend_call > MODULE.code
             > MODULE.callees
  < PROGRAM.entities
  < MODULE.code
```

The called function could be defined by this property:

```
PREPEND_CALL "MY_TRACK"
```

but it is not.

5.5 Add a pragma to a module

This phase prepend or appends a pragma to a module.

```
alias add_pragma 'Prepends or append a pragma to the code of a module'

add_pragma > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

The pragma name can be defined by this property:

```
PRAGMA_NAME "MY_PRAGMA"
```

The pragma can be append or prepend thanks to this property:

```
PRAGMA_PREPEND TRUE
```

The pass `clear_pragma` 5.5 clears all pragma, this should be done on any input with unhandled pragma, we don't what semantic we might break.

```
clear_pragma > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

The pass `pragma_outliner` 5.5 is used for outlining a sequence of statements contained between two given sentinel pragmas using properties `PRAGMA_OUTLINER_BEGIN` 5.5 and `PRAGMA_OUTLINER_END` 5.5. The name of the new function is controlled using `PRAGMA_OUTLINER_PREFIX` 5.5.

```
pragma_outliner      >     MODULE.code
    > MODULE.callees
    < PROGRAM.entities
    < MODULE.code
    < MODULE.cumulated_effects
    < MODULE.regions
    < CALLEES.summary_regions
    < MODULE.summary_regions
    < MODULE.transformers
    < MODULE.preconditions
```

```
PRAGMA_OUTLINER_BEGIN "begin"
```

```
PRAGMA_OUTLINER_END "end"
```

```
PRAGMA_OUTLINER_PREFIX "pips_outlined"
```

Remove labels that are not usefull

```
remove_useless_label > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

Loop labels can be kept thanks to this property:

```
REMOVE_USELESS_LABEL_KEEP_LOOP_LABEL FALSE
```

Chapter 6

Static Analyses

Analyses encompass the computations of call graphs, the memory effects, reductions, use-def chains, dependence graphs, interprocedural checks (*flinter*), semantics information (transformers and preconditions), continuations, complexities, convex array regions, dynamic aliases and complementary regions.

6.1 Call Graph

All lists of callees are needed to build the global lists of callers for each module. The callers and callees lists are used by `pipsmake` to control top-down and bottom-up analyses. The call graph is assumed to be a DAG, i.e. no recursive cycle exists, but it is not necessarily connected.

The height of a module can be used to schedule bottom-up analyses. It is zero if the module has no callees. Else, it is the maximal height of the callees plus one.

The depth of a module can be used to schedule top-down analyses. It is zero if the module has no callers. Else, it it the maximal depth of the callers plus one.

```
callgraph                       > ALL.callers
                                > ALL.height
                                > ALL.depth
        < ALL.callees
```

The following pass generates a *uDrawGraph*[1] version of the callgraph. Its quite partial since it should rely on an hypothetical *all callees*, direct and indirect, resource.

```
alias dvcg_file 'Graphical Call Graph'
alias graph_of_calls 'For current module'
alias full_graph_of_calls 'For all modules'

graph_of_calls              > MODULE.dvcg_file
        < ALL.callees
```

[1] `http://www.informatik.uni-bremen.de/uDrawGraph`

```
full_graph_of_calls          > PROGRAM.dvcg_file
        < ALL.callees
```

6.2 Memory Effects

The data structures used to represent memory effects and their computation are described in [34]. Another description is available on line, in *PIPS Internal Representation of Fortran and C code*[2] Technical Report.

Note that the standard name in the Dragon book is likely to be *Gen* and *Kill* sets in the standard data flow analysis framework, but PIPS uses the more general concept of *effect* developped by P. Jouvelot and D. Gifford [38] and its analyses are mostly based on the abstract syntac tree (AST) rather than the control flow graph (CFG).

6.2.1 Proper Memory Effects

The proper memory effects of a statement basically are a list of variables that may be read (used) or written (defined) by the statement. They are used to build use-def chains (see [1] or a later edition) and then the dependence graph.

Proper means that the effects of a compound statement do not include the effects of lower level statements. For instance, the body of a loop, true and false branches of a test statement, control nodes in an unstructured statement ... are ignored to compute the proper effects of a loop, a test or an unstructured.

Two families of effects are computed : `pointer_effects` are effects in which intermediary access paths may refer to different memory locations at different program points; regular `effects` are constant path effects, which means that their intermediary access paths all refer to unique memory locations. The same distinction holds for convex array regions (see section 6.12).

`proper_effects_with_points_to` and `proper_effects_with_pointer_values` are alternatives to compute constant path proper effects using points-to (see subsection 6.14.5) or pointer values analyses (see subsection 6.14.7). This is still at an experimental stage.

Summary effects (see Section 6.2.4) of a called module are used to compute the proper effects at the corresponding call sites. They are translated from the callee's scope into the caller's scope. The translation is based on the actual-to-formal binding. If too many actual arguments are defined, a user warning is issued but the processing goes on because a simple semantics is available: ignore useless actual arguments. If too few actual arguments are provided, a user error is issued because the effects of the call are not defined.

Variables private to loops are handled like regular variable.

See `proper_effects` 6.2.1

See `proper_effects` 6.2.1

```
proper_pointer_effects                  > MODULE.proper_pointer_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_pointer_effects
```

[2] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

```
proper_effects                  > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects
```

When pointers are used, points-to information is useful to obtain precise proper memory effects.

Because points-to analysis is able to detect some cases of segfaults, variables that are not defined/written can nevertheless have a different abstract value at the beginning and at the end of a piece of code.

This leads to difficulties with memory effects. Firstly, if a piece of code is not reachable because a segfault always occurs before it is executed, it has no memory effects (as usual, the PIPS output is pretty surprising...). Secondly, cumulated memory effects can either be any effect that is linked to any execution of a piece of code or any effect that happens when the executions reach the end of the piece of code.

So EffectsWithPointsTo contains weird results either because the code always segfault somewhere or because the code might segfault because a function argument is not checked before it is used. Hence, effects disappear or must effects become may effects.

```
proper_effects_with_points_to > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.points_to
        < CALLEES.summary_effects

proper_effects_with_pointer_values > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.simple_pointer_values
        < CALLEES.summary_effects
```

6.2.2 Filtered Proper Memory Effects

To be continued...by whom?

This phase collects information about where a given global variable is actually modified in the program.

```
filter_proper_effects         > MODULE.filtered_proper_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < CALLEES.summary_effects
```

6.2.3 Cumulated Memory Effects

Cumulated effects of statements are lists of read or written variables, just like the proper effects (see Section 6.2.1).

Cumulated means that the effects of a compound statement, do loop, test or unstructured, include the effects of the lower level statements such as a loop body or a test branch.

For return, exit and abort statements (only for the main function or what is consider as the main function), cumulated effects will also add the read on LUNS (Logical Units) that are present for the function. The goal of adding read LUNS for these statements is to improve OUT Effects (and Regions) especially for the last statements that make a write on LUNS.

```
cumulated_effects            > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

TODO: inline documentation

```
cumulated_effects_with_points_to        > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

TODO: inline documentation

```
cumulated_effects_with_pointer_values   > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

TODO: inline documentation

```
cumulated_pointer_effects   > MODULE.cumulated_pointer_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_pointer_effects
```

TODO: inline documentation

```
cumulated_pointer_effects_with_points_to > MODULE.cumulated_pointer_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_pointer_effects
        < MODULE.points_to
```

TODO: inline documentation

```
cumulated_pointer_effects_with_pointer_values > MODULE.cumulated_pointer_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_pointer_effects
        < MODULE.simple_pointer_values
```

6.2.4 Summary Data Flow Information (SDFI)

Summary data flow information is the simplest interprocedural information needed to take procedure into account in a parallelizer. It was introduced in Parafrase (see [40]) under this name, but should be called summary memory effects in PIPS context.

The `summary_effects` 6.2.4 of a module are the cumulated memory effects of its top level statement (see Section 6.2.3), but effects on local dynamic variables are ignored (because they cannot be observed by the callers[3]) and subscript expressions of remaining effects are eliminated.

```
summary_pointer_effects                  > MODULE.summary_pointer_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_pointer_effects

summary_effects                 > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

6.2.5 IN and OUT Effects

IN and OUT memory effects of a statement s are memory locations whose input values are used by statement s or whose output values are used by statement s continuation. Variables allocated in the statement are not part of the IN or OUT effects. Variables defined before they are used ar not part of the IN effects. OUT effects require an interprocedural analysis[4]

```
in_effects > MODULE.in_effects
           > MODULE.cumulated_in_effects
         < PROGRAM.entities
         < MODULE.code
         < MODULE.cumulated_effects
         < CALLEES.in_summary_effects

in_summary_effects > MODULE.in_summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_effects

out_summary_effects > MODULE.out_summary_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.out_effects

out_effects > MODULE.out_effects
        < PROGRAM.entities
        < MODULE.code
```

[3] Unless it accesses illegally the stack: see Tom Reps, http://pages.cs.wisc.edu/ reps

[4] They are not validated as of June 21, 2008 (FI).

```
        < MODULE.out_summary_effects
        < MODULE.cumulated_in_effects
```

6.2.6 Proper and Cumulated References

The concept of *proper references* is not yet clearly defined. The original idea is to keep track of the actual objects of Newgen domain `reference` used in the program representation of the current statement, while retaining if they correspond to a read or a write of the corresponding memory locations. Proper references are represented as effects.

For C programs, where memory accesses are not necessarily represented by objects of Newgen domain `reference`, the semantics of this analysis is unclear.

Cumulated references gather proper references over the program code, without taking into account the modification of memory stores by the program execution.

FC: I should implement real summary references?

```
proper_references       > MODULE.proper_references
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_references    > MODULE.cumulated_references
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_references
```

6.2.7 Effect Properties

Effects are a first or second level analysis. They are analyzed using only some information about pointers, either none, or points-to or pointer values. They are used by many passes such as dependence graph analysis (Rice Pass), semantics analysis (Transformers passes), convex array regions analysis...

It is often tempting, useful or necessary to ignore some effects. It is always safer to ignore effects when they are used by a pass to avoid possible inconsistencies due to other passes using effects. However, it may be necessary to ignore some effects in an effect pass because effects are merged and cannot be unmerged by a later pass. Of course, this is not a standard setting for PIPS and the semantics of the resulting codes or later analyses is unknown in general, but to the person who makes the decision for a subset of input codes or for experimental reasons.

6.2.7.1 Effects Filtering

Filter this variable in phase `filter_proper_effects` 6.2.2.

```
EFFECTS_FILTER_ON_VARIABLE ""
```

Property `USER_EFFECTS_ON_STD_FILES` 6.2.7.1 is used to control the way the user uses `stdout`, `stdin` and `stderr`. The default case (`FALSE`) means that the user does not modify these global variables. When set to `TRUE`, they

are considered as user variables, and dereferencing them through calls to stdio functions leads to less precise effects.

```
USER_EFFECTS_ON_STD_FILES FALSE
```

6.2.7.2 Checking Pointer Updates

When set to TRUE, EFFECTS_POINTER_MODIFICATION_CHECKING 6.2.7.2 enables pointer modification checking during the computation of cumulated effects and/or RW convex array regions. Since this is still at experimentation level, it's default value is FALSE. This property should disappear when pointer modification analyses are more mature.

```
EFFECTS_POINTER_MODIFICATION_CHECKING FALSE
```

6.2.7.3 Dereferencing Effects

The default (and correct) behaviour for the computation of effects is to transform dereferencing paths into constant paths using the information available, either none or points-to or pointer values, and abstract locations used to represent sets of locations.

When property CONSTANT_PATH_EFFECTS 6.2.7.3 is set to FALSE, the latter transformation is skipped. Effects are then equivalent to pointer_effects. This property is available for backward compatibility and experimental purpose. It must be born in mind that analyses and transformations using the resulting effects may yield uncorrect results. This property also affects the computation of convex array regions.

```
CONSTANT_PATH_EFFECTS TRUE
```

Since CONSTANT_PATH_EFFECTS 6.2.7.3 may be set to FALSE erroneously, some tests are included in conflicts testing to avoid generating wrong code. However, these tests are costly, and can be turned off by setting TRUST_CONSTANT_PATH_EFFECTS_IN_CONFLICTS 6. to FALSE. This must be used with care and only when there is no aliasing.

```
  TRUST_CONSTANT_PATH_EFFECTS_IN_CONFLICTS FALSE
```

EFFECTS_IGNORE_DEREFERENCING 6.2.7.3 is set to FALSE. This must be used with extreme care and only when pointer operations are known no to matter with the analysis performed because only a subset of input codes is used. Constant path effects are obtained by filtering constant path effects and by dropping effects due to a pointer-related address computation as in `*p=3;` or `p->i=3;`.

```
EFFECTS_IGNORE_DEREFERENCING FALSE
```

6.2.7.4 Effects of References to a Variable Length Array (VLA)

Property VLA_EFFECT_READ 6.2.7.4 makes a read effect on variables dimension of an variable-length array (vla) at each use of it. For instance, with an array declared as `a[size]`, at each occurence of `a`, like `a[i]`, a `READ effect` will be made for `size`. Normally, no reason to set it to FALSE. For parallelization purpose, maybe want to set it to false?

```
VLA_EFFECT_READ TRUE
```

6.2.7.5 Memory Effects vs Environment Effects

Property `MEMORY_EFFECTS_ONLY` 6.2.7.5 is used to restrict the action kind of an effect action to `store`. In other words, variable declarations and type declarations are not considered to alter the execution state when this property is set to TRUE. This is fine for Fortran code because variables cannot be declared among executable statements and because new type cannot be declared. But this leads to wrong result for C code when loop distribution or use-def elimination is performed.

Currently, PIPS does not have the capability to store default values depending on the source code language. The default value is TRUE to avoid disturbing too many phases of PIPS at the same time while environment and type declaration effects are introduced.

```
MEMORY_EFFECTS_ONLY TRUE
```

6.2.7.6 Time Effects

Some programs do measure execution times. All code placed between measurement points must not be moved out, as can happen when loops are distributed or, more generally, instructions are rescheduled. Since loops using time effects are not parallel, a clock variable is always updated when a time-related function is called. This is sufficient to avoid most problems, but not all of them because time effects of all other executed statements are kept implicit, i.e. the real time clock is not updated: and loops can still be distributed. If time measurements are key, this property must be turned on. By default, it is turned off.

```
TIME_EFFECTS_USED FALSE
```

6.2.7.7 Effects of Unknown Functions

Some source code is sometimes missing. *PIPS*[5] does not have any way to guess the memory effects of functions whose source code is missing. Several approaches are possible to approximate the exact effects. Two optimistic ones are implemented: either we assume that the function only computes a result and has no side effects thru pointer parameters, global variables or static variables (default option), or we assume the maximal possible effects through pointers (this should be clarified: for all pointers `p`, `*p` is written) but not thru static or global variables.

```
MAXIMAL_PARAMETER_EFFECTS_FOR_UNKNOWN_FUNCTIONS FALSE
```

For safety, a pessimitic option is be implemented and a maximal memory effect, `*ANYMODULE*:*ANYWHERE*`, is associated to such unknown functions.

```
MAXIMAL_EFFECTS_FOR_UNKNOWN_FUNCTIONS FALSE
```

These two properties should not be true simultaneously.

[5] `http://www.cri.ensmp.fr/pips`

6.2.7.8 Other Properties Impacting EFfects

Property `ALIASING_ACROSS_TYPES` 6.14.8.1 has an impact on effect computation. When the locations used or defined by a memory effect are unknown, an abstract location is used to represent either all possible locations of the program (TRUE) or all locations of a certain type (FALSE).

6.3 Live Memory Access Paths

6.4 Reductions

The proper reductions are computed from a code.

```
proper_reductions > MODULE.proper_reductions
  < PROGRAM.entities
  < MODULE.code
  < MODULE.proper_references
  < CALLEES.summary_effects
  < CALLEES.summary_reductions
```

The cumulated reductions propagate the reductions in the code, upwards.

```
cumulated_reductions > MODULE.cumulated_reductions
  < PROGRAM.entities
  < MODULE.code
  < MODULE.proper_references
  < MODULE.cumulated_effects
  < MODULE.proper_reductions
```

This pass summarizes the reductions candidates found in a module for export to its callers. The summary effects should be used to restrict attention to variable of interest in the translation?

```
summary_reductions > MODULE.summary_reductions
  < PROGRAM.entities
  < MODULE.code
  < MODULE.cumulated_reductions
  < MODULE.summary_effects
```

Some possible (simple) transformations could be added to the code to mark reductions in loops, for latter use in the parallelization.

The following is NOT implemented. Anyway, should the cumulated_reductions be simply used by the prettyprinter instead?

```
loop_reductions > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.cumulated_reductions
```

6.4.1 Reduction Propagation

tries to transform

```
{
 a = b + c;
 r = r + a;
}
```

into

```
{
 r = r +b  ;
 r = r +c  ;
}
```

```
reduction_propagation > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.proper_reductions
  < MODULE.dg
```

6.4.2 Reduction Detection

tries to transform

```
{
 a = b + c;
 b = d + a;
}
```

which hides a reduction on b into

```
{
 b = b + c  ;
 b = d + b  ;
}
```

when possible

```
reduction_detection > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.dg
```

6.5 Chains (Use-Def Chains)

Use-def and def-use chains are a standard data structure in optimizing compilers [1]. These chains are used as a first approximation of the dependence graph. Chains based on convex array regions (see Section 6.12) are more effective for interprocedural parallelization.

If chains based on convex array regions have been selected, the simplest dependence test must be used because regions carry more information than any kind of preconditions. Preconditions and loop bound information already are included in the region predicate.

6.5.1 Menu for Use-Def Chains

```
alias chains 'Use-Def Chains'

alias atomic_chains 'Standard'
alias region_chains 'Regions'
alias in_out_regions_chains 'In-Out Regions'
```

6.5.2 Standard Use-Def Chains (a.k.a. Atomic Chains)

The algorithm used to compute use-def chains is original because it is based on PIPS hierarchical control flow graph and not on a unique control flow graph.

This algorithm generates inexistent dependencies on loop indices. These dependence arcs appear between DO loop headers and implicit DO loops in IO statements, or between one DO loop header and unrelated DO loop bound expressions using that index variable. It is easy to spot the problem because loop indices are not privatized. A prettyprint option,

PRETTYPRINT_ALL_PRIVATE_VARIABLES 10.2.22.5.1

must be set to true to see if the loop index is privatized or not. The problem disappears when some loop indices are renamed.

The problem is due to the internal representation of DO loops: PIPS has no way to distinguish between initialization effects and increment effects. They have to be merged as proper loop effects. To reduce the problem, proper effects of DO loops do not include the index read effect due to the loop incrementation.

Artificial arcs are added to... (Pierre Jouvelot, help!).

```
atomic_chains                    > MODULE.chains
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

6.5.3 READ/WRITE Region-Based Chains

Such chains are required for effective interprocedural parallelization. The dependence graph is annotated with *proper regions*, to avoid inaccuracy due to summarization at simple statement level (see Section 6.12).

Region-based chains are only compatible with the Rice Fast Dependence Graph option (see Section 6.6.1) which has been extended to deal with them[6]. Other dependence tests do not use region descriptors (their convex system), because they cannot improve the Rice Fast Dependence test based on regions.

Regions chains are built using *proper regions* which are particular READ and WRITE regions. For simple statements (assignments, calls to intrinsic functions), summarization is avoided to preserve accuracy. At this inner level of the program control flow graph, the extra amount of memory necessary to store regions without computing their convex hull should not be too high compared to the expected gain for dependence analysis. For tests and loops, proper regions contain the regions associated to the condition or the range. And for external calls,

[6]When using regions, the *fast* qualifier does not stand anymore, because the dependence test involves dealing with convex systems that contain much more constraints than when using the sole array indices.

proper regions are the *summary regions* of the callee translated into the caller's name space, to which are merely appended the regions of the expressions passed as argument (no summarization for this step).

```
region_chains                    > MODULE.chains
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_regions
```

6.5.4 IN/OUT Region-Based Chains

Beware : this option is for experimental use only; resulting parallel code may not be equivalent to input code (see the explanations below).

When `in_out_regions_chains` 6.5.4 is selected, IN and OUT regions (see Sections 6.12.5 and 6.12.8) are used at call sites instead of READ and WRITE regions. For all other statements, usual READ and WRITE regions are used.

As a consequence, arrays and scalars which could be declared as local in callees, but are exposed to callers because they are statically allocated or are formal parameters, are ignored, increasing the opportunities to detect parallel loops. But as the program transformation which consists in *privatizing* variables in modules is not yet implemented in `PIPS`, the code resulting from the parallelization with `in_out_regions_chains` 6.5.4 may not be equivalent to the original sequential code. The privatization here is non-standard: for instance, variables declared in commons or static should be stack allocated to avoid conflicts.

As for region-based chains (see Section 6.5.3), the simplest dependence test should be selected for best results.

```
in_out_regions_chains            > MODULE.chains
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_regions
        < MODULE.in_regions
        < MODULE.out_regions
```

The following loop in Subroutine `inout` cannot be parallelized legally because Subroutine `foo` uses a static variable, `y`. However, `PIPS` will display this loop as (potentially) parallel if the `in_out` option is selected for use-def chain computation. Remember that IN/OUT regions require MUST regions to obtain interesting results (see Section 6.12.5).

```
      subroutine inout(a,n)
      real a(n)

      do i = 1, n
         call foo(a(i))
      enddo

      end

      subroutine foo(x)
      save y
```

```
y = x
x = x + y

end
```

6.5.5 Chain Properties

6.5.5.1 Add use-use Chains

It is possible to put use-use dependence arcs in the dependence graph. This is useful for estimation of cache memory traffic and of communication for distributed memory machine (e.g. you can parallelize only communication free loops). Beware of use-use dependence on scalar variables. You might expect scalars to be broadcasted and/or replicated on each processor but they are not handled that way by the parallelization process unless you manage to have them declared private with respect to all enclosing loops.

This feature is not supported by PIPS user interfaces. Results may be hard to interpret. It is useful to print the dependence graph.

```
KEEP_READ_READ_DEPENDENCE FALSE
```

6.5.5.2 Remove Some Chains

It is possible to mask effects on local variables in loop bodies. This is dangerous with current version of Allen & Kennedy which assumes that all the edges are present, the ones on private variables being partially ignored but for loop distribution. In other words, this property should always be set to `false`.

```
CHAINS_MASK_EFFECTS FALSE
```

It also is possible to keep only true data-flow (Def – Use) dependences in the dependence graph. This was an attempt at mimicking the effect of direct dependence analysis and at avoiding privatization. However, *direct* dependence analysis is not implemented in the standard tests and spurious def-use dependence arcs are taken into account.

```
CHAINS_DATAFLOW_DEPENDENCE_ONLY FALSE
```

These last two properties are not consistent with PIPS current development (1995/96). It is assumed that *all* dependence arcs are present in the dependence graph. Phases using the latter should be able to filter out irrelevant arcs, e.g. pertaining to privatized variables.

6.6 Dependence Graph (DG)

The dependence graph is used primarily by the parallelization algorithms. A dependence graph is a refinement of use-def chains (Section 6.5). It is *location*-based and not *value*-based.

There are several ways to compute a dependence graph. Some of them are fast (Banerjee's one for instance) but provide poor results, others might be slower (Rémi Triolet's one for instance) but produce better results.

Three different dependence tests are available, all based on Fourier-Motzkin elimination improved with a heuristics for the integer domain. The `fast` version uses subscript expressions only (unless convex array regions were used to compute use-def chains, in which case regions are used instead). The `full` version uses subscript expressions and loop bounds. The `semantics` version uses subscript expressions and preconditions (see 6.9).

Note that, for interprocedural parallelization, precise array regions only are used by the fast dependence test if the proper kind of use-def chains has been previously selected (see Section 6.5.3).

There are several kinds of dependence graphs. Most of them share the same overall data structure: a graph with labels on arcs and vertices. usually, the main differences are in the labels that decorate arcs; for instance, Kennedy's algorithm requires dependence levels (which loop actually creates the dependence) while algorithms originated from CSRD prefer DDVs (relations between loop indices when the dependence occurs). Dependence cones introduced in [26, 35, 36, 37] are even more precise [56].

The computations of dependence level and dependence cone [55] are both implemented in `PIPS`. DDV's are not computed. Currently, only dependence levels are exploited by parallelization algorithms.

The dependence graph can be printed with or without filters (see Section 10.8). The standard dependence graph includes all arcs taken into account by the parallelization process (Allen & Kennedy [2]), except those that are due to scalar private variables and that impact the distribution process only. The loop carried dependence graph does not include intra-iteration dependences and is a good basis for iteration scheduling. The whole graph includes all arcs, but input dependence arcs.

It is possible to gather some statistics about dependences by turning on property `RICEDG_PROVIDE_STATISTICS` 6.6.6.2 (more details in the *properties*). A Shell script from `PIPS` utilities, `print-dg-statistics`, can be used in combination to extract the most relevant information for a whole program.

During the parallelization phases, is is possible to ignore arcs related to states of the libc, such as the heap memory management, because thread-safe libraries do perform the updates within critical sections. But these arcs are part of the use-def chains and of the dependence graph. If they were removed instead of being ignored, use-def elimination would remove all `free` statements.

The main contributors for the design and development of dependence analysis are Rémi Triolet, François Irigoin and Yi-qing Yang [55]. The code was improved by Corinne Ancourt and Béatrice Creusillet.

6.6.1 Menu for Dependence Tests

```
alias dg 'Dependence Test'

alias rice_fast_dependence_graph 'Preconditions Ignored'
alias rice_full_dependence_graph 'Loop Bounds Used'
alias rice_semantics_dependence_graph 'Preconditions Used'
alias rice_regions_dependence_graph 'Regions Used'
```

6.6.2 Fast Dependence Test

Use subscript expressions only, unless convex array regions were used to compute use-def chains, in which case regions are used instead. `rice_regions_dependence_graph` is a synonym for this rule, but emits a warning if `region_chains` is not selected.

```
rice_fast_dependence_graph      > MODULE.dg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
        < MODULE.cumulated_effects
```

6.6.3 Full Dependence Test

Use subscript expressions and loop bounds.

```
rice_full_dependence_graph      > MODULE.dg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
        < MODULE.cumulated_effects
```

6.6.4 Semantics Dependence Test

Uses subscript expressions and preconditions (see 6.9).

```
rice_semantics_dependence_graph > MODULE.dg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

6.6.5 Dependence Test with Convex Array Regions

Synonym for `rice_fast_dependence_graph`, except that it emits a warning when `region_chains` is not selected.

```
rice_regions_dependence_graph      > MODULE.dg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
        < MODULE.cumulated_effects
```

6.6.6 Dependence Properties (Ricedg)

6.6.6.1 Dependence Test Selection

This property seems to be now obsolete. The dependence test choice is now controlled directly and only by rules in `pipsmake`. The procedures called by these rules may use this property. Anyway, it is useless to set it manually.

```
DEPENDENCE_TEST "full"
```

6.6.6.2 Statistics

Provide the following counts during the dependence test. There are three parts: numbers of dependencies and independences (fields 1-10), dimensions of referenced arrays and dependence natures (fields 11-25) and the same information for constant dependencies (fields 26-40), decomposition of the dependence test in elementary steps (fields 41-49), use and complexity of Fourier-Motzkin's pairwise elimination (fields 50, 51 and 52-68).

1 array reference pairs, i.e. number of tests effected (used to be the number of use-def, def-use and def-def pairs on arrays);

2 number of independences found (on array reference pairs);

Note: field 1 minus field 2 is the number of array dependencies.

3 numbers of loop independent dependences between references in the same statement (not useful for program transformation and parallelization if statements are preserved); it should be subtracted from field 2 to compare results with other parallelizers;

4 numbers of constant dependences;

5 numbers of exact dependences;

Note: field 5 must be greater or equal to field 4.

6 numbers of inexact dependences involved only by the elimination of equation;

7 numbers of inexact dependences involved only by the F-M elimination;

8 numbers of inexact dependences involved by both elimination of equation and F-M elimination;

Note: the sum of fields 5 to 8 and field 2 equals field 1

9 number of dependences among scalar variables;

10 numbers of dependences among loop index variables;

11-40 dependence types detail table with the dimensions [5][3] and constant dependence detail table with the dimensions [5][3]; the first index is the array dimension (from 0 to 4 - no larger arrays has ever been found); the second index is the dependence nature (1: d-u, 2: u-d, 3: d-d); both arrays are flatten according to C rule as 5 sequences of 3 natures;

Note: the sum of fields 11 to 25 should be equal to the sum of field 9 and 2 minus field 1.

Note: the fields 26 to 40 must be less than or equal to the corresponding fields 11 to 25

41 numbers of independences found by the test of constant;

42 numbers of independences found by the GCD test;

43 numbers of independences found by the normalize test;

44 numbers of independences found by the lexico-positive test for constant Di variables;

45 numbers of independences found during the projection on Di variables by the elimination of equation;

46 numbers of independences found during the projection on Di variables by the Fourier-Motzkin's elimination;

47 numbers of independences found during the test of faisability of Di subsystem by the elimination of equation;

48 numbers of independences found during the test of faisability of Di sous-system by the Fourier-Motzkin's elimination;

49 numbers of independences found by the test of lexico-positive for Di subsystem;

Note: the sum of fields 41 to 49 equals field 2

50 total number of Fourier-Motzkin's pair-wise eliminations used;

51 number of Fourier-Motzkin's pair-wise elimination in which the system size doesn't augment after the elimination;

52-68 complexity counter table of dimension [17]. The complexity of one projection by F-M is the product of the number of positive inequalities and the number of negatives inequalities that contain the eliminated variable. This is an histogram of the products. Products which are less than or equal to 4 imply that the total number of inequalities does not increase. So if no larger product exists, field 50 and 51 must be equal.

The results are stored in the current workspace in MODULE.resulttestfast, MODULE.resultesttestfull, or MODULE.resulttestseman according to the test selected.

```
RICEDG_PROVIDE_STATISTICS FALSE
```

Provide the statistics above and count all array reference pairs including these involved in call statement.

```
RICEDG_STATISTICS_ALL_ARRAYS FALSE
```

6.6.6.3 Algorithmic Dependences

This property can be set to only take into account true flow dependences (Def – Use) during the computation of SCC by the Allen&Kennendy algorithm.

Note that this is different from the CHAINS_DATAFLOW_DEPENDENCE_ONLY property, which is set to compute a partial data dependence graph.

Warning: if set, this property may potentially yields incorrect parallel code because dynamic single assignment is not guaranteed.

```
RICE_DATAFLOW_DEPENDENCE_ONLY FALSE
```

6.6.6.4 Optimization

The default option is to compute the dependence graph only for loops which can be parallelized using Allen & Kennedy algorithm. However it is possible to compute the dependences in all cases, even for loop containing test, goto, etc... by setting this option to TRUE.

Of course, this information is not used by the parallelization phase which is restricted to loops meeting the A&K conditions. By the way, the hierarchical control flow graph is not exploited either by the parallelization phase.

```
COMPUTE_ALL_DEPENDENCES FALSE
```

6.7 Flinter

Function `flinter` 6.7 performs some intra and interprocedural checks about formal/actual argument pairs, use of COMMONs,... It was developed by Laurent Aniort and Fabien Coelho. Ronan Keryell added the uninitialized variable checking.

```
alias flinted_file 'Flint View'
flinter                          > MODULE.flinted_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.code
        < MODULE.proper_effects
        < MODULE.chains
```

In the past, `flinter` 6.7 used to require `MODULE.summary_effects` to check the parameter passing modes and to make sure that no module would attempt an assignment on an expression. However, this kind of bug is detected by the effect analysis... which was required by flinter.

Resource `CALLEES.code` is not explicitly required but it produces the global symbols which function `flinter` 6.7 needs to check parameter lists.

6.8 Loop Statistics

Computes statistics about loops in module. It computes the number of perfectly and imperfectly nested loops and gives their depths. And it gives the number of nested loops which we can treat with our algorithm.

```
loop_statistics > MODULE.stats_file
        < PROGRAM.entities
        < MODULE.code
```

Note: it does not seem to behave like a standard analysis, associating information to the internal representation. Instead, an ASCII file seems to be created.

6.9 Semantics Analysis

PIPS semantics analysis targets mostly integer scalar variables. It is a two-pass process, with a bottom-up pass computing transformers 6.9.1, and a top-down pass propagating preconditions 6.9.2. Transformers and preconditions are specially powerful case of return and jump functions [12]. They abstract relations between program states with polyhedra and encompass most standard interprocedural constant propagations as well as most interval analyses. It is a powerful *relational* symbolic analysis.

Unlike [16] their computations are based on PIPS Hierarchical Control Flow Graph and on syntactic constructs instead of a standard flow graph. The best presentation of this part of PIPS is in [27].

A similar analysis is available in Parafrase-2 []. It handles *polynomial* equations between scalar integer variables. SUIF [] also performs some kind of semantics analysis.

The semantics analysis part of PIPS was designed and developed by François Irigoin.

6.9.1 Transformers

A transformer is an approximate relation between the symbolic initial values of scalar variables and their values after the execution of a statement, simple or compound (see [34] and [27]). In abstract interpretation terminology, a transformer is an abstract command linking the input abstract state of a statement and its output abstract state.

RK: The following is hard to read without any example for someone that knows PIPS. It ought to have everything in this documentation?

By default, only integer scalar variables are analyzed, but properties can be set to handle boolean, string and floating point scalar variables[7] : SEMANTICS_ANALYZE_SCALAR_INTEGER_VARI
SEMANTICS_ANALYZE_SCALAR_BOOLEAN_VARIABLES 6.9.4.1 SEMANTICS_ANALYZE_SCALAR_STRING_VARIABLES
SEMANTICS_ANALYZE_SCALAR_FLOAT_VARIABLES 6.9.4.1 SEMANTICS_ANALYZE_SCALAR_COMPLEX_VARIABLES 6.
SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES 6.9.4.1 SEMANTICS_ANALYZE_CONSTANT_PATH 6.9.4.1

Transformers can be computed intraprocedurally by looking at each function independently or they can be computed interprocedurally starting with the leaves of the call tree[8].

Intraprocedural algorithms use cumulated_effects 6.2.3 to handle procedure calls correctly. In some respect, they are interprocedural since call statements are accepted. Interprocedural algorithms use the summary_transformer 6.9.1.8 of the called procedures.

Fast algorithms use a very primitive non-iterative transitive closure algorithm (two possible versions: flow sensitive or flow insensitive). Full algorithms use a transitive closure algorithm based on vector subspace (i.e. *à la* Karr [39]) or one based on the discrete derivatives [29, 5]. The iterative fix point algorithm for transformers (i.e. Halbwachs/Cousot [16] is implemented but not used because the results obtained with transitive closure algorithms are faster and up-to-now sufficient. Property SEMANTICS_FIX_POINT_OPERATOR 6.9.4.8 is set to select the transitive closure algorithm used.

[7] Floating point values are combined exactly, which is not correct but still useful when dead code can be eliminated according to some parameter value.

[8] Recursive calls are not handled. Hopefully, they are detected by pipsmake to avoid looping forever.

Additional information, such as array declarations and array references, can be used to improve transformers. See the property documentation for:

SEMANTICS_TRUST_ARRAY_DECLARATIONS 6.9.4.2 SEMANTICS_TRUST_ARRAY_REFERENCES 6.9.4.2

Within one procedure, the transformers can be computed in forward mode, using precondition information gathered along. Transformers can also be recomputed once the preconditions are available. In both cases, more precise transformers are obtained because the statement can be better modelized using precondition information. For instance, a non-linear expression can turn out to be linear because the values of some variables are numerically known and can be used to simplify the initial expression. See properties:

SEMANTICS_RECOMPUTE_EXPRESSION_TRANSFORMERS 6.9.4.6

SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT 6.9.4.6

SEMANTICS_RECOMPUTE_FIX_POINTS_WITH_PRECONDITIONS 6.9.4.8

and phase refine_transformers 6.9.1.7.

Unstructured control flow graphs can lead to very long transformer computations, whose results are usually not interesting. Their sizes are limited by two properties:

SEMANTICS_MAX_CFG_SIZE2 6.9.4.5 SEMANTICS_MAX_CFG_SIZE1 6.9.4.5

discussed below.

Default value were set in the early nineties to obtain results fast enough for live demonstrations. They have not been changed to preserve the non-regression tests. However since 2005, processors are fast enough to use the most precise options in all cases.

A transformer map contains a transformer for each statement of a module. It is a mapping from statements to transformers (type statement_mapping, which is not a NewGen file). Transformers maps are stored on and retrieved from disk by *pipsdbm*.

6.9.1.1 Menu for Transformers

```
alias transformers 'Transformers'
alias transformers_intra_fast 'Quick Intra-Procedural Computation'
alias transformers_inter_fast 'Quick Inter-Procedural Computation'
alias transformers_intra_full 'Full Intra-Procedural Computation'
alias transformers_inter_full 'Full Inter-Procedural Computation'
alias transformers_inter_full_with_points_to 'Full Inter-Procedural with points-to Computa
alias refine_transformers 'Refine Transformers'
```

6.9.1.2 Fast Intraprocedural Transformers

Build the fast intraprocedural transformers.

```
transformers_intra_fast         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.proper_effects
```

6.9.1.3 Full Intraprocedural Transformers

Build the improved intraprocedural transformers.

```
transformers_intra_full         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.proper_effects
```

6.9.1.4 Fast Interprocedural Transformers

Build the fast interprocedural transformers.

```
transformers_inter_fast         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.proper_effects
        < PROGRAM.program_precondition
```

6.9.1.5 Full Interprocedural Transformers

Build the improved interprocedural transformers (This should be used as default option.).

```
transformers_inter_full         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.proper_effects
        < PROGRAM.program_precondition
```

6.9.1.6 Full Interprocedural Transformers with points-to

Build the improved interprocedural transformers with points-to informations

```
transformers_inter_full_with_points_to         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.points_to
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.proper_effects
        < PROGRAM.program_precondition
```

6.9.1.7 Refine Full Interprocedural Transformers

Rebuild the interprocedural transformers using interprocedural preconditions. Intraprocedural preconditions are also used to refine all transformers.

```
refine_transformers          > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.proper_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.summary_precondition
        < PROGRAM.program_precondition
```

Rebuild the interprocedural transformers using interprocedural preconditions and points-to information. Intraprocedural preconditions are also used to refine all transformers.

```
refine_transformers_with_points_to          > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code
        < MODULE.points_to
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.proper_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.summary_precondition
        < PROGRAM.program_precondition
```

6.9.1.8 Summary Transformer

A summary transformer is an interprocedural version of the module statement transformer, obtained by eliminating dynamic local, a.k.a. stack allocated, variables. The filtering is based on the module summary effects. Note: each module has a UNIQUE top-level statement.

A `summary_transformer` 6.9.1.8 is of Newgen type `transformer`.

```
summary_transformer             > MODULE.summary_transformer
        < PROGRAM.entities
        < MODULE.transformers
        < MODULE.summary_effects
```

6.9.2 Preconditions

A precondition for a statement s in a module m is a predicate true for every state reachable from the initial state of m, in which s is executed. A precondition

is of NewGen type "transformer" (see *PIPS Internal Representation of Fortran and C code*[9]) and preconditions is of type `statement_mapping`.

Option `preconditions_intra` 6.9.2.5 associates a precondition to each statement, assuming that no information is available at the module entry point.

Inter-procedural preconditions may be computed with *intra*-procedural transformers but the benefit is not clear. Intra-procedural preconditions may be computed with *inter*-procedural transformers. This is faster that a full interprocedural analysis because there is no need for a top-down propagation of summary preconditions. This is compatible with code transformations like `partial_eval` 9.4.2, `simplify_control` 9.3.1 and `dead_code_elimination` 9.3.2.

Since these two options for transformer and precondition computations are independent and that `transformers_inter_full` 6.9.1.5 and `preconditions_inter_full` 6.9.2.7 must be both (independently) selected to obtain the best possible results. These two options are recommended.

6.9.2.1 Initial Precondition or Program Precondition

All `DATA` initializations contribute to the global initial state of the program. The contribution of each module is computed independently. Note that variables statically initialized behave as static variables and are preserved between calls according to Fortran standard. The module initial states are abstracted by an initial precondition based on integer scalar variables only.

Note: To be extended to handle C code. To be extended to handle properly unknown modules.

```
initial_precondition     > MODULE.initial_precondition
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

All initial preconditions, including the initial precondition for the main, are combined to define the program precondition which is an abstraction of the program initial state.

```
program_precondition     > PROGRAM.program_precondition
        < PROGRAM.entities
        < ALL.initial_precondition
```

The program precondition can only be used for the initial state of the main procedure. Although it appears below for all interprocedural analyses and it always is computed, it only is used when a main procedure is available.

6.9.2.2 Intraprocedural Summary Precondition

A summary precondition is of type "transformer", but the argument list must be empty as it is a simple predicate on the initial state. So in fact it is a state predicate.

The intraprocedural summary precondition uses DATA statement for the main module and is the TRUE constant for all other modules.

[9] `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

```
intraprocedural_summary_precondition            > MODULE.summary_precondition
        < PROGRAM.entities
        < MODULE.initial_precondition
```

Interprocedural summary preconditions can be requested instead. They are not described in the same section in order to introduce the summary precondition resource at the right place in `pipsmake.rc`.

No menu is declared to select either intra- or interprocedural summary preconditions.

6.9.2.3 Interprocedural Summary Precondition

By default, summary preconditions are computed intraprocedurally. The interprocedural option must be explicitly activated.

An interprocedural summary precondition for a module is derived from all its call sites. Of course, preconditions must be known for all its callers' statements. The summary precondition is the convex hull of all call sites preconditions, translated into a proper environment which is *not* necessarily the module's frame. Because of *invisible* global and static variables and aliasing, it is difficult for a caller to know which variables might be used by the caller to represent a given memory location. To avoid the problem, the current summary precondition is always translated into the *caller*'s frame. So each module must first translate its summary precondition, when receiving it from the resource manager (*pipsdbm*) before using it.

Note: the previous algorithm was based on a on-the-fly reduction by convex hull. Each time a call site was encountered while computing a module preconditions, the callee's summary precondition was updated. This old scheme was more efficient but not compatible with program transformations because it was impossible to know when the summary preconditions of the modules had to be reset to the infeasible (a.k.a. empty) precondition.

An infeasible precondition means that the module is never called although a main is present in the workspace. If no main module is available, a TRUE precondition is generated. Note that, in both cases, the impact of static initializations propagated by link edition is taken into account although this is prohibited by the Fortran Standard which requires a BLOCKDATA construct for such initializations. In other words, a module which is never called has an impact on the program execution and its declarations should not be destroyed.

```
interprocedural_summary_precondition            > MODULE.summary_precondition
        < PROGRAM.entities
        < PROGRAM.program_precondition
        < CALLERS.preconditions
        < MODULE.callers
```

An interprocedural summary precondition for a module is derived from all its call sites.

```
interprocedural_summary_precondition_with_points_to            > MODULE.summary_preconditi
        < PROGRAM.entities
        < PROGRAM.program_precondition
        < CALLERS.preconditions
```

```
        < MODULE.callers
        < MODULE.points_to
```

The following rule is obsolete. It is context sensitive and its results depends on the history of commands performed on the workspace.

```
summary_precondition            > MODULE.summary_precondition
        < PROGRAM.entities
        < CALLERS.preconditions
        < MODULE.callers
```

6.9.2.4 Menu for Preconditions

```
alias preconditions 'Preconditions'

alias preconditions_intra 'Intra-Procedural Analysis'
alias preconditions_inter_fast 'Quick Inter-Procedural Analysis'
alias preconditions_inter_full 'Full Inter-Procedural Analysis'
alias preconditions_intra_fast 'Fast intra-Procedural Analysis'
```

6.9.2.5 Intra-Procedural Preconditions

Only build the preconditions in a module without any interprocedural propagation. The fast version uses a fast but crude approximation of preconditions for unstructured code.

```
preconditions_intra            > MODULE.preconditions
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_effects
        < MODULE.summary_transformer
        < MODULE.summary_precondition
        < MODULE.code

preconditions_intra_fast            > MODULE.preconditions
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_effects
        < MODULE.summary_transformer
        < MODULE.summary_precondition
        < MODULE.code
```

6.9.2.6 Fast Inter-Procedural Preconditions

Option `preconditions_inter_fast` 6.9.2.6 uses the module own precondition derived from its callers as initial state value and propagates it downwards in the module statement.

The *fast* versions use no fix-point operations for loops.

```
preconditions_inter_fast        > MODULE.preconditions
        < PROGRAM.entities
        < PROGRAM.program_precondition
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < CALLEES.summary_effects
        < MODULE.summary_transformer
```

6.9.2.7 Full Inter-Procedural Preconditions

Option `preconditions_inter_full` 6.9.2.7 uses the module own precondition derived from its callers as initial state value and propagates it downwards in the module statement.

The *full* versions use fix-point operations for loops.

```
preconditions_inter_full        > MODULE.preconditions
        < PROGRAM.entities
        < PROGRAM.program_precondition
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.summary_transformer
```

Option `preconditions_inter_full_with_points_to` 6.9.2.7 uses the module own precondition derived from its callers as initial state value and propagates it downwards in the module statement.

```
preconditions_inter_full_with_points_to        > MODULE.preconditions
        < PROGRAM.entities
        < PROGRAM.program_precondition
        < MODULE.code
        < MODULE.points_to
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < CALLEES.summary_transformer
        < MODULE.summary_transformer
```

6.9.3 Total Preconditions

Total preconditions are interesting to optimize the nominal behavior of a terminating application. It is assumed that the application ends in the main proce-

dure. All other exits, aborts or stops, explicit or implicit such as buffer overflows and zero divide and null pointer dereferencing, are considered exceptions. This also applies at the module level. Modules nominally return. Other control flows are considered exceptions. Non-terminating modules have an empty total precondition[10]. The standard preconditions can be refined by anding with the total preconditions to get information about the nominal behavior. Similar sources of increased accuracy are the array declarations and the array references, which can be exploited directly with properties described in section 6.9.4.2. These two properties should be set to true whenever possible.

Hence, a total precondition for a statement s in a module m is a predicate true for every state from which the final state of m, in which s is executed, is reached. It is an over-approximation of the theoretical total precondition. So, if the predicate is false, the final control state cannot be reached. A total precondition is of NewGen type "transformer" (see *PIPS Internal Representation of Fortran and C code*[11]) and total_preconditions is of type `statement_mapping`.

The relationship with continuations (see Section 6.10) is not clear. Total preconditions should be more general but no must version exist.

Option `total_preconditions_intra` 6.9.3.2 associates a precondition to each statement, assuming that no information is available at the module return point.

Inter-procedural total preconditions may be computed with *intra*-procedural transformers but the benefit is not clear. Intra-procedural total preconditions may be computed with *inter*-procedural transformers. This is faster than a full interprocedural analysis because there is no need for a top-down propagation of summary total postconditions.

Since these two options for transformer and total precondition computations are independent, `transformers_inter_full` 6.9.1.5 and `total_preconditions_inter` 6.9.3.3 must be both (independently) selected to obtain the best possible results.

6.9.3.0.1 Status: This is a set of experimental passes. The intraprocedural part is implemented. The interprocedural part is not implemented yet, waiting for an expressed practical interest. Neither C for loops nor repeat loops are supported.

6.9.3.1 Menu for Total Preconditions

```
alias total_preconditions 'Total Preconditions'

alias total_preconditions_intra 'Total Intra-Procedural Analysis'
alias total_preconditions_inter 'Total Inter-Procedural Analysis'
```

6.9.3.2 Intra-Procedural Total Preconditions

Only build the total preconditions in a module without any interprocedural propagation. No specific condition must be met when reaching a RETURN statement.

[10]Non-termination conditions could also be propagated backwards to provide an over-approximation of the conditions under which an application never terminates, i.e. conditions for liveness.

[11]`http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`

```
total_preconditions_intra            > MODULE.total_preconditions
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.summary_effects
        < MODULE.summary_transformer
        < MODULE.code
```

6.9.3.3 Inter-Procedural Total Preconditions

Option `total_preconditions_inter` 6.9.3.3 uses the module own total postcondition derived from its callers as final state value and propagates it backwards in the module statement. This total module postcondition must be true when the RETURN statement is reached.

```
total_preconditions_inter        > MODULE.total_preconditions
        < PROGRAM.entities
        < PROGRAM.program_postcondition
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.summary_total_postcondition
        < MODULE.summary_effects
        < CALLEES.summary_effects
        < MODULE.summary_transformer
```

The program postcondition is only used for the main module.

6.9.3.4 Summary Total Precondition

The summary total precondition of a module is the total precondition of its statement limited to information observable by callers, just like a summary transformer (see Section 6.9.1.8).

A summary total precondition is of type "transformer".

```
summary_total_precondition            > MODULE.summary_total_precondition
        < PROGRAM.entities
        < CALLERS.total_preconditions
```

6.9.3.5 Summary Total Postcondition

A final postcondition for a module is derived from all its call sites. Of course, total postconditions must be known for all its callers' statements. The summary total postcondition is the convex hull of all call sites total postconditions, translated into a proper environment which is *not* necessarily the module's frame. Because of *invisible* global and static variables and aliasing, it is difficult for a caller to know which variables might be used by the caller to represent a given memory location. To avoid the problem, the current summary total postcondition is always translated into the *caller*'s frame. So each module must first

translate its summary total postcondition, when receiving it from the resource manager (*pipsdbm*) before using it.

A summary total postcondition is of type "transformer".

```
summary_total_postcondition            > MODULE.summary_total_postcondition
        < PROGRAM.entities
        < CALLERS.total_preconditions
        < MODULE.callers
```

6.9.3.6 Final Postcondition

The program postcondition cannot be derived from the source code. It should be defined explicitly by the user. By default, the predicate is always true. But you might want some variables to have specific values, e.g. `KMAX==1`, or signs,`KMAX>1` or relationships `KMAX>JMAX`.

```
program_postcondition     > PROGRAM.program_postcondition
```

6.9.4 Semantic Analysis Properties

6.9.4.1 Value types

By default, the semantic analysis is restricted to scalar integer variables as they are key variables to understand scientific code behavior. However it is possible to analyze scalar variables with other data types. Fortran LOGICAL variables are represented as 0/1 integers. Character string constants and floating point constants are represented as undefined values.

The analysis is thus limited to constant propagation for character strings and floating point values whereas integer, boolean and pointer variables are processed with a relational analysis.

Character string constants of fixed maximal length could be translated into integers but the benefit is not yet assessed because they are not much used in the benchmark and commercial applications we have studied. The risk is to increase significantly the number of overflows encountered during the analysis.

For the pointer analysis, it's strongly recommended to activate `proper_effects_with_points_to` 6.2.1 before performing this analysis.

In interprocedural analysis, or in presence of formal parameter, to performe the pointer analysis, it's strongly recommended to set `SEMANTICS_ANALYZE_CONSTANT_PATH` 6.9.4.1 at TRUE. `SEMANTICS_ANALYZE_CONSTANT_PATH` 6.9.4.1 can also serve to analyse the structures?

```
SEMANTICS_ANALYZE_SCALAR_INTEGER_VARIABLES TRUE
```

```
SEMANTICS_ANALYZE_SCALAR_BOOLEAN_VARIABLES FALSE
```

```
SEMANTICS_ANALYZE_SCALAR_STRING_VARIABLES FALSE
```

```
SEMANTICS_ANALYZE_SCALAR_FLOAT_VARIABLES FALSE
```

```
SEMANTICS_ANALYZE_SCALAR_COMPLEX_VARIABLES FALSE
```

```
SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES FALSE
```

```
SEMANTICS_ANALYZE_CONSTANT_PATH FALSE
```

6.9.4.2 Array Declarations and Accesses

For every module, array declaration are assumed to be correct with respect to the standard: the upper bound must be greater than or equal to the lower bound. When implicit, the lower bound is one. The star upper bound is neglected.

This property is turned off by default because it might slow down PIPS quite a lot without adding any useful information because loop bounds are usually different from array bounds.

```
SEMANTICS_TRUST_ARRAY_DECLARATIONS FALSE
```

For every module, array references are assumed to be correct with respect to the declarations: the subscript expressions must have values lower than or equal to the upper bound and greater than or equal to the lower bound.

This property is turned off by default because it might slow down PIPS quite a lot without adding any useful information.

```
SEMANTICS_TRUST_ARRAY_REFERENCES FALSE
```

6.9.4.3 Type Information

Type information for integer variables is ignored by default. The behavior of natural integer is assumed and no wrap-around is assumed to ever happen. The properties described here could as well be named:

SEMANTICS_ASSUME_NO_INTEGER_OVERFLOW

Type range information is difficult to turn into useful information. It implies some handling of wrap-around behaviors. It is likely to cause lots of overflows with `int` and `long int` variables. It should be used for `unsigned char` and `unsigned short int` only with the current implementation.

This is still an experimental development. By default this property is not set, and it should only be set by *PIPS*[12] developpers.

This property is turned off by default because it might slow down PIPS quite a lot without adding any useful information. It is also turned off because it is experimental and should only be used by developpers.

```
SEMANTICS_USE_TYPE_INFORMATION FALSE
```

Type information can also be used only when computing transformers or when computing preconditions:

```
SEMANTICS_USE_TYPE_INFORMATION_IN_TRANSFORMERS FALSE
```

```
SEMANTICS_USE_TYPE_INFORMATION_IN_PRECONDITIONS FALSE
```

[12] http://www.cri.ensmp.fr/pips

It is not clear why you would like to assume overflows when computing transformers and not when computing preconditions, but the opposite makes sense.

Note that simple statements such as `i++` have no precise convex transformer because of the wrap-around to 0. Assuming the type declaration `unsigned char i`, the transformer maps the new value of `i` to the interval `[0..256]`.

If standard transformers are used, the variable values defined by the integer preconditions must be remapped in the type interval λ using a modulo definition. For instance, Value v is defined as $v = \lambda v_1 + v_2$, v and v_1 are projected and v_2 is renamed v.

Each time a precondition is used in to compute a transformer, it must be normalized according to its type, even when the condition happens to be found without precondition, as a test condition or a loop bound.

For instance, in the example below:

```
void foo(unsigned char i, unsigned char j) {
if(i<j) {
  i++, j++;
  if(i<j)
    // true branch
  else
    // false branch
}}
```

you might wrongly assume that the false branch is never reached. But it is not true if `j==255` initially.

In the same way, the sequence:

```
unsigned char i, j;
i = 257;
j = 3/i;
```

will not be analyzed properly if the precondition for the division is not fixed with type information.

As a consequence, transformers should not be computed *in context* (see `SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT` 6.9.4.6) with the current implementation if type information has an impact on the result. It is necessary to compute the precondition first and then to refine the transformers with them (see `refine_transformers` 6.9.1.7).

To sum up, the basic semantics analysis assumes that no integer overflow occurs during an execution. If integer overflows are known to occur, it is safer to set `SEMANTICS_USE_TYPE_INFORMATION` 6.9.4.3. But this property destroys information gathered about arithmetic information. To obtain more accurate results, set property `SEMANTICS_USE_TYPE_INFORMATION_IN_PRECONDITIONS` **??** to compute transformers without overflows and to remap the preconditions later. The transformers can then be refined with these first preconditions and more accurate preconditions found. As explained above, this is not safe because all these developements are experimental and because precondition information given by test and loop conditions is used without paying attention to type information.

6.9.4.4 Integer Division

Integer divisions are defined by an equation linking the quotient q, the dividendd_1, the divisor d_2 and the remainder r.

$$d_1 = q \times d_2 + r$$

Programming languages like C and Fortran specify that the divident d_1 and the remainder r have the same sign. If d_1 is positive, the remainder is constrained by:

$$0 \leq r < |d_2|$$

Else, it is constrained by:

$$|d_2| < r \leq 0$$

Hence, if the sign of d_1 is unknown, the remainder is less constrained:

$$|d_2| < r < |d_2|$$

Since integer divisions are usually used with positive integer variables used to index arrays, the accuracy of the analysis can be improved by setting the following property to true:

```
SEMANTICS_ASSUME_POSITIVE_REMAINDERS TRUE
```

Since the result is not always correct, this property should be set to false, but for historic reasons it is true by default.

6.9.4.5 Flow Sensitivity

Perform "meet" operations for semantics analysis. This property is managed by `pipsmake` which often sets it to TRUE. See comments in `pipsmake` documentation to turn off convex hull operations for a module or more if they last too long.

```
SEMANTICS_FLOW_SENSITIVE FALSE
```

Complex control flow graph may require excessive computation resources. This may happen when analyzing a parser for instance.

```
SEMANTICS_ANALYZE_UNSTRUCTURED TRUE
```

To reduce execution time, this property is complemented with a heuristics to turn off the analysis of very complex unstructured.

If the control flow graph counts more than `SEMANTICS_MAX_CFG_SIZE1` 6.9.4.5 vertices, use effects only.

```
SEMANTICS_MAX_CFG_SIZE2 20
```

If the control flow graph counts more than `SEMANTICS_MAX_CFG_SIZE1` 6.9.4.5 but less than `SEMANTICS_MAX_CFG_SIZE2` 6.9.4.5 vertices, perform the convex hull of its elementary transformers and take the fixpoint of it. Note that `SEMANTICS_MAX_CFG_SIZE2` 6.9.4.5 is assumed to be greater than or equal to `SEMANTICS_MAX_CFG_SIZE1` 6.9.4.5.

```
SEMANTICS_MAX_CFG_SIZE1 20
```

6.9.4.6 Context for statement and expression transformers

Without preconditions, transformers can be precise only for affine expressions. Approximate transformers can sometimes be derived for other expressions, involving for instance products of variables or divisions.

However, a precondition of an expression can be used to refine the approximation. For instance, some non-linear expressions can become affine because some of the variables have constant values, and some non-linear expressions can be better approximated because the variables signs or ranges are known.

To be backward compatible and to be conservative for PIPS execution time, the default value is false.

Not implemented yet.

```
SEMANTICS_RECOMPUTE_EXPRESSION_TRANSFORMERS FALSE
```

Intraprocedural preconditions can be computed at the same time as transformers and used to improve the accuracy of expression and statement transformers. Non-linear expressions can sometimes have linear approximations over the subset of all possible stores defined by a precondition. In the same way, the number of convex hulls can be reduced if a test branch is never used or if a loop is always entered.

```
SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT FALSE
```

The default value is false for reverse compatibility and for speed.

6.9.4.7 Interprocedural Semantics Analysis

To be refined later; basically, use callee's transformers instead of callee's effects when computing transformers bottom-up in the call graph; when going top-down with preconditions, should we care about unique call site and/or perform meet operation on call site preconditions ?

```
SEMANTICS_INTERPROCEDURAL FALSE
```

This property is used internally and is not user selectable.

6.9.4.8 Fix Point and Transitive Closure Operators

CPU time and memory space are cheap enough to compute loop fix points for *transformers*. This property implies `SEMANTICS_FLOW_SENSITIVE` 6.9.4.5 and is not user-selectable.

```
SEMANTICS_FIX_POINT FALSE
```

The default fix point operator, called *transfer*, is good for induction variables but it is not good for all kinds of code. The default fix point operator is based on the transition function associated to a loop body. A computation of eigenvectors for eigenvalue 1 is used to detect loop invariants. This fails when no transition function but only a transition relation is available. Only equations can be found.

The second fix point operator, called *pattern*, is based on a pattern matching of elementary equations and inequalities of the loop body transformer. Obvious invariants are detected. This fix point operator is not better than the previous one for induction variables but it can detect invariant equations and inequalities.

A third fix point operator, called *derivative*, is based on finite differences. It was developed to handled DO loops desugared into WHILE loops as well as standard DO loops. The loop body transformer on variable values is projected onto their finite differences. Invariants, both equations and inequalities, are deduced directly from the constraints on the differences and after integration. This third fix point operator should be able to find at least as many invariants as the two previous one, but at least some inequalities are missed because of the technique used. For instance, constraints on a flip-flop variable can be missed. Unlike Cousot-Halbwachs fix point (see below), it does not use Chernikova steps and it should not slow down analyses.

This property is user selectable and its default value is *derivative*. The default value is the only one which is now seriously maintained.

```
SEMANTICS_FIX_POINT_OPERATOR "derivative"
```

The next property is experimental and its default value is 1. It is used to unroll while loops virtually, i.e. at the semantics equation level, to cope with periodic behaviors such as flip-flops. It is effective only for standard while loops and the only possible value other than 1 is 2.

```
SEMANTICS_K_FIX_POINT 1
```

The next property `SEMANTICS_PATTERN_MATCHING_FIX_POINT` has been removed and replaced by option *pattern* of the previous property.

This property was defined to select one of Cousot-Halbwachs's heuristics and to compute fix points with inequalities and equalities for loops. These heuristics could be used to compute fix points for transformers and/or preconditions. This option implies `SEMANTICS_FIX_POINT` 6.9.4.8 and `SEMANTICS_FLOW_SENSITIVE` 6.9.4.5. It has not been implemented yet in PIPS[13] because its accuracy has not yet been required, but is now badly named because there is no direct link between *inequality* and *Halbwachs*. Its default value is *false* and it is not user selectable.

```
SEMANTICS_INEQUALITY_INVARIANT FALSE
```

Because of convexity, some fix points may be improved by using some of the information carried by the preconditions. Hence, it may be profitable to recompute loop fix point transformer when preconditions are being computed.

The default value is false because this option slows down PIPS and does not seem to add much useful information in general.

```
SEMANTICS_RECOMPUTE_FIX_POINTS_WITH_PRECONDITIONS FALSE
```

The next property is used to refine the computation of preconditions inside nested loops. The loop body is reanalyzed to get one transformer for each control path and the identity transformer is left aside because it is useless to compute the loop body precondition. This development is experimental and turned off by default.

```
SEMANTICS_USE_TRANSFORMER_LISTS FALSE
```

The next property is only useful if the previous one is set to true. Instead of computing the fix point of the convex hull of the transformer list, it computes

[13]But some fix point functions are part of the C3 linear library.

the convex hull of the derivative constraints. Since it is a new feature, it is set to false by default, but it should become the default option because it should always be more accurate, at least indirectly because the systems are smaller. The number of overflows is reduced, as well as the execution time. In practice, these improvements have not been measured. This development is experimental and turned off by default.

```
SEMANTICS_USE_DERIVATIVE_LIST FALSE
```

The next property is only useful if Property SEMANTICS_USE_TRANSFORMER_LISTS 6.9.4.8 is set to true. Instead of computing the precondition derived from the transitive closure of a transformer list, semantics also computes the preconditions associated to different projections of the transformer list and use as loop precondition the intersection of these preconditions. Although it is a new feature, it is set to true by default for the validation's sake. See test case Semantics/maison-neuve09.c: it improves the accuracy, but not as much as SEMANTICS_USE_DERIVATIVE_LIST 6.9.4.8. This development is experimental and turned off by default.

```
SEMANTICS_USE_LIST_PROJECTION TRUE
```

The string Property SEMANTICS_LIST_FIX_POINT_OPERATOR 6.9.4.8 is used to select a particular heuristic to compute an approximation of the transitive closure of a list of transformers. It is only useful if Property SEMANTICS_USE_TRANSFORMER_LISTS 6.9.4.8 is selected. The current default value is "depth_two". An experimental value is "max_depth".

```
SEMANTICS_LIST_FIX_POINT_OPERATOR "depth_two"
```

Preconditions can (used to) preserve initial values of the formal parameters. This is not often useful in C because programmers usually avoid modifying scalar parameters, especially integer ones. However, old values create problems in region computation because preconditions seem to be used instead of tranformer ranges. Filtering out the initial value does reduce the precision of the precondition analysis, but this does not impact the transformer analysis. Since the advantage is really limited to preconditions and for the region's sake, the default value is set to true. Turn it to false if you have a doubt about the preconditions really available.

The loop index is usually dead on loop exit. So keeping information about its value is useless... most of the times. However, it is preserved by default.

```
SEMANTICS_KEEP_DO_LOOP_EXIT_CONDITION TRUE
```

```
SEMANTICS_FILTER_INITIAL_VALUES TRUE
```

6.9.4.9 Normalization Level

Normalizing transformer and preconditions systems is a delicate issue which is not mathematically defined, and as such is highly empirical. It's a tradeoff between eliminating redundant information, keeping an internal storage not too far from the prettyprinting for non-regression testing, exposing useful information for subsequent analyses,... all this at a reasonable cost.

Several *levels* of normalization are possible. These levels do not correspond to graduations on a normalization scale, but are different normalization heuristics. A level of 4 includes a preliminary lexicographic sort of contraints, which is very user friendly, but currently implies strings manipulations which are quite costly. It has been recently chosen to perform this normalization only before storing transformers and preconditions to the database (`SEMANTICS_NORMALIZATION_LEVEL_BEFORE_STORAGE` with a default value of 4). However, this can still have a serious impact on performances. With any other value, the normalization level is equel to 2.

```
SEMANTICS_NORMALIZATION_LEVEL_BEFORE_STORAGE 4
```

6.9.4.10 Evaluation of sizeof

Property `EVAL_SIZEOF` 9.4.2 can be set to true to force the static evaluation of size of. Potentially, the computed transformers and preconditions are only valid for the target architecture defined in `ri-util-local.h`.

6.9.4.11 Prettyprint

Preconditions reflect by default all knowledge gathered about the current state (i.e. store). However, it is possible to restrict the information to variables actually read or written, directly or indirectly, by the statement following the precondition.

```
SEMANTICS_FILTERED_PRECONDITIONS FALSE
```

6.9.4.12 Debugging

Output semantics results on stdout

```
SEMANTICS_STDOUT FALSE
```

Debug level for semantics used to be controlled by a property. A Shell variable, `SEMANTICS_DEBUG_LEVEL`, is used instead.

6.9.5 Reachability Analysis: The Path Transformer

A set of operations on array regions are defined in PIPS such as the function `regions_intersection`. However, this is not sufficient because regions should be combined with what we call path transformers in order to propagate the memory stores used in them. Indeed, regions operations must be performed with respect to the abstraction of the same memory store.

A path transformer permits to compare array regions of statements originally defined in different memory stores.

The path transformer between two statements computes the possible changes performed by a piece of code delimited by two statements S_{begin} and S_{end} enclosed within a statement S. A path transformer is represented by a convex polyhedron over program variables and its computation is based on transformers; thus, it is a system of linear inequalities. The goal of the following phase is to compute the path transformer between two statements labeled by sb and se.

```
path_transformer                > MODULE.path_transformer_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

The next properties are used by Phase `path_transformer` to label the two statements S_{begin} and S_{end}.

```
PATH_TRANSFORMER_BEGIN "sb"
```

```
PATH_TRANSFORMER_END "se"
```

The next property is used to allow an empty path or not which is necessary for the test of dependence. If its value is false, we return an empty transformer for an empty path. Otherwise, an identity transformer is returned.

```
IDENTITY_EMPTY_PATH_TRANSFORMER TRUE
```

6.10 Continuation conditions

Continuation conditions are attached to each statement. They represent the conditions under which the program will not stop in this statement. Under- and over-approximations of these conditions are computed.

```
continuation_conditions > MODULE.must_continuation
                        > MODULE.may_continuation
                        > MODULE.must_summary_continuation
                        > MODULE.may_summary_continuation
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < CALLEES.must_summary_continuation
        < CALLEES.may_summary_continuation
```

6.11 Complexities

Complexities are symbolic approximations of the execution times of statements. They are computed interprocedurally and based on polynomial approximations of execution times. Non-polynomial execution times are represented by *unknown* variables which are not free with respect to the program variables. Thus non-polynomial expressions are equivalent to polynomial expressions over a larger set of variables.

Probabilities for tests should also result in unknown variables (still to be implemented). See [57].

A `summary_complexity` is the approximation of a module execution times. It is translated and used at call sites.

Complexity estimation could be refined (i.e. the number of unknown variables reduced) by using transformers to combine elementary complexities using local states, rather than preconditions to combine elementary complexities relatively to the module initial state. The same options exist for region computation. The initial version [45] used the initial state for combinations. The new version [18] delays evaluation of variable values as long as possible but does not really use local states.

The first version of the complexity estimator was designed and developed by Pierre Berthomier. It was restricted to intra-procedural analysis. This first version was enlarged and validated on real code for SPARC-2 machines by Lei Zhou [57]. Since, it has been modified slightly by François Irigoin. For simple programs, complexity estimation are strongly correlated with execution times. The estimations can be used to see if program transformations are beneficial.

Known bugs: tests and while loops are not correctly handled because a fixed probably of 0.5 is systematically assumed.

6.11.1 Menu for Complexities

```
alias complexities      'Complexities'
alias uniform_complexities      'Uniform'
alias fp_complexities   'FLOPs'
alias any_complexities  'Any'
```

6.11.2 Uniform Complexities

Complexity estimation is based on a set of basic operations and fixed execution times for these basic operation. The choice of the set is critical but fixed. Experiments by Lei Zhou showed that it should be enlarged. However, the basic times, which also are critical, are tabulated. New sets of tables can easily be developed for new processors.

Uniform complexity tables contain a unit execution time for all basic operations. They nevertheless give interesting estimations for SPARC SS-10, especially for -O2/-O3 optimized code.

```
uniform_complexities                    > MODULE.complexities
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < CALLEES.summary_complexity
```

6.11.3 Summary Complexity

Local variables are eliminated from the complexity associated to the top statement of a module in order to obtain the modules' summary complexity.

```
summary_complexity              > MODULE.summary_complexity
        < PROGRAM.entities
        < MODULE.code
        < MODULE.complexities
```

6.11.4 Floating Point Complexities

Tables for floating point complexity estimation are set to 0 for non-floating point operations, and to 1 for all floating point operations, including intrinsics like `SIN`.

```
fp_complexities                    > MODULE.complexities
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < CALLEES.summary_complexity
```

This enables the default specification within the properties to be considered.

```
any_complexities                    > MODULE.complexities
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < CALLEES.summary_complexity
```

6.11.5 Complexity properties

The following properties control the static estimation of dynamic code execution time.

6.11.5.1 Debugging

Trace the walk across a module's internal representation:

```
COMPLEXITY_TRACE_CALLS FALSE
```

Trace all intermediate complexities:

```
COMPLEXITY_INTERMEDIATES FALSE
```

Print the complete cost table at the beginning of the execution:

```
COMPLEXITY_PRINT_COST_TABLE FALSE
```

The cost table(s) contain machine and compiler dependent information about basic execution times, e.g. time for a load or a store.

6.11.5.2 Fine Tuning

It is possible to specify a list of variables which must remain literally in the complexity formula, although their numerical values are known (this is OK) or although they have multiple unknown and unrelated values during any execution (this leads to an incorrect result).

Formal parameters and imported global variables are left unevaluated.

They have relatively high priority (FI: I do not understand this comment by Lei).

This list should be empty by default (but is not for unknown historical reasons):

```
COMPLEXITY_PARAMETERS "IMAX LOOP"
```

Controls the printing of *accuracy* statistics:

- 0: do not prettyprint any statistics with complexities (to give the user a false sense of accuracy and/or to avoid cluttering his/her display); this is the default value;
- 1: prettyprint statistics only for loop/block/test/unstr. statements and not for basic statements, since they should not cause accuracy problems;
- 2: prettyprint statistics for all statements

```
COMPLEXITY_PRINT_STATISTICS 0
```

6.11.5.3 Target Machine and Compiler Selection

This property is used to select a set of basic execution times. These times depend on the target machine, the compiler and the compilation options used. It is shown in [57] that fixed basic times can be used to obtain accurate execution times, if enough basic times are considered, and if the target machine has a simple RISC processor. For instance, it is not possible to use only one time for a register load. It is necessary to take into account the nature of the variable, i.e. formal parameter, dynamic variable, global variable, and the nature of the access, e.g. the dimension of an accessed array. The cache can be ignored an replacer by an average hit ratio.

Different set of elementary cost tables are available:

- `all_1`: each basic operation cost is 1;
- `fp_1`: only floating point operations are taken into account and have cost unit 1; all other operations have a null cost.

In the future, we might add a sparc-2 table...

The different elementary table names are defined in `complexity-local.h`. They presently are `operation`, `memory`, `index`, `transcend` and `trigo`.

The different tables required are to be found in `$PIPS_LIBDIR/complexity/xyz`, where `xyz` is specified by this property:

```
COMPLEXITY_COST_TABLE "all_1"
```

6.11.5.4 Evaluation Strategy

For the moment, we have designed two ways to solve the complexity combination problem. Since symbolic complexity formulae use program variables it is necessary to specify in which store they are evaluated. If two complexity formulae are computed relatively to two different stores, they cannot be directly added.

The first approach, which is implemented, uses the module initial store as universal store for all formulae (but possibly for the complexity of elementary

statements). In some way, symbolic variable are evaluated as early as possible as soon as it is known that they won't make it in the module summary complexity.

This first method is easy to implement when the preconditions are available but it has at least two drawbacks:

- if a variable is used in different places with the same unknown value, each occurrence will be replaced by a different unknown value symbol (the infamous `UU_xx` symbols in formulae).
- since variables are replaced by numerical values as soon as possible as early as possible, the user is shown a numerical execution time instead of a symbolic formulae which would likely be more useful (see property `COMPLEXITY_PARAMETERS` 6.11.5.2). This is especially true with interprocedural constant propagation.

The second approach, which is not implemented, delay variable evaluation as late as possible. Complexities are computed and given relatively to the stores used by each statements. Two elementary complexities are combined together using the earliest store. The two stores are related by a *transformer* (see Section 6.9.4). Such an approach is used to compute `MUST` regions as precisely as possible (see Section 6.12.9).

A simplified version of the late evaluation was implemented. The initial store of the procedure is the only reference store used as with the early evaluation, but variables are not evaluated right away. They only are evaluated when it is necessary to do so. This not an ideal solution, but it is easy to implement and reduces considerably the number of unknown values which have to be put in the formulae to have correct results.

```
COMPLEXITY_EARLY_EVALUATION FALSE
```

6.12 Convex Array Regions

Convex array regions are functions mapping a memory store onto a convex set of array elements. They are used to represent the memory effects of modules or statements. Hence, they are expressed with respect to the initial store of the module or to the store immediately preceding the execution of the statement they are associated with. The latter is now standard in PIPS. Comprehensive information about convex array regions and their associated algorithms is available in Creusillet's PhD Dissertation [20].

Apart from the array name and its dimension descriptors (or ϕ variables), an array region contains three additional informations:

- The *type* of the region: `READ` (`R`) or `WRITE` (`W`) to represent the effects of statements and procedures; `IN` and `OUT` to represent the flow of array elements.
- The *approximation* of the region: `EXACT` when the region exactly represents the requested set of array elements, or `MAY` or `MUST` if it is an over- or under-approximation (`MUST` $\subseteq$ `EXACT` $\subseteq$ `MAY`).

Unfortunately, for historical reasons, `MUST` is still used in the implementation instead of `EXACT`, and actual `MUST` regions are not computed. Moreover, the `must_regions` option in fact computes exact and may regions.

`MAY` regions are flow-insensitive regions, whereas `MUST` regions are flow sensitive. Any array element touched by any execution of a statement is in the `MAY` region of this statement. Any array element in the `MUST` region of a statement is accessed by any execution of this statement.

- a convex polyhedron containing equalities and inequalities: they link the ϕ variables that represent the array dimensions, to the values of the program integer scalar variables.

  For a performance purpose, this convex polyhedron is never add for scalar variables, except for the computation of IN/OUT-Regions for loops whose his algorithm required it.

For instance, the convex array region:

<A(ϕ_1,ϕ_2)-W-EXACT-{ϕ_1==I, ϕ_1==ϕ_2}>

where the region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of `A`, corresponds to an assignment of the element `A(I,I)`.

Internally, convex array regions are of type `effect` and as such can be used to build use-def chains (see Section 6.5.3). Regions chains are built using *proper regions* which are particular `READ` and `WRITE` regions. For simple statements (assignments, calls to intrinsic functions), summarization is avoided to preserve accuracy. At this inner level of the program control flow graph, the extra amount of memory necessary to store regions without computing their convex hull should not be too high compared to the expected gain for dependence analysis. For tests and loops, proper regions contain the regions associated to the condition or the range. And for external calls, proper regions are the *summary regions* of the callee translated into the caller's name space, to which are merely appended the regions of the expressions passed as argument (no summarization for this step).

Ressource `proper_regions` is equivalent to `proper_effects` (see Section 6.2.1), and `regions` to `cumlulated_effects` (see Section 6.2.3). So they share some features, like LUNS present for `regions`/`cumlulated_effects` on return/exit/abort statements.

Together with READ/WRITE regions and IN regions are computed their invariant versions for loop bodies (`MODULE.inv_regions` and `MODULE.inv_in_regions`). For a given loop body, they are equal to the corresponding regions in which all variables that may be modified by the loop body (except the current loop index) are eliminated from the descriptors (convex polyhedron). For other statements, they are equal to the empty list of regions.

In the following trivial example,

```
k = 0;
for(i=0; i<N; i++)
```

```
{
// regions for loop body:
//     <a[phi1]-W-EXACT-{PHI1==K,K==I}>
// invariant regions for loop body:
//     <a[phi1]-W-EXACT-{PHI1==I}>
  k = k+1;
  a[k] = k;
}
```

notice that the variable k which is modified in the loop body, and which appears in the loop body region polyhedron, does not appear anymore in the invariant region polyhedron.

`MAY READ` and `WRITE` region analysis was first designed by Rémi Triolet [48] and then revisited by François Irigoin [49]. Alexis Platonoff [45] implemented the first version of region analysis in `PIPS`. These regions were computed with respect to the initial stores of the modules. François Irigoin and, mainly, Béatrice Creusillet [18, 19, 20], added new functionalities to this first version as well as functions to compute `MUST` regions, and `IN` and `OUT` regions.

`MAY` and `MUST` regions also compute useful_variables_regions ressource. This ressource computes the regions used by a variable at the memory state of is declaration. So it associates for each entity variable of a module, a R/W regions. It was already compute during the computation if the regions but not memorize. The store of this ressource was added by Nelson Lossing.

Array regions for C programs are currently under development.

6.12.1 Menu for Convex Array Regions

```
alias regions 'Array regions'

alias may_regions 'MAY regions'
alias must_regions 'EXACT or MAY regions'
alias useful_variables_regions 'Useful Variables regions'
```

6.12.2 MAY READ/WRITE Convex Array Regions

This function computes the `MAY` pointer regions in a module.

```
may_pointer_regions             > MODULE.proper_pointer_regions
                                > MODULE.pointer_regions
                                > MODULE.inv_pointer_regions
                                > MODULE.useful_variables_pointer_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_pointer_regions
```

This function computes the `MAY` regions in a module.

```
may_regions                     > MODULE.proper_regions
```

```
                                > MODULE.regions
                                > MODULE.inv_regions
                                > MODULE.useful_variables_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_regions
```

6.12.3 MUST READ/WRITE Convex Array Regions

This function computes the MUST regions in a module.

```
must_pointer_regions            > MODULE.proper_pointer_regions
                                > MODULE.pointer_regions
                                > MODULE.inv_pointer_regions
                                > MODULE.useful_variables_pointer_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_pointer_regions
```

This function computes the MUST pointer regions in a module using simple points_to information to disambiguate dereferencing paths.

```
must_pointer_regions_with_points_to > MODULE.proper_pointer_regions
                                    > MODULE.pointer_regions
                                    > MODULE.inv_pointer_regions
                                    > MODULE.useful_variables_pointer_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.points_to
        < CALLEES.summary_pointer_regions
```

This function computes the MUST regions in a module.

```
must_regions                    > MODULE.proper_regions
                                > MODULE.regions
                                > MODULE.inv_regions
                                > MODULE.useful_variables_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_regions
```

This function computes the MUST regions in a module using information on pointer targets given by points-to.

```
must_regions_with_points_to     > MODULE.proper_regions
                                > MODULE.regions
                                > MODULE.inv_regions
                                > MODULE.useful_variables_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.points_to
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_regions
```

This function computes the MUST regions in a module using information on pointer targets given by pointer values.

```
must_regions_with_pointer_values > MODULE.proper_regions
                                > MODULE.regions
                                > MODULE.inv_regions
                                > MODULE.useful_variables_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.simple_pointer_values
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_regions
```

6.12.4 Summary READ/WRITE Convex Array Regions

Module summary regions provides an approximation of the effects it's execution has on its callers variables as well as on global and static variables of its callees.

```
summary_pointer_regions                 > MODULE.summary_pointer_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.pointer_regions

summary_regions                 > MODULE.summary_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
```

6.12.5 IN Convex Array Regions

IN convex array regions are flow sensitive regions. They are read regions not covered (i.e. not previously written) by assignments in the local hierarchical control-flow graph. There is no way with the current pipsmake-rc and pipsmake to express the fact that IN (and OUT) regions must be calculated using must_regions 6.12.3 (a new kind of resources, must_regions 6.12.3, should be

added). The user must be knowledgeable enough to select `must_regions` 6.12.3 first.

```
in_regions                      > MODULE.in_regions
                                > MODULE.cumulated_in_regions
                                > MODULE.inv_in_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.summary_effects
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.regions
        < MODULE.inv_regions
        < CALLEES.in_summary_regions
```

6.12.6 IN Summary Convex Array Regions

This pass computes the IN convex array regions of a module. They contain the array elements and scalars whose values impact the output of the module.

```
in_summary_regions              > MODULE.in_summary_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.in_regions
```

6.12.7 OUT Summary Convex Array Regions

This pass computes the OUT convex array regions of a module. They contain the array elements and scalars whose values impact the continuation of the module.

See Section 6.12.8.

```
out_summary_regions             > MODULE.out_summary_regions
        < PROGRAM.entities
        < CALLERS.out_regions
```

6.12.8 OUT Convex Array Regions

`OUT` convex array regions are also flow sensitive regions. They are downward exposed written regions which are also used (i.e. imported) in the continuation of the program. They are also called *exported* regions. Unlike `READ`, `WRITE` and `IN` regions, they are propagated downward in the call graph and in the hierarchical control flow graphs of the subroutines.

```
out_regions                     > MODULE.out_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
```

```
< MODULE.preconditions
< MODULE.regions
< MODULE.inv_regions
< MODULE.summary_effects
< MODULE.cumulated_effects
< MODULE.cumulated_in_regions
< MODULE.inv_in_regions
< MODULE.out_summary_regions
```

6.12.9 Properties for Convex Array Regions

If `MUST_REGIONS` is true, then it computes convex array regions using the algorithm described in report E/181/CRI, called T^{-1} *algorithm*. It provides more accurate regions, and preserve MUST approximations more often. As it is more costly, its default value is FALSE. EXACT_REGIONS is true for the moment for backward compatibility only.

```
EXACT_REGIONS TRUE
```

```
MUST_REGIONS FALSE
```

The default option is to compute regions without taking into account declared array bounds. The next property can be turned to `TRUE` to systematically add them in the region descriptors. Both options have their advantages and drawbacks, but the second one implies that the *PIPS*[14] user is sure that her/his program is correct with respect to array accesses. In case of doubt, you might want to run pass `array_bound_check_bottom_up` 7.1.1 or `array_bound_check_top_down` 7.1.2.

```
REGIONS_WITH_ARRAY_BOUNDS FALSE
```

Property `MEMORY_IN_OUT_EFFECTS_ONLY` 6.12.9's default value is set to `TRUE` to avoid computing IN and OUT effects or regions on non-memory effects, even if `MEMORY_EFFECTS_ONLY` 6.2.7.5 is set to `FALSE`.

```
MEMORY_IN_OUT_EFFECTS_ONLY TRUE
```

The current implementation of effects, simple effects as well as convex array regions, relies on a generic engine which is independent of the effect descriptor representation. The current representation for array regions, parameterized integer convex polyhedra, allows various patterns an provides the ability to exploit context information at a reasonable expense. However, some very common patterns such as nine-point stencils used in seismic computations or red-black patterns cannot be represented. It has been a long lasting temptation to try other representations [20].

A Complementary sections (see Section 6.15) implementation was formerly began as a set of new phases by Manjunathaiah Muniyappa, but is not maintained anymore.

And Nga Nguyen more recently created two properties to switch between regions and disjunctions of regions (she has already prepared basic operators). For the moment, they are always `FALSE`.

[14] `http://www.cri.ensmp.fr/pips`

```
DISJUNCT_REGIONS FALSE
```

```
DISJUNCT_IN_OUT_REGIONS FALSE
```

Statistics may be obtained about the computation of convex array regions. When the next property (REGIONS_OP_STATISTICS) is set to TRUE statistics are provided about operators on regions (union, intersection, projection,...). The second next property turns on the collection of statistics about the interprocedural translation.

```
REGIONS_OP_STATISTICS FALSE
```

```
REGIONS_TRANSLATION_STATISTICS FALSE
```

6.13 Live Memory Access Paths

There are many cases in which it is necessary to know if a variable may be used in the remainder of the execution of the analyzed application. For instance, a global variable cannot be privatized, or a global array be scalarized, if we do not know whether their values are used afterwards, or a copy-out must be generated, which is not currently implemented in the simplest algorithms. Similarly, preconditions do not need to propagate information about variables that are no longer alive.

6.13.1 Live Paths

Traditional liveness analyzes deals with scalar variables. However, with C code, it is interesting to be able to distinguish between different structure fields for instance, and it may also be interesting to deal with array regions. So we have retained for these analyzes an internal representation as *effects*, thus allowing to deal with general memory access paths, and to rely on the existing machinery of effects/regions computations.
Memory access paths can be equivalent to the constant path? **??**

For each statement or function, we compute two sets: a *Live_in* set contains the memory paths which are alive in the store preceding the statement execution, while a *Live_out* set contains the memory paths alive in the store immediately following the statement execution. For sequences of instructions, the *Live_out* set of an instruction is equal to the *Live_in* set of the next instruction. However, this not true for the last statement of conditional or loop bodies and the first statements of the next instructions.

```
live_out_summary_paths     > MODULE.live_out_summary_paths
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.live_out_paths
```

```
live_paths      > MODULE.live_in_paths
                > MODULE.live_out_paths
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.in_effects
        < MODULE.live_out_summary_paths


live_in_summary_paths     > MODULE.live_in_summary_paths
        < PROGRAM.entities
        < MODULE.code
        < MODULE.live_in_paths
```

6.13.2 Live Out Regions

This pass only keeps constraints on live variables of out-regions when it's possible. So the polyhedral constraints for a variable will be less constraint. But we assume that if a variable is not alive, there is no reason to appear on the constraints of the out-regions.

This pass can be usefull to have regions that wil be use to generate some code. It has better insure to not revive a "die" variable.

See Section 6.12.8 and 6.13.1.

Note: Only constraints with variable are considered, since regions don't have constraints on cell (PHI¡a[0] not possible).

```
live_out_regions  > MODULE.live_out_regions
                  < PROGRAM.entities
                  < MODULE.code
                  < MODULE.out_regions
                  < MODULE.live_out_paths
                  < MODULE.live_in_paths
```

6.13.3 Live In/Out Effect

There is no reason to ave live IN/OUT effects, because variables in IN/OUT effects must be present in Live IN/OUT. If it's not the case, it must be a bug somewhere?

6.14 Alias Analysis

6.14.1 Dynamic Aliases

Dynamic aliases are pairs (formal parameter, actual parameter) of convex array regions generated at call sites. An "IN alias pair" is generated for each IN region of a called module and an "OUT alias pair" for each OUT region. For EXACT regions, the transitive, symmetric and reflexive closure of the dynamic alias relation results in the creation of equivalence classes of regions (for MAY regions, the closure is different and does not result in an equivalence relation, but nonetheless allows us to define alias classes). A set of alias classes is generated

for a module, based on the IN and OUT alias pairs of all the modules below it in the callgraph. The alias classes for the whole workspace are those of the module which is at the root of the callgraph, if the callgraph has a unique root. As an intermediate phase between the creation of the IN and OUT alias pairs and the creation of the alias classes, "alias lists" are created for each module. An alias list for a module is the transitive closure of the alias pairs (IN or OUT) for a particular path through the callgraph subtree rooted in this module.

```
in_alias_pairs > MODULE.in_alias_pairs
        < PROGRAM.entities
        < MODULE.callers
        < MODULE.in_summary_regions
        < CALLERS.code
        < CALLERS.cumulated_effects
        < CALLERS.preconditions

out_alias_pairs > MODULE.out_alias_pairs
        < PROGRAM.entities
        < MODULE.callers
        < MODULE.out_summary_regions
        < CALLERS.code
        < CALLERS.cumulated_effects
        < CALLERS.preconditions

alias_lists > MODULE.alias_lists
        < PROGRAM.entities
        < MODULE.in_alias_pairs
        < MODULE.out_alias_pairs
        < CALLEES.alias_lists

alias_classes > MODULE.alias_classes
        < PROGRAM.entities
        < MODULE.alias_lists
```

6.14.2 Init Points-to Analysis

This phase generates synthetic points-to relations for formal parameters. It creates synthetic sinks, i.e. stubs, for formal parameters and provides an initial set of points-to to the `intraprocedural_points_to_analysis` 6.14.5.

Currently, it assumes that no sharing exists between the formal parameters and within the data structures pointed to by the formal parameters. Two properties should control this behavior, `ALIASING_ACROSS_FORMAL_PARAMETERS` 6.14.8.1 and `ALIASING_ACROSS_TYPES` 6.14.8.1. The first one supersedes the property `ALIASING_INSIDE_DATA_STRUCTURE` 6.14.8.1.

```
alias init_points_to_analysis  'Init Points To Analysis'

init_points_to_analysis > MODULE.init_points_to_list
           < PROGRAM.entities
           < MODULE.code
```

6.14.3 Interprocedural Points to Analysis

This pass is being implemented by Amira MENSI. The `interprocedural_points_to_analysis` 6.14.3 is implemented in order to compute points-to relations in an interprocedural way, based on WILSON algorithm. This phase computes both Gen and Kill sets at the level of the call site.It requires another resource which is computed by `intraprocedural_points_to_analysis` 6.14.5.

```
alias interprocedural_points_to_analysis  'Interprocedural Points To Analysis'

interprocedural_points_to_analysis > MODULE.points_to
                   > MODULE.points_to_out
                   > MODULE.points_to_in
        ! SELECT.proper_effects_with_points_to
        ! SELECT.cumulated_effects_with_points_to
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects
        < CALLEES.points_to_out
        < CALLEES.points_to_in
```

6.14.4 Fast Interprocedural Points to Analysis

This pass is being implemented by Amira MENSI. The `fast_interprocedural_points_to_analysis` 6.14.4 is implemented in order to compute points-to relations in an interprocedural way, based on WILSON algorithm. This phase computes only Kill sets at the call site level. It requires another resource which is computed by `intraprocedural_points_to_analysis` 6.14.5.

```
alias fast_interprocedural_points_to_analysis  'Fast Interprocedural Points To Analysis'

fast_interprocedural_points_to_analysis > MODULE.points_to
                   > MODULE.points_to_out
                   > MODULE.points_to_in
        ! SELECT.proper_effects_with_points_to
        ! SELECT.cumulated_effects_with_points_to
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects
        < CALLEES.points_to_out
        < CALLEES.points_to_in
```

6.14.5 Intraprocedural Points to Analysis

This function is being implemented by Amira MENSI. The `intraprocedural_points_to_analysis` 6.14.5 is implemented in order to compute points-to relations, based on Emami algorithm. EMAMI algorithm is a top-down analysis which calcules the points-to relations by applying specific rules to each assignement pattern identified. This phase requires another resource which is `init_points_to_analysis` 6.14.2.

Ressources points_to_in and points_to_out will be used to compute the transfer function later. They represent points-to relation at the beginning of functions where sources are formal parameters or global variables. Points_to_out are points-to relations at the end of function's body, it contains return value and it's sink, formal parameters, gloabl variables and heap allocated variables which can be visible beyond function's scope. And using effects to compute calls impact on points-to analysis.

```
alias intraprocedural_points_to_analysis  'Intraprocedural Points To Analysis'

intraprocedural_points_to_analysis > MODULE.points_to
                   > MODULE.points_to_out
                   > MODULE.points_to_in
        ! SELECT.proper_effects_with_points_to
        ! SELECT.cumulated_effects_with_points_to
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects
        < CALLEES.points_to_out
        < CALLEES.points_to_in
```

The pointer effects are useful, but they are recomputed for each expression and subexpression by the points-to analysis.

6.14.6 Initial Points-to or Program Points-to

Because no top-down points-to analysis is available, this two passes are useless. A top-down points-to analysis would be useful to check that the restrict assumption about formal parameters is met by the actual parameters. It might make possible a slighlty more precise points-to information in the functions. Hopefully, the formal context and the points-to stubs provide enough equivalent information to the passes that use points-to information.

```
initial_points_to     > MODULE.initial_points_to
        < PROGRAM.entities
        < MODULE.code
        < MODULE.points_to_out
```

All initial points-to are combined to define the program points-to which is an abstraction of the program initial state.

```
program_points_to     > PROGRAM.program_points_to
        < PROGRAM.entities
        < ALL.initial_points_to
```

The program points-to can only be used for the initial state of the main procedure. Although it appears below for all interprocedural analyses and it always is computed, it only is used when a main procedure is available.

6.14.7 Pointer Values Analyses

Computes the initial pointer values from the global or static declarations of the module.

```
initial_simple_pointer_values > MODULE.initial_simple_pointer_values
           < PROGRAM.entities
           < MODULE.code
```

Computes the initial pointer values of the program from the global declarations and the static declarations inside the program modules. They are computed by merging the initial pointer values of all the modules (this may include those which do not belong to actually realizable paths).

```
program_simple_pointer_values > PROGRAM.program_simple_pointer_values
           < PROGRAM.entities
           < ALL.initial_simple_pointer_values
```

Pointer values analysis is another kind of pointer analysis which tries to gather *Pointer Values* both in terms of other pointer values but also of memory addresses. This phase is under development.

```
alias simple_pointer_values  'Pointer Values Analysis'

simple_pointer_values > MODULE.simple_pointer_values
                      > MODULE.in_simple_pointer_values
                      > MODULE.out_simple_pointer_values
           < PROGRAM.entities
           < MODULE.code
           < PROGRAM.program_simple_pointer_values
           < CALLEES.in_simple_pointer_values
           < CALLEES.out_simple_pointer_values
```

6.14.8 Properties for pointer analyses

The following properties are defined to ensure the safe use of `intraprocedural_points_to_analysis` 6.14.5.

6.14.8.1 Impact of Types

The property `ALIASING_ACROSS_TYPES` 6.14.8.1 specifies that two pointers of different effective types can be aliased. The default and safe value is `TRUE`; when it is turned to `FALSE` two pointers of different types are never aliased.

```
ALIASING_ACROSS_TYPES TRUE
```

The property `ALIASING_ACROSS_FORMAL_PARAMETERS` 6.14.8.1 is used to handle the aliasing between formal parameters and global variables of pointer type. When it is set to `TRUE`, two formal parameters or a formal one and a global pointer or two global pointers can be aliased. If it is turned to `FALSE`, such pointers are assumed to be unaliased for intraprocedural analysis and generally for root module(i.e. modules without callers). The default value is `FALSE`. It is the only value currently implemented.

```
ALIASING_ACROSS_FORMAL_PARAMETERS FALSE
```

The nest property specifies that one data structure can recursively contain two pointers pointing to the same location. If it is turned to `FALSE`, it is assumed that two different not included memory access paths cannot point to the same memory locations. The safe value is `TRUE`, but parallelization is hindered. Often, the user can guarantee that data structures do not exhibit any sharing. Optimistically, `FALSE` is the default value.

```
ALIASING_INSIDE_DATA_STRUCTURE FALSE
```

Property `ALIASING_ACROSS_IO_STREAMS` 6.14.8.1 can be set to `FALSE` to specify that two io streams (two variables declared as `FILE *`) cannot be aliased, neither the locations to which they point. The safe and default value is `TRUE`

```
ALIASING_ACROSS_IO_STREAMS TRUE
```

6.14.8.2 Heap Modeling

The following string property defines the lattice of maximal elements to use when precise information is lost. Three values are possible: "unique", "function" and "area". The first value is the default value. A unique identifier is defined to represent any set of unknown locations. The second value defines a separate identifier for each function and compilation unit. Note that compilation units require more explanation about this definition and about the conflict detection scheme. The third value, "area", requires a separate identifier for each area of each function or compilation unit. These abstract lcoation lattice values are further refined if the property `ALIASING_ACROSS_TYPES` 6.14.8.1 is set to FALSE. The abstract location API hides all these local maximal values from its callers. Note that the dereferencing of any such top abstract location returns the very top of all abstract locations.

The `ABSTRACT_HEAP_LOCATIONS` 6.14.8.2 specifies the modeling of the heap. The possible values are "unique", "insensitive", "flow-sensitive" and "context-sensitive". Each value defines a strictly refined analysis with respect to analyses defined by previous values [This may not be a good idea, since flow and context sensitivity are orthogonal].

The default value, "unique", implies that the heap is a unique array. It is enough to parallelize simple loops containing pointer-based references such as "p[i]".

In the "insensitive" case and all other cases, one array is allocated in each function to modelize the heap.

In the "flow-sensitive" case, the statement numbers of the malloc() call sites are used to subscribe this array, as well as all indices of the surrounding loops [Two improvements in one property...]. Only the first half of the property is implemented.

In the "context_sensitive" case, the interprocedural translation of memory acces paths based on the abstract heap are prefixed by the same information regarding the call site: function containing the call site, statement number of the call site and indices of surrounding loops. This is not implemented.

Note that the naming of options is not fully compatible with the usual notations in pointer analyses. Note also that the insensitive case is redundant

with context sensitive case: in the later case, a unique heap associated to malloc() would carry exactly the same amount of information [flow and context sensitivity are orthogonal].

Finally, note that abstract heap arrays are distinguished according to their types if the property ALIASING_ACROSS_TYPES 6.14.8.1 is set to FALSE [impact on abstract heap location API]. Else, the heap array is of type unknown. If a heap abstract location is dereferenced without any point-to information nor heap aliasing information, the safe result is the top abstract location.

```
ABSTRACT_HEAP_LOCATIONS "unique"
```

Property POINTS_TO_SUCCESSFUL_MALLOC_ASSUMED 6.14.8.2 is used to control the analysis of a malloc call. The call may return either a unique target in the heap, or a pair of targets, one in the heap and NULL. The default value is true for historical reasons and because the result is shorter and correct almost all the time.

```
POINTS_TO_SUCCESSFUL_MALLOC_ASSUMED TRUE
```

6.14.8.3 Type Handling

The property POINTS_TO_STRICT_POINTER_TYPES 6.14.8.3 is used to handle pointer arithmetic. According to C standard(section 6.5.6, item 8) the following C code :

```
int *p, i;
p = \&i;
p++ ;
```

is correct and p points to the same area, expressed by the points to analysis as i[*]. The default value is FALSE, meaning that p points to an array element. When it's set to TRUE typing becone strict ; meaning that p points to an integer and the behavior is undefined. So the analysis stops with a pips_user_error(illegal pointer arithmetic)

```
POINTS_TO_STRICT_POINTER_TYPES FALSE
```

6.14.8.4 Dereferenceing of Null and Undefined Pointers

The property POINTS_TO_UNINITIALIZED_POINTER_DEREFERENCING 6.14.8.4 specifies what to do when an uninitialized pointer is or may be dereferenced. The safe value is FALSE. The points-to analysis assumed that no undefined pointer is ever dereferenced. So if a pointer may be undefined and is dereferenced, the arc is considered impossible and removed from the points-to information. If not other arc provides some value for this pointer, the code is assumed dead and the current points-to set is reduced to the empty set. A warning about dead code is emitted. However the property can be set to TRUE and the dereferencing of an undefined pointer is accepted and results in an anywhere location.

```
POINTS_TO_UNINITIALIZED_POINTER_DEREFERENCING FALSE
```

The property POINTS_TO_NULL_POINTER_DEREFERENCING 6.14.8.4 is very similar to the previous one. It specifies what to do when an null pointer is or may be dereferenced. The safe value is FALSE. The points-to analysis assumed that no null pointer is ever dereferenced. So if a pointer may be undefined and is dereferenced, the arc is considered impossible and removed from the points-to information. If not other arc provides some value for this pointer, the code is assumed dead and the current points-to set is reduced to the empty set. A warning about dead code is emitted. However the property can be set to TRUE and the dereferencing of an undefined pointer is accepted and results in an anywhere location.

```
POINTS_TO_NULL_POINTER_DEREFERENCING FALSE
```

The property POINTS_TO_NULL_POINTER_INITIALIZATION 6.14.8.4 allows the initialization of pointers that are formal parameters or global variables to NULL when computing a calling context. The most accurate property value is TRUE, which makes sure that generated points-to stubs are different from NULL because two arcs are always generated: an arc towards the new points-to stub and an arc towards the NULL location. Thus it prevents from dereferencing a null pointer when dereferencing a points-to stub and it allows the comparison of two points-to stubs when a condition such as p!=q is interpreted or the comparison of one points-to stub to NULL as in p!=NULL. This property must be set to TRUE for the points-to analysis to return valid results since the constant path lattice used implies that NULL is not included in points-to stubes. Also, setting it to FALSE make any formal recursive data structures infinite since NULL is never found by the analyzer. Basically, this property should be removed.

```
POINTS_TO_NULL_POINTER_INITIALIZATION TRUE
```

6.14.8.5 Limits of Points-to Analyses

The integer property POINTS_TO_OUT_DEGREE_LIMIT 6.14.8.5 specifies the maximum number of arcs exiting a given vertex of a poins-to graph. When the maximum out degree is reached for a given source vertex, all the corresponding sink vertices are fused into one new vertex, the minimal upper bound of the initial vertices according to the abstract address lattice, and the points-to graph is updated accordingly. New nodes are created as long as the limit is not reached. The freeing of a list spine can generate an unbounded out degree (see for instance Pointers/list05.c).

```
POINTS_TO_OUT_DEGREE_LIMIT 5
```

The integer property POINTS_TO_PATH_LIMIT 6.14.8.5 specifies the maximum number of occurences of an object of a given type in a non-cyclic path generated by the points-to graph. New nodes are created as long as no such path exists. When the limit is reached, a cycle is created.

```
POINTS_TO_PATH_LIMIT 2
```

The integer property POINTS_TO_SUBSCRIPT_LIMIT 6.14.8.5 specifies the maximum number of subscript of an object can be generated via pointer arithmetic. When the limit is reached, an unbounded subscript, *, is used to model any possible subscript value.

```
POINTS_TO_SUBSCRIPT_LIMIT 2
```

6.14.9 Menu for Alias Views

```
alias alias_file 'Alias View'

alias print_in_alias_pairs 'In Alias Pairs'
alias print_out_alias_pairs 'Out Alias Pairs'
alias print_alias_lists 'Alias Lists'
alias print_alias_classes 'Alias Classes'
```

Display the dynamic alias pairs (formal region, actual region) for the IN regions of the module.

```
print_in_alias_pairs > MODULE.alias_file
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.in_alias_pairs
```

Display the dynamic alias pairs (formal region, actual region) for the OUT regions of the module.

```
print_out_alias_pairs > MODULE.alias_file
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.out_alias_pairs
```

Display the transitive closure of the dynamic aliases for the module.

```
print_alias_lists > MODULE.alias_file
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.alias_lists
```

Display the dynamic alias equivalence classes for this module and those below it in the callgraph.

```
print_alias_classes > MODULE.alias_file
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.alias_classes
```

6.15 Complementary Sections

```
alias compsec 'Complementary Sections'
```

A new representation of array regions added in PIPS by Manjunathaiah Muniyappa. This anlysis is not maintained anymore.

6.15.1 READ/WRITE Complementary Sections

This function computes the complementary sections in a module.

```
complementary_sections > MODULE.compsec
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < CALLEES.summary_compsec
```

6.15.2 Summary READ/WRITE Complementary Sections

```
summary_complementary_sections > MODULE.summary_compsec
        < PROGRAM.entities
        < MODULE.code
        < MODULE.compsec
```

Chapter 7

Dynamic Analyses (Instrumentation)

Dynamic analyses are performed at run-time. At compile-time, the property than can be proved or disproved are exploited, but in doubt a run-time is added to the source code. The current dynamic analyses implemented in PIPSare array bound checking, Fortran alias and used-before-set analyses.

7.1 Array Bound Checking

Array bound checking refers to determining whether all array references are within their declared range in all of their uses in a program. These array bound checks may be analysed intraprocedurally or interprocedurally, depending on the need for accuracy.

There are two versions of intraprocedural array bounds checking: array bound check bottom up, array bound check top down. The first approach relies on checking every array access and on the elimination of redundant tests by advanced dead code elimination based on preconditions. The second approach is based on exact convex array regions. They are used to prove that all accessed in a compound statement are correct.

These two dynamic analyses are implemented for Fortran. They are described in Nga Nguyen's PhD (see [42]) and in [43]. They may work for C code, but this has not been validated.

7.1.1 Elimination of Redundant Tests: Bottom-Up Approach

This transformation takes as input the current module, adds array range checks (lower and upper bound checks) to every statement that has one or more array accesses. The output is the module with those added tests.

If one test is trivial or exists already for the same statement, it is no need to be generated in order to reduce the number of tests. As Fortran language permits an assumed-size array declarator with the unbounded upper bound of the last dimension, no range check is generated for this case also.

Associated with each test is a bound violation error message and in case of real access violation, a STOP statement will be put before the current statement.

This phase should always be followed by the `partial_redundancy_elimination` 9.2.2 for logical expression in order to reduce the number of bound checks.

```
alias array_bound_check_bottom_up 'Elimination of Redundant Tests'
array_bound_check_bottom_up            > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

7.1.2 Insertion of Unavoidable Tests

This second implementation is based on the array region analyses phase which benefits some interesting proven properties:

1. If a MAY region correspond to one node in the control flow graph that represents a block of code of program is included in the declared dimensions of the array, no bound check is needed for this block of code.

2. If a MUST region correspond to one node in the control flow graph that represents a block of code of program contains elements which are outside the declared dimensions of the array, there is certainly bound violation in this block of code. An error can be detected just in compilation time.

If none of these two properties are satisfied, we consider the approximation of region. In case of MUST region, if the exact bound checks can be generated, they will be inserted before the block of code. If not, like in case of MAY region, we continue to go down to the children nodes in the control flow graph.

The main advantage of this algorithm is that it permits to detect the sure bound violations or to tell that there is certainly no bound violation as soon as possible, thanks to the context given by preconditions and the top-down analyses.

```
alias array_bound_check_top_down 'Insertion of Unavoidable Tests'
array_bound_check_top_down   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
```

7.1.3 Interprocedural Array Bound Checking

This phase checks for out of bound errors when passing arrays or array elements as arguments in procedure call. It ensures that there is no bound violation in every array access in the callee procedure, with respect to the array declarations in the caller procedure.

```
alias array_bound_check_interprocedural 'Interprocedural Array Bound Checking'
array_bound_check_interprocedural             > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
```

7.1.4 Array Bound Checking Instrumentation

We provide here a tool to calculate the number of dynamic bound checks from both initial and PIPS generated code.

These transformations are implemented by Thi Viet Nga NGUYEN (see [42]).

```
alias array_bound_check_instrumentation 'Array Bound Checking Instrumentation'
array_bound_check_instrumentation > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Array bounds checking refers to determining whether all array reference are within their declared range in all of its uses in a program. Here are array bounds checking options for code instrumentation, in order to compute the number of bound checks added. We can use only one property for these two case, but the meaning is not clear. To be changed ?

```
INITIAL_CODE_ARRAY_BOUND_CHECK_INSTRUMENTATION TRUE
```

```
PIPS_CODE_ARRAY_BOUND_CHECK_INSTRUMENTATION FALSE
```

In practice, bound violations may often occur with arrays in a common block. The standard is violated, but programmers think that they are not dangerous because the allocated size of the common is not reached. The following property deals with this kind of bad programming practice. If the array is a common variable, it checks if the reference goes beyond the size of the common block or not.

```
ARRAY_BOUND_CHECKING_WITH_ALLOCATION_SIZE FALSE
```

The following property tells the verification phases (array bound checking, alias checking or uninitialized variables checking) to instrument codes with the STOP or the PRINT message. Logically, if a standard violation is detected, the program will stop immediately. Furthermore, the STOP message gives the partial redundancy elimination phase more information to remove redundant tests occurred after this STOP. However, for the debugging purposes, one may need to display all possible violations such as out-of-bound or used-before-set errors, but not to stop the program. In this case, a PRINT message is chosen. By default, we use the STOP message.

```
PROGRAM_VERIFICATION_WITH_PRINT_MESSAGE FALSE
```

7.2 Alias Verification

7.2.1 Alias Propagation

Aliasing occurs when two or more variables refer to the same storage location at the same program point. Alias analysis is critical for performing most optimizations correctly because we must know for certain that we have to take into account all the ways a location, or the value of a variable, may (or must) be used or changed. Compile-time alias information is also important for program verification, debugging and understanding.

In Fortran 77, parameters are passed by address in such a way that, as long as the actual argument is associated with a named storage location, the called subprogram can change the value of the actual argument by assigning a value to the corresponding formal parameter. So new aliases can be created between formal parameters if the same actual argument is passed to two or more formal parameters, or between formal parameters and global parameters if an actual argument is an object in common storage which is also visible in the called subprogram or other subprograms in the call chain below it.

Both *intraprocedural* and *interprocedural* alias determinations are important for program analysis. Intraprocedural aliases occur due to pointers in languages like LISP, C, C++ or Fortran 90, union construct in C or `EQUIVALENCE` in Fortran. Interprocedural aliases are generally created by parameter passing and by access to global variables, which propagates intraprocedural aliases across procedures and introduces new aliases.

The basic idea for computing interprocedural aliases is to follow all the possible chains of argument-parameters and nonlocal variable-parameter bindings at all call sites. We introduce a *naming memory locations* technique which guarantees the correctness and enhances the precision of data-flow analysis. The technique associates sections, offsets of actual parameters to formal parameters following a certain call path. Precise alias information are computed for both scalar and array variables. The analysis is called alias propagation.

This analysis is implemented by Thi Viet Nga Nguyen (see [42]).

```
alias_propagation           > MODULE.alias_associations
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.alias_associations
        < CALLERS.code
```

7.2.2 Alias Checking

With the call-by-reference mechanism in Fortran 77, new aliases can be created between formal parameters if the same actual argument is passed to two or more formal parameters, or between formal parameters and global parameters if an actual argument is an object in common storage which is also visible in the called subprogram or other subprograms in the call chain below it.

Restrictions on association of entities in Fortran 77 (Section 15.9.3.6 [7]) say that neither aliased formal parameters nor the variable in the common block may become defined during execution of the called subprogram or the others subprograms in the call chain.

This phase uses information from the `alias_propagation` 7.2.1 analysis and computes the definition informations of variables in a program, and then to verify statically if the program violates the standard restriction on alias or not. If these informations are not known at compile-time, we instrument the code with tests that check the violation dynamically during execution of program.

This verification is implemented by Thi Viet Nga Nguyen (see [42]).

```
alias alias_check 'Alias Check'
```

```
alias_check   > MODULE.code
        < PROGRAM.entities
        < MODULE.alias_associations
        < MODULE.cumulated_effects
        < ALL.code
```

This is a property to control whether the alias propagation and alias checking phases use information from MAIN program or not. If the current module is never called by the main program, we do no alias propagation and alias checking for this module if the property is on. However, we can do nothing with modules that have no callers at all, because this is a top-down approach.

```
ALIAS_CHECKING_USING_MAIN_PROGRAM FALSE
```

7.3 Used Before Set

This analysis checks if the program uses a variable or an array element which has not been assigned a value. In this case, anything may happen: the program may appear to run normally, or may crash, or may behave unpredictably. We use IN regions that give a set of read variables not previously written. Depending on the nature of the variable: local, formal or global, we have different cases. In principle, it works as follows: if we have a MUST IN region at the module statement, the corresponding variable must be used before being defined, a STOP is inserted. Else, we insert an initialization function and go down, insert a verification function before each MUST IN at each sub-statements.

This is a top-down analysis that process a procedure before all its callees. Information given by callers is used to verify if we have to check for the formal parameters in the current module or not. In addition, we produce information in the resource MODULE.ubs to tell if the formal parameters of the called procedures have to be checked or not.

This verification is implemented by Thi Viet Nga Nguyen (see [42]).

```
alias used_before_set 'Used Before Set'
used_before_set   > MODULE.ubs
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < CALLERS.ubs
```

Chapter 8

Parallelization, Distribution and Code Generation

8.1 Code Parallelization

`PIPS` basic parallelization function, `rice_all_dependence` 8.1.3, produces a new version of the Module code with DOALL loops exhibited using Allen & Kennedy's algorithm. The DOALL syntactic construct is non-standard but easy to understand and usual in text book like [54]. As parallel prettyprinter option, it is possible to use Fortran 90 array syntax (see Section 10.4). For C, the loops can be output as `for`-loop decorated with OpenMP pragma.

Remember that Allen & Kennedy's algorithm can only be applied on loops with simple bodies, i.e. sequences of assignments, because it performs loop distribution and loop regeneration without taking control dependencies into account. If the loop body contains tests and branches, the coarse grain parallelization algorithm should be used (see 8.1.6).

Loop index variables are privatized whenever possible, using a simple algorithm. Dependence arcs related to the index variable and stemming from the loop body must end up inside the loop body. Else, the loop index is not privatized because its final value is likely to be needed after the loop end and because no copy-out scheme is supported.

A better privatization algorithm for all scalar variable may be used as a preliminary code transformation. An array privatizer is also available (see Section 9.7.11). A non-standard `PRIVATE` declaration is used to specify which variables should be allocated on stack for each loop iteration. An HPF or OpenMP format can also be selected.

Objects of type `parallelized_code` differs from objects of type `code` for historic reasons, to simplify the user interface and because most algorithms cannot be applied on `DOALL` loops. This used to be true for pre-condition computation, dependence testing and so on... It is possible neither to re-analyze parallel code, nor to re-parse it (although it would be interesting to compute the complexity of a parallel code) right now but it should evolves. See § 8.1.8.

8.1.1 Parallelization properties

There are few properties that control the parallelization behaviour.

8.1.1.1 Properties controlling Rice parallelization

TRUE to make all possible parallel loops, FALSE to generate real (vector, innermost parallel?) code:

```
GENERATE_NESTED_PARALLEL_LOOPS TRUE
```

Show statistics on the number of loops parallelized by pips:

```
PARALLELIZATION_STATISTICS FALSE
```

To select whether parallelization and loop distribution is done again for already parallel loops:

```
PARALLELIZE_AGAIN_PARALLEL_CODE FALSE
```

The motivation is we may want to parallelize with a coarse grain method first, and finish with a fine grain method here to try to parallelize what has not been parallelized. When applying *à la* Rice parallelizing to parallelize some (still) sequential code, we may not want loop distribution on already parallel code to preserve cache resources, etc.

Thread-safe libraries are protected by critical sections. Their functions can be called safely from different execution threads. For instance, a loop whose body contains calls to `malloc` can be parallelized. The underlying state changes do no hinder parallelization, at least if the code is not sensitive to pointer values.

```
PARALLELIZATION_IGNORE_THREAD_SAFE_VARIABLES FALSE
```

Since this property is used to mask arcs in the dependence graph, it must be exploited by each parallelization phase independently. It is not used to derived a simplified version of the use-def chains or of the dependence graph to avoid wrong result with use-def elimination, which is based on the same graph.

8.1.2 Menu for Parallelization Algorithm Selection

Entries in menu for the resource `parallelized_code` and for the different parallelization algorithms with may be activated or selected. Note that the nest parallelization algorithm is not debugged.

```
alias parallelized_code 'Parallelization'

alias rice_all_dependence 'All Dependences'
alias rice_data_dependence 'True Dependences Only'
alias rice_cray 'CRAY Microtasking'
alias nest_parallelization 'Loop Nest Parallelization'
alias coarse_grain_parallelization 'Coarse Grain Parallelization'
alias internalize_parallel_code 'Consider a parallel code as a sequential one'
```

8.1.3 Allen & Kennedy's Parallelization Algorithm

Use Allen & Kennedy's algorithm and consider all dependences.

```
rice_all_dependence             > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code MODULE.dg
```

8.1.4 Def-Use Based Parallelization Algorithm

Several other parallelization functions for shared-memory target machines are available. Function `rice_data_dependence` 8.1.4 only takes into account data flow dependences, a.k.a true dependences. It is of limited interest because transitive dependences are computed. It is not equivalent at all to performing array and scalar expansion based on direct dependence computation (Brandes, Feautrier, Pugh). It is not safe when privatization is performed before parallelization.

This phase is named after the historical classification of data dependencies in output dependence, anti-dependence and true or data dependence. It should not be used for standard parallelization, but only for experimental parallelization by knowledgeable users, aware that the output code may be illegal.

```
rice_data_dependence            > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code MODULE.dg
```

8.1.5 Parallelization and Vectorization for Cray Multiprocessors

Function `rice_cray` 8.1.5 targets Cray vector multiprocessors. It selects one outermost parallel loop to use multiple processors and one innermost loop for the vector units. It uses Cray microtasking directives. Note that a prettyprinter option must also be selected independently (see Section 10.4).

```
rice_cray                   > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code MODULE.dg
```

8.1.6 Coarse Grain Parallelization

Function `coarse_grain_parallelization` 8.1.6 implements a loop parallelization algorithm based on convex array regions. It considers only one loop at a time, its body being abstracted by its invariant read and write regions. No loop distribution is performed, but any kind of loop body is acceptable whereas Allen & Kennedy algorithm only copes with very simple loop bodies.

For nasty reasons about effects that are statement addresses to effects mapping, this pass changes the code instead of producing a `parallelized_code` resource. It is not a big deal since often we want to modify the code again and we should use `internalize_parallel_code` 8.1.8 just after if its behavior were modified.

```
coarse_grain_parallelization > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.preconditions
        < MODULE.inv_regions
```

Function `coarse_grain_parallelization_with_reduction` 8.1.6 extend the standard `coarse_grain_parallelization` 8.1.6 by using reduction detection informations.

```
coarse_grain_parallelization_with_reduction > MODULE.reduction_parallel_loops
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.cumulated_reductions
        < MODULE.proper_reductions
        < MODULE.inv_regions
```

8.1.7 Global Loop Nest Parallelization

Function `nest_parallelization` 8.1.7 is an attempt at combining loop transformations and parallelization for perfectly nested loops. Different parameters are computed like loop ranges and contiguous directions for references. Loops with small ranges are fully unrolled. Loops with large ranges are strip-mined to obtain vector and parallel loops. Loops with medium ranges simply are parallelized. Loops with unknown range also are simply parallelized.

For each loop direction, the amount of spatial and temporal localities is estimated. The loop with maximal locality is chosen as innermost loop.

This algorithm still is in the development stage. It can be tried to check that loops are interchanged when locality can be improved. An alternative for static control section, is to use the interface with PoCC (see Section 10.11). Internship!

```
nest_parallelization                    > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code MODULE.dg
```

8.1.8 Coerce Parallel Code into Sequential Code

To simplify the user interface and to display with one click a parallelized program, programs in PIPS are *parallelized code* instead of standard *code*. As a consequence, parallelized programs cannot be further analyzed and transformed because sequential code and parallelized code do not have the same resource type. Most pipsmake rules apply to *code* but not to *parallelized code*. Unfortunately, improving the parallelized code with some other transformations such as dead-code elimination is also useful. Thus this pseudo-transformation is added to coerce a parallel code into a classical (sequential) one. Parallelization is made an internal code transformation in PIPS with this rule. PV:not clear

Although this is not the effective process, parallel loops are tagged as parallel and loop local variables may be added in a *code* resource because of a previous privatization phase.

If you display the "generated" code, it may not be displayed as a parallel one if the PRETTYPRINT_SEQUENTIAL_STYLE 10.2.22.3.2 is set to a parallel output style (such as omp). Anyway, the information is available in *code*.

Note this transformation may no be usable with some special parallelizations in PIPS such as WP65 or HPFC that generate other resource types that may be quite different.

```
internalize_parallel_code              > MODULE.code
        < MODULE.parallelized_code
```

8.1.9 Detect Computation Intensive Loops

Generate a pragma on each loop that seems to be computation intensive according to a simple cost model.

The computation intensity is derived from the complexity and the memory footprint. It assumes the cost model:

$$execution\_time = startup\_overhead + \frac{memory\_footprint}{bandwidth} + \frac{complexity}{frequency}$$

A loop is marked with pragma COMPUTATION_INTENSITY_PRAGMA 8.1.9 if the communication costs are lower than the execution cost as given by uniform_complexities 6.11.2.

```
computation_intensity > MODULE.code
< MODULE.code
< MODULE.regions
< MODULE.complexities
```

This correspond to the transfer startup overhead. Time unit is the same as in complexities.

```
COMPUTATION_INTENSITY_STARTUP_OVERHEAD 10
```

This corresponds to the memory bandwidth in octet per time unit.

```
COMPUTATION_INTENSITY_BANDWIDTH 100
```

And This is the processor frequency, in operation per time unit.

```
COMPUTATION_INTENSITY_FREQUENCY 1000
```

This is the generated pragma.

```
COMPUTATION_INTENSITY_PRAGMA "pips␣intensive␣loop"
```

Those values have limited meaning here, only their ratio have some. Having COMPUTATION_INTENSITY_FREQUENCY 8.1.9 and COMPUTATION_INTENSITY_BANDWIDTH 8.1.9 of the same magnitude clearly limits the number of generated pragmas...

8.1.10 Limit parallelism using complexity

Parallel loops which are considered as not complex enough are replaced by sequential ones using a simple cost model based on complexity (see uniform_complexities 6.11.2).

```
limit_parallelism_using_complexity > MODULE.code
< MODULE.code
< MODULE.complexities
```

8.1.11 Limit Parallelism in Parallel Loop Nests

This phase restricts the parallelism of parallel do-loop nests by limiting the number of top-level parallel do-loops to be below a given limit. The too many innermost parallel loops are replaced by sequential loops, if any. This is useful to keep enough coarse-grain parallelism and respecting some hardware or optimization constraints. For example on GPU, in CUDA there is a 2D limitation on grids of thread blocks, in OpenCL it is limited to 3D. Of course, since the phase works onto parallel loop nest, it might be interesting to use a parallelizing phase such as `internalize_parallel_code` (see § 8.1.8) or coarse grain parallelization before applying `limit_nested_parallelism`.

```
limit_nested_parallelism          > MODULE.code
        < MODULE.code
```

PIPS relies on the property `NESTED_PARALLELISM_THRESHOLD` 8.1.11 to determine the desired level of nested parallelism.

```
NESTED_PARALLELISM_THRESHOLD 0
```

8.2 SIMDizer for SIMD Multimedia Instruction Set

The SAC project aims at generating efficient code for processors with SIMD extension instruction set such as VMX, SSE4, etc. which are also refered to as Superword Level Parallelism (SLP). For more information, see `https://info.enstb.org/projets/sac`, or better, see Serge Guelton's PhD dissertation.

Some phases use `ACCEL_LOAD` 8.2 and `ACCEL_STORE` 8.2 to generate DMA calls and `ACCEL_WORK` 8.2.

```
ACCEL_LOAD "SIMD_LOAD"
```

```
ACCEL_STORE "SIMD_STORE"
```

```
ACCEL_WORK "SIMD_"
```

8.2.1 SIMD Atomizer

Here is yet another atomizer, based on `new_atomizer` (see Section 9.4.1.2), used to reduce complex statements to three-address code close to assembly code. There are only some minor differences with respect to `new_atomizer`, except that it does not break down *simple* expressions, that is, expressions that are the sum of a reference and a constant such as tt i+1. This is needed to generate code that could potentially be efficient, whereas the original atomizer would most of the time generate inefficient code.

```
alias simd_atomizer 'SIMD Atomizer'

simd_atomizer                     > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Use the SIMD_ATOMIZER_ATOMIZE_REFERENCE 8.2.1 property to make the SIMD Atomizer go wild: unlike other atomizer, it will break the content of a reference. SIMD_ATOMIZER_ATOMIZE_LHS 8.2.1 can be used to tell the atomizer to atomize both lhs and rhs.

```
SIMD_ATOMIZER_ATOMIZE_REFERENCE FALSE
```

```
SIMD_ATOMIZER_ATOMIZE_LHS FALSE
```

The SIMD_OVERRIDE_CONSTANT_TYPE_INFERENCE 8.2.1 property is used by the SAC library to know if it must override C constant type inference. In C, an integer constant always as the minimum size needed to hold its value, starting from an **int**. In sac we may want to have it converted to a smaller size, in situation like **char** b;/*...*/;**char** a = 2 + b;. Otherwise the result of 2+b is considered as an **int**. if SIMD_OVERRIDE_CONSTANT_TYPE_INFERENCE 8.2.1 is set to TRUE, the result of 2+b will be a **char**.

```
SIMD_OVERRIDE_CONSTANT_TYPE_INFERENCE FALSE
```

8.2.2 Loop Unrollling for SIMD Code Generation

Tries to unroll the code for making the simdizing process more efficient. It thus tries to compute the optimal unroll factor, allowing to pack the most instructions together. Sensible to SIMDIZER_AUTO_UNROLL_MINIMIZE_UNROLL 8.2.11.1 and SIMDIZER_AUTO_UNROLL_SIMPLE_CALCULATION 8.2.11.1.

```
alias simdizer_auto_unroll 'SIMD-Auto Unroll'

simdizer_auto_unroll        > MODULE.code
< PROGRAM.simd_treematch
< PROGRAM.simd_operator_mappings
        < PROGRAM.entities
        < MODULE.code
```

Similiar to simdizer_auto_unroll 8.2.2 but at the loop level.
Sensible to LOOP_LABEL 9.1.1.

```
loop_auto_unroll        > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

8.2.3 Tiling for SIMD Code Generation

Tries to tile the code to make the simdizing process more efficient.
Sensible to LOOP_LABEL 9.1.1 to select the loop nest to tile.

```
simdizer_auto_tile        > MODULE.code
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.code
```

8.2.4 Preprocessing of Reductions for SIMD Code Generation

This phase tries to pre-process reductions, so that they can be vectorized efficiently by the `simdizer` 8.2.10 phase. When multiple reduction statements operating on the same variable with the same operation are detected inside a loop body, each "instance" of the reduction is renamed, and some code is added before and after the loop to initialize the new variables and compute the final result.

```
alias simd_remove_reductions 'SIMD Remove Reductions'

simd_remove_reductions      > MODULE.code
                            > MODULE.callees
        ! MODULE.simdizer_init
        < PROGRAM.entities
        < MODULE.cumulated_reductions
        < MODULE.code
        < MODULE.dg
```

```
SIMD_REMOVE_REDUCTIONS_PREFIX "RED"
```

```
SIMD_REMOVE_REDUCTIONS_PRELUDE ""
```

```
SIMD_REMOVE_REDUCTIONS_POSTLUDE ""
```

8.2.5 Redundant Load-Store Elimination

Remove useless load store calls (and more).

```
redundant_load_store_elimination      > MODULE.code
> MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.out_regions
        < MODULE.chains
```

If `REDUNDANT_LOAD_STORE_ELIMINATION_CONSERVATIVE` 8.2.5 is set to false, `redundant_load_store_elimination` 8.2.5 will remove any statement not implied in the computation of out regions, otherwise it will not remove statement that modifies aprameters reference.

```
REDUNDANT_LOAD_STORE_ELIMINATION_CONSERVATIVE TRUE
```

8.2.6 Undo Some Atomizer Transformations (?)

...

```
alias deatomizer 'Deatomizer'

deatomizer                  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
```

8.2.7 If Conversion

This phase is the first phase of the if-conversion algorithm. The complete if conversion algorithm is performed by applying the three following phase: `if_conversion_init` 8.2.7, `if_conversion` 8.2.7 and `if_conversion_compact` 8.2.7.

Use `IF_CONVERSION_INIT_THRESHOLD` 8.2.7 to control whether if conversion will occur or not: beyhond this number of call, no conversion is done.

```
IF_CONVERSION_INIT_THRESHOLD 40
```

```
alias if_conversion_init 'If-conversion init'

if_conversion_init                  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.summary_complexity
```

This phase is the second phase of the if-conversion algorithm. The complete if conversion algorithm is performed by applying the three following phase: `if_conversion_init` 8.2.7, `if_conversion` 8.2.7 and `if_conversion_compact` 8.2.7.

```
IF_CONVERSION_PHI "__C-conditional__"
```

```
alias if_conversion 'If-conversion'

if_conversion                   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

This phase is the third phase of the *if-conversion* algorithm. The complete if conversion algorithm is performed by applying the three following phase: `if_conversion_init` 8.2.7, `if_conversion` 8.2.7 and `if_conversion_compact` 8.2.7.

```
alias if_conversion_compact 'If-conversion compact'
```

```
if_conversion_compact                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
```

Converts max in loop bounds into tests. Also sets variable so that `simplify_control` 9.3.1 works afterwards.

8.2.8 Loop Unswitching

This phase apply loop distribution on a loop-invariant test. So it transforms a loop with an if/then/else inside into an if/then/else with the loop duplicated into the "then" and "else" branches.

```
loop_nest_unswitching                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

8.2.9 Scalar Renaming

The Scalar Renaming pass tries to minimize dependencies in the code by renaming scalars when legal.

```
scalar_renaming           > MODULE.code
        < PROGRAM.entities
        < MODULE.dg
        < MODULE.proper_effects
```

8.2.10 Tree Matching for SIMD Code Generation

This function initialize a treematch used by `simdizer` 8.2.10 for simd-oriented pattern matching

```
simd_treematcher > PROGRAM.simd_treematch
```

This function initialize operator matchings used by `simdizer` 8.2.10 for simd-oriented pattern matching

```
simd_operator_mappings > PROGRAM.simd_operator_mappings
```

```
simdizer_init  > MODULE.code
< PROGRAM.entities
< MODULE.code
```

Function `simdizer` 8.2.10 is an attempt at generating SIMD code for SIMD multimedia instruction set such as MMX, SSE2, VIS,... This transformation performs the core vectorization, transforming sequences of similar statements into vector operations.

```
alias simdizer 'Generate SIMD code'

simdizer                    > MODULE.code
                            > MODULE.callees
                            > PROGRAM.entities
! MODULE.simdizer_init
< PROGRAM.simd_treematch
< PROGRAM.simd_operator_mappings
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg
```

When set to true, following property tells the simdizer to try to padd arrays when it seems to be profitable

```
SIMDIZER_ALLOW_PADDING FALSE
```

Skip generation of load and stores, using generic functions instead.

```
SIMDIZER_GENERATE_DATA_TRANSFERS TRUE
```

This phase is to be called after simdization of affectation operator. It performs type substitution from char/short array to in array using the packing from the simdization phase For example, four consecutive load from a char array could be a single load from an int array. This prove to be useful for *c to vhdl* compilers such as `c2h`.

```
alias simd_memory_packing 'Generate Optimized Load Store'

simd_memory_packing  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

8.2.11 SIMD properties

This property is used to set the target register size, expressed in bits, for places where this is needed (for instance, auto-unroll with simple algorithm).

```
SAC_SIMD_REGISTER_WIDTH 64
```

8.2.11.1 Auto-Unroll

This property is used to control how the auto unroll phase computes the unroll factor. By default, the minimum unroll factor is used. It is computed by using the minimum of the optimal factor for each statement. If the property is set to FALSE, then the maximum unroll factor is used instead.

```
SIMDIZER_AUTO_UNROLL_MINIMIZE_UNROLL TRUE
```

This property controls how the "optimal" unroll factor is computed. Two algorithms can be used. By default, a simple algorithm is used, which simply compares the actual size of the variables used to the size of the registers to find out the best unroll factor. If the property is set to FALSE, a more complex algorithm is used, which takes into account the actual SIMD instructions.

```
SIMDIZER_AUTO_UNROLL_SIMPLE_CALCULATION TRUE
```

8.2.11.2 Memory Organisation

This property is used by the sac library to know which elements of multi-dimensional array are consecutive in memory. Let us consider the three following references a(i,j,k), a(i,j,k+1) and a(i+1,j,k). Then, if `SIMD_FORTRAN_MEM_ORGANISATION` 8.2.11.2 is set to `TRUE`, it means that a(i,j,k) and a(i+1,j,k) are consecutive in memory but a(i,j,k) and a(i,j,k+1) are not. However, if `SIMD_FORTRAN_MEM_ORGANISATION` 8.2.11.2 is set to `FALSE`, a(i,j,k) and a(i,j,k+1) are consecutive in memory but a(i,j,k) and a(i+1,j,k) are not.

```
SIMD_FORTRAN_MEM_ORGANISATION TRUE
```

8.2.11.3 Pattern file

This property is used by the sac library to know the path of the pattern definition file. If the file is not found, the execution fails.

```
SIMD_PATTERN_FILE "patterns.def"
```

8.3 Code Distribution

Different automatic code distribution techniques are implemented in `PIPS` for distributed-memory machines. The first one is based on the emulation of a shared-memory. The second one is based on HPF. A third one target architectures with hardware coprocessors. Another one is currently developed at IT Sud Paris that generate MPI code from OpenMP one.

8.3.1 Shared-Memory Emulation

WP65[1] [30, 31, 32] produces a new version of a module transformed to be executed on a distributed memory machine. Each module is transformed into two modules. One module, `wp65_compute_file`, performs the computations, while the other one, `wp65_bank_file`, emulates a shared memory.

This rule does not have data structure outputs, as the two new program generated have computed names. This does not fit the `pipsmake` framework too well, but is OK as long as nobody wishes to apply `PIPS` on the generated code, e.g. to propagate constant or eliminate dead code.

Note that use-use dependencies are used to allocate temporary arrays in local memory (i.e. in the software cache).

[1] `http://www.cri.ensmp.fr/pips/wp65.html`

This compilation scheme was designed by Corinne Ancourt and François Irigoin. It uses theoretical results in [6]. Its input is a very small subset of Fortran program (e.g. procedure calls are not supported). It was implemented by the designers, with help from Lei Zhou.

```
alias wp65_compute_file 'Distributed View'
alias wp65_bank_file 'Bank Distributed View'
wp65                            > MODULE.wp65_compute_file
                                > MODULE.wp65_bank_file
        ! MODULE.privatize_module
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
        < MODULE.cumulated_effects
        < MODULE.chains
        < MODULE.proper_effects
```

Name of the file for the target model:

```
WP65_MODEL_FILE "model.rc"
```

8.3.2 HPF Compiler

The *HPF compiler*[2] is a project by itself, developed by Fabien Coelho in the `PIPS` framework.

A whole set of rules is used by the `PIPS` *HPF compiler*[3], *HPFC*[4]. By the way, the whole compiler is just a big hack according to Fabien Coelho.

8.3.2.1 HPFC Filter

The first rule is used to apply a shell to put HPF-directives in an f77 parsable form. Some shell script based on sed is used. The `hpfc_parser` 4.2.2 must be called to analyze the right file. This is triggered automatically by the bang selection in the `hpfc_close` 8.3.2.5 phase.

```
hpfc_filter             > MODULE.hpfc_filtered_file
    < MODULE.source_file
```

8.3.2.2 HPFC Initialization

The second HPFC rule is used to initialize the hpfc status and other data structures global to the compiler. The HPF compiler status is bootstrapped. The compiler status stores (or should store) all relevant information about the HPF part of the program (data distribution, IO functions and so on).

```
hpfc_init               > PROGRAM.entities
                        > PROGRAM.hpfc_status
    < PROGRAM.entities
```

[2] `http://www.cri.ensmp.fr/pips/hpfc.html`
[3] `http://www.cri.ensmp.fr/pips/hpfc.html`
[4] `http://www.cri.ensmp.fr/pips/hpfc.html`

8.3.2.3 HPF Directive removal

This phase removes the directives (some special calls) from the code. The remappings (implicit or explicit) are also managed at this level, through copies between differently shaped arrays.

To manage calls with distributed arguments, I need to apply the directive extraction bottom-up, so that the callers will know about the callees through the `hpfc_status`. In order to do that, I first thought of an intermediate resource, but there was obscure problem with my fake calls. Thus the dependence static then dynamic directive analyses is enforced at the bang sequence request level in the `hpfc_close` 8.3.2.5 phase.

The `hpfc_static_directives` 8.3.2.3 phase analyses static mapping directives for the specified module. The `hpfc_dynamic_directives` 8.3.2.3 phase does manages realigns and function calls with prescriptive argument mappings. In order to do so it needs its callees' required mappings, hence the need to analyze beforehand static directives. The code is cleaned from the `hpfc_filter` 8.3.2.1 artifacts after this phase, and all the proper information about the HPF stuff included in the routines is stored in `hpfc_status`.

```
hpfc_static_directives          > MODULE.code
                        > PROGRAM.hpfc_status
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MODULE.code

hpfc_dynamic_directives          > MODULE.code
                        > PROGRAM.hpfc_status
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MODULE.code
    < MODULE.proper_effects
```

8.3.2.4 HPFC actual compilation

This rule launches the actual compilation. Four files are generated:

1. the host code that mainly deals with I/Os,
2. the SPMD node code,
3. and some initialization stuff for the runtime (2 files).

Between this phase and the previous one, many PIPS standard analyses are performed, especially the regions and preconditions. Then this phase will perform the actual translation of the program into a host and SPMD node code.

```
hpfc_compile            > MODULE.hpfc_host
                        > MODULE.hpfc_node
                        > MODULE.hpfc_parameters
                        > MODULE.hpfc_rtinit
                        > PROGRAM.hpfc_status
    < PROGRAM.entities
```

```
    < PROGRAM.hpfc_status
    < MODULE.regions
    < MODULE.summary_regions
    < MODULE.preconditions
    < MODULE.code
    < MODULE.cumulated_references
    < CALLEES.hpfc_host
```

8.3.2.5 HPFC completion

This rule deals with the compiler closing. It must deal with commons. The hpfc parser selection is put here.

```
hpfc_close              > PROGRAM.hpfc_commons
    ! SELECT.hpfc_parser
    ! SELECT.must_regions
    ! ALL.hpfc_static_directives
    ! ALL.hpfc_dynamic_directives
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MAIN.hpfc_host
```

8.3.2.6 HPFC install

This rule performs the installation of HPFC generated files in a separate directory. This rule is added to make hpfc usable from wpips and epips. I got problems with the make and run rules, because it was trying to recompute everything from scratch. To be investigated later on.

```
hpfc_install            > PROGRAM.hpfc_installation
    < PROGRAM.hpfc_commons

hpfc_make

hpfc_run
```

8.3.2.7 HPFC *High Performance Fortran Compiler* properties

Debugging levels considered by HPFC: `HPFC_{,DIRECTIVES,IO,REMAPPING}_DEBUG_LEVEL`.

These booleans control whether some computations are directly generated in the output code, or computed through calls to dedicated runtime functions. The default is the direct expansion.

```
HPFC_EXPAND_COMPUTE_LOCAL_INDEX TRUE
```

```
HPFC_EXPAND_COMPUTE_COMPUTER TRUE
```

```
HPFC_EXPAND_COMPUTE_OWNER TRUE
```

```
HPFC_EXPAND_CMPLID TRUE
```

```
HPFC_NO_WARNING FALSE
```

Hacks control...

```
HPFC_FILTER_CALLEES FALSE
```

```
GLOBAL_EFFECTS_TRANSLATION TRUE
```

These booleans control the I/O generation.

```
HPFC_SYNCHRONIZE_IO FALSE
```

```
HPFC_IGNORE_MAY_IN_IO FALSE
```

Whether to use lazy or non-lazy communications

```
HPFC_LAZY_MESSAGES TRUE
```

Whether to ignore FCD (Fabien Coelho Directives...) or not. These directives are used to instrument the code for testing purposes.

```
HPFC_IGNORE_FCD_SYNCHRO FALSE
```

```
HPFC_IGNORE_FCD_TIME FALSE
```

```
HPFC_IGNORE_FCD_SET FALSE
```

Whether to measure and display the compilation times for remappings, and whether to generate outward redundant code for remappings. Also whether to generate code that keeps track dynamically of live mappings. Also whether not to send data to a twin (a processor that holds the very same data for a given array).

```
HPFC_TIME_REMAPPINGS FALSE
```

```
HPFC_REDUNDANT_SYSTEMS_FOR_REMAPS FALSE
```

```
HPFC_OPTIMIZE_REMAPPINGS TRUE
```

```
HPFC_DYNAMIC_LIVENESS TRUE
```

```
HPFC_GUARDED_TWINS TRUE
```

Whether to use the local buffer management. 1 MB of buffer is allocated.

```
HPFC_BUFFER_SIZE 1000000
```

```
HPFC_USE_BUFFERS TRUE
```

Whether to use in and out convex array regions for input/output compiling

```
HPFC_IGNORE_IN_OUT_REGIONS TRUE
```

Whether to extract more equalities from a system, if possible.

```
HPFC_EXTRACT_EQUALITIES TRUE
```

Whether to try to extract the underlying lattice when generating code for systems with equalities.

```
HPFC_EXTRACT_LATTICE TRUE
```

8.3.3 STEP: MPI code generation from OpenMP programs

8.3.3.1 STEP Directives

RK: IT Sud-Paris : insert your documentation here; FI: or a pointer towards you documentation

The `step_parser` 8.3.3.1 phase identifies the OpenMP constructs. The directive semantics are stored in the `MODULE.step_directives` ressource.

```
step_parser                 > MODULE.step_directives
                            > MODULE.code
   < MODULE.code
```

8.3.3.2 STEP Analysis

The `step_analyse_init` 8.3.3.2 phase init the `PROGRAM.step_comm` ressources

```
step_analyse_init           > PROGRAM.step_comm
```

The `step_analyse` 8.3.3.2 phase triggers the convex array regions analyses to compute SEND and RECV regions leading to MPI messages and checks whether a given SEND region corresponding to a directive construct is consumed by a RECV region corresponding to a directive construct. In this case, communications can be optimized.

```
step_analyse                > PROGRAM.step_comm
                            > MODULE.step_send_regions
                            > MODULE.step_recv_regions
   < PROGRAM.entities
   < PROGRAM.step_comm
   < MODULE.step_directives
   < MODULE.code
   < MODULE.preconditions
   < MODULE.transformers
   < MODULE.cumulated_effects
   < MODULE.regions
   < MODULE.in_regions
   < MODULE.out_regions
   < MODULE.chains
   < CALLEES.code
   < CALLEES.step_send_regions
   < CALLEES.step_recv_regions
```

8.3.3.3 STEP code generation

Based on the OpenMP construct and analyses, new modules are generated to translate the original code with OpenMP directives. The default code transformation for OpenMP construct is driven by the `STEP_DEFAULT_TRANSFORMATION` 8.3.3.3 property. The different value allowed are :

- `"HYBRID"` : for OpenMP and MPI parallel code
- `"MPI"` : for MPI parallel code
- `"OMP"` : for OpenMP parallel code

```
STEP_DEFAULT_TRANSFORMATION "HYBRID"
```

The `step_compile` 8.3.3.3 phase generates source code for OpenMP constructs depending of the transformation desired. Each OpenMP construct could have a specific transformation define by STEP clauses (without specific clauses, the `STEP_DEFAULT_TRANSFORMATION` 8.3.3.3 is used). The specific STEP clauses allowed are :

- `"!\$step hybrid"` : for OpenMP and MPI parallel code
- `"!\$step no\_mpi"` : for OpenMP parallel code
- `"!\$step mpi"` : for MPI parallel code
- `"!\$step ignore"` : for sequential code

```
step_compile               > MODULE.step_file
   < PROGRAM.entities
   < PROGRAM.step_comm
   < MODULE.step_directives
   < MODULE.code
```

The `step_install` 8.3.3.3 phase copy the generated source files in the directory specified by the `STEP_INSTALL_PATH` 8.3.3.3 property.

```
step_install
   < ALL.step_file
```

```
STEP_INSTALL_PATH ""
```

8.3.4 PHRASE: high-level language transformation for partial evaluation in reconfigurable logic

The PHRASE project is an attempt to automatically (or semi-automatically) transform high-level language programs into code with partial execution on some accelerators such as reconfigurable logic (such as FPGAs) or data-paths.

This phases allow to split the code into portions of code delimited by PHRASE-pragma (written by the programmer) and a control program managing them. Those portions of code are intended, after transformations, to be executed in reconfigurable logic. In the PHRASE project, the reconfigurable logic is synthesized with the Madeo tool that take SmallTalk code as input. This is why we have a SmallTalk pretty-printer (see section 10.10).

8.3.4.1 Phrase Distributor Initialisation

This phase is a preparation phase for the Phrase Distributor `phrase_distributor` 8.3.4.2: the portions of code to externalize are identified and isolated here. Comments are modified by this phase.

```
alias phrase_distributor_init 'PHRASE Distributor initialization'

phrase_distributor_init                  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This phase is automatically called by the following `phrase_distributor` 8.3.4.2.

8.3.4.2 Phrase Distributor

The job of distribution is done here. This phase should be applied after the initialization (Phrase Distributor Initialisation `phrase_distributor_init` 8.3.4.1), so this one is automatically applied first.

```
alias phrase_distributor 'PHRASE Distributor'

phrase_distributor                       > MODULE.code
                                         > MODULE.callees
        ! MODULE.phrase_distributor_init
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.dg
```

8.3.4.3 Phrase Distributor Control Code

This phase add control code for PHRASE distribution. All calls to externalized code portions are transformed into START and WAIT calls. Parameters communication (send and receive) are also handled here

```
alias phrase_distributor_control_code 'PHRASE Distributor Control Code'

phrase_distributor_control_code          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.dg
```

8.3.5 Safescale

The Safescale project is an attempt to automatically (or semi-automatically) transform sequential code written in C language for the Kaapi runtime.

8.3.5.1 Distribution init

This phase is intended for the analysis of a module given with the aim of finding blocks of code delimited by specific pragmas from it.

```
alias safescale_distributor_init 'Safescale distributor init'

safescale_distributor_init                  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

8.3.5.2 Statement Externalization

This phase is intended for the externalization of a block of code.

```
alias safescale_distributor 'Safescale distributor'

safescale_distributor                  > MODULE.code
                                       > MODULE.callees
        ! MODULE.safescale_distributor_init
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
```

8.3.6 CoMap: Code Generation for Accelerators with DMA

8.3.6.1 Phrase Remove Dependences

```
alias phrase_remove_dependences 'Phrase Remove Dependences'

phrase_remove_dependences                      > MODULE.code
                                               > MODULE.callees
        ! MODULE.phrase_distributor_init
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.dg
```

8.3.6.2 Phrase comEngine Distributor

This phase should be applied after the initialization (Phrase Distributor Initialisation or `phrase_distributor_init` 8.3.4.1). The job of comEngine distribution is done here.

```
alias phrase_comEngine_distributor 'PHRASE comEngine Distributor'

phrase_comEngine_distributor                       > MODULE.code
                                                   > MODULE.callees
        ! MODULE.phrase_distributor_init
```

```
< PROGRAM.entities
< MODULE.code
< MODULE.in_regions
< MODULE.out_regions
< MODULE.dg
< MODULE.summary_complexity
```

8.3.6.3 PHRASE ComEngine properties

This property is set to TRUE if we want to synthesize only one process on the HRE.

```
COMENGINE_CONTROL_IN_HRE TRUE
```

This property holds the fifo size of the ComEngine.

```
COMENGINE_SIZE_OF_FIFO 128
```

8.3.7 Parallelization for Terapix architecture

8.3.7.1 Isolate Statement

Isolate the statement given in `ISOLATE_STATEMENT_LABEL` 8.3.7.1 in a separated memory. Data transfer are generated using the same DMA as `kernel_load_store` 8.3.7.5.

The algorithm is based on Read and write regions (no in / out yet) to compute the data that must be copied and allocated. Rectangular hull of regions are used to match allocator and data transfers prototypes. If an analysis fails, definition regions are use instead. If a sizeof is involved, `EVAL_SIZEOF` 9.4.2 must be set to true.

```
isolate_statement > MODULE.code
> MODULE.callees
< MODULE.code
< MODULE.regions
< PROGRAM.entities
```

```
ISOLATE_STATEMENT_LABEL ""
```

As a side effect of isolate_statement pass, some new variables are declared into the function. A prefix can be used for the names of those variables using the property `ISOLATE_STATEMENT_VAR_PREFIX`. It is also possible to insert a suffix using the property `ISOLATE_STATEMENT_VAR_SUFFIX`. The suffix will be inserted between the original variable name and the instance number of the copy.

```
ISOLATE_STATEMENT_VAR_PREFIX ""
```

```
ISOLATE_STATEMENT_VAR_SUFFIX ""
```

By default we cannot isolate a statement with some complex effects on the non local memory. But if we know we can (for example), we can override this behaviour by setting the following property:

```
ISOLATE_STATEMENT_EVEN_NON_LOCAL FALSE
```

8.3.7.2 GPU XML Output

Dump XML for a function, intended to be used for SPEAR. Track back the parameters that are used for the iteration space.

```
gpu_xml_dump > MODULE.gpu_xml_file
         < PROGRAM.entities
         < MODULE.code
```

8.3.7.3 Delay Communications

Optimize the load/store dma by delaying the stores and performing the stores as soon as possible. Interprocedural version.

It uses `ACCEL_LOAD` 8.2 and `ACCEL_STORE` 8.2 to distinguish loads and stores from other calls.

The communication elimination makes the assumption that a load/store pair can always be removed.

```
delay_communications_inter                    > MODULE.code
        > MODULE.callees
        ! CALLEES.delay_communications_inter
        ! MODULE.delay_load_communications_inter
        ! MODULE.delay_store_communications_inter
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.dg

delay_load_communications_inter                  > MODULE.code
        > MODULE.callees
        > CALLERS.code
        > CALLERS.callees
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg

delay_store_communications_inter                   > MODULE.code
        > MODULE.callees
        > CALLERS.code
        > CALLERS.callees
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg
```

Optimize the load/store dma by delaying the stores and performing the stores as soon as possible. Intra Procedural version.

It uses ACCEL_LOAD 8.2 and ACCEL_STORE 8.2 to distinguish loads and stores from other calls.

The communication elimination makes the assumption that a load/store pair can always be removed.

```
delay_communications_intra                    > MODULE.code
        > MODULE.callees
        ! MODULE.delay_load_communications_intra
        ! MODULE.delay_store_communications_intra
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.dg

delay_load_communications_intra                  > MODULE.code
        > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg

delay_store_communications_intra                  > MODULE.code
        > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg
```

8.3.7.4 Hardware Constraints Solver

if SOLVE_HARDWARE_CONSTRAINTS_TYPE 8.3.7.4 is set to **VOLUME**, Given a loop label, a maximum memory footprint and an unknown entity, try to find the best value for SOLVE_HARDWARE_CONSTRAINTS_UNKNOWN 8.3.7.4 to make memory footprint of SOLVE_HARDWARE_CONSTRAINTS_LABEL 8.3.7.4 reach but not exceed SOLVE_HARDWARE_CONSTRAINTS_LIMIT 8.3.7.4. If it is set to **NB_PROC**, it tries to find the best value for SOLVE_HARDWARE_CONSTRAINTS_UNKNOWN 8.3.7.4 to make the maximum range of first dimension of all regions accessed by SOLVE_HARDWARE_CONSTRAINTS_LABEL equals to SOLVE_HARDWARE_CONSTRAINTS_LIMIT 8.3.7.4.

```
solve_hardware_constraints > MODULE.code
< MODULE.code
< MODULE.regions
< PROGRAM.entities
```

```
SOLVE_HARDWARE_CONSTRAINTS_LABEL ""
```

```
SOLVE_HARDWARE_CONSTRAINTS_LIMIT 0
```

```
SOLVE_HARDWARE_CONSTRAINTS_UNKNOWN ""
```

```
SOLVE_HARDWARE_CONSTRAINTS_TYPE ""
```

8.3.7.5 kernelize

Bootstraps the kernel ressource

```
bootstrap_kernels > PROGRAM.kernels
```

Add a kernel to the list of kernels known to pips

```
flag_kernel > PROGRAM.kernels
< PROGRAM.kernels
```

Generate unoptimized load / store information for each call to the module.

```
kernel_load_store > CALLERS.code
> CALLERS.callees
> PROGRAM.kernels
< PROGRAM.kernels
< CALLERS.code
    < CALLERS.regions
        < CALLERS.preconditions
```

The legacy `kernel_load_store` 8.3.7.5 approach is limited because it generates the DMA around a call, and `isolate_statement` 8.3.7.1 engine does not perform well in interprocedural.

The following properties are used to specify the names of runtime functions. Since they are used in Par4All, their default names begin with `P4A_`. To have an idea about their prototype, have a look to the Par4All accelerator runtime or in `validation/AcceleratorUtils/include/par4all.c`.

Enable/disable the scalar handling by kernel load store.

```
KERNEL_LOAD_STORE_SCALAR FALSE
```

The `ISOLATE_STATEMENT_EVEN_NON_LOCAL` 8.3.7.1 property can be used to force the generation even with non local memory access. But beware it would not solve all the issues...

The following properties can be used to customized the allocate/load/store functions:

```
KERNEL_LOAD_STORE_ALLOCATE_FUNCTION "P4A_accel_malloc"
```

```
KERNEL_LOAD_STORE_DEALLOCATE_FUNCTION "P4A_accel_free"
```

The following properties are used to name the DMA functions to use for scalars:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION "P4A_copy_to_accel"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION "P4A_copy_from_accel"
```

and for 1-dimension arrays:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_1D "P4A_copy_to_accel_1d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_1D "P4A_copy_from_accel_1d"
```

and in 2 dimensions:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_2D "P4A_copy_to_accel_2d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_2D "P4A_copy_from_accel_2d"
```

and in 3 dimensions:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_3D "P4A_copy_to_accel_3d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_3D "P4A_copy_from_accel_3d"
```

and in 4 dimensions:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_4D "P4A_copy_to_accel_4d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_4D "P4A_copy_from_accel_4d"
```

and in 5 dimensions:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_5D "P4A_copy_to_accel_5d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_5D "P4A_copy_from_accel_5d"
```

and in 6 dimensions:

```
KERNEL_LOAD_STORE_LOAD_FUNCTION_6D "P4A_copy_to_accel_6d"
```

```
KERNEL_LOAD_STORE_STORE_FUNCTION_6D "P4A_copy_from_accel_6d"
```

As a side effect of kernel load store pass, some new variables are declared into the function. A prefix can be used for the names of those variables using the property KERNEL_LOAD_STORE_VAR_PREFIX 8.3.7.5. It is also possible to insert a suffix using the property KERNEL_LOAD_STORE_VAR_PREFIX 8.3.7.5. The suffix will be inserted between the original variable name and the instance number of the copy.

```
KERNEL_LOAD_STORE_VAR_PREFIX "p4a_var_"
```

```
KERNEL_LOAD_STORE_VAR_SUFFIX ""
```

Split a parallel loop with a local index into three parts: a host side part, a kernel part and an intermediate part. The intermediate part simulates the parallel code to the kernel from the host

```
kernelize > MODULE.code
> MODULE.callees
> PROGRAM.kernels
                ! MODULE.privatize_module
! MODULE.coarse_grain_parallelization
< PROGRAM.entities
< MODULE.code
< PROGRAM.kernels
```

The property KERNELIZE_NBNODES 8.3.7.5 is used to set the number of nodes for this kernel. KERNELIZE_KERNEL_NAME 8.3.7.5 is used to set the name of generated kernel. KERNELIZE_HOST_CALL_NAME 8.3.7.5 is used to set the name of generated call to kernel (host side).

```
KERNELIZE_NBNODES 128
```

```
KERNELIZE_KERNEL_NAME ""
```

```
KERNELIZE_HOST_CALL_NAME ""
```

```
OUTLINE_LOOP_STATEMENT FALSE
```

Gather all constants from a module and put them in a single array. Relevant for Terapix code generation, and maybe for other accelerators as well

```
group_constants > MODULE.code
< PROGRAM.entities
< MODULE.code
< MODULE.regions
```

You may want to group constants only for a particular statement, in that case use GROUP_CONSTANTS_STATEMENT_LABEL 8.3.7.5

```
GROUP_CONSTANTS_STATEMENT_LABEL ""
```

The way variables are grouped is control by GROUP_CONSTANTS_LAYOUT 8.3.7.5, the only relevant value as of now is "terapix".

```
GROUP_CONSTANTS_LAYOUT ""
```

The name of the variable holding constants can be set using GROUP_CONSTANTS_HOLDER 8.3.7.5.

```
GROUP_CONSTANTS_HOLDER "caillou"
```

You may want to skip loop bounds from the grouping

```
GROUP_CONSTANTS_SKIP_LOOP_RANGE FALSE
```

You may want to skip litterals too.

```
GROUP_CONSTANTS_LITERAL TRUE
```

Perform various checks on a Terapix microcode to make sure it can be synthesized. GROUP_CONSTANTS_HOLDER 8.3.7.5 is used to differentiate mask and image.

```
normalize_microcode > MODULE.code
                                > CALLERS.code
                                > COMPILATION_UNIT.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.callers
        < MODULE.cumulated_effects
```

normalize array access in a loop nest for terapixification.

This pass is meaningless for any other target :(.

```
terapix_warmup > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

converts divide operator into multiply operator using formula $a/cste = a * (1/b) \simeq a * (128/cste)/128$

```
terapix_remove_divide > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Use this property for accuracy of divide to multiply conversion.

```
TERAPIX_REMOVE_DIVIDE_ACCURACY 4
```

8.3.7.6 Communication Generation

This phase computes the mapping of data on the accelarators. It records the set of data that have to be copied on the GPU before each statement in the module, and the set of data that have to be copied back from the GPU after the execution of each statement.

Then according to this information, the copy-in and copy-out transfers are generated using same set of properties as `kernel_load_store` 8.3.7.5.

This work has been described in [4][3]. However, the implementation is more complex than the published equations because of PIPS' HCFG and because of a heuristic to generate transfers as high as possible in the HCFG.

```
alias kernel_data_mapping "Kernel data mapping"
kernel_data_mapping > MODULE.kernel_copy_in
    > MODULE.kernel_copy_out
        < PROGRAM.entities
        < PROGRAM.kernels
        < MODULE.code
        < MODULE.summary_effects
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
```

```
        < MODULE.callees
        < MODULE.callers
        < CALLEES.kernel_copy_out
        < CALLEES.kernel_copy_in
```

This phase wrap argument at call site with an access function. The wrapper name is controlled with `WRAP_KERNEL_ARGUMENT_FUNCTION_NAME` 8.3.7.6. Currently the purpose of this is to insert call to a runtime to resolve addresses in accelerator memory corresponding to addresses in host memory.

```
WRAP_KERNEL_ARGUMENT_FUNCTION_NAME "P4A_runtime_host_ptr_to_accel_ptr"
```

```
alias wrap_kernel_argument "wrap_kernel_argument"
wrap_kernel_argument > CALLERS.code
   > CALLERS.callees
        < PROGRAM.entities
        < CALLERS.code
        < CALLERS.callees
        < MODULE.callers
```

8.3.8 Code Distribution on GPU

This phase generate GPU kernels from perfect parallel loop nests. `GPU_IFY_ANNOTATE_LOOP_NESTS` 8.3.8 property triggers automatically the annotation of the loop nest (see `gpu_loop_nest_annotate` 8.3.8).

```
GPU_IFY_ANNOTATE_LOOP_NESTS FALSE
```

```
alias gpu_ify 'Distribute // loop nests on GPU'
gpu_ify                    > MODULE.code
> MODULE.callees
> PROGRAM.entities
< MODULE.privatized
< PROGRAM.entities
< MODULE.code
        < MODULE.cumulated_effects
```

For example from

```
for(i = 1; i <= 499; i += 1)
   for(j = 1; j <= 499; j += 1)
      save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]+space[i][j-1]+space[i][
```

it generates something like

```
p4a_kernel_launcher_0(save, space);

[...]
void p4a_kernel_launcher_0(float_t save[501][501], float_t space[501][501])
{
  int i;
  int j;
```

```
for(i = 1; i <= 499; i += 1)
   for(j = 1; j <= 499; j += 1)

      p4a_kernel_wrapper_0(save, space, i, j);
}

void p4a_kernel_wrapper_0(float_t save[501][501], float_t space[501][501], i
{
  i = P4A_pv_0(i);
  j = P4A_pv_1(j);
  p4a_kernel_0(save, space, i, j);
}
void p4a_kernel_0(float_t save[501][501], float_t space[501][501], int
                  i, int j) {
  save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]+space[i][j-1]+space[i][j+1
}
```

The launcher, wrapper and kernel prefix names to be used during the generation:

```
GPU_LAUNCHER_PREFIX "p4a_launcher"
```

```
GPU_WRAPPER_PREFIX "p4a_wrapper"
```

```
GPU_KERNEL_PREFIX "p4a_kernel"
```

This boolean property control wherever the outliner use the original function name as a suffix instead of only numerical suffix.

```
GPU_OUTLINE_SUFFIX_WITH_OWNER_NAME TRUE
```

For Fortran output you may need to have these prefix name in uppercase.

Indeed, each level of outlining can be enabled or disabled according to the following properties:

```
GPU_USE_LAUNCHER TRUE
```

```
GPU_USE_WRAPPER TRUE
```

```
GPU_USE_KERNEL TRUE
```

Each generated function can go in its own source file according to the following properties:

```
GPU_USE_KERNEL_INDEPENDENT_COMPILATION_UNIT FALSE
```

```
GPU_USE_LAUNCHER_INDEPENDENT_COMPILATION_UNIT FALSE
```

```
GPU_USE_WRAPPER_INDEPENDENT_COMPILATION_UNIT FALSE
```

By default they are set to `FALSE` for languages like CUDA that allow kernel and host codes mixed in a same file but for OpenCL it is not the case.

When the original code is in Fortran it might be useful to wrap the kernel launcher in an independent C file. The `GPU_USE_FORTRAN_WRAPPER` 8.3.8 can be used for that purpose. The name of the function wrapper can be configured using the property `GPU_FORTRAN_WRAPPER_PREFIX` 8.3.8. As specified before it is safe to use prefix name in uppercase.

```
GPU_USE_FORTRAN_WRAPPER FALSE
```

```
GPU_FORTRAN_WRAPPER_PREFIX "P4A_FORTRAN_WRAPPER"
```

The phase generates a wrapper function to get the iteration coordinate from intrinsics functions instead of the initial loop indices. Using this kind of wrapper is the normal behaviour but for simulation of an accelerator code, not using a wrapper is useful.

The intrinsics function names to get an i^{th} coordinate in the iteration space are defined by this GNU *à la* `printf` format:

```
GPU_COORDINATE_INTRINSICS_FORMAT "P4A_vp_%d"
```

where `%d` is used to get the dimension number. Here `vp` stands for *virtual processor* dimension and is a reminiscence from PompC and HyperC...

Please, do not use this feature for buffer-overflow attack...

Annotates loop nests with comments and guards for further generation of CUDA calls.

```
alias gpu_loop_nest_annotate 'Decorate loop nests with iteration spaces and add iteration

gpu_loop_nest_annotate         > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

To annotate only outer parallel loop nests, set the following variable to true:

```
GPU_LOOP_NEST_ANNOTATE_PARALLEL TRUE
```

Clear annotation previously added by `gpu_loop_nest_annotate` 8.3.8.

```
alias gpu_clear_annotations_on_loop_nest 'Clear annotation previously added by gpu_loop_ne

gpu_clear_annotations_on_loop_nest   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Parallelize annotated loop nests based on the sentinel comments.

```
gpu_parallelize_annotated_loop_nest        > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This phase promote a whole function body into a parallel loop with one thread.

```
one_thread_parallelize > MODULE.code
> PROGRAM.entities
< PROGRAM.entities
< MODULE.code
```

This phase promote sequential code in GPU kernels to avoid memory transfers.

```
gpu_promote_sequential > MODULE.code
> PROGRAM.entities
< PROGRAM.entities
< MODULE.code
```

OpenCL 1.X requires function parameters to be qualified as either *local* or *global*, and this information must be propagated in local declarations within functions.

```
gpu_qualify_pointers  > MODULE.code
> PROGRAM.entities
                < PROGRAM.entities
                < MODULE.code
                < MODULE.callees
```

Note: this pass does mostly changes entities (parameter types, variable types). Only casts in the code may be affected.

Whether to add qualifiers to pointer casts:

```
GPU_QUALIFY_POINTERS_DO_CASTS TRUE
```

8.3.9 Task code generation for StarPU runtime

This pass outlines code parts so that every single piece of computation is located in a function that makes no use of global or static variable. Heuristic is trivial for now, this is an experimental work. `TASKIFY_TASK_PREFIX` **??**.

```
taskify                         > MODULE.code
                                > MODULE.callees
                                > PROGRAM.entities
                < PROGRAM.entities
                < MODULE.code
                < MODULE.cumulated_effects
                < MODULE.privatized
                < MODULE.regions
```

```
TASKIFY_TASK_PREFIX "P4A_task"
```

This pass generates pragmas for the StarPU GCC plugin. This is still experimental, so is the GCC plugin.

```
generate_starpu_pragma          > MODULE.code
                                > MODULE.callees
                                > PROGRAM.entities
                < PROGRAM.entities
                < MODULE.code
                < MODULE.cumulated_effects
                < MODULE.privatized
                < MODULE.regions
```

8.3.10 SCALOPES: task code generation for the SCMP architecture with SESAM HAL

The SCMP architecture, an asymmetric multiprocessor system-on-chip for dynamic applications, is described in [52]. SESAM is a simulation tool built up to help the design of such architectures. It relies on a specific programming model based on the explicit separation of the control and computation tasks, and its HAL provides high level memory allocation, shared memory access and synchronization functions.

The goal of the project was to generate applications for this architecture from sequential C code. Two different approaches were implemented. The first tries to identify tasks but is more specific. The second one is more general but tasks must have been previously identified with labels; this is currently performed manually.

8.3.10.1 First approach

The goal of the following phase is to generate SCMP tasks from C functions. The tasks are linked and scheduled using the SCMP Hardware Adaptation Layer (HAL). Pass `sesamify` 8.3.10.1 takes as input a module and analyzes all its callees. For instance, the 'main' module can be submitted to sesamify after the `gpu_ify` 8.3.8 or `scalopragma` 8.3.10.1 pass have been applied. Each analyzed module is transformed into a SCMP task if its name begins with `P4A_scmp_task`. To generate the final files for the SCMP simulator, the pass output must be transformed by a specific python parser.

This pass outlines code parts based on pragma. It can outline blocs or loops with a `#pragma scmp task` flag. It is based on the outline pass.

```
scalopragma                     > MODULE.code
                                > MODULE.callees
                                > PROGRAM.entities
                < PROGRAM.entities
                < MODULE.code
                < MODULE.cumulated_effects
```

The goal of Bufferization is to generate a dataflow communication through buffers between modules. The communication is done by special function call generated by `kernel_load_store` 8.3.7.5. To keep flows consistent outside the module `scalopify` 8.3.10.1 surrounds variable call with a special function too. A C file with stubs is needed.

Note that you must also set `KERNEL_LOAD_STORE_DEALLOCATE_FUNCTION` 8.3.7.5 to "" in order to have it generate relevant code.

The goal of this pass is to keep consistent flows outside the tasks.

```
scalopify                     > MODULE.code
> MODULE.callees
> PROGRAM.entities
< PROGRAM.entities
< MODULE.code
                < MODULE.cumulated_effects


sesamify                     > MODULE.code
> MODULE.callees
> PROGRAM.entities
< PROGRAM.entities
< MODULE.code
                < MODULE.cumulated_effects
```

8.3.10.2 General Solution

This code generation flow first relies on phase `isolate_statement` 8.3.7.1 to isolate the memory spaces of tasks identified by labels beginning by `SCALOPES_KERNEL_TASK_PREFIX` 8.3.10.2. Then phase `sesam_buffers_processing` 8.3.10.2 generates a header file describing how kernel and server tasks use the SESAM shared buffers. A post-processing phase is necessary to actually generate all the tasks of the distributed application. This is implemented in Par4All.

This solution is extensively described in [17]. A less technical presentation can be found in [53].

Phase `sesam_buffers_processing` is to be run after `isolate_statement` has been applied to all tasks statements. It then produces a header file to be included by the future SESAM application individual tasks. This header file describes how kernel and server tasks use the SESAM buffers.

```
sesam_buffers_processing    > MODULE.sesam_buffers_file
              < PROGRAM.entities
              < MODULE.code
              < MODULE.cumulated_effects
```

The next two properties are used by phase `sesam_buffers_processing` to detect kernel tasks statements in the input module and to generate server tasks names in the output header file.

```
SCALOPES_KERNEL_TASK_PREFIX "P4A_sesam_task_"
```

```
SCALOPES_SERVER_TASK_PREFIX "P4A_sesam_server_"
```

8.4 Automatic Resource-Constrained Static Task Parallelization

8.4.1 Sequence Dependence DAG (SDG)

A Sequence Dependence DAG G is a data dependence DAG where task vertices τ are labeled with statements, while control dependences are encoded in the abstract syntax trees of statements. Any statement S can label a DAG vertex, i.e. each vertex τ contains a statement S, which corresponds to the code it runs when scheduled. An SDG is not built only on simple instructions, represented as *call* statements; compound statements such as test statements (both true and false branches) and loop nests may constitute indivisible vertices of the SDG

Phase `sequence_dependence_graph` generates the Sequence Dependence Graph (SDG) in the dg ressource and in a dot file.

```
sequence_dependence_graph                    > MODULE.sdg
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

8.4.2 BDSC-Based Hierarchical Task Parallelization (HBDSC)

The goal of the following phase is to generate the scheduled task graph using BDSC: A Ressource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems, in order to automate the task-based parallelization of sequential applications.

Phase `bdsc_kdg_parallelization` applies BDSC and generates the scheduled Clustered Dependence Graph (KDG) in the dg ressource and in a dot file.

```
hbdsc_parallelization                        > MODULE.sdg
        < PROGRAM.entities                   > MODULE.schedule
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.sdg
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.complexities
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

The next properties are used by Phase `bdsc_kdg_parallelization` to permit the user defining the number of clusters and the memory size based on

the properties of the target architecture, generating communications or not in SPIRE code.

```
BDSC_NB_CLUSTERS 4
```

```
BDSC_MEMORY_SIZE -1
```

```
BDSC_DISTRIBUTED_MEMORY TRUE
```

In order to control the granularity of parallel tasks, we introduce this property. By default, we generate the maximum parallelism in a code. Otherwise, i.e. COSTLY_TASKS_ONLY value is TRUE, only loops and call functions are generated as parallel tasks. This is important in order to make a trade-off between the cost of thread creation and task execution.

```
COSTLY_TASKS_ONLY FALSE
```

This property is used to evaluate BDSC scheduling robustness. Since our BDSC scheduling heuristic relies on the numerical approximations of the execution time and communication costs of tasks, one needs to assess its sensitivity over the accuracy of these estimations. Since a mathematical analysis of this issue is made difficult by the heuristic nature of BDSC and, in fact, of scheduling processes in general, we ran multiple versions of each application using various static execution and communication cost models using a biased BDSC cost model, where we modulated each execution time and communication cost value randomly by at most $\Delta\%$, that is our propertu BDSC_SENSITIVITY (the default BDSC cost model would thus correspond to $\Delta = 0$).

```
BDSC_SENSITIVITY 0
```

We represent cost, data and time information of different tasks in terms of polynomials. We instrument using the phase `bdsc_code_instrumentation` the input sequential code and run it once in order to obtain the numerical values of the polynomials. The instrumented code contains the initial user code plus instructions that compute the values of the cost polynomials for each statement.

```
bdsc_code_instrumentation                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.chains
        < MODULE.sdg
        < MODULE.regions
        < MODULE.complexities
```

The next property is used also by Phase `bdsc_kdg_parallelization`. It decides if cost, data and time information will be extracted from the result of the instrumentation which is a dynamic analysis: `BDSC_INSTRUMENTED_FILE` file or not, i.e, static analysis.

```
BDSC_INSTRUMENTED_FILE ""
```

We add this pass since we use in our expriments, comparison between DSC and BDSC. In order to make this comparison automatic and simple, we apply Phase `dsc_code_parallelization` that performs DSC algorithm instead of BDSC algorithm and generates the parallel code SPIRE.

```
dsc_code_parallelization                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.complexities
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

8.4.3 SPIRE(PIPS) generation

The parallel code is represented using SPIRE of PIPS: A Generic Sequential to Parallel Intermediate Representation Extension applied to PIPS.

Phase `spire_shared_unstructured_to_structured` applies BDSC and generates the parallel code SPIRE.

```
spire_shared_unstructured_to_structured                 > MODULE.shared_spire_code
        < PROGRAM.entities                              > MODULE.code
        < MODULE.code
        < MODULE.sdg
        < MODULE.schedule
```

Phase `spire_distributed_unstructured_to_structured` applies BDSC, generates the parallel code SPIRE (spawn and barrier constructs) and inserts send and recv primitives.

```
spire_distributed_unstructured_to_structured                > MODULE.distributed_spire_
        < PROGRAM.entities                                  > MODULE.code
        < MODULE.code
        < MODULE.schedule
        < MODULE.sdg
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.proper_effects
        < MODULE.cumulated_effects
```

8.4.4 SPIRE-Based Parallel Code Generation

SPIRE is designed in such a way that it can facilitate the generation of code for different types of parallel systems. Different parallel languages can be mapped

into SPIRE. The two following phases show how this intermediate representation simplifies the task of producing code for various languages such as OpenMP and MPI.

Generate OpenMP task parallel code using SPIRE-based PIPS parallel intermediate representation.

```
openmp_task_generation                  > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.shared_spire_code
```

And to output the code decorated with OpenMP (omp task) directives, we use the existing pass print_parallelizedOMP_code

Generate MPI parallel code using SPIRE-based PIPS parallel intermediate representation.

```
mpi_task_generation                   > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.distributed_spire_code
```

Output the code decorated with MPI instructions.

```
print_parallelizedMPI_code     > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

8.4.5 MPI Code Generation

taskmapping in variable of the same name with `MPI_DUPLICATE_VARIABLE_PREFIX` 8.4.5 num proc

```
task_mapping  > MODULE.task
                < PROGRAM.entities
                < MODULE.code
```

copyvalueofwrite in variable of the same name with `MPI_DUPLICATE_VARIABLE_PREFIX` 8.4.5 num proc

```
copy_value_of_write  > MODULE.code
                < PROGRAM.entities
                < MODULE.proper_effects
                < MODULE.cumulated_effects
                < MODULE.summary_effects
                < MODULE.code
                < MODULE.task
```

copyvalueofwrite in variable of the same name with `MPI_DUPLICATE_VARIABLE_PREFIX` 8.4.5 num proc

```
copy_value_of_write_with_cumulated_regions  > MODULE.code
                < PROGRAM.entities
                < MODULE.proper_effects
                < MODULE.cumulated_effects
                < MODULE.summary_effects
```

```
                < MODULE.code
                < MODULE.task
                < MODULE.live_out_regions
```

```
MPI_DUPLICATE_VARIABLE_PREFIX "__dpvar__"
```

```
MPI_NBR_CLUSTER 4
```

```
MPI_LOCAL_VARIABLES_LIST ""
```

```
MPI_LOCAL_PARAMETER TRUE
```

variablereplication replicate variable and declaration with same name and prefix `MPI_DUPLICATE_VARIABLE_PREFIX` 8.4.5 and suffix num proc

```
variable_replication  > MODULE.code
                < PROGRAM.entities
                < MODULE.code
                < MODULE.task
```

eliminateoriginalvariables eliminate original variables of the code to replace them by there copy with same name and prefix `MPI_DUPLICATE_VARIABLE_PREFIX` 8.4.5 and suffix num proc

```
eliminate_original_variables  > MODULE.code
                < PROGRAM.entities
                < MODULE.code
                < MODULE.task


mpi_conversion  > MODULE.code
                < PROGRAM.entities
                < MODULE.code
                < MODULE.task
```

Chapter 9

Program Transformations

A program transformation is a special phase which takes a code as input, modifies it, possibly using results from several different analyses, and puts back this modified code as result.

A rule describing a program transformation will never be chosen automatically by `pipsmake` to generate some code since every transformation rule contains a cycle for the `MODULE.code` resource. Since the first rule producing code, described in this file, is `controlizer` 4.3 and since it is the only non-cyclic rule, the internal representation always is initialized with it.

As program transformations produce nothing else, `pipsmake` cannot guess when to apply these rules automatically. This is exactly what the user want most of the time: program transformations are under explicit control by the user. Transformations are applied when the user pushes one of *wpips* transformation buttons or when (s)he enters an *apply* command when running `tpips`[1], or by executing a `Perform` Shell script. See the introduction for pointers to the user interfaces.

Unfortunately, it is sometime nice to be able to chain several transformations without any user interaction. No general macro mechanism is available in `pipsmake`, but it is possible to impose some program transformations with the '!' command.

User inputs are not well-integrated although a `user_query` rule and a `string` resource could easily be added. User interaction with a phase are performed directly without notifying `pipsmake` to be more flexible and to allow dialogues between a transformation and the user.

9.1 Loop Transformations

9.1.1 Introduction

Most loop transformations require the user to give a valid loop label to locate the loop to be transformed. This is done interactively or by setting the following property to the valid label:

```
LOOP_LABEL ""
```

[1] `http://www.cri.ensmp.fr/pips/line-interface.html`

Put a label on unlabelled loops for further interactive processing. Unless `FLAG_LOOPS_DO_LOOPS_ONLY` 9.1.1 is set to false, only do loops are considered.

```
flag_loops                    > MODULE.code
  > MODULE.loops
        < PROGRAM.entities
        < MODULE.code
```

```
FLAG_LOOPS_DO_LOOPS_ONLY TRUE
```

Display label of all modules loops

```
print_loops > MODULE.loops_file
< MODULE.loops
```

9.1.2 Loop range Normalization

Use intermediate variables as loop upper and lower bound when they are not affine.

```
linearize_loop_range > MODULE.code
    < PROGRAM.entities
< MODULE.code
```

9.1.3 Label Elimination

Clean all statement labels. No legality check is performed. This transformation is particularly useful after a sequence of loop transformations is applied on a structured code.

```
  clean_labels > MODULE.code
  < PROGRAM.entities
  < MODULE.code
```

If `LABEL_EXCEPTION` 9.1.3 is set to a string, all the labels containing this string will not be eliminated.

```
LABEL_EXCEPTION ""
```

9.1.4 Loop Distribution

Function `distributer` 9.1.4 is a restricted version of the parallelization function `rice*` (see Section 8.1.3).

Distribute all the loops of the module.

Allen & Kennedy's algorithm [2] is used in both cases. The only difference is that `distributer` 9.1.4 does not produce DOALL loops, but just distributes loops as much as possible.

```
alias distributer 'Distribute Loops'
distributer                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

Partial distribution distributes the statements of a loop nest except the isolated statements,that have no dependences at the common level l, are gathered in the same l-th loop.

```
PARTIAL_DISTRIBUTION FALSE
```

9.1.5 Statement Insertion

Check if the statement flagged by `STATEMENT_INSERTION_PRAGMA` 9.1.5 can be safely inserted in the current control flow. This pass should be reserved to internal use only, another pass should create and insert a flagged statement and then call this one to verify the validity of the insertion

```
statement_insertion > MODULE.code
    < PROGRAM.entities
    < ALL.code
    > ALL.code
< MODULE.regions
< MODULE.out_regions
```

```
STATEMENT_INSERTION_PRAGMA "pips␣inserted␣statement␣to␣check"
```

```
STATEMENT_INSERTION_SUCCESS_PRAGMA "pips␣inserted␣statement"
```

```
STATEMENT_INSERTION_FAILURE_PRAGMA "pips␣inserted␣statement␣to␣remove"
```

9.1.6 Loop Expansion

Prepare the loop expansion by creating a new statement (that may be invalid) for further processing by `statement_insertion` 9.1.5. Use `STATEMENT_INSERTION_PRAGMA` 9.1.5 to identify the created statement. Otherwise `LOOP_LABEL` 9.1.1 and `LOOP_EXPANSION_SIZE` 9.1.6 have the same meaning as in `loop_expansion` 9.1.6

```
loop_expansion_init  > MODULE.code
    < PROGRAM.entities
    < MODULE.code
```

Extends the range of a loop given by `LOOP_LABEL` 9.1.1 to fit a size given by `LOOP_EXPANSION_SIZE` 9.1.6. An offset can be set if `LOOP_EXPANSION_CENTER` 9.1.6 is set to True. The new loop is guarded to prevent illegal iterations, further transformations can elaborate on this.

```
loop_expansion  > MODULE.code
    < PROGRAM.entities
    < MODULE.code
< MODULE.cumulated_effects
```

```
LOOP_EXPANSION_SIZE ""
```

```
LOOP_EXPANSION_CENTER FALSE
```

Extends the dimension of all declared arrays so that no access is illegal.

```
array_expansion > PROGRAM.entities
    < PROGRAM.entities
< MODULE.code
< MODULE.regions
```

9.1.7 Loop Fusion

This pass fuses as many loops as possible in a greedy manner. The loops must appear in a sequence and have exactly the same loop bounds and if possible the same loop indices. We'll always try first to fuse loops where there is a dependence between their body. We expect that this policy will maximize possibilities for further optimizations.

Property `LOOP_FUSION_GREEDY` 9.1.7 allows to control whether it'll try to fuse as many loop as possible even without any reuse. This will be done in a second pass.

Property `LOOP_FUSION_MAXIMIZE_PARALLELISM` 9.1.7 is used to control if loop fusion has to preserve parallelism while fusing. If this property is true, a parallel loop is never fused with a sequential loop.

Property `LOOP_FUSION_KEEP_PERFECT_PARALLEL_LOOP_NESTS` 9.1.7 prevents to lose parallelism when fusing outer loops from a loop nests without being able to fuse inner loops.

Property `LOOP_FUSION_MAX_FUSED_PER_LOOP` 9.1.7 limit the number of fusion per loop. A negative value means that no limit will be enforced.

The fusion legality is checked in the standard way by comparing the dependence graphs obtained before and after fusion.

This pass is still in the experimental stage. It may have side effects on the source code when the fusion is attempted but not performed in case loop index are different.

```
alias Fusion 'Fusion Loops'
loop_fusion            > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.dg
```

This pass is the same is `loop_fusion` 9.1.7 excepts that it uses regions instead of dependence graph. Properties are the same as before: `LOOP_FUSION_GREEDY` 9.1.7, `LOOP_FUSION_MAXIMIZE_PARALLELISM` 9.1.7, `LOOP_FUSION_KEEP_PERFECT_PARALLEL_LOOP_NESTS` 9.1.7, and `LOOP_FUSION_MAX_FUSED_PER_LOOP` 9.1.7 control the algorithm.

```
alias Fusion 'Fusion Loops With Regions'
loop_fusion_with_regions            > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

```
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.preconditions
        < MODULE.inv_regions
        < MODULE.dg
```

```
LOOP_FUSION_MAXIMIZE_PARALLELISM TRUE
```

```
LOOP_FUSION_GREEDY FALSE
```

```
LOOP_FUSION_KEEP_PERFECT_PARALLEL_LOOP_NESTS TRUE
```

```
LOOP_FUSION_MAX_FUSED_PER_LOOP -1
```

9.1.8 Index Set Splitting

Index Set Splitting [24] splits the loop referenced by property `LOOP_LABEL` 9.1.1 into two loops. The first loop ends at an iteration designated by property `INDEX_SET_SPLITTING_BOUND` 9.1.8 and the second start thereafter. It currently only works for do loops. This transformation is always legal. Index set splitting in combination with loop unrolling could be used to perform loop peeling.

```
alias index_set_splitting 'Index Set Splitting'
index_set_splitting      > MODULE.code
        > PROGRAM.entities
        < PROGRAM.entities
        < MODULE.code
```

Index Set Splitting *requires* the following globals to be set :

- `LOOP_LABEL` 9.1.1 is the loop label
- `INDEX_SET_SPLITTING_BOUND` 9.1.8 is the splitting bound

```
INDEX_SET_SPLITTING_BOUND ""
```

Additionnaly, `INDEX_SET_SPLITTING_SPLIT_BEFORE_BOUND` 9.1.8 can be used to accurately tell to split the loop before or after the bound given in `INDEX_SET_SPLITTING_BOUND` 9.1.8

```
INDEX_SET_SPLITTING_SPLIT_BEFORE_BOUND FALSE
```

9.1.9 Loop Unrolling

9.1.9.1 Regular Loop Unroll

Unroll requests a loop label and an unrolling factor from the user. Then it unrolls the specified loop as specified. The transformation is very general, and it is interesting to run `partial_eval` 9.4.2, `simplify_control` 9.3.1 and `dead_code_elimination` 9.3.2 after this transformation. When the number of

iterations cannot be proven to be a multiple of the unrolling factor, the extra iterations can be executed first or last (see LOOP_UNROLL_WITH_PROLOGUE 9.1.9.1).

Labels in the body are deleted. To unroll nested loops, start with the innermost loop.

This transformation is always legal.

```
alias unroll 'Loop Unroll'
unroll                          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Use LOOP_LABEL 9.1.1 and UNROLL_RATE 9.1.9.1 if you do not want to unroll interactively You can also set LOOP_UNROLL_MERGE 9.1.9.1 to use the same declarations among all the unrolled statement (only meaningful in C).

```
UNROLL_RATE 0
```

```
LOOP_UNROLL_MERGE FALSE
```

The unrolling rate does not always divide exactly the number of iterations. So an extra loop must be added to execute the remaining iterations. This extra loop can be executed with the first iterations (prologue option) or the last iterations (epilogue option). Property LOOP_UNROLL_WITH_PROLOGUE 9.1.9.1 can be set to FALSE to use the epilogue when possible. The current implementation of the unrolling with prologue is general, while the implementation of the unrolling with epilogue is restricted to loops with a statically knonw increment of one. The epilogue option may reduce misalignments.

```
LOOP_UNROLL_WITH_PROLOGUE TRUE
```

Another option might be to require unrolling of the prologue or epilogue loop when possible.

9.1.9.2 Full Loop Unroll

A loop can also be fully unrolled if the range is numerically known. "Partial Eval" may be usefully applied first.

This is only useful for small loop ranges.

Unrolling can be interactively applied and the user is requested a loop label:

```
alias full_unroll 'Full Loop Unroll (Interactive)'
full_unroll                          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Or directives can be inserted as comments for loops to be unrolled with:

```
alias full_unroll_pragma 'Full Loop Unroll (Pragma)'
full_unroll_pragma                   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

The directive is a comment containing the string `Cxxx` just before a loop to fully unroll (it is reserved to Fortran right now and should be generalized).

Full loop unrolling is applied one loop at a time by default. The user must specify the loop label. This default feature can be turned off and all loops with constant loop bounds and constant increment are fully unrolled.

Use `LOOP_LABEL` 9.1.1 to pass the desired label if you do not want to give it interactively

Property `FULL_LOOP_UNROLL_EXCEPTIONS` 9.1.9.2 is used to forbid loop unrolling when specific user functions are called in the loop body. The function names are separated by SPACEs. The default value is the empy set, i.e. the empry string.

```
FULL_LOOP_UNROLL_EXCEPTIONS ""
```

9.1.10 Loop Fusion

This pass applies unconditionnally a loop fusion between the loop designated by the property `LOOP_LABEL` 9.1.1 and the following loop. They must have the same loop index and the same iteration set. No legality check is performed.

```
force_loop_fusion > MODULE.code
        < PROGRAM.entities
< MODULE.code
```

9.1.11 Strip-mining

Strip-mine requests a loop label and either a chunk size or a chunk number. Then it strip-mines the specified loop, if it is found. Note that the DO/ENDDO construct is not compatible with such local program transformations.

```
alias strip_mine 'Strip Mining'
strip_mine                          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Behavior of strip mining can be controlled by the following properties:

- `LOOP_LABEL` 9.1.1 selects the loop to strip mine
- `STRIP_MINE_KIND` 9.1.11 can be set to 0 (fixed-size chunks) or 1 (fixed number of chunks). Negative value is used for interactive prompt.
- `STRIP_MINE_FACTOR` 9.1.11 controls the size of the chunk or the number of chunk depending on `STRIP_MINE_KIND` 9.1.11. Negative value is used for interactive prompt.

```
STRIP_MINE_KIND -1
```

```
STRIP_MINE_FACTOR -1
```

9.1.12 Loop Interchange

`loop_interchange` 9.1.12 requests a loop label and exchange the outer-most loop with this label and the inner-most one in the same loop nest, if such a loop nest exists.

Presently, legality is not checked.

```
alias loop_interchange 'Loop Interchange'
loop_interchange                              > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Property `LOOP_LABEL` 9.1.1 can be set to a loop label instead of using the default interactive method.

9.1.13 Hyperplane Method

`loop_hyperplane` 9.1.13 requests a loop label and a hyperplane direction vector and applies the hyperplane method to the loop nest starting with this loop label, if such a loop nest exists.

Presently, legality is not checked.

```
alias loop_hyperplane 'Hyperplane Method'
loop_hyperplane                               > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.1.14 Loop Nest Tiling

`loop_tiling` 9.1.14 requests from the user a numerical loop label and a numerical partitioning matrix and applies the tiling method to the loop nest starting with this loop label, if such a loop nest exists.

The partitioning matrix must be of dimension $n \times n$ where n is the loop nest depth. The default origin for the tiling is 0, but lower loop bounds are used to adjust it and decrease the control overhead. For instance, if each loop is of the usual kind, `DO I = 1, N`, the tiling origin is point (1, 1,...). The code generation is performed according to the PPoPP'91 paper but redundancy elimination may results in different loop bounds.

Presently, legality is not checked. There is no decision procedure to select automatically an *optimal* partitioning matrix. Since the matrix must be numerically known, it is not possible to generate a block distribution unless all loop bounds are numerically known. It is assumed that the loop nest is fully parallel.

Jingling Xue published an advanced code generation algorithm for tiling in Parallel Processing Letters (`http://cs.une.edu.au/~xue/pub.html`).

```
alias loop_tiling 'Tiling'
loop_tiling                                   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This transformations prompts the user for a partition matrix. Alternatively, this matrix can be provided through the `LOOP_TILING_MATRIX` 9.1.14 property. The format of the matrix is a00 a01 a02,a10 a11 a12,a20 a21 a22

```
LOOP_TILING_MATRIX ""
```

It is sometimes useful to apply a partial loop tiling, on the external loops of the loop nest for instance. The PARTIAL_LOOP_TILING 9.1.14 property should be TRUE. The LOOP_TILING_MATRIX 9.1.14 information is used.

```
PARTIAL_LOOP_TILING FALSE
```

Likewise, one can use the LOOP_LABEL 9.1.1 property to specify the targeted loop.

The implementation of the parallel loop tiling transformation is ongoing. It uses the hyperplane direction to generate the code with potential parallel loops scanning tiles.

```
alias parallel_loop_tiling 'Parallel Tiling'
parallel_loop_tiling                                > MODULE.code
        < PROGRAM.entities
        < MODULE.dg
        < MODULE.code
```

Loop tiling is valid only if it respects data dependencies. To check the legality of the transformation application the CHECK_TRANSFORMATION_LEGALITY 9.1.14 property should be TRUE. If the transformation is not legal the code does not changed.

```
CHECK_TRANSFORMATION_LEGALITY TRUE
```

Different codes can be generated after tiling, each having different behavior and performance. TILE_DIRECTION 9.1.14 and LOCAL_TILE_DIRECTION 9.1.14 properties specify the chosen directions to scan respectively the tiles and the local elements into each tile.

```
TILE_DIRECTION "TP"
```

TILE_DIRECTION 9.1.14 is set to TS when the direction is colinear to the partitioning vectors. It is set to TP when the hyperplane direction associated to orthogonal directions are used.

```
LOCAL_TILE_DIRECTION "LI"
```

The local elements of each tile are scanned 1) according to the partitioning vectors when LOCAL_TILE_DIRECTION 9.1.14 is set LS, 2) using the hyperplane direction associated to orthogonal directions if set to LP, 3) along the original basis when set to LI.

9.1.15 Symbolic Tiling

Tiles a loop nest using a partitioning vector that can contain symbolic values. The tiling only works for parallelepiped tiles. Use LOOP_LABEL 9.1.1 to specify the loop to tile. Use SYMBOLIC_TILING_VECTOR 9.1.15 as a comma-separated list to specify tile sizes. Use SYMBOLIC_TILING_FORCE 9.1.15 to bypass condition checks. Consider using loop_nest_unswitching 8.2.8 if generated max disturbs further analyses

```
symbolic_tiling > MODULE.code
! MODULE.coarse_grain_parallelization
< PROGRAM.entities
< MODULE.code
< MODULE.cumulated_effects
```

```
SYMBOLIC_TILING_VECTOR ""
```

```
SYMBOLIC_TILING_FORCE FALSE
```

9.1.16 Loop Normalize

The loop normalization consists in transforming all the loops of a given module into a normal form. In this normal form, the lower bound and the increment are equal to one (1).

Property `LOOP_NORMALIZE_PARALLEL_LOOPS_ONLY` 9.1.16 control whether we want to normalize only parallel loops or all loops.

If we note the initial DO loop as:

```
DO I = lower, upper, incre
   ...
ENDDO
```

the transformation gives the folowing code:

```
DO NLC = 0, (upper - lower + incre)/incre - 1, 1
   I = incre*NLC + lower
   ...
ENDDO
I = incre * MAX((upper - lower + incre)/incre, 0) + lower
```

The normalization is done only if the initial increment is a constant number. The normalization produces two assignment statements on the initial loop index. The first one (at the beginning of the loop body) assigns it to its value function of the new index and the second one (after the end of the loop) assigns it to its final value.

```
alias loop_normalize 'Loop Normalize'
loop_normalize          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

If the increment is 1, the loop is considered already normalized. To have a 1-increment loop normalized too, set the following property

```
LOOP_NORMALIZE_ONE_INCREMENT FALSE
```

This is useful to have iteration spaces that begin at 0 for GPU for example.

The loop normalization has been defined in some days only Fortran was available, so having loops starting at 1 like the default for arrays too make sense in Fortran.

Anyway, no we could generalize for C (starting at 0 is more natural) or why not from any other value that can be chosen with the following property:

```
LOOP_NORMALIZE_LOWER_BOUND 1
```

If you are sure the final assignment is useless, you can skip it with the following property.

```
LOOP_NORMALIZE_SKIP_INDEX_SIDE_EFFECT FALSE
```

```
 LOOP_NORMALIZE_PARALLEL_LOOPS_ONLY FALSE
```

9.1.17 Guard Elimination and Loop Transformations

Youcef Bouchebaba's implementation of unimodular loop transformations...

```
guard_elimination       > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.1.18 Tiling for sequences of loop nests

Tiling for sequences of loop nests

Youcef Bouchebaba's implementation of tiling for sequences of loop nests ...

```
alias tiling_sequence 'Tiling sequence of loop nests'

tiling_sequence      > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.2 Redundancy Elimination

9.2.1 Loop Invariant Code Motion

This is a test to implement a loop-invariant code motion. This phase hoist loop-invariant code out of the loop.

A side effect of this transformation is that the code is parallelized too with some loop distribution. If you don't want this side effect, you can check section **??** which does a pretty nice job too.

The original algorithm used is described in Chapters 12, 13 and 14 of Julien Zory's PhD dissertation [58].

```
invariant_code_motion             > MODULE.code
        < PROGRAM.entities
        < MODULE.proper_effects
        < MODULE.code MODULE.dg
```

Note: this pass deals with loop invariant code motion while the `icm` pass deals with expressions.

9.2.2 Partial Redundancy Elimination

In essence, a *partial redundancy* [41] is a computation that is done more than once on some path through a flowgraph. We implement here a partial redundancy elimination transformation for logical expressions such as bound checks by using informations given by precondition analyses.

This transformation is implemented by Thi Viet Nga NGUYEN.

See also the transformation in 9.4.10, the partial evaluation, and so on.

```
alias partial_redundancy_elimination 'Partial Redundancy Elimination'

partial_redundancy_elimination              > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
```

9.2.3 Identity Elimination

This pass was done by Nelson LOSSING.

Function `identity_elimination` 9.2.3 deletes identity statements like x=x; when x is an expression without side effect.

This pass also imply that if the instruction x=x; raise an exception, this exception disapear with the instructon.

```
alias identity_elimination 'Identity Elimination'

identity_elimination              > MODULE.code
        < PROGRAM.entities
        < MODULE.proper_effects
        < MODULE.code
```

Function `identity_elimination_with_points_to` 9.2.3 is an extension of `identity_elimination` 9.2.3 to also consider identity with pointer. So we want to eliminate x=*p or *p=x, when p is a pointer who points-to x.

This extension is not implemented and only compute an `identity_elimination` 9.2.3.

```
alias identity_elimination_with_points_to 'Identity Elimination with Points-to'

identity_elimination_with_points_to > MODULE.code
        < PROGRAM.entities
        < MODULE.points_to
        < MODULE.proper_effects
        < MODULE.code
```

9.3 Control-Flow Optimizations

9.3.1 Control Simplification (a.k.a. Dead Code Elimination)

Function `simplify_control` 9.3.1 is used to delete non-executed code, such as empty loop nests or zero-trip loops, for example after strip-mining or partial

evaluation.

Preconditions are used to find always true conditions in tests and to eliminate such tests. In some cases, tests cannot be eliminated, but test conditions can be simplified. One-trip loops are replaced by an index initialization and the loop body. Zero-trip loops are replaced by an index initialization. Effects in bound computations are preserved.

A lot of unexecuted code can simply be eliminated by testing its precondition feasibility. A very simple and fast test may be used if the preconditions are normalized when they are computed, but this slows down the precondition computation. Or non-normalized preconditions are stored in the database and an accurate and slow feasibility test must be used. Currently, the first option is used for assignments, calls, IOs and IF statements but a stronger feasibility test is used for loops.

FORMAT statements are suppressed because they behave like a NOP command. They should be gathered at the beginning or at the end of the module using property `GATHER_FORMATS_AT_BEGINNING` 4.3 or `GATHER_FORMATS_AT_END` 4.3. The property must be set before the control flow graph of the module is computed.

The cumulated effects are used in debug mode to display information.

The `simplify_control` 9.3.1 phase also performs some *If Simplifications* and *Loop Simplifications* [41].

This function was designed and implemented by Ronan Keryell.

```
alias simplify_control 'Simplify Control'
simplify_control          > MODULE.code
                          > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.preconditions
```

This pass is the same as `simplify_control` 9.3.1. It is used under this obsolete name in some validation scripts. The name has been preserved for backward compatibility.

```
suppress_dead_code          > MODULE.code
                            > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.preconditions
```

This pass is very similar to `simplify_control` 9.3.1, but it does not require the preconditions. Only local information is used. It can be useful to clean up input code with constant tests, e.g. `3>4`, and constant loop bounds. It can also be used after `partial_eval` 9.4.2 to avoid recomputing the preconditions yet another time. The property `SIMPLIFY_CONTROL_DIRECTLY_PRIVATE_LOOP_INDICES` 9.3.1 assert that the loop indices don't need a copy out, i.e. the value at the exit of the loop can be forgotten.

```
SIMPLIFY_CONTROL_DIRECTLY_PRIVATE_LOOP_INDICES FALSE
```

Whether to try to simplify do/while loops (property introduced as a special workaround for FREIA).

```
SIMPLIFY_CONTROL_DO_WHILE TRUE
```

```
alias simplify_control_directly 'Simplify Control Directly'
simplify_control_directly    > MODULE.code
                             > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

9.3.1.1 Properties for Control Simplification

It is sometimes useful to display statistics on what has been found useless and removed in a function, this property controls the statistics display:

```
DEAD_CODE_DISPLAY_STATISTICS TRUE
```

9.3.2 Dead Code Elimination (a.k.a. Use-Def Elimination)

Function `dead_code_elimination` 9.3.2 deletes statements whose def references are all dead, i.e. are not used by later executions of statements. It was developed by Ronan Keryell. The algorithm compute the set of live statements without fix-point. An initial set of live statements is extended with new statements reached thru use-def chains, control dependences and....

The initial set of live statements contains IO statements, RETURN, STOP, return, exit, abort...

Note that use-def chains are computed intraproceduraly and not interproceduraly. Hence some statements may be preserved because they update a formal parameter although this formal parameter is no longer used by the callers.

The dependence graph may be used instead of the use-def chains, but Ronan Keryell, designer and implementer of the initial Fortran version, did not produce convincing evidence of the benefit... The drawback is the additional CPU time required.

This pass was extended to C by Mehdi Amini in 2009-2010, but it is not yet stabilized. For C code, this pass requires that effects are calculated with property `MEMORY_EFFECTS_ONLY` set to `FALSE` because we need that the DG includes arcs for declarations as these latter are separate statements now.

`clean_declarations` 9.7.1 is automatically performed at the end, this is why cumulated effects are needed.

The following properties are intended to force some function calls to be preserved by the algorithm, `DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS` 9.3.2 expect a space separated list of function names while `DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS_PREFIX` 9.3.2 expect a space separated list of prefix for function name.

Does it really work?

```
DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS ""
```

```
DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS_PREFIX ""
```

```
alias dead_code_elimination 'Dead Code Elimination'
dead_code_elimination          > MODULE.code
                               > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.chains
```

Historical comments from Nga NGUYEN: According to [1] p. 595, and [41] p. 592, a variable is *dead* if it is not used on any path from the location in the code where it is defined to the exit point of the routine in the question; an instruction is *dead* if it computes only values that are not used on any executable path leading from the instruction. The transformation that identifies and removes such dead code is called dead code elimination. So in fact, the *Use-def elimination* pass in PIPS is a *Dead code elimination* pass and the *Suppress dead code* pass (see Section 9.3.1) does not have a standard name. It could be the *control simplification* pass. The wrong initial naming has been fixed, but it shows in PIPS source code, in tpips scripts and in validation test cases.

For backward compatibility, the next pass name is preserved.

```
alias use_def_elimination 'Use-Def elimination'
use_def_elimination          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.chains
```

`out_regions` 6.12.8 are harder to compute than a simple data dependence graph, but they provide some advantages over a standard dead-code elimination. They are computed interprocedurally. And they can be used to reduce the iteration sets. Since `out_regions` 6.12.8 are also flow sensitive regions, `dead_code_elimination_with_out_regions` 9.3.2 will also done an equivalent simplification that `simplify_control` 9.3.1 will do when a standard dead-code elimination can't detect it.

`loop_bound_minimization_with_out_regions` 9.3.3 is automatically performed at the end (after `clean_declarations` 9.7.1). It reduces the iteration sets. The preconditions requirement comes from it.

`DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS` 9.3.2 and `DEAD_CODE_ELIMINATION_KEEP_FUNCTIONS_PREFIX` doesn't work with `dead_code_elimination_with_out_regions` 9.3.2.

An equivalent result can/must be obtain by using `dead_code_elimination` 9.3.2 and applying `region_chains` 6.5.3 to compute use-def chains for intraprocedural result, or `in_out_regions_chains` 6.5.4 for interprocedural result. Except for the possible optimization to reduce the iteration sets. (The equivalence of the results are not tested)

```
alias dead_code_elimination_with_out_regions 'Dead Code Elimination with OUT Regions'
dead_code_elimination_with_out_regions          > MODULE.code
                                                > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.preconditions
        < MODULE.out_regions
```

9.3.3 Loop bound minimization

`loop_bound_minimization_with_out_regions` 9.3.3 has been implemented by Nelson LOSSING.

Loop bound minimization tries to minimize the loop bounds by filtering the iterations which have no effects on the code executed after the loop. For this purpose, it needs the `out_regions` 6.12.8 that will indicate which part of arrays are computed in the loop and will be useful on the loop continuation.

This pass only modifies the loop bounds when it is possible, and not the loop body nor the loop increment. Think to use `loop_normalize` 9.1.16, if you want to start at 0 or 1, or have an increment of 1.

The new loop bounds can be expressed by variables or constants which differ from the original bounds.

This pass can be launched successively several time and continues to make some improvements in the code. The refinement of an execution of this pass can provide more precise OUT_Regions and so allows better minimization for other loop bounds. As a consequence one launch of this pass is not equivalent to two successive launches.

For n loops, at most n executions of this pass should provide an optimal result. Any aditional execution is useless. No proof of this last assumption has been done.

Only work for Fortran's DO loop kind of loop.

```
alias loop_bound_minimization_with_out_regions 'Loop bound minimization with out regions'

loop_bound_minimization_with_out_regions      > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < MODULE.out_regions
```

9.3.4 Control Restructurers

Two control restructurers are available: `unspaghettify` 9.3.4.1 which is used by default in conjunction with `controlizer` 4.3 and `restructure_control` 9.3.4.2 which must be explicitly applied[2]

[2] A property can be used to force the call to `restructurer` by the `controlizer` 4.3.

9.3.4.1 Unspaghettify

The *unspaghettifier* is a heuristic to clean up and to simplify the control graphs of a module. It is useful because the controlizer (see Section 4.3) or some transformation phases can generate some *spaghetti* code with a lot of useless unstructured code which can confuse some other parts of `PIPS`. Dead code elimination, for example, uses `unspaghettify` 9.3.4.1.

This control restructuring transformation can be automatically applied in the `controlizer` 4.3 phase (see Section 4.3) if the `UNSPAGHETTIFY_IN_CONTROLIZER` 4.3 property is true.

To add flexibility, the behavior of `unspaghettify` 9.3.4.1 is controlled by the properties `UNSPAGHETTIFY_TEST_RESTRUCTURING` 9.3.4.1 and `UNSPAGHETTIFY_RECURSIVE_DECOMPOSITION` 9.3 to allow more restructuring from `restructure_control` 9.3.4.2 to be added in the `controlizer` 4.3 for example.

This function was designed and implemented by Ronan Keryell.

```
alias unspaghettify 'Unspaghettify the Control Graph'

unspaghettify           > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

To display the statistics about `unspaghettify` 9.3.4.1 and control graph restructuring `restructure_control` 9.3.4.2.

```
UNSPAGHETTIFY_DISPLAY_STATISTICS TRUE
```

The following option enables the use of IF/THEN/ELSE restructuring when applying unspaghettify:

```
UNSPAGHETTIFY_TEST_RESTRUCTURING FALSE
```

It is assumed as true for `restructure_control` 9.3.4.2. It recursively implement TEST restructuring (replacing IF/THEN/ELSE with GOTOs with structured IF/THEN/ELSE without any GOTOs when possible) by applying pattern matching methods.

The following option enables the use of control graph hierarchisation when applying unspaghettify:

```
UNSPAGHETTIFY_RECURSIVE_DECOMPOSITION FALSE
```

It is assumed as true for `restructure_control` 9.3.4.2. It implements a recursive decomposition of the control flow graph by an interval graph partitioning method.

The restructurer can recover some while loops if this property is set:

```
UNSPAGHETTIFY_WHILE_RECOVER FALSE
```

9.3.4.2 Restructure Control

`restructure_control` 9.3.4.2 is a more complete restructuring phase that is useful to improve the accuracy of various `PIPS` phases.

It is implemented by calling `unspaghettify` 9.3.4.1 (§ 9.3.4.1) with the properties `UNSPAGHETTIFY_TEST_RESTRUCTURING` 9.3.4.1 and `UNSPAGHETTIFY_RECURSIVE_DECOMPOSITION` 9.3.4.1 set to `TRUE`.

Other restructuring methods are available in `PIPS` with the TOOLPACK's restructurer (see Section 9.3.5).

```
alias restructure_control 'Restructure the Control Graph'

restructure_control          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.4.3 DO Loop Recovery

This control-flow transformation transforms while loops into DO loops by recovering an index variable, an initial value, a final value and an increment.

Useful to be run after transformations ?!?

```
alias recover_for_loop 'Recover for-loops from while-loops'

recover_for_loop          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

This phase cannot be called from inside the control restructurer since it needs many higher-level analysis. This is why it is in a separate phase.

9.3.4.4 For Loop to DO Loop Conversion

Since in PIPS some transformations and analysis are more precise for Fortran code, this is a transformation than try to transform the C-like for-loops into Fortran-like do-loops.

Don't worry about the C-code output: the prettyprinter output do-loop as for-loop if the C-output is selected. The do-loop construct is interesting since the iteration set is computed at the loop entry (for example it is not sensible to the index modification from the inside of the loop) and this simplifies abstract interpretation a lot.

This transformation transform for example a

```
for (i = lb; i < ub; i += stride)
  body;
```

into a

```
do i = lb, ub - 1, stride
  body
end do
```

```
alias for_loop_to_do_loop 'For-loop to do-loop transformation'

for_loop_to_do_loop          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.4.5 For Loop to While Loop Conversion

Since in PIPS some transformations and analysis may not be implemented for C for loops but may be implemented for while loops, it is interesting to have this for loop to while loop conversion.

This transformation transforms a

```
for (init; cond; update)
  body;
```

into a

```
{
  init;
  while(cond) {
    body;
    update;
  }
}
```

Since analysis are more precise on do-loops, you should apply a `for_loop_to_do_loop` 9.3.4.4 transformation first , and only after, apply this `for_loop_to_while_loop` 9.3.4.5 transformation that will transform the remaining for-loops into while loops.

```
alias for_loop_to_while_loop 'For-loop to while-loop transformation'

for_loop_to_while_loop          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.4.6 Do While to While Loop Conversion

Some transformations only work on while loops, thus it is useful to have this transformation that transforms a

```
do {
  body;
} while (cond);
```

into a

```
{
  body;
}
while (cond) {
  body;
}
```

It is a transformation useful before while loop to for loop recovery for example (see § 9.3.4.3).

```
alias dowhile_to_while 'Do-while to while-loop transformation'

dowhile_to_while          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.4.7 Spaghettify

`spaghettify` 9.3.4.7 is used in the context of the PHRASE project while creating "Finite State Machine"-like code portions in order to synthesize them in reconfigurable units.

This phases transform structured code portions (eg. loops) in unstructured statements.

`spaghettify` 9.3.4.7 transforms the module in a unstructured code with hierarchical unstructured portions of code corresponding to the old control flow structures.

To add flexibility, the behavior of `spaghettify` 9.3.4.7 is controlled by the properties

- DESTRUCTURE_TESTS
- DESTRUCTURE_LOOPS
- DESTRUCTURE_WHILELOOPS
- DESTRUCTURE_FORLOOPS

to allow more or less destruction power.

```
alias spaghettify 'Spaghettify the Control Graph'

spaghettify          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Thoses properties allow to fine tune `spaghettify` 9.3.4.7 phase

```
DESTRUCTURE_TESTS TRUE
```

```
DESTRUCTURE_LOOPS TRUE
```

```
DESTRUCTURE_WHILELOOPS TRUE
```

```
DESTRUCTURE_FORLOOPS TRUE
```

9.3.4.8 Full Spaghettify

The `spaghettify` 9.3.4.7 is used in context of PHRASE project while creating"Finite State Machine"-like code portions in order to synthesize them in reconfigurable units.

This phases transforms all the module in a unique flat unstructured statement.

Whereas the `spaghettify` 9.3.4.7 transforms the module in a unstructured code with hierarchical unstructured portions of code corresponding to the old structures, the `full_spaghettify` 9.3.4.8 transform the code in a sequence statement with a beginning statement, a unique and flattened unstructured (all the unstructured and sequences are flattened), and a final statement.

```
alias full_spaghettify 'Spaghettify the Control Graph for the entire module'

full_spaghettify          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.5 Control Flow Normalisation (STF)

This pass is now obsolete. Use `restructure_control` 9.3.4.2 instead.

Transformation `stf` 9.3.5 is a C interface to a Shell script used to restructure a Fortran program using ISTST (via the combined tool fragment ISTLY = ISTLX/ISTYP and then ISTST) from TOOLPACK [47, 44].

Be careful, since TOOLPACK is written in Fortran, you need the Fortran runtime libraries to run STF if is has not been statically compiled...

Known bug/feature: `stf` 9.3.5 does not change resource `code` like other transformations, but the `source` file. Transformations applied before `stf` 9.3.5 are lost.

This transformation is now assumed redundant with respect to the native PIPS control restructurers, which deal with other languages too.

```
alias stf 'Restructure with STF'
stf                       > MODULE.source_file
        < MODULE.source_file
```

9.3.6 Trivial Test Elimination

Function `suppress_trivial_test` 9.3.6 is used to delete the TRUE branch of trivial test instruction. After apply `suppress_trivial_test` 9.3.6, the condition of the new test instruction is the condition correspondent to the FALSE branch of the initial test.

This function was designed and implemented by Trinh Quoc Anh.

```
alias suppress_trivial_test 'Trivial Test Elimination'
suppress_trivial_test          > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.7 Finite State Machine Generation

Theses phases are used for PHRASE project.

NB: The PHRASE project is an attempt to automatically (or semi-automatically) transform high-level language for partial evaluation in reconfigurable logic (such as FPGAs or DataPaths).

This library provides phases allowing to build and modify "Finite State Machine"-like code portions which will be later synthesized in reconfigurable units. This was implemented by Sylvain Guérin.

9.3.7.1 FSM Generation

This phase tries to generate finite state machine from arbitrary code by applying rules numeroting branches of the syntax tree and using it as state variable for the finite state machine.

This phase recursively transforms each UNSTRUCTURED statement in a WHILE-LOOP statement controlled by a state variable, whose different values are associated to the different statements.

To add flexibility, the behavior of `fsm_generation` 9.3.7.1 is controlled by the property `FSMIZE_WITH_GLOBAL_VARIABLE` 9.3.7.5 which controls the fact that the same global variable (global to the current module) must be used for each FSMized statements.

```
alias fsm_generation 'FSM Generation'

fsm_generation        > MODULE.code
                      > PROGRAM.entities
        < PROGRAM.entities
        < MODULE.code
```

To generate a hierarchical finite state machine, apply first `spaghettify` 9.3.4.7 (§ 9.3.4.7) and then `fsm_generation` 9.3.7.1.

To generate a flat finite state machine, apply first `full_spaghettify` 9.3.4.8 (§ 9.3.4.8) and then `fsm_generation` 9.3.7.1 or use the aggregate phase `full_fsm_generation` 9.3.7.2.

9.3.7.2 Full FSM Generation

This phase tries to generate a flat finite state machine from arbitrary code by applying rules numeroting branches of the syntax tree and using it as state variable for the finite state machine.

This phase transform all the module in a FSM-like code, which is a WHILE-LOOP statement controlled by a state variable, whose different values are associated to the different statements.

In fact, this phase do nothing but rely on pipsmake to apply the succession of the 2 phases `full_spaghettify` 9.3.4.8 and `fsm_generation` 9.3.7.1 (§ 9.3.7.1)

```
alias full_fsm_generation 'Full FSM Generation'

full_fsm_generation   > MODULE.code
                      > PROGRAM.entities
        ! MODULE.full_spaghettify
```

```
        ! MODULE.fsm_generation
        < PROGRAM.entities
        < MODULE.code
```

9.3.7.3 FSM Split State

This phase is not yet implemented and do nothing right now...

This phase transform a state of a FSM-like statement and split it into n new states where the portion of code to execute is smaller.

NB: Phase `full_spaghettify` 9.3.4.8 must have been applied first !

```
alias fsm_split_state 'FSM split state

fsm_split_state    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.7.4 FSM Merge States

This phase is not yet implemented and do nothing right now...

This phase transform 2 or more states of a FSM-like statement and merge them into a new state where the portion of code to execute is bigger.

NB: Phase `full_spaghettify` 9.3.4.8 must have been applied first !

```
alias fsm_merge_states 'FSM merge states

fsm_merge_states    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.3.7.5 FSM Properties

Control the fact that the same global variable (global to the current module) must be used for each FSMized statements.

```
FSMIZE_WITH_GLOBAL_VARIABLE FALSE
```

9.3.8 Control Counters

A code instrumentation that adds local integer counters in tests and loops to know how many times a path is taken. This transformation may help some semantical analyses.

```
alias add_control_counters 'Control counters

add_control_counters    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.4 Expression Transformations

9.4.1 Atomizers

Atomizer produces, or should produce, three-address like instructions, in Fortran. An atomic instructions is an instruction that contains no more than three variables, such as `A = B op C`. The result is a program in a low-level Fortran on which you are able to use all the others passes of `PIPS`.

Atomizers are used to simplify the statement encountered by automatic distribution phases. For instance, indirect addressing like `A(B(I)) = ...` is replaced by `T=B(I);A(T) = ....`

9.4.1.1 General Atomizer

```
alias atomizer 'Atomizer'
atomizer                        > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.dg
```

9.4.1.2 Limited Atomizer

This pass performs subscripts atomization so that they can be converted in reference for more accruate analysis.

```
simplify_subscripts > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This pass evaluates expression of the form $*$"ae" that can be found in COLD output.

I doubt it can be useful elsewhere ...

```
simplify_constant_address_expressions > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This pass performs a conversion from complex to real. `SIMPLIFY_COMPLEX_USE_ARRAY_OF_STRUCTS` 9.4.1.2 controls the new layout

```
simplify_complex > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

```
SIMPLIFY_COMPLEX_USE_ARRAY_OF_STRUCTS TRUE
```

Split structures in separated variables when possible, that is remove the structure variable and replaces all fields by different variables.

```
split_structures > MODULE.code
    < PROGRAM.entities
    < MODULE.code
```

Here is a new version of the atomizer using a small atomizer from the HPF compiler (see Section 8.3.2).

```
alias new_atomizer 'New Atomizer'
new_atomizer                        > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

An atomizer is also used by WP65 (see Section 8.3.1)

9.4.1.3 Atomizer Properties

This transformation only atomizes indirect references of array access functions.

```
ATOMIZE_INDIRECT_REF_ONLY FALSE
```

By default, simple array accesses such as `X(I+2)` are atomized, although it is not necessary to generate assembly code:

```
ATOMIZE_ARRAY_ACCESSES_WITH_OFFSETS TRUE
```

The purpose of the default option is to maximise common subexpression elimination.

Once a code has been atomized, you can use this transformation to generate two address code only It can be useful for asm generation

```
generate_two_addresses_code > MODULE.code
< MODULE.code
< MODULE.cumulated_effects
< PROGRAM.entities
```

Set following property to false if you want to split dereferencing:

```
GENERATE_TWO_ADDRESSES_CODE_SKIP_DEREFERENCING TRUE
```

9.4.2 Partial Evaluation

Function `partial_eval` 9.4.2 produces code where *numerical* constant expressions or subexpressions are replaced by their value. Using the preconditions, some variables are evaluated to a integer constant, and replaced wherever possible. They are not replaced in user function calls because Fortran uses a call-by-reference mechanism and because they might be updated by the function. For the same conservative reason, they are not replaced in intrinsics calls.

Note that *symbolic* constants were left unevaluated because they already are constant. However it was found unfriendly by users because the principle of least surprise was not enforced: symbolic constants were sometimes replaced in the middle of an expression but not when the whole expression was a reference to a symbolic constant. Symbolic integer constants are now replaced by their values systematically.

Transformations `simplify_control` 9.3.1 and `dead_code_elimination` 9.3.2 should be performed after partial evaluation. It is sometimes important to run more than one partial evaluation in a row, because the first partial evaluation

may linearize some initially non-linear expressions. Perfect Club benchmark `ocean` is a case in point.

Comments from Nga NGUYEN: According to [1] and [41], the name of this optimization should be Constant-Expression Evaluation or Constant Folding for integer values. This transformation produces well error message at compile time indicating potential error such as division by zero.

```
alias partial_eval 'Partial Eval'
partial_eval                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.preconditions
```

PIPS[3] default behavior in various places is to evaluate symbolic constants. While meaningful, this approach is not source-to-source compliant, so one can set property `EVAL_SYMBOLIC_CONSTANT` 9.4.2 to FALSE to prevent some of those evaluations.

```
EVAL_SYMBOLIC_CONSTANT TRUE
```

One can also set `PARTIAL_EVAL_ALWAYS_SIMPLIFY` 9.4.2 to TRUE in order to force distribution, even when it does not seem profitable

```
PARTIAL_EVAL_ALWAYS_SIMPLIFY FALSE
```

Likewise, one can turn following property to true if he wants to use hard-coded value for size of types. The C standard is broken down into two parts, one is target independent, and one is target dependent. Currently, there is no way to specify different architecture models such as 32/32, 32/64 or 64/64, as the architecture model is hardwired in ri-util-local.h.

```
EVAL_SIZEOF FALSE
```

Regardless of the modelization, the C source code generated when this property is set to true is potentially no longer portable to other targets. This property is also used by the semantics analysis.

This function was implemented initially by Bruno BARON.

9.4.3 Reduction Detection

Phase `Reductions` detects generalized instructions and replaces them by calls to a run-time library supporting parallel reductions. It was developed by Pierre JOUVELOT in CommonLISP, as a prototype, to show than NewGen data structures were language-neutral. Thus it by-passes some of `pipsmake`/dbm facilities.

This phase is now obsolete, although reduction detection is critical for code restructuring and optimization... A new reduction detection phase was implemented by Fabien COELHO. Have a look at § 6.4 but it does not include a code transformation. Its result could be prettyprinted in an HPF style (FC: implementation?).

[3] `http://www.cri.ensmp.fr/pips`

```
old_reductions                        > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

9.4.4 Reduction Replacement

`replace_reduction_with_atomic` 9.4.4 replace all reduction in loop that are marked as parallel with reduction by `coarse_grain_parallelization_with_reduction` 8.1.6.

The property `ATOMIC_OPERATION_PROFILE` 9.4.4 control the set of atomic operations and operand allowed. At that time only "cuda" is supported.

`flag_parallel_reduced_loops_with_atomic` 9.4.4 flag as parallel all loops that were detected by `coarse_grain_parallelization_with_reduction` 8.1.6.

The property `ATOMIC_OPERATION_PROFILE` 9.4.4 control the set of atomic operations and operand allowed. At that time only "cuda" is supported.

```
ATOMIC_OPERATION_PROFILE "cuda"
```

```
replace_reduction_with_atomic > MODULE.code
      > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.reduction_parallel_loops
        < MODULE.cumulated_reductions

flag_parallel_reduced_loops_with_atomic > MODULE.code
> MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.reduction_parallel_loops
        < MODULE.cumulated_reductions
```

Flag loops with openmp directives, taking into account reductions.

```
flag_parallel_reduced_loops_with_openmp_directives > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.reduction_parallel_loops
        < MODULE.cumulated_reductions
```

9.4.5 Forward Substitution

CSE)

Scalars can be forward substituted. The effect is to undo already performed optimizations such as invariant code motion and common subexpression elimination, or manual atomization. However we hope to do a better job automatically!

```
alias forward_substitute 'Forward Substitution'

forward_substitute      > MODULE.code
```

```
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
        < MODULE.cumulated_effects
```

One can set `FORWARD_SUBSTITUTE_OPTIMISTIC_CLEAN` 9.4.5 to TRUE in order to clean (without check) forward - substituted assignments. Use cautiously !

```
FORWARD_SUBSTITUTE_OPTIMISTIC_CLEAN FALSE
```

9.4.6 Expression Substitution

This transformation is quickly developed to fulfill the need of a simple pattern matcher in pips. The user provide a module name through `EXPRESSION_SUBSTITUTION_PATTERN` 9.4.6 property and all expression similar to those contained in `EXPRESSION_SUBSTITUTION_PATTERN` 9.4.6 will be substituted to a call to this module. It is a kind of simple outlining transformations, it proves to be useful during simdization to recognize some idioms. Note that the pattern must contain only a single return instruction!

This phase was developed by Serge Guelton during his PhD.

```
alias expression_substitution 'Expression Substitution'

expression_substitution > MODULE.code
        > MODULE.callee
        < PROGRAM.entities
        < ALL.code
```

Set `RELAX_FLOAT_ASSOCIATIVITY` 9.4.6 to TRUE if you want to consider all floating point operations as really associative[4]:

```
RELAX_FLOAT_ASSOCIATIVITY FALSE
```

This property is used to set the one-liner module used during expression substitution. It must be the name of a module already loaded in pips and containing only one return instruction (the instruction to be matched).

```
EXPRESSION_SUBSTITUTION_PATTERN ""
```

9.4.7 Rename Operators

This transformation replaces all language operators by function calls.

```
rename_operator > MODULE.code
                < MODULE.code
                < PROGRAM.entities
```

[4]Floating point computations are not associative in real hardware because of finite precision and rounding errors. For example $(10^{50} \ominus 10^{-60}) \oplus 1 = 1$ but $10^{50} \oplus (-10^{-60} \oplus 1) = 0$.

The function name is derived from the operator name, the operator arguments type(s) and a common prefix. Each function name is built using the pattern [PREFIX][OP NAME][SUFFIX] (eg: *int + int* will lead to *op_addi*). The replacement function must have been declared, otherwise a warning is emited and the operator is ignored.

For instance, the following code:

```
float foo(float a, float b)
{
    return a + b;
}
```

becomes, using the default configuration:

```
float foo(float a, float b)
{
    return op_addf(a, b);
}
```

OP NAME is defined by the following table:

post++	post_inc	*	mul	−=	minus_up
++pre	inc_pre	/	div	<=	leq
post−−	post_dec	%	mod	<	lt
−−pre	dec_pre	=	assign	>=	geq
+	plus	*=	mul_up	>	gt
unary +	un_plus	/=	div_up	==	eq
−	minus	%=	mod_up	!=	neq
unary −	un_minus	+=	plus_up		

Using the property `RENAME_OPERATOR_OPS` 9.4.7, it is possible to give a restrictive list of operator names on which operator renaming should be applied. Operator that are not in this list are ignored.

```
RENAME_OPERATOR_OPS "plus␣minus␣mul␣div␣mod␣un_plus␣un_minus␣assign␣mul_up␣div_up␣m
```

Assuming that all arguments of the operator have the same type. *SUFFIX* is deduced using the following table:

char	c	long	l	_Bool	b
short	s	float	f	_Complex	C
int	i	double	d	_Imaginary	I

Using the property `RENAME_OPERATOR_SUFFIXES` 9.4.7, it is possible to give a restrictive list of suffix on which operator renaming should be applied. Every type not listed in this list will be ignored.

```
RENAME_OPERATOR_SUFFIXES "f␣d␣C␣I"
```

The *PREFIX* is a common prefix defined by the property `RENAME_OPERATOR_PREFIX` 9.4.7 which is applied to each operators. It can be used to choose between multiple implementations of the same operator. The default value is *op_*.

```
RENAME_OPERATOR_PREFIX "op_"
```

In Pips, C For loop like **for**(i=0; i < n; i++) is represented by a Fortran-like range-based Do loop **do** i = 1,n−1. Thus, the code:

```
for(i=0; i < n; i++)
```

will be rewritten :

```
for(i=0; i <= op_subi(n,1); i++)
```

If you want it to be rewritten :

```
for(op_assigni(&i,0); op_leqi(i,op_subi(n,1)); op_inci(i,1))
```

you should set the property `RENAME_OPERATOR_REWRITE_DO_LOOP_RANGE` 9.4.7 to *TRUE*. This is not the default behaviour, because in most case you don't want to rewrite For loop like this.

```
RENAME_OPERATOR_REWRITE_DO_LOOP_RANGE FALSE
```

Some operators (=, +=, ...) takes a modifiable lvalue. In this case, the expected function signature for a type T is T (T*, T). For instance, the code:

```
float a, b;
a += b;
```

would be rewritten:

```
float a, b;
op_add_upf(&a, b);
```

9.4.8 Array to Pointer Conversion

This transformation replaces all arrays in the module by equivalent linearized arrays. Eventually using array/pointer equivalence.

```
linearize_array > MODULE.code
                > COMPILATION_UNIT.code
 > CALLERS.code
        > PROGRAM.entities
< PROGRAM.entities
< MODULE.code
       < COMPILATION_UNIT.code
< CALLERS.code
```

This transformation replaces all arrays in the module by equivalent linearized arrays. This only makes the arrays starting their index from one.

```
linearize_array_fortran > MODULE.code
                        > CALLERS.code
        > PROGRAM.entities
        < PROGRAM.entities
        < MODULE.code
        < CALLERS.code
```

Use `LINEARIZE_ARRAY_USE_POINTERS` 9.4.8 to control whether arrays are declared as 1D arrays or pointers. Pointers are accessed using dereferencement and arrays using subscripts. This property does not apply to the fortran case.

```
LINEARIZE_ARRAY_USE_POINTERS FALSE
```

Use `LINEARIZE_ARRAY_MODIFY_CALL_SITE` 9.4.8 to control whether the call site is modified or not.

```
LINEARIZE_ARRAY_MODIFY_CALL_SITE TRUE
```

Use `LINEARIZE_ARRAY_CAST_AT_CALL_SITE` 9.4.8 to control whether a cast is inserted at call sites. Turning it on break further effects analysis, but without the cast it might break compilation or at least generate warnings for type mismatch. This property does not apply to the fortran case.

```
LINEARIZE_ARRAY_CAST_AT_CALL_SITE FALSE
```

Use `LINEARIZE_ARRAY_SKIP_STATIC_LENGTH_ARRAYS` 9.4.8 to skip the array to pointer conversion for static length arrays. Linearization is always done.

```
LINEARIZE_ARRAY_SKIP_STATIC_LENGTH_ARRAYS FALSE
```

Use `LINEARIZE_ARRAY_SKIP_LOCAL_ARRAYS` 9.4.8 to skip the array to pointer conversion for locally declared arrays. Linearization is always done.

```
LINEARIZE_ARRAY_SKIP_LOCAL_ARRAYS FALSE
```

9.4.9 Expression Optimization Using Algebraic Properties

This is an experimental section developed by Julien Zory as PhD work [58]. This phase aims at optimizing Fortran expression evaluation using algebraic properties such as associativity, commutativity, neutral elements and so forth. It is unfortunately obsolete because it relies on external pieces of software.

This phase restructure arithmetic expressions in order (1) to decrease the number of operations (e.g. through factorization), (2) to increase the ILP by keeping the corresponding DAG wide enough, (3) to facilitate the detection of composite instructions such as multiply-add, (4) to provide additional opportunities for (4a) invariant code motion (ICM) and (4b) common subexpression elimination (CSE).

Large arithmetic expressions are first built up via forward substitution when the programmer has already applied ICM and CSE by hand.

The optimal restructuring of expressions depends on the target defined by a combination of the computer architecture and the compiler. The target is specified by a string property called `EOLE_OPTIMIZATION_STRATEGY` 9.4.9 which can take values such as `"P2SC"` for IBM Power-2 architecture and XLF 4.3. To activate all sub-transformations such as ICM and CSE set it to `"FULL"`. See *properties* for more information about values for this property and about other properties controlling the behavior of this phase.

The current implementation is still shaky and does not handle well expressions of mixed types such as `X+1` where `1` is implictly promoted from integer to real.

Warning: this phase relies on an external (and unavailable) binary. To use it partly, you can set `EOLE_OPTIMIZATION_STRATEGY` 9.4.9 to `"CSE"` or `"ICM"`,

or even ICMCSE to have both. This will only activate common subexpressions elimination or invariant code motion (in fact, invariant sub-expression hoisting). Since it is a quite common use case, they have been defined as independent phases too. See 9.4.10.

```
alias optimize_expressions 'Optimize Expressions'

optimize_expressions    > MODULE.code
        < PROGRAM.entities
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.code

alias instruction_selection 'Select Instructions'

instruction_selection > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

EOLE: Evaluation Optimization of Loops and Expressions. Julien Zory stuff integrated within pips [58]. It relies on an external tool named eole. The version and options set can be controlled from the following properties. The status is experimental. See the optimize_expressions 9.4.9 pass for more details about the advanced transformations performed.

```
EOLE "newgen_eole"
```

```
EOLE_FLAGS "-nfd"
```

```
EOLE_OPTIONS ""
```

```
EOLE_OPTIMIZATION_STRATEGY "P2SC"
```

9.4.10 Common Subexpression Elimination

Here are described two interesting cases of the one in § 9.4.9.

Run common sub-expression elimination to factorize out some redundant expressions in the code.

One can use COMMON_SUBEXPRESSION_ELIMINATION_SKIP_ADDED_CONSTANT 9.4.10 to skip expression of the form a+2 and COMMON_SUBEXPRESSION_ELIMINATION_SKIP_LHS 9.4.10 to prevent elimination of left hand side of assignment.

The heuristic used for common subexpression elimination is described in Chapter 15 of Julien Zory's PhD dissertation [58]. It was designed for Fortran code. Its use for C code is experimental and flawed in general.

```
alias common_subexpression_elimination 'Common Subexpression Elimination'

common_subexpression_elimination > MODULE.code
        < PROGRAM.entities
```

```
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.code

alias icm 'Invariant Code Motion'

icm > MODULE.code
        < PROGRAM.entities
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.code
```

Note: the `icm` deals with expressions while the `invariant_code_motion` deals with loop invariant code.

The following property is used in sac to limit the subexpressions: When set to true, only subexpressions without "+constant" terms are eligible.

```
COMMON_SUBEXPRESSION_ELIMINATION_SKIP_ADDED_CONSTANT FALSE
```

```
COMMON_SUBEXPRESSION_ELIMINATION_SKIP_LHS TRUE
```

The icm pass performs invariant code motion over sub-expressions.

9.5 Hardware Accelerator

Generate code from a FREIA application possibly targeting hardware accelerator, such as SPoC, Terapix, or GPGPU. I'm unsure about the right granularity (now it is at the function level) and the resource which is produced (should it be an accelerated file?). The current choice does not allow to easily mix different accelerators.

9.5.1 FREIA Software

Generate code for a software FREIA implementation, by applying various optimizations at the library API level, but without generating accelerated functions.

```
freia_aipo_compiler   > MODULE.code
                      > MODULE.callees
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

The following properties are generic to all FREIA accelerator targets.

Whether to label arcs in dag dot output with the image name, and to label nodes with the statement number, and whether to filter out unused scalar nodes.

```
FREIA_DAG_LABEL_ARCS FALSE
```

```
FREIA_DAG_LABEL_NODES TRUE
```

```
FREIA_DAG_FILTER_NODES TRUE
```

Whether to compile lone operations, i.e. operations which do not belong to a sequence.

```
FREIA_COMPILE_LONE_OPERATIONS TRUE
```

Whether to normalize some operations:

```
FREIA_NORMALIZE_OPERATIONS TRUE
```

Whether to simplify the DAG using algebraic properties.

```
FREIA_SIMPLIFY_OPERATIONS TRUE
```

Whether to remove dead image operations in the DAG. Should always be beneficial.

```
FREIA_REMOVE_DEAD_OPERATIONS TRUE
```

Whether to remove duplicate operations in the DAG, including algebraic optimizations with commutators. Should be always beneficial to terapix, but it may depend for spoc.

```
FREIA_REMOVE_DUPLICATE_OPERATIONS TRUE
```

Whether to remove useless image copies from the expression DAG.

```
FREIA_REMOVE_USELESS_COPIES TRUE
```

Whether to move image copies within an expression DAG outside as external copies, if possible.

```
FREIA_MOVE_DIRECT_COPIES TRUE
```

Whether to merge identical arguments, especially kernels, when calling an accelerated function:

```
FREIA_MERGE_ARGUMENTS TRUE
```

Whether to attempt to reuse initial images if possible, instead of keeping possibly newly introduced temporary images.

```
FREIA_REUSE_INITIAL_IMAGES TRUE
```

Try to allow shuffling image pointers, but this is not allowed by default because it may lead to wrong code as the compiler currently ignores the information and mixes up images.

```
FREIA_ALLOW_IMAGE_SHUFFLE FALSE
```

Whether to assume that casts are simple image copies. Default is to keep a cast as cast, which is not accelerated.

```
FREIA_CAST_IS_COPY FALSE
```

Whether to cleanup freia returned status, as the code is assumed correct when compiled.

```
FREIA_CLEANUP_STATUS TRUE
```

Assume this pixel size in bits:

```
FREIA_PIXEL_SIZE 16
```

If set to a non-zero value, assume this image size when generating code. If zero, try generic code. In particular, the height is useful to compute a better imagelet size when generating code for the Terapix hardware accelerator.

```
FREIA_IMAGE_HEIGHT 0
```

```
FREIA_IMAGE_WIDTH 0
```

If a FREIA application uses a function to transpose a morpho kernel, the following property can be used to store the function name. Pips expects a function having the following signature: `func_name(int32_t kernelTransposed[9], const int32_t kernelIn[9])`

```
FREIA_TRANSPOSE_KERNEL_FUNC ""
```

Ad-hoc transformation to remove particular scalar write-write dependencies in sequences. They are introduced by the do-while to while conversion on FREIA convergence loops. There is an underlying generic transformation on sequences that could be implemented with more thoughts on the subject.

```
freia_remove_scalar_ww_deps > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.5.2 FREIA SPoC

FREIA Compiler for SPoC target.

Consider applying `freia_unroll_while` beforehand to unroll convergence use with the right number of iterations to make the best use of the available hardware. Note that the same transformation would also make sense somehow on sequences when the do-while to while transformation as been applied, but the unrolling factor is much harder to decide in the sequence case as it would depend on previous operations.

```
freia_spoc_compiler   > MODULE.code
                      > MODULE.callees
                      > MODULE.spoc_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

freia_unroll_while > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Default depth of the target SPoC accelerator:

```
HWAC_SPOC_DEPTH 8
```

9.5.3 FREIA Terapix

FREIA compiler for Terapix target.

```
freia_terapix_compiler  > MODULE.code
                        > MODULE.callees
                        > MODULE.terapix_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

Number of processing elements (PE) for the Terapix accelerator:

```
HWAC_TERAPIX_NPE 128
```

Default size of memory, in pixel, for the Terapix accelerator (RAMPE is *RAM* of *PE*):

```
HWAC_TERAPIX_RAMPE 1024
```

Terapix DMA bandwidth. How many terapix cycles to transfer an imagelet row (size of which is necessarily the number of pe):

```
HWAC_TERAPIX_DMABW 24
```

Yes, *twenty-four*, this is not a typo. It does not seem to depend whether pixels are 8-bit or 16-bit. The DDR tics are a little faster than the Terapix tics.

Terapix 2D global RAM (GRAM) width and height:

```
HWAC_TERAPIX_GRAM_WIDTH 64
```

```
HWAC_TERAPIX_GRAM_HEIGHT 32
```

Whether and how to further cut the dag for terapix. Expected values, by order of compilation costs, are: *none*, *compute*, *enumerate*.

```
HWAC_TERAPIX_DAG_CUT "compute"
```

Whether input and output memory transfers overlap one with the other, that is we have a full duplex DMA.

```
HWAC_TERAPIX_OVERLAP_IO FALSE
```

Note that it is already assumed that computations overlap with communications. This adds the information that host-accelerator loads and stores run in parallel. This has two impacts: the communication apparent time is reduced thanks to the overlapping, which is good, but the imagelet memory cannot be reused for inputs because it is still live while being stored, which is bad for memory pressure.

Use this maximum size (height) of an imagelet if set to non-zero. It may be useful to set this value to the image height (if known) so that compiler generates code for smaller imagelets, so that the runtime is not surprised. This is rather use of debug to impose an imagelet size.

```
HWAC_TERAPIX_IMAGELET_MAX_SIZE 0
```

Whether to reduce graphs to subgraphs of connected components. This should be always beneficial for Terapix, so this is the default. This option is added as a workaround against potential issues with the runtime.

```
HWAC_TERAPIX_REDUCE_TO_CONNECTED_COMPONENTS TRUE
```

9.5.4 FREIA OpenCL

FREIA compiler for OpenCL target, which may run on both multi-core and GPU.

```
freia_opencl_compiler  > MODULE.code
                       > MODULE.callees
                       > MODULE.opencl_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

Whether we should attempt to generate merged OpenCL image operations. If not, the result will be simular to simple AIPO compilation, no actual helper functions will be generated.

```
HWAC_OPENCL_MERGE_OPERATIONS TRUE
```

Whether to merge OpenCL constant-kernel operations.

```
HWAC_OPENCL_MERGE_KERNEL_OPERATIONS TRUE
```

Whether to generate OpenCL specialized constant-kernel operations.

```
HWAC_OPENCL_GENERATE_SPECIAL_KERNEL_OPS TRUE
```

Whether merged OpenCL image operations should include reductions as well. Added to help debugging the code.

```
HWAC_OPENCL_MERGE_REDUCTIONS TRUE
```

Whether to generate OpenCL code for one operation on its own. It is not interesting to do so because it is just equivalent to the already existing AIPO implementation, but it can be useful for debug.

```
HWAC_OPENCL_COMPILE_ONE_OPERATION FALSE
```

Whether to add an explicit synchronization in the generated codes, before calling a kernel. This property was added for debug.

```
HWAC_OPENCL_SYNCHRONIZE_KERNELS FALSE
```

Whether to preload pixels before using them in generated kernels. Should be better for GPGPU, but not necessarily for CPU implementations.

```
HWAC_OPENCL_PRELOAD_PIXELS TRUE
```

Tell which image dimension is handled by the first OpenCL thread dimension. Value is either *height* or *width*.

```
HWAC_OPENCL_FIRST_THREAD_DIMENSION "height"
```

Tell which is the first loop in the kernel code: either on the image *height* or *width*.

```
HWAC_OPENCL_FIRST_KERNEL_LOOP "height"
```

Whether to generate *tiled* kernels on the first thread dimension,

```
HWAC_OPENCL_TILING FALSE
```

9.5.5 FREIA Sigma-C for Kalray MPPA-256

FREIA compiler for Kalray MPPA-256 Sigma-C target.

```
freia_sigmac_compiler > MODULE.code
                      > MODULE.callees
                      > MODULE.sigmac_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

Minimal number of aggregated operators in a compound arithmetic agent

```
  HWAC_SIGMAC_MERGE_ARITH 0
```

Generation of kernel-specific morphological operators

```
  HWAC_SIGMAC_SPECIFIC_MORPHO TRUE
```

9.5.6 FREIA OpenMP + Async communications for Kalray MPPA-256

FREIA compiler for Kalray MPPA-256 OpenMP + Async communications target.

```
freia_mppa_compiler > MODULE.code
                    > MODULE.callees
                    > MODULE.mppa_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
```

Compute clusters shared memory avalaible slots

```
  HWAC_MPPA_MAX_SMEM_SLOTS 4
```

Maximum number of instructions in command structure

```
  HWAC_MPPA_MAX_INSTRS_CMD 50
```

9.6 Function Level Transformations

9.6.1 Inlining

Inlining is a well known technique. Basically, it replaces a function call by the function body. The current implementation does not work if the function has static declarations, access global variables .... Actually it (seems to) work(s) for pure, non-recursive functions ... and not to work for any other kind of call.

Property `INLINING_CALLERS` 9.6.1 can be set to define the list of functions where the call sites have to be inlined. By default, all call sites of the inlined function are inlined.

Only for C because of pipsmake output declaration !

```
inlining      > CALLERS.c_source_file
              > PROGRAM.entities
              > MODULE.callers
! MODULE.split_initializations
        < PROGRAM.entities
        < CALLERS.code
        < CALLERS.printed_file
< MODULE.code
        < MODULE.cumulated_effects
        * ALL.restructure_control
* ALL.remove_useless_label
```

Use following property to control how generated variables are initialized

```
INLINING_USE_INITIALIZATION_LIST TRUE
```

Use following property to control whether inlining should ignore stubs:

```
INLINING_IGNORE_STUBS TRUE
```

Set the following property to TRUE to add comments on inlined statements to keep track of their origin.

```
INLINING_COMMENT_ORIGIN FALSE
```

Same as inlining but always simulate the by-copy argument passing

Only for C because of pipsmake output declaration !

```
inlining_simple      > CALLERS.c_source_file
                     > PROGRAM.entities
                     > MODULE.callers
! MODULE.split_initializations
        < PROGRAM.entities
        < CALLERS.code
        < CALLERS.printed_file
< MODULE.code
        < MODULE.callers
        * ALL.restructure_control
* ALL.remove_useless_label
```

Regenerate the ri from the ri ...

Only for C because of pipsmake output declaration !

```
recompile_module > MODULE.c_source_file
< MODULE.code
```

The default behavior of inlining is to inline the given module in **all** call sites. Use `INLINING_CALLERS` 9.6.1 property to filter the call sites: only given module names will be considered.

```
INLINING_CALLERS ""
```

9.6.2 Unfolding

Unfolding is a complementary transformation of inlining 9.6.1. While inlining inlines all call sites to a given module in other modules, unfolding inlines recursively all call sites in a given module, thus unfolding the content of the module. An unfolded source code does not contain any call anymore. If you run it recursievly, you should set `INLINING_USE_INITIALIZATION_LIST` 9.6.1 to false.

Only for C because of output declaration in pipsmake rule!

```
unfolding      > MODULE.c_source_file
               > MODULE.callees
               > PROGRAM.entities
! CALLERS.split_initializations
        < PROGRAM.entities
        < MODULE.code
        < MODULE.printed_file
        < MODULE.cumulated_effects
        < CALLEES.code
        * ALL.restructure_control
* ALL.remove_useless_label
```

Same as unfolding, but cumulated effects are not used, and the resulting code always simulates the by-copy argument passing.

Only for C because of output declaration in pipsmake rule!

```
unfolding_simple      > MODULE.c_source_file
                      > MODULE.callees
                      > PROGRAM.entities
! CALLERS.split_initializations
        < PROGRAM.entities
        < MODULE.code
        < MODULE.printed_file
        < CALLEES.code
* ALL.restructure_control
* ALL.remove_useless_label
```

Use `UNFOLDING_CALLEES` 9.6.2, to specify which modules you want to inline in the unfolded module. The unfolding will be performed as long as one of the module in `UNFOLDING_CALLEES` 9.6.2 is called. More than one module can be specified, they are separated by blank spaces.

```
UNFOLDING_CALLEES ""
```

The default behavior of the `unfolding` 9.6.2 pass is to recursively inline all callees from the current module or from the argument modules of the pass, as long as a callee remains. You can use `UNFOLDING_FILTER` 9.6.2 to inline all call sites to a module not present in the space separated module list defined by the string property:

```
UNFOLDING_FILTER ""
```

By default this list is empty and hence all call sites are inlined.

9.6.3 Outlining

This documentation is a work in progress, as well as the documented topic.

Serge? Still true? What do you mean?

Outlining is the opposite transformation of `inlining` 9.6.1. It replaces some statements in an existing module by a call site to a new function whose execution is equivalent to the execution of the replaced statements. The body of the new function is similar to the piece of code replaced in the existing module. The user is prompted for various pieces of information in order to perform the outlining:

- a new module name,
- the statement number of the first outlined statement,
- number of statements to outline.

The statements are a subset of a sequence. They are counted in the sequence.

Serge: I invent...(FI)

`OUTLINE_WRITTEN_SCALAR_BY_REFERENCE` 9.6.3 controls whether we pass written scalar by reference or not. This property might lead to incorrect code!

```
outline      > MODULE.code
        > PROGRAM.entities
< MODULE.privatized
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.regions
        < MODULE.code
```

The property `OUTLINE_SMART_REFERENCE_COMPUTATION` 9.6.3 is used to limit the number of entities passed by reference. With it, a [0][0] is passed as an a[n][m] entity, without it it is passed as an **int** or **int**$*$ depending on the cumulated read/write memory effects of the outlined statements.

Serge: but cumulated effects are always required?

If you need to pass the upper bound expression of a particular loop as a parameter, which is used in Ter@pix code generation (see Section ???), set `OUTLINE_LOOP_BOUND_AS_PARAMETER` 9.6.3 to the loop label.

The property `OUTLINE_MODULE_NAME` 9.6.3 is used to specify the new module name. The user is prompted if it is set to its default value, the empty string.

Serge: a few words of motivation?

Serge: unconditionally?

But first the pass scans the code for any statement flagged with the pragma defined by the string property `OUTLINE_PRAGMA` 9.6.3.

Serge: What happens when it is set to the empty strin?

If set, the string property OUTLINE_LABEL 9.6.3 is used to choose the statement to outline.

The boolean property OUTLINE_ALLOW_GLOBALS 9.6.3 controls whether global variables whose initial values are not used are passed as parameters or not. It is suggested to addressed this issue with a previous privatization pass.

Finally, the boolean property OUTLINE_INDEPENDENT_COMPILATION_UNIT 9.6.3 can be set to true to outline the new module into a newly created compilation unit. It is named after the OUTLINE_MODULE_NAME 9.6.3. All necessary types, global variables and functions are declared into this new compilation unit. All the functions brought in the new compilation unit through the sub-callgraph are declared static and prefixed with OUTLINE_CALLEES_PREFIX 9.6.3.

Two properties were added for R-Stream compatibility. If OUTLINE_REMOVE_VARIABLE_RSTREAM_IMAGE 9.6.3 is set to True, instead of outlining loop indexes and written scalar variables the outliner declares them as local variables inside the outlined function. Therefore they are not added to the effective or formal parameters.

The behavior of OUTLINE_REMOVE_VARIABLE_RSTREAM_SCOP 9.6.3 is the same as above. Except that this property only excludes loop indexes from being outlined.

Serge: why is it called "globals"? It seems that this makes sense for any local variable.

Serge: you outline a set of statements, is it reduced to a singleton in that case?

Serge: Consistency between properties? Name conflicts?

```
OUTLINE_MODULE_NAME ""
```

```
OUTLINE_PRAGMA "pips␣outline"
```

```
OUTLINE_LABEL ""
```

```
OUTLINE_ALLOW_GLOBALS FALSE
```

```
OUTLINE_SMART_REFERENCE_COMPUTATION FALSE
```

```
OUTLINE_LOOP_BOUND_AS_PARAMETER ""
```

```
OUTLINE_INDEPENDENT_COMPILATION_UNIT FALSE
```

```
OUTLINE_WRITTEN_SCALAR_BY_REFERENCE TRUE
```

```
OUTLINE_CALLEES_PREFIX ""
```

```
OUTLINE_REMOVE_VARIABLE_RSTREAM_IMAGE FALSE
```

```
OUTLINE_REMOVE_VARIABLE_RSTREAM_SCOP FALSE
```

This pass was developped by Serge Guelton, as part of his PhD work.

9.6.4 Cloning

Procedures can be cloned to obtain several specialized versions. The call sites must be updated to refer to the desired version.

Cloning can be automatic or assisted by the user. Several examples are available in clone validation suite.

Slicing and specialization slicing have been introduced in [8, 9] and are more general than cloning. For the time being, no slicing transformation is available in *PIPS*[5] since they usually do not preserve the program semantics.

```
alias clone 'Manual Clone'

clone                   > CALLERS.code
                        > CALLERS.callees
        < MODULE.code
        < MODULE.callers
        < MODULE.user_file
        < CALLERS.callees
        < CALLERS.code

alias clone_substitute 'Manual Clone Substitution'

clone_substitute        > CALLERS.code
                        > CALLERS.callees
        < MODULE.code
        < MODULE.callers
        < MODULE.user_file
        < CALLERS.callees
        < CALLERS.code
```

Cloning of a subroutine according to an integer scalar argument. The argument is specified through integer property `TRANSFORMATION_CLONE_ON_ARGUMENT` 9.6.4. If set to 0, a user request is performed.

```
alias clone_on_argument 'Clone On Argument'

clone_on_argument       > CALLERS.code
                        > CALLERS.callees
                        > MODULE.callers
        < MODULE.code
        < MODULE.callers
        < MODULE.user_file
        < CALLERS.callees
        < CALLERS.preconditions
        < CALLERS.code
```

Not use assisted version of cloning it just perform the cloning without any substitution Use the `CLONE_NAME` 9.6.4 property if you want a particular clone name. It's up to another phase to perform the substitution.

[5] `http://www.cri.ensmp.fr/pips`

```
alias clone_only 'Simple Clone'

clone_only
        < MODULE.code
        < MODULE.user_file
```

There are two cloning properties. Cloning on an argument. If 0, a user request is performed.

```
TRANSFORMATION_CLONE_ON_ARGUMENT 0
```

Clone name can be given using the CLONE_NAME properties Otherwise, a new one is generated

RSTREAM_CLONE_SUFFIX 9.6.4 is the suffix appended to cloned function for R-Stream pre-compilation.

```
CLONE_NAME ""
```

```
RSTREAM_CLONE_SUFFIX ""
```

9.7 Declaration Transformations

9.7.1 Declarations Cleaning

Clean the declarations of unused variables and commons and so. It is also a code transformation, since not only the module entity are updated by the process, but also the declaration statements, some useless writes...

Clean the declarations of unused variables and commons and so.

```
alias clean_declarations 'Clean Declarations'

clean_declarations      > MODULE.code
        < PROGRAM.entities
        < MODULE.code
< MODULE.cumulated_effects
```

In C, dynamic variables which are allocated and freed but otherwise never used can be removed. This phase removes the calls to the dynamic allocation functions (*malloc* and *free* or user defined equivalents), and remove their declarations.

Clean unused local dynamic variables by removing malloc/free calls.

```
clean_unused_dynamic_variables > MODULE.code
< PROGRAM.entities
< MODULE.code
```

It may be a regular expression instead of a function name?

```
DYNAMIC_ALLOCATION "malloc"
```

```
DYNAMIC_DEALLOCATION "free"
```

Detecting and forcing variables in the `register` storage class can help subsequent analyses, as they cannot be referenced by pointers.

```
force_register_declarations > PROGRAM.entities
                            > MODULE.code
         < PROGRAM.entities
         < MODULE.code
```

Whether to allow arrays to be qualified as registers:

```
FORCE_REGISTER_ARRAY TRUE
```

Whether to allow pointers to be qualified as registers:

```
FORCE_REGISTER_POINTER TRUE
```

Whether to allow formal parameters to be qualified as registers:

```
FORCE_REGISTER_FORMAL TRUE
```

Remove some simple cases of pointers used on scalars.

```
remove_simple_scalar_pointers > MODULE.code
         < PROGRAM.entities
         < MODULE.code
```

9.7.2 Array Resizing

One problem of Fortran code is the unnormalized array bound declarations. In many program, the programmer put an asterisk (assumed-size array declarator), even 1 for every upper bound of last dimension of array declaration. This feature affects code quality and prevents others analyses such as array bound checking or alias analysis. We developed in PIPS two new methods to find out automatically the proper upper bound for the unnormalized and assumed-size array declarations, a process we call *array resizing*. Both approaches have advantages and drawbacks and maybe a combination of these ones is needed.

To have 100% resized arrays, we implement also the code instrumentation task, in the top-down approach.

Different options to compute new declarations for different kinds of arrays are described in properties-rc.tex. You can combine the two approaches to have a better results by using these options.

How to use these approaches: after generating new declarations in the logfile, you have to use the script $PIPS_ROOT/Src/Script/misc/array_resizing_instrumentation.pl to replace the unnormalized declarations and add new assignments in the source code.

9.7.2.1 Top Down Array Resizing

The method uses the relationship between actual and formal arguments from parameter-passing rules. New array declarations in the called procedure are computed with respect to the declarations in the calling procedures. It is faster than the first one because convex array regions are not needed.

This phase is implemented by Thi Viet Nga Nguyen (see [42]).

```
alias array_resizing_top_down 'Top Down Array Resizing'
array_resizing_top_down          > MODULE.new_declarations
                                 > PROGRAM.entities
        < PROGRAM.entities
        < CALLERS.code
        < CALLERS.new_declarations
        < CALLERS.preconditions
```

9.7.2.2 Bottom Up Array Resizing

The approach is based on an convex array region analysis that gives information about the set of array elements accessed during the execution of code. The regions READ and WRITE of each array in each module are merged and a new value for the upper bound of the last dimension is calculated and then it will replace the 1 or *.

This function is firstly implemented by Trinh Quoc ANH, and ameliorated by Corinne ANCOURT and Thi Viet Nga NGUYEN (see [42]).

```
alias array_resizing_bottom_up 'Bottom Up Array Resizing'
array_resizing_bottom_up         > MODULE.code
        > PROGRAM.entities
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < MODULE.regions
```

9.7.2.3 Full Bottom Up Array Resizing

This pass is base on `Useful Variables Regions` (see 6.12) that give for each variable the read/write region for this variable for the whole program at declaration time with his declaration memory state.

Contrary to the two previous array resizing passes, this pass was tested for C code but may also work for Fortran (not tested). It doesn't generate an intrumentation file that will be used to modify the code but directly modify the code.

This pass also computes new upper AND lower bounds for the array for all the dimension. For language that doesn't allow array notation (a[begin:end] or a[begin:nbr_elem]), a shift for all the access of the array is apply depending of the lower boud. Since we modify all the dimension of the array, the data structure in the memory are not kept.

Some prettyprint option may be added to allow C array notation.

Note : Property `ARRAY_RESIZING_ASSUMED_SIZE_ONLY` 9.7.2.5 has to be set Ã `FALSE` to compute something in C code.

This pass is implemented by Nelson Lossing, with the solver made by Thi Viet Nga NGUYEN for `Bottom Up Array Resizing` (see previous section 9.7.2.2).

```
alias array_resizing_full_bottom_up 'Full Bottom Up Array Resizing'
array_resizing_full_bottom_up        > MODULE.code
        > PROGRAM.entities
        < PROGRAM.entities
        < MODULE.code
```

```
        < MODULE.preconditions
        < MODULE.useful_variables_regions
```

9.7.2.4 Array Resizing Statistic

We provide here a tool to calculate the number of pointer-type A(,1) and assumed-size A(,*) array declarators as well as other information.

```
alias array_resizing_statistic 'Array Resizing Statistic'
array_resizing_statistic   > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.7.2.5 Array Resizing Properties

This phase is firstly designed to infer automatically new array declarations for assumed-size (A(*)) and one (A(1) or also called ugly assumed-size) array declarators. But it also can be used for all kinds of array : local or formal array arguments, unnormalized or all kinds of declarations. There are two different approaches that can be combined to have better results.

Top-down Array Resizing

There are three different options:

- Using information from the MAIN program or not (1 or 0). If you use this option, modules that are never called by the MAIN program are not taken into account. By default, we do not use this information (0).
- Compute new declarations for all kinds of formal array arguments, not only assumed-size and one declarations (1 or 0). By default, we compute for assumed-size and one only (0).
- Compute new declarations for assumed-size array only, not for ugly assumed-size (one) array (1 or 0). By default, we compute for both kinds (0).

So the combination of the three above options gives us a number from 0 to 7 (binary representation : 000, 001,..., 111). You must pay attention to the order of options. For example, if you want to use information from MAIN program to compute new declarations for assumed-size and one array declarations, both of them, the option is 4 (100). The default option is 0 (000).

```
ARRAY_RESIZING_TOP_DOWN_OPTION 0
```

Bottom-up Array Resizing

There are also three different options:

- Infer new declarations for arrays with declarations created by the top-down approach or not (1 or 0). This is a special option because we want to combine the two approaches: apply top-down first and then bottom-up on the instrumented arrays (their declarations are of from: I_PIPS_MODULE_ARRAY). By default, we do not use this option (0).

- Compute new declarations for all kinds of array arguments, not only assumed-size and one declarations (1 or 0). By default, we compute for assumed-size and one only (0).
- Compute new declarations for local array arguments or not (1 or 0). By default, we compute for formal array arguments only (0).

So the combination of the three above options gives us a number from 0 to 7 (binary representation : 000, 001,..., 111). You must pay attention to the order of options. There are some options that exclude others, such as the option to compute new declarations for instrumented array (I_PIPS_MODULE_ARRAY). The default option is 0 (000).

```
ARRAY_RESIZING_BOTTOM_UP_OPTION 0
```

Full Bottom-up Array Resizing

There are two possibles options that correspond to the second and third options of bottom-up Array Resizing, but explicited it. This two options can certainly replace the bit number combinaison for previous property?

Property `ARRAY_RESIZING_ASSUMED_SIZE_ONLY` 9.7.2.5 correspond to the middle bit of `ARRAY_RESIZING_BOTTOM_UP_OPTION` 9.7.2.5. It has the same default value, `TRUE`, that mean computation only for assumed-size and one declarations (* or 1 for Fortran). But to compute C code, you have to change this value for `FALSE`.

```
ARRAY_RESIZING_ASSUMED_SIZE_ONLY TRUE
```

Property `ARRAY_RESIZING_ARGUMENT` 9.7.2.5 correspond to the right bit of `ARRAY_RESIZING_BOTTOM_UP_OPTION` 9.7.2.5. It permits to recompute bound for argument array. By default, it's at `FALSE`.

WARNING : The `TRUE` case is not tested because it can be very dangerous since we don't also modify the call site, and the data structure is also completly modify.

```
ARRAY_RESIZING_ARGUMENT FALSE
```

9.7.3 Scalarization

Three scalarization pass have been developped. The first one, `scalarization` 9.7.3.1, is based on convex array regions. The second one, `constant_array_scalarization` 9.7.3.2, is based on constant array references. The third one, `quick_scalarization` 9.7.3.3, is based on proper and cumulated effects and on the dependence graph. It is supposed equivalent to the first one, but much faster because convex array regions may be slow to compute.

Pass `constant_array_scalarization` 9.7.3.2 intends to eliminate array references, e.g. for VHDL generation, for scalar constant propagations, e.g. to propagate constant arrays, and, ultimately, to explicit the control of automata encoded with a array containing pointers to transition functions and the next state, e.g. a protocol controler.

The other two scalarization passes intend to reduce the number of array accesses by replacing them with scalar accesses. Combined with loop fusion and/or used on automatically generated code, scalarization may eliminate intermediate arrays and improve locality.

9.7.3.1 Scalarization Based on Convex Array Regions

Scalarization is the process of replacing array references with references to scalar copies of the array elements wherever appropriate. Expected benefits include lower memory footprint and access time because registers can be used instead of temporary stack variables or memory accesses, and hence, shorter execution times, as long as the register pressure is not so high that spill code is generated.

Scalarizing a given array reference is subject to two successive criteria, a Legality criterion and a Profitability criterion:

- The Legality criterion is evaluated first. It tries and determines if replacing the reference might lead to breaking dependence arcs, due to hidden references to the element, e.g. "get(A,i)" instead of "A[i]". Also, one and only one array element must be accessed at each iteration because the algorithm used is based on convex array regions and because their implementation assumes only one region per array and per access type. If these two conditions are not met, no scalarization takes place.

- The Profitability criterion is then evaluated, to try and eliminate cases where scalarization would yield no satisfactory performance gains, e.g. when a scalarized reference has to be immediately copied back into the original reference. This is performed at the source level and depends on the effective register allocation and code generation.

The legality test is based on convex array regions, and not on the dependence graph as is the Carr's algorithm [13][14]. Currently, loop carried dependence arcs prevent scalarization by PIPS, although some can be processed by Carr's algorithm. See non-regression tests Transformations/scalarization30 to 36. Procedure calls do not derail our legality test, but Carr's benchmark does not include any procedure call. Also, the convex array regions related to the same array are merged in PIPS as a unique region, even when they have provably no intersection. As a result, each array can result in at most one scalar when Carr's algorithm can generate several ones.

This transformation is useful 1) to improve the readability of the source code with constructs similar to *let x be ...*, 2) to improve the modelizations of the execution, time or energy, by using source instructions closer to the machine instruction, 3) to perform an optimization at the source level because the code generator does not include a powerful (partially) redundant load elimination, and 4) to be a useful pass in a fully source-to-source compiler. This transformation is useful to reduce the expansion caused by the different atomizer passes.

The new scalar variables use the default prefix `___scalar__` and are thus easily identified, but a new prefix can be user defined with Property `SCALARIZATION_PREFIX` 9.7.3.1. If`SCALARIZATION_PREFIX` 9.7.3.1 is the empty string, the names of the scalarized variables are used.

If needed according to the IN and OUT convex array regions, the new variables are initialized, e.g. `__scalar0__ = A[i]`, and/or copied back into the initial array, e.g. `A[i] = __scalar0__`.

Scalarization is currently applicable both to Fortran and C code, but the code generation slightly differs because local declarations are possible with C. However, this may destroy perfect loop nests required by other passes. Property `SCALARIZATION_PRESERVE_PERFECT_LOOP_NEST` 9.7.3.1 can be used to preserve C perfect loop nests, this property is currently not completely implemented. `SCALARIZATION_KEEP_PERFECT_PARALLEL_LOOP_NESTS` 9.7.3.1 can be used to preserve C perfect parallel loop nests.

Pass `scalarization` 9.7.3.1 uses the read and written convex array regions to decide if the scalarization is possible, the IN and OUT regions to decide if it is useful to initialize the scalar copy or to restore the value of the array element.

```
alias scalarization 'Scalarization'
scalarization  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
        < CALLEES.summary_effects
```

Since OUT regions are computed interprocedurally, strange code may result when a small or not so small function without any real output is scalarized. The problem can be fixed by adding a `PRINT` or a `printf` to force an OUT region, or by setting Property `SCALARIZATION_FORCE_OUT` 9.7.3.1 to true.

Also, it may be useful to use Property `SEMANTICS_TRUST_ARRAY_DECLARATIONS` 6.9.4.2 and/or `SEMANTICS_TRUST_ARRAY_REFERENCES` 6.9.4.2 to make sure that as many loops as possible are entered. If not, loop bounds may act like guards, which prevents PIPS from hoisting a reference out of a loop, although this is puzzling for programmers because they expect all loops to be entered at least once...

As explained above, Property `SCALARIZATION_PREFIX` 9.7.3.1 is used to select the names of the new scalar variables. If it is set to the empty string, `""`, the scalarized variable name is used as prefix, which improves readability.

```
SCALARIZATION_PREFIX ""
```

When property `SCALARIZATION_USE_REGISTERS` 9.7.3.1 is set to `TRUE` (default value), new variables are declared with the `register` qualifier. However, this is not always compatible with other phases. Set it to `FALSE` to toggle off this behavior.

```
SCALARIZATION_USE_REGISTERS TRUE
```

Also, as explained above, Property `SCALARIZATION_PRESERVE_PERFECT_LOOP_NEST` 9.7.3.1 is used to control the way new scalar variables are declared and initialized in C. The current default value is FALSE because the locality of declarations is better for automatic loop parallelization, but this could be changed as a privatization pass should do as well while preserving perfect loop nests. Note that initializations may have to be placed in such a way that a perfect loop nest is nevertheless destroyed, even with this property set to true.

```
SCALARIZATION_PRESERVE_PERFECT_LOOP_NEST FALSE
```

```
SCALARIZATION_KEEP_PERFECT_PARALLEL_LOOP_NESTS FALSE
```

Property SCALARIZATION_FORCE_OUT 9.7.3.1 forces the generation of a copy-out statement when it is useless according to the OUT region, for instance when dealing with a library function.

```
SCALARIZATION_FORCE_OUT FALSE
```

Numerical property SCALARIZATION_THRESHOLD 9.7.3.1 is used to decide profitability: the estimated complexity of the scalarized code must be less than the initial complexity. Its minimal value is 2. It can be set to around 5 to forbid scalarization in sequences with no loop benefit.

```
SCALARIZATION_THRESHOLD 2
```

Property SCALARIZATION_ACROSS_CONTROL_TEST 9.7.3.1 is used to control the place where new memory accesses are inserted.

A memory access may be moved out of a loop, which is the best case for performance, but it then is no longer control dependent on the loop bounds. A new access is thus added if the compiler cannot prove that the loop is always entered. However, the burden of the proof may be too much, if only because nothing can be proven when the loop is sometimes entered and sometimes not. Also, the memory access may be always perfectly legal.

The default value for SCALARIZATION_ACROSS_CONTROL_TEST 9.7.3.1 is "exactness", allying safety to performance. It forbids scalarization when both read and written regions for the current piece of code are approximate regions, and, in addition, it applies a cheap test. This is the default option.

When it is set to "strict", property SCALARIZATION_ACROSS_CONTROL_TEST 9.7.3.1 goes for safety before performance, allowing candidates from approximated regions, but performing a strict test to check the validity of the transformation.

When the property is set to "cheap", memory accesses are moved outside of control structures with only the cheapest test. Use this last value with caution, since it is unsafe in the general case.

These properties are pretty hard to understand, as well as their impact. See sequence06, 07, 08 and 44 in the validation suite of the scalarization pass.

```
SCALARIZATION_ACROSS_CONTROL_TEST "exactness"
```

This property tells that variables which are used with an & (address-of) operator (a pointer is generated which references them) should not be scalarized.

```
SCALARIZATION_SKIP_ADDRESS_OF_VARIABLES FALSE
```

This pass was designed by François Irigoin and implemented by Laurent Daverio and François Irigoin.

9.7.3.2 Scalarization Based on Constant Array References

Similar to scalarization 9.7.3.1, but with a different criterion: if an array is only accessed with numerical constant subscript expressions, it is replaced by a

set of scalars and all its references are replaced by references to the corresponding scalars.

```
constant_array_scalarization > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This pass may be useful to make C source code more palatable to C to VHDL converters that handle scalars better than arrays. It is also useful to propagate constants contained in an array since the semantics passes only analyze scalar variables.

This pass was designed and implemented by Serge Guelton.

9.7.3.3 Scalarization Based on Memory Effects and Dependence Graph

Pass `scalarization` 9.7.3.1 may be costly because it relies on array regions analyses and uses them to check its legality criteria. The next phase alleviates these drawbacks.

This pass solely relies on proper and cumulated effets, and as such may fail to scalarize some accesses. However, it is expected to give good results in usual cases, especially after `loop_fusion` 9.1.7.

It basically uses the same algorithm as scalar privatization, but performs on the dependence graph rather than the chains graph for more precision about array dependences. Several legality criteria are then tested to ensure the safety of the transformation. In particular, it is checked that candidate references are not accessed through hidden references (for instance in calls), and that only one kind of reference is scalarized in the loop.

```
alias quick_scalarization 'Quick Scalarization'
quick_scalarization  > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.proper_effects
        < MODULE.dg
```

Pass `quick_scalarization` 9.7.3.3 was implemented by Béatrice Creusillet.

9.7.4 Induction Variable Substitution

Induction substitution is the process of replacing scalar variables by a linear expression of the loop indices.

```
alias induction_substitution 'Induction variable substitution'
induction_substitution > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

This pass was designed and implemented by Mehdi Amini.

9.7.5 Strength Reduction

Reduce the complexity of expression computation by generating induction variables when possible. E.g.

```
for (i=0;i<n;i++)
  a[i]=2;
```

Would become

```
for (i=0;i<n;i++) {
  *a=2;
  a++;
}
```

```
strength_reduction > MODULE.code
                   > PROGRAM.entities
    < MODULE.code
    < MODULE.cumulated_effects
    < MODULE.transformers
```

9.7.6 Flatten Code

The goal of this program transformation is to enlarge basic blocks as much as possible to increase the opportunities for optimization. The input code is assumed serial: parallel loops are declared sequential.

This transformation has been developed in PIPS for heterogeneous computing and is combined with inlining to increase the size of the code executed by an external accelerator while reducing the externalization overhead[6]. Other transformations, such as partial evaluation and dead code elimination (including use-def elimination) can be applied to streamline the resulting code further.

The transformation `flatten_code` 9.7.6 firstly moves declarations up in the abstract syntax tree, secondly remove useless braces and thirdly fully unroll loops when there iteration counts are known and the `FLATTEN_CODE_UNROLL` property is true. Unrolling can also be controlled using Property `FULL_LOOP_UNROLL_EXCEPTIONS` 9.1.9.2. The declarations are moved up: some parallel loops may become sequential. To avoid inconsistency, all loops are declared sequential by `flatten_code` 9.7.6.

Inlining(s), which must be performed explicitly by the user with tpips or another PIPS interface, can be used first to create lots of opportunities. The basic block size increase is first due to brace removals made possible when declarations have been moved up, and then to loop unrollings. Finally, partial evaluation, dead code elimination and use-def based elimination can also straighten-out the code and enlarge basic blocks by removing useless tests or assignments.

The code externalization and adaptation for a given hardware accelerator is performed by another phase, see for instance Section 9.5.

Initially developed in August 2009 by Laurent Daverio, with help from Fabien Coelho and François Irigoin.

```
alias flatten_code 'Flatten Code'
flatten_code > MODULE.code
```

[6] FREIA project

```
        < PROGRAM.entities
        < MODULE.code
```

If the following property is set, loop unrolling is applied too for loops with static bounds.

```
FLATTEN_CODE_UNROLL TRUE
```

9.7.7 Split Update Operators

Split C operators such as `a += b`, `a *= b`, `a >>= b`, etc. into their expanded form such as `a = a + b`.

Note that if the left hand side expression lhs implies side effects, the transformed code is not equivalent since lhs be evaluated twice in the transformed code. The left hand side is not checked for side effects. The legality of the transformation is not guaranteed.

```
split_update_operator > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

Two improvements could be used. Check the side effects with a function such as `expression_to_proper_constant_path_effects()` and/or use a pointer to evaluate lhs `a` once to obtain `p = &a; *p = *p +b;`.

9.7.8 Split Initializations (C Code)

The purpose of this transformation is to separate the initialization part from the declaration part in C code in order to make static code analyses simpler.

This transformation recurses through all variable declarations, and creates a new statement each time an initial value is specified in the declaration, if the initial value can be assigned and if the variable is not static. The declarations are modified by eliminating the initial value, and a new assignment statement with the initial value is added to the source code.

This transformation can be used, for instance, to improve reduction detection (see TRAC Ticket 181).

Note that C array and structure initializations, which use braces, cannot be converted into assignments. In such cases, the initial declaration is either left untouched or expanded into a statement list.

```
alias split_initializations 'Split Initializations'
split_initializations > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

This transformation uses the `C89_CODE_GENERATION` property to generate either C89 or C99 code.

This pass has been developed by Serge Guelton [25]. Its validation suite is still too limited.

9.7.9 Set Return Type

The purpose of this transformation is to change the return type of a function. The new type will be a typedef whose name is controlled by SET_RETURN_TYPE_AS_TYPEDEF_NEW_TYPE 9.7.9. The corresponding typedef must exist in the symbol table.

This transformation loops over the symbols in the symbol table, and for each of them which is a typedef, compare the local name to the property SET_RETURN_TYPE_AS_TYPEDEF_NEW_TYPE 9.7.9. This approach is unsafe because there can be different typedef with the same name in different compilation units, resulting in differents entries in the symbol table for a same local name. The return type can also be incoherent with the return statement, thus it is not safe to run it on a non-void function.

However this pass has been created for special need in par4all, and considering restrictions described above, it does the job.

```
SET_RETURN_TYPE_AS_TYPEDEF_NEW_TYPE "P4A_accel_kernel_wrapper"
```

```
alias set_return_type_as_typedef 'Set return type as typedef'
set_return_type_as_typedef > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.7.10 Cast Actual Parameters at Call Sites

The purpose of this transformation is to cast parameters at call sites according to the prototype, a.k.a. the signature, of the called functions.

```
alias cast_at_call_sites 'Cast parameters at call sites'
cast_at_call_sites > CALLERS.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.callers
        < CALLERS.code
```

9.7.11 Scalar and Array Privatization

Variable privatization consists in discovering variables whose values are local to a particular scope, usually a loop iteration.

Three different privatization functions are available. The *quick* privatization is restricted to loop indices and is included in the dependence graph computation (see Section 6.6). The scalar privatization should be applied before any serious parallelization attempt. The array privatization is much more expensive and is still mainly experimental.

You should keep in mind that, although they modify the internal repesentation of the code, scalar and array privatizations are only latent program transformations, and no actual local variable declaration is generated. This is the responsibility of code generation phases, which may use this information differently depending on their target.

9.7.11.1 Scalar Privatization

Two phases implement scalar privatization. Both detect variables which are local to a loop nest. They differ in the way they handle global variables: `privatize_module` 9.7.11.1 privatizes only local variables and function parameters, whereas `privatize_module_even_globals` 9.7.11.1 also tries to privatize global variables. Both phases produce a fake resource, `MODULE.privatized`, so that one or the other can be activated. The default phase is `privatize_module` 9.7.11.1.

Privatizer detects variables that are local to a loop nest and marks these variables as private. A variable is private to a loop if the values assigned to this variable inside the loop cannot reach a statement outside the loop body.

Note that illegal code, for instance code with uninitialized variables, can lead to surprising privatizations, which are still correct since the initial semantics is unspecified.

```
alias privatize_module 'Privatize Scalars'
privatize_module                    > MODULE.code
                                    > MODULE.privatized
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.chains
```

This pass is similar to `privatize_module` 9.7.11.1, but it also privatizes global scalar variables, using information from `live_paths` 6.13.1 analyses to avoid privatizing global variables which are the values of which are used afterwards, and callees `summary_effects` 6.2.4 to ensure that global variables are not used in callees. This last property is necessary to keep the transformation as local as possible. If it were not true, it would require to clone the sub call-trees.

```
alias privatize_module_even_globals 'Privatize Local and Global Scalars'
privatize_module_even_globals       > MODULE.code
                                    > MODULE.privatized
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
        < MODULE.chains
        < MODULE.live_out_paths
        < MODULE.live_in_paths
        < CALLEES.summary_effects
```

The results of this phase are not always as good as expected. this happens when global variables are also used/defined in called modules. Requiring that `privatize_module_even_globals` 9.7.11.1 is called on callees first (by using a bang rule), is not sufficient, because declaring variable as private does not remove the effects on the variables. Enforcing that `localize_declaration` 9.7.11.2 is first performed on the called modules partially solves the problems, except for loop indices of outermost loops which are not localized.

9.7.11.2 Declaration Localization

Use informations from `privatize_module` 9.7.11.1 to move `C` variable declarations as close as possible to their uses. For instance

```
int i,j;
for(i=0;i<10;i++)
        for(j=0;j<10;j++)
        ...
```

becomes

```
int i;
for(i=0;i<10;i++)
{
        int j;
        for(j=0;j<10;j++)
        ...
}
```

```
localize_declaration > MODULE.code
        < MODULE.privatized
        < PROGRAM.entities
        < MODULE.code
```

```
LOCALIZE_DECLARATION_SKIP_LOOP_INDICES FALSE
```

9.7.11.3 Array Privatization

Array privatization aims at privatizing whole arrays (`array_privatizer` 9.7.11.3) or sets of array elements (`array_section_privatizer` 9.7.11.3) instead of scalar variables only. The algorithm, developed by Béatrice Creusillet [20], is very different from the algorithm used for solely privatizing scalar variables and relies on IN and OUT regions analyses. Of course, it also privatizes scalar variables, although the algorithm is much more expensive and should be used only when necessary.

Array sections privatization is still experimental and should be used with great care. In particular, it is not compatible with the next steps of the parallelization process, i.e. dependence tests and code generation, because it does not modify the code, but solely produces a new region resource.

Another transformation, which can also be called a *privatization*, consists in declaring as *local* to a procedure or function the variables which are used only locally. This happens quite frequently in old Fortran codes where variables are declared as `SAVE`d to avoid allocations at each invocation of the routine. However, this prevents parallelization of the loop surrounding the calls. The function which performs this transformation is called `declarations_privatizer` 9.7.11.3.

```
alias array_privatizer 'Privatize Scalars & Arrays'
alias array_section_privatizer 'Scalar and Array Section Privatization'
alias declarations_privatizer 'Declaration Privatization'
```

```
array_privatizer             > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions

array_section_privatizer             > MODULE.code
                                     > MODULE.privatized_regions
                                     > MODULE.copy_out_regions
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions

declarations_privatizer              > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.regions
        < MODULE.in_regions
        < MODULE.out_regions
```

Several privatizability criterions can be applied for array section privatization, and its not yet clear which one should be used. The default case is to remove potential false dependences between iterations. The first option, when set to false, removes this constraint. It is useful for single assignment programs, to discover what section is really local to each iteration.

When the second option is set to false, copy-out is not allowed, which means only array regions that are not further reused in the program continuation can be privatized.

```
ARRAY_PRIV_FALSE_DEP_ONLY TRUE
```

```
ARRAY_SECTION_PRIV_COPY_OUT TRUE
```

9.7.12 Scalar and Array Expansion

Variable expansion consists in adding new dimensions to a variable so as to parallelize surrounding loops. There is no known advantage for expansion against privatization, but expansion is used when parallel loops must be distributed, for instance to generate SIMD code.

It is assumed that the variables to be expanded are the private variables. So this phase only is useful if a privatization has been performed earlier.

9.7.12.1 Scalar Expansion

Loop private scalar variables are expanded

```
alias variable_expansion 'Expand Scalar'
variable_expansion                    > MODULE.code
        < MODULE.privatized
        < PROGRAM.entities
        < MODULE.code
```

Uses `LOOP_LABEL` 9.1.1 to select a particular loop, then finds all reduction in this loop and performs variable expension on all reduction variables.

```
reduction_variable_expansion      > MODULE.code
        < PROGRAM.entities
        < MODULE.cumulated_reductions
        < MODULE.code
```

A variant of atomization that splits expressions but keep as much reduction as possible. E.g: r+=a+b becomes r+=a ; r+=b;

```
reduction_atomization      > MODULE.code
        < PROGRAM.entities
        < MODULE.cumulated_reductions
        < MODULE.code
```

9.7.12.2 Array Expansion

Not implemented yet.

9.7.13 Variable Length Array

These passes are designed to analyze C code.

These passes verify if the length variable of a variable length array is initialized and add some initilization if necessary.

For instance

```
int n;
int a[n];
...
```

the pass detects that `a[n]` needs `n` and verifies that `n` is initialized.

They have been developped by Nelson Lossing.

Note: Can `used_before_set` 7.3 be equivalent, but statically?

9.7.13.1 Check Initialize Variable Length Array

We can check the intialization of vla using two different ways:

The first one `check_initialize_vla_with_preconditions` 9.7.13.1 uses the preconditions and verifies that the variable has a value.

The other way `check_initialize_vla_with_effects` 9.7.13.1/ `check_initialize_vla_with_regions` 9 uses the effects/regions to check if the variable was written before use.

```
alias check_initialize_vla_with_preconditions 'Check Initialize Variable Length Array With
alias check_initialize_vla_with_effects 'Check Initialize Variable Length Array With Effec
alias check_initialize_vla_with_regions 'Check Initialize Variable Length Array With Regio
```

All these passes use property `ERROR_ON_UNINITIALIZE_VLA` 9.7.13.1 to generate a warning or an error when an uninitialized length variable is detected. Indeed, we can have some false positive due to the precision of used ressources, especially when the length variable is itself an array or a pointer.

```
ERROR_ON_UNINITIALIZE_VLA FALSE
```

`check_initialize_vla_with_preconditions` 9.7.13.1 uses the preconditions to check whether the length variable is initialized.

For this purpose, it checks if the length variable has a value, but also verifies, if it is possible, that this value is greater than or equal to zero. It only checks if it is greater than or equal to zero because the passes 9.7.13.2 initialize them at 0 by default.

The norm doesn't allow array initialization with 0. But gcc and clang allow it with a warning (-pedantic).

It may be useful to activate `SEMANTICS_ANALYZE_CONSTANT_PATH` 6.9.4.1 and/or `SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES` 6.9.4.1 to reduce false positive if the length variable is itself an array or a pointer. But it does not work everytime. (it's not implemented yet.)

```
check_initialize_vla_with_preconditions
  < PROGRAM.entities
  < MODULE.code
  < MODULE.preconditions
```

`check_initialize_vla_with_effects` 9.7.13.1 is not implemented yet.

It can required to set `VLA_EFFECT_READ` 6.2.7.4 as `TRUE`? Depending if cumulated effects or out effects are used.

The idea is to check for each vla declaration, if the length variable is exactly written before the declaration (It can correspond to an exact write, or to be present in exact out effects with `VLA_EFFECT_READ` 6.2.7.4 at `TRUE`).

This method doesn't permit to verify if the length variable is positive.

This pass can only be run bottom-up.

```
check_initialize_vla_with_effects
  < PROGRAM.entities
  < MODULE.code
  < MODULE.cumulated_effects
```

`check_initialize_vla_with_regions` 9.7.13.1 is not implemented yet.

It can required to set `VLA_EFFECT_READ` 6.2.7.4 as `TRUE`? Depending if regions or out regions are used.

The idea is to check for each vla declaration, if the length variable is exactly written before the declaration (It can correspond to an exact write, or to be present in exact out regions with `VLA_EFFECT_READ` 6.2.7.4 at `TRUE`).

This method doesn't permit to verify if the length variable is positive.

It may suppress some false positive for vla declare with an array cell as length.

This pass can only be run bottom-up.

```
check_initialize_vla_with_regions
  < PROGRAM.entities
  < MODULE.code
  < MODULE.regions
```

9.7.13.2 Initialize Variable Length Array

Similarly to the check 9.7.13.1, there is two different ways to determine the variable length to initialize: by using the preconditions or by using the effects/regions.

So we have different passes to generate the initialization.

```
alias initialize_vla_with_preconditions 'Initialize Variable Length Array With Preconditio
alias initialize_vla_with_effects 'Initialize Variable Length Array With Effects'
alias initialize_vla_with_regions 'Initialize Variable Length Array With Regions'
```

The only thing that differs in these passes is how the list of variables to initialize is generated, and so the PIPS dependence ressources that will be needed.

These passes initialize length variables at declaration times for scalar. When the length variable is a cell of another array, it is initialized just after the declaration of the array (not implemented yet). For instance,

```
int n, l[1];
...
int a[n];
int b[l[0]];
...
```

becomes

```
int n=0, l[1];
l[0]=0;
...
int a[n];
int b[l[0]];
...
```

The passes uses the propoerty `INITIALIZE_VLA_VALUE` 9.7.13.2 to know the initial value, 0 by default.

```
INITIALIZE_VLA_VALUE 0
```

```
initialize_vla_with_preconditions > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.preconditions
```

```
initialize_vla_with_effects > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.cumulated_effects


initialize_vla_with_regions > MODULE.code
  < PROGRAM.entities
  < MODULE.code
  < MODULE.regions
```

9.7.14 Freeze variables

Function `freeze_variables` 9.7.14 produces code where variables interactively specified by the user are transformed into constants. This is useful when the functionality of a code must be reduced. For instance, a code designed for N dimensions could be reduced to a 3-D code by setting N to 3. This is not obvious when N changes within the code. This is useful to specialize a code according to specific input data[7].

CA? More information? The variable names are requested from the PIPS user?

```
alias freeze_variables 'Freeze Variables'
freeze_variables                    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.cumulated_effects
```

9.8 Miscellaneous transformations

The following warning paragraphs should not be located here, but the whole introduction has to be updated to take into account the merger with properties-rc.tex, the new content (the transformation section has been exploded) and the new passes such as gpips. No time right now. FI.

All PIPS transformations assume that the initial code is legal according to the language standard. In other words, its semantics is well defined. Otherwise, it is impossible to maintain a constant semantics through program transformations. So uninitialized variables, for instance, can lead to codes that seem wrong, because they are likely to give different outputs than the initial code. But this does not matter as the initial code output is undefined and could well be the new output,

Also, remember that dead code does not impact the semantics in an observable way. Hence dead code can be transformed in apparently weird ways. For instance, all loops that are part of a dead code section can be found parallel, although they are obviously sequential, because all the references will carry an unfeasible predicate. In fact, reference A(I), placed in a dead code section, does not reference the memory and does not have to be part of the dependence graph.

Dead code can crop out in many modules when a whole application linked with a library is analyzed. All unused library modules are dead for PIPS.

[7] See the CHiLL tool.

On the other hand, missing source modules synthesized by PIPS may also lead to weird results because they are integrated in the application with empty definitions. Their call sites have no impact on the application semantics.

9.8.1 Type Checker

Typecheck code according to Fortran standard + double-complex. Typechecking is performed interprocedurally for user-defined functions. Insert type conversions where implicitly needed. Use typed intrinsics instead of generic ones. Precompute constant conversions if appropriate (e.g. 16 to 16.0E0). Add comments about type errors detected in the code. Report back how much was done.

```
type_checker            > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.code
```

Here are type checker options. Whether to deal with double complex or to refuse them. Whether to add a summary of errors, conversions and simplifications as a comment to the routine. Whether to always show complex constructors.

```
TYPE_CHECKER_DOUBLE_COMPLEX_EXTENSION FALSE
```

```
TYPE_CHECKER_LONG_DOUBLE_COMPLEX_EXTENSION FALSE
```

```
TYPE_CHECKER_ADD_SUMMARY FALSE
```

```
TYPE_CHECKER_EXPLICIT_COMPLEX_CONSTANTS FALSE
```

9.8.2 Manual Editing

The window interfaces let the user edit the source files, because it is very useful to demonstrate `PIPS`. As with `stf` 9.3.5, editing is not integrated like other program transformations, and previously applied transformations are lost. Consistency is however always preserved.

A general edit facility fully integrated in `pipsmake` is planned for the (not so) near future. Not so near because user demand for this feature is low.

Since `tpips` can invoque any Shell command, it is also possible to touch and edit source files.

9.8.3 Transformation Test

This is plug to implement quickly a program transformation requested by a user. Currently, it is a full loop distribution suggested by Alain Darte to compare different implementations, namely Nestor and PIPS.

```
alias transformation_test 'Transformation Test'

transformation_test    > MODULE.code
        < PROGRAM.entities
        < MODULE.code
```

9.9 Extensions Transformations

9.9.1 OpenMP Pragma

The following transformation reads the sequential code and generates OpenMP pragma as an extension to statements. The pragmas produced are based on the information previously computed by differents phases and already stores in the pips internal representation of the sequential code. It might be interesting to use the phase internalize_parallel_code (see § 8.1.8) before to apply ompify_code in order to maximize the number of parallel information available.

```
ompify_code             > MODULE.code
        < MODULE.code
```

As defined in the ri, the pragma can be of different types. The following property can be set to `str` or `expr`. Obviously, if the property is set to `str` then pragmas would be generated as strings otherwise pragmas would be generated as expressions.

```
PRAGMA_TYPE "expr"
```

The PIPS phase OMP_LOOP_PARALLEL_THRESHOLD_SET allows to add the OpenMP `if` clause to all the OpenMP pragmas. Afterwards, the number of iterations in the loop is evaluated dynamically and compared to the defined threshold. The loop is parallelized only if the threshold is reached.

```
omp_loop_parallel_threshold_set             > MODULE.code
        < MODULE.code
```

The OMP_LOOP_PARALLEL_THRESHOLD_VALUE property , is used as a parameter by the PIPS phase OMP_LOOP_PARALLEL_THRESHOLD_SET. The number of iteration of the parallel loop will be compared to that value in an omp if clause. The OpenMP run time will decide dynamicaly to parallelize the loop if the number of iteration is above this threshold.

```
OMP_LOOP_PARALLEL_THRESHOLD_VALUE 0
```

The OMP_IF_CLAUSE_RECURSIVE property , is used as a parameter by the PIPS phase OMP_LOOP_PARALLEL_THRESHOLD_SET. If set to TRUE the number of iterations of the inner loops will be used to test if the threshold is reached. Otherwise only the nunber of iteration of the processed loop will be used.

```
OMP_IF_CLAUSE_RECURSIVE TRUE
```

Compiler tends to produce many parallel loops which is generally not optimal for performance. The following transformation merges nested omp pragma in a unique omp pragma.

```
omp_merge_pragma              > MODULE.code
        < MODULE.code
```

PIPS merges the omp pragma on the inner or outer loop depending on the property OMP_MERGE_POLICY. This string property can be set to either `outer` or `inner`.

```
OMP_MERGE_POLICY "outer"
```

The OMP_MERGE_PRAGMA phase with the `inner` mode can be used after the phase limit_nested_parallelism (see § 8.1.11). Such a combinaison allows to fine choose the loop depth you really want to parallelize with OpenMP.

The merging of the if clause of the omp pragma follows its own rule. This clause can be ignore without changing the output of the program, it only changes the program perfomances. Then three policies are offered to manage the if clause merging. The if clause can simply be ignored. Or the if clauses can be merged alltogether using the boolean opertaion `or` or `and`. When ignored, the if clause can be later regenerated using the appropriated PIPS phase : OMP_LOOP_PARALLEL_THRESHOLD_SET. To summarize, remenber that the property can be set to `ignore` `or` or `and`

```
OMP_IF_MERGE_POLICY "ignore"
```

Chapter 10

Output Files (Prettyprinted Files)

PIPS results for any analysis and/or transformations can be displayed in several different formats. User views are the closest one to the initial user source code. Sequential views are obtained by prettyprinting the PIPS internal representation of modules. Code can also be displayed graphically or using Emacs facilities (through a property). Of course, parallelized versions are available. At the program level, call graph and interprocedural control flow graphs, with different degrees of ellipse, provide interesting summaries.

Dependence graphs can be shown, but they are not user-friendly. No filtering interface is available. They mainly are useful for debugging and for teaching purposes.

10.1 Parsed Printed Files (User View)

These are files containing a pretty-printed version of the *parsed* code, before the controlizer is applied. It is the code display closest to the user source code, because arcs in control flow graphs do not have to be rewritten as GOTO statements. However, it is inconsistent with the internal representation of the code as soon a a code transformation has been applied.

Bug: the inconsistence between the user view and the internal code representation presently is not detected. Solution: do not use user views.

The Fortran statements may be decorated with preconditions or transformers or complexities or any kind of effects, including convex array regions,... depending on the prettyprinter selected used to produce this file.

Transformers and preconditions require cumulated effects to build the module value basis.

10.1.1 Menu for User Views

```
alias parsed_printed_file 'User View'

alias print_source 'Basic'
alias print_source_transformers 'With Transformers'
```

```
alias print_source_preconditions 'With Preconditions'
alias print_source_total_preconditions 'With Total Preconditions'
alias print_source_regions 'With Regions'
alias print_source_in_regions 'With IN Regions'
alias print_source_out_regions 'With OUT Regions'
alias print_source_complexities 'With Complexities'
alias print_source_proper_effects 'With Proper Effects'
alias print_source_cumulated_effects 'With Cumulated Effects'
alias print_source_in_effects 'With IN Effects'
alias print_source_out_effects 'With OUT Effects'
alias print_source_continuation_conditions 'With Continuation Conditions'
```

10.1.2 Standard User View

Display the code without any decoration.

```
print_source         > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
```

10.1.3 User View with Transformers

Display the code decorated with the transformers.

```
print_source_transformers         > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.1.4 User View with Preconditions

Display the code decorated with the preconditions.

```
print_source_preconditions        > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.preconditions
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < MODULE.cumulated_effects
```

10.1.5 User View with Total Preconditions

Display the code decorated with the total preconditions.

```
print_source_total_preconditions        > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
```

```
        < MODULE.total_preconditions
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < MODULE.cumulated_effects
```

10.1.6 User View with Continuation Conditions

Display the code decorated with the continuation conditions.

```
print_source_continuation_conditions   > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.must_continuation
        < MODULE.may_continuation
        < MODULE.must_summary_continuation
        < MODULE.may_summary_continuation
        < MODULE.cumulated_effects
```

10.1.7 User View with Convex Array Regions

Display the code decorated with the regions.

```
print_source_regions              > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.1.8 User View with Invariant Convex Array Regions

Display the code decorated with the regions.

```
print_source_inv_regions              > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.inv_regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.1.9 User View with IN Convex Array Regions

Display the code decorated with the IN regions.

```
print_source_in_regions              > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
```

```
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.1.10 User View with OUT Convex Array Regions

Display the code decorated with the OUT regions.

```
print_source_out_regions              > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.1.11 User View with Complexities

Display the code decorated with the complexities.

```
print_source_complexities         > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.complexities
        < MODULE.summary_complexity
```

10.1.12 User View with Proper Effects

Display the code decorated with the proper effects.

```
print_source_proper_effects       > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        <  MODULE.proper_effects
```

10.1.13 User View with Cumulated Effects

Display the code decorated with the cumulated effects.

```
print_source_cumulated_effects    > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.1.14 User View with IN Effects

Display the code decorated with its IN effects.

```
print_source_in_effects       > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_effects
        < MODULE.in_summary_effects
```

10.1.15 User View with OUT Effects

Display the code decorated with its OUT effects.

```
print_source_out_effects       > MODULE.parsed_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.out_effects
        < MODULE.out_summary_effects
```

10.2 Printed File (Sequential Views)

These are files containing a pretty-printed version of the internal representation, `code`.

The statements may be decorated with the result of any analysis, e.g.complexities, preconditions, transformers, convex array regions,... depending on the pretty printer used to produce this file.

To view C programs, it is a good idea to select a C pretty printer, for example in `tpips` with:

```
setproperty PRETTYPRINT_C_CODE TRUE (obsolete, replaced by PRETTYPRINT_LANGUAGE ``C'')
```

Transformers and preconditions (and regions?) require cumulated effects to build the module value basis.

10.2.1 Html output

This is intended to be used with PIPS IR Navigator (tm).

Produce a html version of the internal representation of a PIPS Module. The property `HTML_PRETTYPRINT_SYMBOL_TABLE` 10.2.1 control whether the symbol table should be included in the output.

```
html_prettyprint > MODULE.html_ir_file
         < PROGRAM.entities
         < MODULE.code
```

Produce a html version of the symbol table, it's module-independent, it'll produce the same output for each module (the symbol table is global/unique).

```
html_prettyprint_symbol_table > PROGRAM.html_ir_file
         < PROGRAM.entities
         < MODULE.code
```

```
HTML_PRETTYPRINT_SYMBOL_TABLE FALSE
```

10.2.2 Menu for Sequential Views

```
alias printed_file 'Sequential View'

alias print_code 'Statements Only'
alias print_code_transformers 'Statements & Transformers'
alias print_code_complexities 'Statements & Complexities'
alias print_code_preconditions 'Statements & Preconditions'
alias print_code_total_preconditions 'Statements & Total Preconditions'
alias print_code_regions 'Statements & Regions'
alias print_code_regions 'Statements & Invariant Regions'
alias print_code_complementary_sections 'Statements & Complementary Sections'
alias print_code_in_regions 'Statements & IN Regions'
alias print_code_out_regions 'Statements & OUT Regions'
alias print_code_privatized_regions 'Statements & Privatized Regions'
alias print_code_proper_effects 'Statements & Proper Effects'
alias print_code_in_effects 'Statements & IN Effects'
alias print_code_out_effects 'Statements & OUT Effects'
alias print_code_cumulated_effects 'Statements & Cumulated Effects'
alias print_code_proper_reductions 'Statements & Proper Reductions'
alias print_code_cumulated_reductions 'Statements & Cumulated Reductions'
alias print_code_static_control 'Statements & Static Controls'
alias print_code_continuation_conditions 'Statements & Continuation Conditions'
alias print_code_proper_regions 'Statements & Proper Regions'
alias print_code_proper_references 'Statements & Proper References'
alias print_code_cumulated_references 'Statements & Cumulated References'
alias print_initial_precondition 'Initial Preconditions'
alias print_code_points_to_list 'Statements & Points To'
alias print_code_simple_pointer_values 'Statements & Simple Pointer Values'
```

10.2.3 Standard Sequential View

Display the code without any decoration.

```
print_code                      > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
```

10.2.4 Sequential View with Transformers

Display the code statements decorated with their transformers, except for loops, which are decorated with the transformer from the loop entering states to the loop body states. The effective loop transformer, linking the input to the output state of a loop, is recomputed when needed and can be deduced from the

precondition of the next statement after the loop[1].

```
print_code_transformers         > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.2.5 Sequential View with Initial Preconditions

```
print_initial_precondition  > MODULE.printed_file
        < MODULE.initial_precondition
        < PROGRAM.entities

print_program_precondition  > PROGRAM.printed_file
        < PROGRAM.program_precondition
        < PROGRAM.entities
```

10.2.6 Sequential View with Complexities

Display the code decorated with the complexities.

```
print_code_complexities         > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.complexities
        < MODULE.summary_complexity
```

10.2.7 Sequential View with Preconditions

Display the code decorated with the preconditions.

```
print_code_preconditions        > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < MODULE.summary_precondition
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

[1]PIPS design maps statement to decorations. For one loop statement, we need two transformers: one transformer to propagate the loop precondition as loop body precondition and a second transformer to propagate the loop precondition as loop postcondition. The second transformer can be deduced from the first one, but not the first one from the second one, and the second transformer is not used to compute the loop postcondition as it is more accurate to use the body postcondition. It is however computed to derive a compound statement transformer, e.g. the loop is part of block, which is part of a module statement, and then junked.

10.2.8 Sequential View with Total Preconditions

Display the code decorated with the total preconditions.

```
print_code_total_preconditions        > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.total_preconditions
        < MODULE.summary_total_precondition
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.2.9 Sequential View with Continuation Conditions

Display the code decorated with the continuation preconditions.

```
print_code_continuation_conditions   > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.parsed_code
        < MODULE.must_continuation
        < MODULE.may_continuation
        < MODULE.must_summary_continuation
        < MODULE.may_summary_continuation
        < MODULE.cumulated_effects
```

10.2.10 Sequential View with Convex Array Regions

10.2.10.1 Sequential View with Plain Pointer Regions

Display the code decorated with the pointer regions.

```
print_code_pointer_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.pointer_regions
        < MODULE.summary_pointer_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.2 Sequential View with Proper Pointer Regions

Display the code decorated with the proper pointer regions.

```
print_code_proper_pointer_regions          > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_pointer_regions
        < MODULE.summary_pointer_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.3 Sequential View with Invariant Pointer Regions

Display the code decorated with the invariant read/write pointer regions.

```
print_code_inv_pointer_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.inv_pointer_regions
        < MODULE.summary_pointer_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.4 Sequential View with Plain Convex Array Regions

Display the code decorated with the regions.

```
print_code_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.5 Sequential View with Proper Convex Array Regions

Display the code decorated with the proper regions.

```
print_code_proper_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.6 Sequential View with Invariant Convex Array Regions

Display the code decorated with the invariant read/write regions.

```
print_code_inv_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.inv_regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.7 Sequential View with IN Convex Array Regions

Display the code decorated with the IN regions.

```
print_code_in_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.8 Sequential View with OUT Convex Array Regions

Display the code decorated with the OUT regions.

```
print_code_out_regions              > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.10.9 Sequential View with Privatized Convex Array Regions

Display the code decorated with the privatized regions.

```
print_code_privatized_regions       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.privatized_regions
        < MODULE.copy_out_regions
```

10.2.11 Sequential View with Complementary Sections

Display the code decorated with complementary sections.

```
print_code_complementary_sections > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.compsec
        < MODULE.summary_compsec
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.2.12 Sequential View with Proper Effects

Display the code decorated with the proper pointer effects.

```
print_code_proper_pointer_effects       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_pointer_effects
```

Display the code decorated with the proper effects.

```
print_code_proper_effects       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

Display the code decorated with the proper references.

```
print_code_proper_references       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_references
```

10.2.13 Sequential View with Cumulated Effects

Display the code decorated with the cumulated effects.

```
print_code_cumulated_pointer_effects    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_pointer_effects
        < MODULE.summary_pointer_effects
```

Display the code decorated with the cumulated effects.

```
print_code_cumulated_effects    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

Display the code decorated with the cumulated references.

```
print_code_cumulated_references    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_references
```

10.2.14 Sequential View with IN Effects

Display the code decorated with its IN effects.

```
print_code_in_effects       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_effects
        < MODULE.in_summary_effects
```

10.2.15 Sequential View with OUT Effects

Display the code decorated with its OUT effects.

```
print_code_out_effects       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.out_effects
        < MODULE.out_summary_effects
```

10.2.16 Sequential View with Live Paths

Display the code decorated with the live in paths.

```
print_code_live_in_paths    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.live_in_paths
        < MODULE.live_in_summary_paths
```

Display the code decorated with the live out paths.

```
print_code_live_out_paths    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.live_out_paths
        < MODULE.live_out_summary_paths
```

Display the code decorated with the live out regions.

```
print_code_live_out_regions    > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.live_out_regions
```

10.2.17 Sequential View with Proper Reductions

Display the code decorated with the proper reductions.

```
print_code_proper_reductions > MODULE.printed_file
  < PROGRAM.entities
  < MODULE.code
  < MODULE.proper_reductions
```

10.2.18 Sequential View with Cumulated Reductions

Display the code decorated with the cumulated reductions.

```
print_code_cumulated_reductions > MODULE.printed_file
  < PROGRAM.entities
  < MODULE.code
  < MODULE.cumulated_reductions
  < MODULE.summary_reductions
```

10.2.19 Sequential View with Static Control Information

Display the code decorated with the static control.

```
print_code_static_control       > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.static_control
```

10.2.20 Sequential View with Points-To Information

Display the code decorated with the points to information.

```
print_code_points_to_list      > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.points_to
```

10.2.21 Sequential View with Simple Pointer Values

Displays the code with simple pointer values relationships.

```
print_code_simple_pointer_values      > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.simple_pointer_values
```

Displays the code with simple gen pointer values and kill sets.

```
print_code_simple_gen_kill_pointer_values      > MODULE.printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.simple_gen_pointer_values
        < MODULE.simple_kill_pointer_values
```

10.2.22 Prettyprint Properties

10.2.22.1 Language

PIPS can handle many different languages. By default the PrettyPrinter uses the native language as an output but it is also possible to prettyprint Fortran code as C code. Possible values for the PRETTYPRINT_LANGUAGE property are: `native F95 F77 C`.

```
PRETTYPRINT_LANGUAGE "native"
```

10.2.22.2 Layout

When prettyprinting semantic information (preconditions, transformers and regions), add a line before and after each piece of information if set to `TRUE`. The resulting code is more readable, but is larger.

```
PRETTYPRINT_LOOSE TRUE
```

By default, each prettyprinted line of Fortran or C code is terminated by its statement number in columns 73-80, unless no significative statement number is available. This feature is used to trace the origin of statements after program transformations and parallelization steps.

This feature may be inconvenient for some compilers or because it generates large source files. It may be turned off.

Note that the statement number is equal to the line number in the function file, that is the source file obtained after PIPS preprocessing[2] and filtering[3], and not the user file, which is the file submitted by the user and which may contain several functions.

Note also that some phases in `pips` may add new statement that are not present in the original file. In this case the number of the statement that requires such a transformation, is used for the added statement.

```
PRETTYPRINT_STATEMENT_NUMBER TRUE
```

Note: this default value is overriden to FALSE by `activate_language()` for C and Fortran 95.

The structured control structure is shown by using an indentation. The default value is 3.

```
PRETTYPRINT_INDENTATION 3
```

Some people prefer to use a space after a comma to separate items in lists such as declaration lists or parameter lists in order to improve readability. Other people would rather pack more information per line. The default option is chosen for readability.

```
PRETTYPRINT_LISTS_WITH_SPACES TRUE
```

Depending on the user goal, it may be better to isolate comments used to display results of PIPS analyses from the source code statement. This is the default option.

```
PRETTYPRINT_ANALYSES_WITH_LF TRUE
```

These qualifiers are not parsed and the prettyprinter ignores them by default, set the following properties for seeing them. They are set by `gpu_qualify_pointers`.

```
PRETTYPRINT_GLOBAL_QUALIFIER ""
```

[2]PIPS preprocessing usually includes the standard C or Fortran preprocessing phase but also breaks down user files into compilation units and function files, a.k.a. initial files in Fortran and source files in C.

[3]Filtering is applied on Fortran files only to perform file includes. It is implemented in Perl.

```
PRETTYPRINT_LOCAL_QUALIFIER ""
```

```
PRETTYPRINT_CONSTANT_QUALIFIER ""
```

```
PRETTYPRINT_PRIVATE_QUALIFIER ""
```

This feature only exists for the semantics analyses.

How to prettyprint C integer types:

```
PRETTYPRINT_C_CHAR_TYPE "char"
```

```
PRETTYPRINT_C_SHORT_TYPE "short"
```

```
PRETTYPRINT_C_INT_TYPE "int"
```

```
PRETTYPRINT_C_LONG_TYPE "long␣int"
```

```
PRETTYPRINT_C_LONGLONG_TYPE "long␣long␣int"
```

```
PRETTYPRINT_C_INT128_TYPE "__int128_t"
```

```
PRETTYPRINT_C_UCHAR_TYPE "unsigned␣char"
```

```
PRETTYPRINT_C_USHORT_TYPE "unsigned␣short"
```

```
PRETTYPRINT_C_UINT_TYPE "unsigned␣int"
```

```
PRETTYPRINT_C_ULONG_TYPE "unsigned␣long␣int"
```

```
PRETTYPRINT_C_ULONGLONG_TYPE "unsigned␣long␣long␣int"
```

```
PRETTYPRINT_C_UINT128_TYPE "__uint128_t"
```

```
PRETTYPRINT_C_SCHAR_TYPE "signed␣char"
```

```
PRETTYPRINT_C_SSHORT_TYPE "signed␣short"
```

```
PRETTYPRINT_C_SINT_TYPE "signed␣int"
```

```
PRETTYPRINT_C_SLONG_TYPE "signed␣long␣int"
```

```
PRETTYPRINT_C_SLONGLONG_TYPE "signed␣long␣long␣int"
```

10.2.22.3 Target Language Selection

10.2.22.3.1 Parallel output style How to print, from a syntactic point of view, a parallel do loop. Possible values are: `do doall f90 hpf cray craft cmf omp`.

```
PRETTYPRINT_PARALLEL "do"
```

10.2.22.3.2 Default sequential output style How to print, from a syntactic point of view, a parallel do loop for a sequential code. Of course, by default, the sequential output is sequential by definition, so the default value is `"do"`.

But we may interested to change this behaviour to display after an application of `internalize_parallel_code` 8.1.8 the parallel code that is hidden in the sequential code. Possible values are: `do doall f90 hpf cray craft cmf omp`.

By default, parallel information is displayed with am OpenMP flavor since it is widely used nowadays.

```
PRETTYPRINT_SEQUENTIAL_STYLE "omp"
```

10.2.22.4 Display Analysis Results

Add statement effects as comments in output; not implemented (that way) yet.

```
PRETTYPRINT_EFFECTS FALSE
```

The next property, `PRETTYPRINT_IO_EFFECTS` 10.2.22.4, is used to control the computation of implicit statement IO effects and display them as comments in output. The implicit effects on the logical unit are simulated by a read/write action to an element of the array `TOP-LEVEL:LUNS()`, or to the whole array when the element is not known at compile time. This is the standard behavior for PIPS. Some phases, e.g. hpfc, may turn this option off, but it is much more risky than to filter out abstract effects. Furthermore, the filtering is better because it takes into account all abstract effects, not only IO effects on logical units. PIPS users should definitely not turn off this property as the semantic equivalence between the inout and the output program is no longer guaranteed.

```
PRETTYPRINT_IO_EFFECTS TRUE
```

To transform C source code properly, variable and type declarations as well as variable and type references must be tracked alhtough standard use and def information is restricted to memory loads and stores because the optimizations are performed at a lower level. Fortran 77 analyses do not need information about variable declarations and there is not possibility of type definition. So the added information about variable declarations and references may be pure noise. It is possible to get rid of it by setting this property to TRUE, which is its default value before August 2010. For C code, it is better to set it to FALSE. For the time being, the default value cannot depend on the code language.

```
PRETTYPRINT_MEMORY_EFFECTS_ONLY FALSE
```

Transform DOALL loops into sequential loops with an opposed increment to check validity of the parallelization on a sequential machine. This property is not implemented.

```
PRETTYPRINT_REVERSE_DOALL FALSE
```

It is possible to print statement transformers as comments in code. This property is not intended for PIPS users, but is used internally. Transformers can be prettyprinted by using `activate` and `PRINT_CODE_TRANSFORMERS`

```
PRETTYPRINT_TRANSFORMER FALSE
```

It is possible to print statement preconditions as comments in code. This property is not intended for PIPS users, but is used internally. Preconditions can be prettyprinted by using `activate` and `PRINT_CODE_PRECONDITIONS`

```
PRETTYPRINT_EXECUTION_CONTEXT FALSE
```

It is possible to print statement with convex array region information as comments in code. This property is not intended for PIPS users, but is used internally. Convex array regions can be prettyprinted by using `activate` and `PRINT_CODE_REGIONS` or `PRINT_CODE_PROPER_REGIONS`

```
PRETTYPRINT_REGION FALSE
```

By default, convex array regions are printed for arrays only, but the internal representation includes scalar variables as well. The default option can be overriden with this property.

```
PRETTYPRINT_SCALAR_REGIONS FALSE
```

10.2.22.5 Display Internals for Debugging

All these debugging options should be set to FALSE for normal operation, when the prettyprinter is expected to produce code as close as possible to the input form. When they are turned on, the output is closer to the PIPS internal representation.

Sequences are implicit in Fortran and in many programming languages but they are internally represented. It is possible to print pieces of information gathered about sequences by turning on this property.

```
PRETTYPRINT_BLOCKS FALSE
```

To print all the C blocks (the `{ }` in C, you can set the following property:

```
PRETTYPRINT_ALL_C_BLOCKS FALSE
```

This property is a C-specialized version of `PRETTYPRINT_BLOCKS`, since in C you can represent the blocks. You can combine this property with a `PRETTYPRINT_EMPTY_BLOCKS` set to true too. Right now, the prettyprint of the C block is done in the wrong way, so if you use this option, you will have redundant blocks inside instructions, but you will have all the other hidden blocks too...

To print unstructured statements:

```
PRETTYPRINT_UNSTRUCTURED FALSE
```

Print all effects for all statements regardless of PRETTYPRINT_BLOCKS 10.2.22.5 and PRETTYPRINT_UNSTRUCTURED 10.2.22.5.

```
PRETTYPRINT_ALL_EFFECTS FALSE
```

Print empty statement blocks (false by default):

```
PRETTYPRINT_EMPTY_BLOCKS FALSE
```

Print statement ordering information (false by default):

```
PRETTYPRINT_STATEMENT_ORDERING FALSE
```

The next property controls the print out of DO loops and CONTINUE statement. The code may be prettyprinted with DO label and CONTINUE instead of DO-ENDDO, as well as with other useless CONTINUE (This property encompasses a virtual PRETTYPRINT_ALL_CONTINUE_STATEMENTS). If set to FALSE, the default option, all useless CONTINUE statements are NOT prettyprinted (ie. all those in structured parts of the code). This mostly is a debugging option useful to understand better what is in the internal representation.

10.2.22.5.1 Warning: if set to TRUE, generated code may be wrong after some code transformations like distribution...

```
PRETTYPRINT_ALL_LABELS FALSE
```

Print code with DO label as comment.

```
PRETTYPRINT_DO_LABEL_AS_COMMENT FALSE
```

Print private variables without regard for their effective use. By default, private variables are shown only for parallel DO loops.

```
PRETTYPRINT_ALL_PRIVATE_VARIABLES FALSE
```

Non-standard variables and tests are generated to simulate the control effect of Fortran IO statements. If an end-of-file condition is encountered or if an io-error is raised, a jump to relevant labels may occur if clauses ERR= or END= are defined in the IO control list. These tests are normally not printed because they could not be compiled by a standard Fortran compiler and because they are redundant with the IO statement itself.

```
PRETTYPRINT_CHECK_IO_STATEMENTS FALSE
```

Print the final RETURN statement, although this is useless according to Fortran standard. Note that comments attached to the final return are lost if it is not printed. Note also that the final RETURN may be part of an unstructured in which case the previous property is required.

```
PRETTYPRINT_FINAL_RETURN FALSE
```

The internal representation is based on a standard IF structure, known as *block if* in Fortran jargon. When possible, the prettyprinter uses the *logical if* syntactical form to save lines and to produce an output assumed closer to the input. When statements are decorated, information gathered by PIPS may be lost. This property can be turned on to have an output closer to the internal

representation. Note that edges of the control flow graphs may still be displayed as *logical if* since they never carry any useful information[4].

```
PRETTYPRINT_BLOCK_IF_ONLY FALSE
```

Effects give data that may be read and written in a procedure. These data are represented by their entity name. By default the entity name used is the shortest nom-ambiguous one. The `PRETTYPRINT_EFFECT_WITH_FULL_ENTITY_NAME` 10.2.22.5.1 property can be used to force the usage of full entity name (module name + scope + local name).

```
PRETTYPRINT_EFFECT_WITH_FULL_ENTITY_NAME FALSE
```

In order to have information on the scope of commons, we need to know the common in which the entity is declared if any. To get this information the `PRETTYPRINT_WITH_COMMON_NAMES` 10.2.22.5.1 property has to set to TRUE.

```
PRETTYPRINT_WITH_COMMON_NAMES FALSE
```

By default, expressions are simplified according to operator precedences. It is possible to override this prettyprinting option and to reflect the abstract tree with redundant parentheses.

```
PRETTYPRINT_ALL_PARENTHESES FALSE
```

By default, the C prettyprinter uses a minimum of braces to improve readability. However, gcc advocates the use of more braces to avoid some ambiguities about else clauses. In order to run succesfully gcc with options -Wall and -Werror, it is possible to force the print-out of all possible braces.

```
PRETTYPRINT_ALL_C_BRACES FALSE
```

The previous property leads to hard to read source code. Property `PRETTYPRINT_GCC_C_BRACES` 10.2.22.5.1 is used to print only a few additional braces required by gcc to avoid *ambiguous else* warning messages.

```
PRETTYPRINT_GCC_C_BRACES FALSE
```

10.2.22.6 Declarations

By default in Fortran (and not in C), module declarations are preserved as huge strings to produce an output as close as possible to the input (see field `decls_text` in type `code`). However, large program transformations and code generation phases, e.g. hpfc, require updated declarations.

Regenerate all variable declarations, including those variables not declared in the user program. By default in Fortran, when possible, the user declaration *text* is used to preserve comments.

```
PRETTYPRINT_ALL_DECLARATIONS FALSE
```

[4]Information is carried by the vertices (i.e. nodes). A CONTINUE statement is generated to have an attachment node when some information must be stored and displayed.

If the prettyprint of the header and the declarations are done by PIPS, try to display the genuine comments. Unfortunately, there is no longer order relation between the comments and the declarations since these are sorted by PIPS. By default, do not try to display the comments when PIPS is generating the header.

```
PRETTYPRINT_HEADER_COMMENTS FALSE
```

How to regenerate the common declarations. It can be *none*, *declaration*, or *include*.

```
PRETTYPRINT_COMMONS "declaration"
```

`DATA` declarations are partially handled presently.

```
PRETTYPRINT_DATA_STATEMENTS TRUE
```

Where to put the dimension information, which must appear once. The default is associated to the type information. It can be associated to The type, or preferably to the common if any, or maybe to a dimension statement, which is not implemented.

```
PRETTYPRINT_VARIABLE_DIMENSIONS "type"
```

10.2.22.7 FORESYS Interface

Print transformers, preconditions and regions in a format accepted by Foresys and Partita. Not maintained.

```
PRETTYPRINT_FOR_FORESYS FALSE
```

10.2.22.8 HPFC Prettyprinter

To deal specifically with the prettyprint for hpfc.

```
PRETTYPRINT_HPFC FALSE
```

10.2.22.9 C Internal Prettyprinter

To unify code generation, Fortran intrinsics such as MIN, MAX, DIV, INT and MOD are always used. MAX and MIN are especially convenient as they are overloaded and accept any number of arguments. The generated code can be prettyprinted as is and C code be fixed with macroprocessing.

This does not work for MIN and MAX because of their varying number of arguments. Some code generating passes are using them with only two arguments, which make the code hard to read if easy to preprocess.

Another option is to let the prettyprinter deal with the issue and generate calls to some PIPS run-time functions or macros such as `pips_min()` and `pips_max`. This is the default behavior.

However, some passes such as function cloning use the prettyprinter and then the parser to regenerate an internal representation. To preserve the intrinsics used and to debug *PIPS*[5], the next property must be set.

[5] `http://www.cri.ensmp.fr/pips`

```
PRETTYPRINT_INTERNAL_INTRINSICS FALSE
```

To generate correct code, some transformations such as loop tiling, need to use the floor division with negative numbers. It can be implemented as PIPS run-time function or macro such as `pips_div()`. The default option is to keep the Fortran and C intrinsics "/".

```
PRETTYPRINT_DIV_INTRINSICS TRUE
```

10.2.22.10 Interface to Emacs

The following property tells PIPS to attach various Emacs properties for interactive purpose. Used internally by the Emacs pretyyprinter and the *epip* user interface.

```
PRETTYPRINT_ADD_EMACS_PROPERTIES FALSE
```

10.3 Printed Files with the Intraprocedural Control Graph

These are files containing a pretty-printed version of code to be displayed with its intraprocedural control graph as a graph, for example using the *uDrawGraph*[6] program (formerly known as daVinci) or DOT/GRAPHVIZ tools. More concretely, use some scripts like `pips_unstructured2daVinci` or `pips_unstructured2dot` to display graphically these `.pref-graph` files.

The statements may be decorated with complexities, preconditions, transformers, regions,... depending on the printer used to produce this file.

10.3.1 Menu for Graph Views

```
alias graph_printed_file 'Control Graph Sequential View'

alias print_code_as_a_graph 'Graph with Statements Only'
alias print_code_as_a_graph_transformers 'Graph with Statements & Transformers'
alias print_code_as_a_graph_complexities 'Graph with Statements & Complexities'
alias print_code_as_a_graph_preconditions 'Graph with Statements & Preconditions'
alias print_code_as_a_graph_total_preconditions 'Graph with Statements & Total Preconditio
alias print_code_as_a_graph_regions 'Graph with Statements & Regions'
alias print_code_as_a_graph_in_regions 'Graph with Statements & IN Regions'
alias print_code_as_a_graph_out_regions 'Graph with Statements & OUT Regions'
alias print_code_as_a_graph_proper_effects 'Graph with Statements & Proper Effects'
alias print_code_as_a_graph_cumulated_effects 'Graph with Statements & Cumulated Effects'
```

10.3.2 Standard Graph View

Display the code without any decoration.

[6] `http://www.informatik.uni-bremen.de/uDrawGraph`

```
print_code_as_a_graph                    > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
```

10.3.3 Graph View with Transformers

Display the code decorated with the transformers.

```
print_code_as_a_graph_transformers       > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.3.4 Graph View with Complexities

Display the code decorated with the complexities.

```
print_code_as_a_graph_complexities       > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.complexities
        < MODULE.summary_complexity
```

10.3.5 Graph View with Preconditions

Display the code decorated with the preconditions.

```
print_code_as_a_graph_preconditions      > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < MODULE.summary_precondition
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.3.6 Graph View with Preconditions

Display the code decorated with the preconditions.

```
print_code_as_a_graph_total_preconditions      > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.total_preconditions
        < MODULE.summary_total_postcondition
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.3.7 Graph View with Regions

Display the code decorated with the regions.

```
print_code_as_a_graph_regions              > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.3.8 Graph View with IN Regions

Display the code decorated with the IN regions.

```
print_code_as_a_graph_in_regions              > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.3.9 Graph View with OUT Regions

Display the code decorated with the OUT regions.

```
print_code_as_a_graph_out_regions              > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.3.10 Graph View with Proper Effects

Display the code decorated with the proper effects.

```
print_code_as_a_graph_proper_effects       > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
```

10.3.11 Graph View with Cumulated Effects

Display the code decorated with the cumulated effects.

```
print_code_as_a_graph_cumulated_effects    > MODULE.graph_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.3.12 ICFG Properties

This prettyprinter is NOT a call graph prettyprinter (see Section 6.1). Control flow information can be displayed and every call site is shown, possibly with some annotation like precondition or region

This prettyprinter uses the module codes in the workspace database to build the ICFG.

Print IF statements controlling call sites:

```
ICFG_IFs FALSE
```

Print DO loops enclosing call sites:

```
ICFG_DOs FALSE
```

It is possible to print the interprocedural control flow graph as text or as a graph using daVinci format. By default, the text output is selected.

```
ICFG_DV FALSE
```

To be destroyed:

```
ICFG_CALLEES_TOPO_SORT FALSE
```

```
ICFG_DRAW TRUE
```

ICFG default indentation when going into a function or a structure.

```
ICFG_INDENTATION 4
```

Debugging level (should be `ICFG_DEBUG_LEVEL` and numeric instead of boolean!):

```
ICFG_DEBUG FALSE
```

Effects are often much too numerous to produce a useful interprocedural control flow graph.

The integer property `RW_FILTERED_EFFECTS` 10.3.12 is used to specify a filtering criterion.

- 0: `READ_ALL`,
- 1: `WRITE_ALL`,
- 2: `READWRITE_ALL`,
- 3: `READ_END`,

- 4: WRITE_END,
- 5: READWRITE_END, .

```
RW_FILTERED_EFFECTS 0
```

10.3.13 Graph Properties

10.3.13.1 Interface to Graphics Prettyprinters

To output a code with a hierarchical view of the control graph with markers instead of a flat one. It purposes a display with a graph browser such as *daVinci*[7]:

```
PRETTYPRINT_UNSTRUCTURED_AS_A_GRAPH FALSE
```

and to have a decorated output with the hexadecimal addresses of the control nodes:

```
PRETTYPRINT_UNSTRUCTURED_AS_A_GRAPH_VERBOSE FALSE
```

10.4 Parallel Printed Files

File containing a pretty-printed version of a parallelized_code. Several versions are available. The first one is based on Fortran-77, extended with a DOALL construct. The second one is based on Fortran-90. The third one generates CRAY Research directives as comments if and only if the correspondent parallelization option was selected (see Sectionsubsection-parallelization).

No one knows why there is no underscore between parallel and printed...

10.4.1 Menu for Parallel View

```
alias parallelprinted_file 'Parallel View'

alias print_parallelized77_code 'Fortran 77'
alias print_parallelizedHPF_code 'HPF directives'
alias print_parallelizedOMP_code 'OMP directives'
alias print_parallelized90_code 'Fortran 90'
alias print_parallelizedcray_code 'Fortran Cray'
alias print_parallelizedMPI_code 'C MPI'
```

10.4.2 Fortran 77 Parallel View

Output a Fortran-77 code extended with DOALL parallel constructs.

```
print_parallelized77_code       > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

[7] http://www.informatik.uni-bremen.de/~davinci

10.4.3 HPF Directives Parallel View

Output the code decorated with HPF directives.

```
print_parallelizedHPF_code     > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

10.4.4 OpenMP Directives Parallel View

Output the code decorated with OpenMP (OMP) directives.

```
print_parallelizedOMP_code     > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

10.4.5 Fortran 90 Parallel View

Output the code with some Fortran-90 array construct style.

```
print_parallelized90_code       > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

10.4.6 Cray Fortran Parallel View

Output the code decorated with parallel Cray directives. Note that the Cray parallelization algorithm should have been used in order to match Cray directives for parallel vector processors.

```
print_parallelizedcray_code     > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
        < MODULE.cumulated_effects
```

10.5 Call Graph Files

This kind of file contains the sub call graph[8] of a module. Of course, the call graph associated to the MAIN module is the program call graph.

Each module can be decorated by *summary* information computed by *one* of `PIPS` analyses.

If one module has different callers, its sub call tree is replicated once for each caller[9].

No fun to read, but how could we avoid it with a text output? But it is useful to check large analyses.

The resource defined in this section is `callgraph_file` (note the missing underscore between call and graph in callgraph...). This is a *file* resource to be displayed, which cannot be loaded in memory by *pipsdbm*.

[8] It is not a graph but a tree.

[9] In the ICFG , the replication would occur for each call site.

Note that the input resource lists could be reduced to one resource, the decoration. `pipsmake` would deduce the other ones. There is no need for a transitive closure, but some people like it that way to make resource usage verification possible...

RK: explain... FI: no idea; we would like to display any set of resources, but the sets are too numerous to have a phase for each.

10.5.1 Menu for Call Graphs

Aliases for call graphs must de different from aliases for interprocedural control flow graphs (ICFG). A simple trick, a trailing SPACE character, is used.

```
alias callgraph_file 'Callgraph View'

alias print_call_graph 'Calls'
alias print_call_graph_with_complexities 'Calls & Complexities'
alias print_call_graph_with_preconditions 'Calls & Preconditions'
alias print_call_graph_with_total_preconditions 'Calls & Total Preconditions'
alias print_call_graph_with_transformers 'Calls & Transformers'
alias print_call_graph_with_proper_effects 'Calls & Proper effects'
alias print_call_graph_with_cumulated_effects 'Calls & Cumulated effects'
alias print_call_graph_with_regions 'Calls & Regions'
alias print_call_graph_with_in_regions 'Calls & In Regions'
alias print_call_graph_with_out_regions 'Calls & Out regions'
```

10.5.2 Standard Call Graphs

To have the call graph without any decoration.

```
print_call_graph                          > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
```

10.5.3 Call Graphs with Complexities

To have the call graph decorated with the complexities.

```
print_call_graph_with_complexities     > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.summary_complexity
        < MODULE.complexities
```

10.5.4 Call Graphs with Preconditions

To have the call graph decorated with the preconditions.

```
print_call_graph_with_preconditions      > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.summary_precondition
```

```
        < MODULE.summary_effects
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

10.5.5 Call Graphs with Total Preconditions

To have the call graph decorated with the total preconditions.

```
print_call_graph_with_total_preconditions      > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.summary_total_postcondition
        < MODULE.summary_effects
        < MODULE.total_preconditions
        < MODULE.cumulated_effects
```

10.5.6 Call Graphs with Transformers

To have the call graph decorated with the transformers.

```
print_call_graph_with_transformers       > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.summary_transformer
        < MODULE.summary_effects
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.5.7 Call Graphs with Proper Effects

To have the call graph decorated with the proper effects.

```
print_call_graph_with_proper_effects     > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.proper_effects
```

10.5.8 Call Graphs with Cumulated Effects

To have the call graph decorated with the cumulated effects.

```
print_call_graph_with_cumulated_effects  > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.5.9 Call Graphs with Regions

To have the call graph decorated with the regions.

```
print_call_graph_with_regions           > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.5.10 Call Graphs with IN Regions

To have the call graph decorated with the IN regions.

```
print_call_graph_with_in_regions        > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.5.11 Call Graphs with OUT Regions

To have the call graph decorated with the OUT regions.

```
print_call_graph_with_out_regions       > MODULE.callgraph_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.callgraph_file
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

This library is used to display the calling relationship between modules. It is different from the interprocedural call flow graph, ICFG (see Section 10.3.12). For example: if A calls B twice, in callgraph, there is only one edge between A and B; while in ICFG (see next section)), there are two edges between A and B, since A contains two call sites.

The call graph is derived from the modules declarations. It does not really the parsed code per se, but the code must have been parsed to have up-to-date declarations in the symbol table.

Because of printout limitations, the call graph is developed into a tree before it is printed. The sub-graph of a module appears as many times as is has callers. The resulting printout may be very long.

There is no option for the callgraph prettyprinter except for debugging.
Debugging level (should be `CALLGRAPH_DEBUG_LEVEL` and numeric!)

```
CALLGRAPH_DEBUG FALSE
```

10.6 DrawGraph Interprocedural Control Flow Graph Files (DVICFG)

This is the file ICFG with format of graph *uDrawGraph*[10] (formerly daVinci). This should be generalized to be less tool-dependent.

10.6.1 Menu for DVICFG's

```
alias dvicfg_file 'DVICFG View'

alias print_dvicfg_with_filtered_proper_effects 'Graphical Calls & Filtered proper effects
```

10.6.2 Minimal ICFG with graphical filtered Proper Effects

Display the ICFG graphically decorated with the write proper effects filtered for a variable.

```
print_dvicfg_with_filtered_proper_effects          > MODULE.dvicfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.dvicfg_file
        < MODULE.proper_effects
        < CALLEES.summary_effects
```

10.7 Interprocedural Control Flow Graph Files (ICFG)

This kind of file contains a more or less precise interprocedural control graph. The graph can be restricted to call sites only, to call sites and enclosing DO loops or to call sites, enclosing DO loops and controlling IF tests. This abstraction option is orthogonal to the set of decorations, but `pipsmake` does not support this orthogonality. All combinations are listed below.

Each call site can be decorated by associated information computed by one of `PIPS` analyses.

10.7.1 Menu for ICFG's

Note: In order to avoid conflicts with callgraph aliases, a space character is appended at each alias shared with call graph related functions (Guillaume Oget).

[10] `http://www.informatik.uni-bremen.de/uDrawGraph`

```
alias icfg_file 'ICFG View'

alias print_icfg 'Calls '
alias print_icfg_with_complexities 'Calls & Complexities '
alias print_icfg_with_preconditions 'Calls & Preconditions '
alias print_icfg_with_total_preconditions 'Calls & Total Preconditions '
alias print_icfg_with_transformers 'Calls & Transformers '
alias print_icfg_with_proper_effects 'Calls & Proper effects '
alias print_icfg_with_filtered_proper_effects 'Calls & Filtered proper effects '
alias print_icfg_with_cumulated_effects 'Calls & Cumulated effects '
alias print_icfg_with_regions 'Calls & Regions '
alias print_icfg_with_in_regions 'Calls & In Regions '
alias print_icfg_with_out_regions 'Calls & Out regions '

alias print_icfg_with_loops 'Calls & Loops'
alias print_icfg_with_loops_complexities 'Calls & Loops & Complexities'
alias print_icfg_with_loops_preconditions 'Calls & Loops & Preconditions'
alias print_icfg_with_loops_total_preconditions 'Calls & Loops & Total Preconditions'
alias print_icfg_with_loops_transformers 'Calls & Loops & Transformers'
alias print_icfg_with_loops_proper_effects 'Calls & Loops & Proper effects'
alias print_icfg_with_loops_cumulated_effects 'Calls & Loops & Cumulated effects'
alias print_icfg_with_loops_regions 'Calls & Loops & Regions'
alias print_icfg_with_loops_in_regions 'Calls & Loops & In Regions'
alias print_icfg_with_loops_out_regions 'Calls & Loops & Out regions'

alias print_icfg_with_control 'Calls & Control'
alias print_icfg_with_control_complexities 'Calls & Control & Complexities'
alias print_icfg_with_control_preconditions 'Calls & Control & Preconditions'
alias print_icfg_with_control_total_preconditions 'Calls & Control & Total Preconditions'
alias print_icfg_with_control_transformers 'Calls & Control & Transformers'
alias print_icfg_with_control_proper_effects 'Calls & Control & Proper effects'
alias print_icfg_with_control_cumulated_effects 'Calls & Control & Cumulated effects'
alias print_icfg_with_control_regions 'Calls & Control & Regions'
alias print_icfg_with_control_in_regions 'Calls & Control & In Regions'
alias print_icfg_with_control_out_regions 'Calls & Control & Out regions'
```

10.7.2 Minimal ICFG

Display the plain ICFG, without any decoration.

```
print_icfg                      > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
```

10.7.3 Minimal ICFG with Complexities

Display the ICFG decorated with complexities.

```
print_icfg_with_complexities            > MODULE.icfg_file
        < PROGRAM.entities
```

```
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_complexity
        < MODULE.complexities
```

10.7.4 Minimal ICFG with Preconditions

Display the ICFG decorated with preconditions. They are expressed in the *callee* name space to evaluate the interest of cloning, depending on the information available to the callee at a given call site.

```
print_icfg_with_preconditions            > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_precondition
        < MODULE.summary_effects
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

10.7.5 Minimal ICFG with Preconditions

Display the ICFG decorated with total preconditions. They are expressed in the *callee* name space to evaluate the interest of cloning, depending on the information available to the callee at a given call site.

```
print_icfg_with_total_preconditions            > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_total_postcondition
        < MODULE.summary_effects
        < MODULE.total_preconditions
        < MODULE.cumulated_effects
```

10.7.6 Minimal ICFG with Transformers

Display the ICFG decorated with transformers.

```
print_icfg_with_transformers             > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
```

10.7.7 Minimal ICFG with Proper Effects

Display the ICFG decorated with the proper effects.

```
print_icfg_with_proper_effects          > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.proper_effects
```

10.7.8 Minimal ICFG with filtered Proper Effects

Display the ICFG decorated with the write proper effects filtered for a variable.

```
print_icfg_with_filtered_proper_effects          > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.proper_effects
        < CALLEES.summary_effects
```

10.7.9 Minimal ICFG with Cumulated Effects

Display the ICFG decorated with cumulated effects.

```
print_icfg_with_cumulated_effects        > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.7.10 Minimal ICFG with Regions

Display the ICFG decorated with regions.

```
print_icfg_with_regions                  > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.11 Minimal ICFG with IN Regions

Display the ICFG decorated with IN regions.

```
print_icfg_with_in_regions               > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.in_regions
        < MODULE.in_summary_regions
```

```
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.12 Minimal ICFG with OUT Regions

Display the ICFG decorated with OUT regions.

```
print_icfg_with_out_regions             > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.13 ICFG with Loops

Display the plain ICFG with loops, without any decoration.

```
print_icfg_with_loops                   > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
```

10.7.14 ICFG with Loops and Complexities

Display the ICFG decorated with loops and complexities.

```
print_icfg_with_loops_complexities      > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_complexity
        < MODULE.complexities
```

10.7.15 ICFG with Loops and Preconditions

Display the ICFG decorated with preconditions.

```
print_icfg_with_loops_preconditions     > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_precondition
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

10.7.16 ICFG with Loops and Total Preconditions

Display the ICFG decorated with total preconditions.

```
print_icfg_with_loops_total_preconditions      > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_total_postcondition
        < MODULE.total_preconditions
        < MODULE.cumulated_effects
```

10.7.17 ICFG with Loops and Transformers

Display the ICFG decorated with transformers.

```
print_icfg_with_loops_transformers       > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
```

10.7.18 ICFG with Loops and Proper Effects

Display the ICFG decorated with proper effects.

```
print_icfg_with_loops_proper_effects     > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.proper_effects
```

10.7.19 ICFG with Loops and Cumulated Effects

Display the ICFG decorated with cumulated effects.

```
print_icfg_with_loops_cumulated_effects        > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.7.20 ICFG with Loops and Regions

Display the ICFG decorated with regions.

```
print_icfg_with_loops_regions            > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
```

```
        < CALLEES.icfg_file
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.21 ICFG with Loops and IN Regions

Display the ICFG decorated with IN regions.

```
print_icfg_with_loops_in_regions         > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.22 ICFG with Loops and OUT Regions

Display the ICFG decorated with the OUT regions.

```
print_icfg_with_loops_out_regions        > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.23 ICFG with Control

Display the plain ICFG with loops, without any decoration.

```
print_icfg_with_control          > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
```

10.7.24 ICFG with Control and Complexities

Display the ICFG decorated with the complexities.

```
print_icfg_with_control_complexities     > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
```

```
        < MODULE.summary_complexity
        < MODULE.complexities
```

10.7.25 ICFG with Control and Preconditions

Display the ICFG decorated with the preconditions.

```
print_icfg_with_control_preconditions   > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_precondition
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

10.7.26 ICFG with Control and Total Preconditions

Display the ICFG decorated with the preconditions.

```
print_icfg_with_control_total_preconditions   > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.summary_total_postcondition
        < MODULE.total_preconditions
        < MODULE.cumulated_effects
```

10.7.27 ICFG with Control and Transformers

Display the ICFG decorated with the transformers.

```
print_icfg_with_control_transformers    > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.transformers
        < MODULE.summary_transformer
        < MODULE.cumulated_effects
```

10.7.28 ICFG with Control and Proper Effects

Display the ICFG decorated with the proper effects.

```
print_icfg_with_control_proper_effects  > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.proper_effects
```

10.7.29 ICFG with Control and Cumulated Effects

Display the ICFG decorated with the cumulated effects.

```
print_icfg_with_control_cumulated_effects   > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.cumulated_effects
        < MODULE.summary_effects
```

10.7.30 ICFG with Control and Regions

Display the ICFG decorated with the regions.

```
print_icfg_with_control_regions         > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.regions
        < MODULE.summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.31 ICFG with Control and IN Regions

Display the ICFG decorated with the IN regions.

```
print_icfg_with_control_in_regions      > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.in_regions
        < MODULE.in_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.7.32 ICFG with Control and OUT Regions

Display the ICFG decorated with the OUT regions.

```
print_icfg_with_control_out_regions     > MODULE.icfg_file
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.icfg_file
        < MODULE.out_regions
        < MODULE.out_summary_regions
        < MODULE.preconditions
        < MODULE.transformers
        < MODULE.cumulated_effects
```

10.8 Data Dependence Graph File

This file contains the data dependence graph.

Known bug: there is no precise relationship between the dependence graph seen by the parallelization algorithms selected and any of its many views...

Two text formats are available: the default format which includes dependence cone and a SRU format which packs all information about one arc on one line and which replaces the dependence cone by the dependence direction vector (DDV). The line numbers given with this format are in fact relative (approximatively...) to the statement line in the `PIPS` output.

The SRU format was defined with researchers at Slippery Rock University (PA). The property

`PRINT_DEPENDENCE_GRAPH_USING_SRU_FORMAT` 10.8.10.1

is set to FALSE by default.

Two graph format, daVinci and GraphViz, are also available.

10.8.1 Menu For Dependence Graph Views

```
alias dg_file 'Dependence Graph View'

alias   print_effective_dependence_graph    'Default'
alias   print_loop_carried_dependence_graph 'Loop Carried Only'
alias   print_whole_dependence_graph        'All arcs'
alias   print_filtered_dependence_graph     'Filtered Arcs'
alias   print_filtered_dependence_daVinci_graph   'Filtered Arcs Output to uDrawGraph'
alias   impact_check                        'Check alias impact'
alias   print_chains_graph                  'Chains'
alias   print_dot_chains_graph              'Chains (for dot)'
alias   print_dot_dependence_graph          'Dependence graph (for dot)'
```

10.8.2 Effective Dependence Graph View

Display dependence levels for loop-carried and non-loop-carried dependence arcs due to non-privatized variables. Do not display dependence cones.

```
print_effective_dependence_graph          > MODULE.dg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.3 Loop-Carried Dependence Graph View

Display dependence levels for loop-carried dependence arcs only. Ignore arcs labeled by private variables and do not print dependence cones.

```
print_loop_carried_dependence_graph          > MODULE.dg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.4 Whole Dependence Graph View

Display dependence levels and dependence polyhedra/cones for all dependence arcs, whether they are loop carried or not, whether they are due to a private variable (and ignored by parallelization algorithms) or not. Dependence cones labeling arcs are printed too.

```
print_whole_dependence_graph          > MODULE.dg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.5 Filtered Dependence Graph View

Same as `print_whole_dependence_graph` 10.8.4 but it's filtered by some variables. Variables to filter is a comma separated list set by user via property "EFFECTS_FILTER_ON_VARIABLE".

```
print_filtered_dependence_graph          > MODULE.dg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.6 Filtered Dependence daVinci Graph View

Same as `print_filtered_dependence_graph` 10.8.5 but its output is *uDrawGraph*[11] format.

```
print_filtered_dependence_daVinci_graph          > MODULE.dvdg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.7 Impact Check

Check impact of alias on the dependance graph.

Note: the signature of the pass implies that it is a transformation pass, not a display pass. The location of its description is wrong.

RK: What is that? FI: Maybe, we should sign our contributions? See validation?

```
impact_check    > MODULE.code
        < PROGRAM.entities
        < MODULE.alias_associations
        < MODULE.cumulated_effects
        < MODULE.summary_effects
        < MODULE.proper_effects
        < MODULE.preconditions
        < MODULE.summary_precondition
        < MODULE.dg
        < ALL.code
```

[11] `http://www.informatik.uni-bremen.de/uDrawGraph`

10.8.8 Chains Graph View

Display the def-use or chains graph in a textual format. The following properties control the output:

PRINT_DEPENDENCE_GRAPH 10.8.10.1
PRINT_DEPENDENCE_GRAPH_WITHOUT_PRIVATIZED_DEPS 10.8.10.1
PRINT_DEPENDENCE_GRAPH_WITHOUT_NOLOOPCARRIED_DEPS 10.8.10.1
PRINT_DEPENDENCE_GRAPH_WITH_DEPENDENCE_CONES 10.8.10.1
PRINT_DEPENDENCE_GRAPH_USING_SRU_FORMAT 10.8.10.1

```
print_chains_graph      > MODULE.dg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
```

10.8.9 Chains Graph Graphviz Dot View

Display the chains graph in graphviz dot format.

```
print_dot_chains_graph  > MODULE.dotdg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.chains
```

10.8.10 Data Dependence Graph Graphviz Dot View

Display the dependence graph in graphviz dot format.

Using pyps, some convenient functions are provide, for instance :

```
import pypsex

w = workspace(...)

w.fun.my_function_name.view_dg()
```

which generate in the current directory a file named my_function_name.png.

```
print_dot_dependence_graph          > MODULE.dotdg_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.dg
```

10.8.10.1 Properties Used to Select Arcs to Display

Here are the properties used to control the printing of dependence graphs in a file called *module_name*.dg. These properties should not be used explicitly because they are set implicitly by the different print-out procedures available in pipsmake.rc. However, not all combinations are available from pipsmake.rc.

```
PRINT_DEPENDENCE_GRAPH FALSE
```

To print the dependence graph without the dependences on privatized variables

```
PRINT_DEPENDENCE_GRAPH_WITHOUT_PRIVATIZED_DEPS FALSE
```

To print the dependence graph without the non-loop-carried dependences:

```
PRINT_DEPENDENCE_GRAPH_WITHOUT_NOLOOPCARRIED_DEPS FALSE
```

To print the dependence graph with the dependence cones:

```
PRINT_DEPENDENCE_GRAPH_WITH_DEPENDENCE_CONES FALSE
```

To print the dependence graph in a computer friendly format defined by Deborah Whitfield (SRU):

```
PRINT_DEPENDENCE_GRAPH_USING_SRU_FORMAT FALSE
```

10.8.11 Properties for Dot output

Here are the properties available to tune the Dot output, read dot documentation for available colors, style, shape, etc.

```
PRINT_DOTDG_STATEMENT TRUE
```

Print statement code and not only ordering inside nodes.

```
  PRINT_DOTDG_TOP_DOWN_ORDERED TRUE
```

Add a constraint on top-down ordering for node instead of free dot placement.

```
PRINT_DOTDG_CENTERED FALSE
```

Should dot produce a centered graph ?

```
PRINT_DOTDG_TITLE ""
```

```
PRINT_DOTDG_TITLE_POSITION "b"
```

Title and title position (t for top and b for bottom) for the graph.

```
PRINT_DOTDG_BACKGROUND "white"
```

Main Background.

```
PRINT_DOTDG_NODE_SHAPE "box"
```

Shape for statement nodes.

```
PRINT_DOTDG_NODE_SHAPE_COLOR "black"
```

```
PRINT_DOTDG_NODE_FILL_COLOR "white"
```

```
PRINT_DOTDG_NODE_FONT_COLOR "black"
```

```
PRINT_DOTDG_NODE_FONT_SIZE "18"
```

```
PRINT_DOTDG_NODE_FONT_FACE "Times-Roman"
```

Color for the shape, background, and font of each node.

```
PRINT_DOTDG_FLOW_DEP_COLOR "red"
```

```
PRINT_DOTDG_ANTI_DEP_COLOR "green"
```

```
PRINT_DOTDG_OUTPUT_DEP_COLOR "blue"
```

```
PRINT_DOTDG_INPUT_DEP_COLOR "black"
```

Color for each type of dependence

```
PRINT_DOTDG_FLOW_DEP_STYLE "solid"
```

```
PRINT_DOTDG_ANTI_DEP_STYLE "solid"
```

```
PRINT_DOTDG_OUTPUT_DEP_STYLE "solid"
```

```
PRINT_DOTDG_INPUT_DEP_STYLE "dashed"
```

Style for each type of dependence

10.8.12 Loop Nest Dependence Cone

Before applying a loop transformation, information on the loop nest dependencies can help to check its legality. This pass enumerates the elements of the dependence cone that represent dependencies inside the loop nest. Loop label should be given.

```
alias dc_file 'Loopnest Dependence Cone'

print_loopnest_dependence_cone  > MODULE.dc_file
              < PROGRAM.entities
              < MODULE.code
              < MODULE.dg
```

10.9 Fortran to C prettyprinter

A basic and experimental C dumper to output a Fortran program in C code. It is not the same as the default pretty printer that is normaly used to pretty print C code in C. This pass is mainly use inside PIPS and Par4All to be able to generate call to CUDA kernels from a fortran code. The kernel is supposed not to use I/O intrinsics (such as WRITE, READ), they are not handle at the moment (and not usefull in the CUDA context) by the crough printer. However it is still possible to print the C code with the name of the fortran intrinsic using the property CROUGH_PRINT_UNKNOWN_INTRINSIC.

```
print_crough  > MODULE.crough
              < PROGRAM.entities
              < MODULE.code
              < MODULE.summary_effects
```

Display the crough output of a fortran function.

```
print_c_code  > MODULE.c_printed_file
              < MODULE.crough
```

Once C version of fortran code has been generated, ones might like to call this C functions from fortran code. A convenient way to do this is to use an interface in the fortran code. PIPS can generate the interface module for any function using the following pass.

```
print_interface  > MODULE.interface
                 < PROGRAM.entities
                 < MODULE.code
```

10.9.1 Properties for Fortran to C prettyprinter

By default the crough pass fails if it encounters a fortran intrinsic that cannot be translated to C. However it is still possible to print the C code with the name of the fortran intrinsic using the following property.

```
CROUGH_PRINT_UNKNOWN_INTRINSIC FALSE
```

By default the crough pass tries to match the best C type for any Fortran type. Here is the matches between Fortran and C variables:

- INTEGER is matched to int.
- INTEGER*4 is matched to int.
- INTEGER*8 is matched to long long int.
- REAL is matched to float.
- REAL*4 is matched to float.
- REAL*8 is matched to double.

However, many Fortran compilers (ifort, gfortran) allow to change the type size at compile time. It is possible to do the same by setting the property CROUGH_USER_DEFINED_TYPE to TRUE. In such a case the include file specified by the property CROUGH_INCLUDE_FILE is included by any file generated using crough. It has to define (using #define or typedef) the two types defined by the properties CROUGH_INTEGER_TYPE and CROUGH_REAL_TYPE. Obviously those types are used in the generated C file to declare variables that has the types INTEGER or REAL in the original Fortan file. When choosing that solution all INTEGER and REAL (including INTEGER*4, INTEGER*8, REAL*4 and REAL*8) variables will be set to the same user defined types.

```
CROUGH_USER_DEFINED_TYPE FALSE
```

```
CROUGH_INCLUDE_FILE "p4a_crough_types.h"
```

```
CROUGH_INTEGER_TYPE "p4a_int"
```

```
CROUGH_REAL_TYPE "p4a_real"
```

Is is possible to prettyprint function parameters that are arrays as pointers using the property CROUGH_ARRAY_PARAMETER_AS_POINTER

```
CROUGH_ARRAY_PARAMETER_AS_POINTER FALSE
```

When PRETTYPRINT_C_FUNCTION_NAME_WITH_UNDERSCORE is set to TRUE, an underscore is added at the end of the module name. This is needed when translating only some part of a Fortran Program to C. This property must be used with great care, so that only interface function names are changed : the function names in subsequent calls are not modified. An other solution to call a C function from a fortran program is to use/declare an interface in the fortran source code (This feature is part of the Fortran 2003 standard but many Fortran95 compilers support it). The property CROUGH_FORTRAN_USES_INTERFACE can be set to TRUE when the Fotran code integrates interfaces. In such a case, the unmodified scalar function parameters (by the function or any of its callees) are expected to be passed by values, the other parameters are passed by pointers. Finally when using interfaces it is also possible to pass all the scalar variables by values using the propety CROUGH_SCALAR_BY_VALUE_IN_FCT_DECL. Now, let's talk about function called from the fortran code PIPS has to print in C. The problem on how to pass scalars (by value or by pointer) also exists. At the moment PIPS is less flexible for function call. One of the solution has to be choosen using the property CROUGH_SCALAR_BY_VALUE_IN_FCT_CALL.

```
PRETTYPRINT_C_FUNCTION_NAME_WITH_UNDERSCORE FALSE
```

```
CROUGH_FORTRAN_USES_INTERFACE FALSE
```

```
CROUGH_SCALAR_BY_VALUE_IN_FCT_DECL FALSE
```

```
CROUGH_SCALAR_BY_VALUE_IN_FCT_CALL FALSE
```

If the property DO_RETURN_TYPE_AS_TYPEDEF is set to TRUE the crough phase additionaly does the same thing that the phase set_return_type_as_typedef does (cf section 2). In such a case the same property SET_RETURN_TYPE_AS_TYPEDEF_NEW_TYPE is taken into account. This is only posible for the SUBROUTINES and the FUNCTIONS but not for the PROGRAMS

```
DO_RETURN_TYPE_AS_TYPEDEF FALSE
```

Using the property INCLUDE_FILES_LIST, it is possible to insert some #include statement before to output the code. The INCLUDE_FILES_LIST is a string interpreted as coma (and/or blank) separated list of files.

```
CROUGH_INCLUDE_FILE_LIST ""
```

10.10 Prettyprinters Smalltalk

This pass is used by the PHRASE project, which is an attempt to automatically (or semi-automatically) transform high-level language application into control code with reconfigurable logic accelerators (such as FPGAs or data-paths with ALU).

This pass is used in context of PHRASE project for synthetisation of reconfigurable logic for a portion of initial code. This function can be viewed as a SmallTalk pretty-printer of a subset of Fortran or C.

It is used as input for the Madeo synthesis tools from UBO/AS that is written in SmallTalk and take circuit behaviour in SmallTalk.

It is an interesting language fusion...

```
alias print_code_smalltalk 'Smalltalk Pretty-Printer'

print_code_smalltalk        > MODULE.smalltalk_code_file
        < PROGRAM.entities
        < MODULE.code
```

10.11 Prettyprinter for the Polyhderal Compiler Collection (PoCC)

This pass is used for printing pragmas `scop` and `endscop` which delimit the static control parts (SCoP) of the code. Instrumented code can be an input for any PoCC compiler.

```
alias pocc_prettyprinter 'pocc_prettyprinter'

pocc_prettyprinter      >    MODULE.code

    < PROGRAM.entities
    < MODULE.code
    < MODULE.static_control
```

For the outlining of the control static parts, function prefix names to be used during the generation:

```
SCOP_PREFIX "SCoP"
```

```
  STATIC_CONTROLIZE_ACROSS_USER_CALLS TRUE
```

Because user function calls are not allowed in PoCC static control parts, whereas they are generally accepted in PIPS (see Arnauld Leservot's PhD), we introduce a property to control the impact of user calls. This property could be called `POCC_COMPATIBILITY` if there were a set of limiting rules to apply for PoCC, but user calls are the only current issue[12] So the property is called `STATIC_CONTROLIZE_ACROSS_USER_CALLS` 10.11 and its default value is TRUE.

[12]The PLUTO compiler does not support intrinsics calls either, althoug it is part of the PoCC collection.

10.11.1 Rstream interface

Detect non-SCoP of the code. Based on R-Stream constraints. Must be preceded by the pocc_prettyprinter phase.

```
alias rstream_interface 'rstream_interface'

rstream_interface > MODULE.code
    < PROGRAM.entities
    < MODULE.code
    < MODULE.static_control
```

10.12 Regions to loops

This phase is designed to replace the body of a function by simple assignments. The new body is representative of the regions described by the old body. When `PSYSTEME_TO_LOOPNEST_FOR_RSTREAM` 10.12 is set to True, the function in charge of computing loop nests from systems chooses the constant values rather than the symbolic values for loop ranges.

```
alias regions_to_loops 'regions_to_loops'

regions_to_loops > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.summary_regions
```

```
PSYSTEME_TO_LOOPNEST_FOR_RSTREAM FALSE
```

10.13 Prettyprinter for CLAIRE

This pass is used for the DREAM-UP project. The internal representation of a C or Fortran program is dumped as CLAIRE objects, either `DATA_ARRAY` or `TASK`. CLAIRE is an object-oriented language used to develop constraint solvers.

The only type constructor is array. Basic types must be storable on a fixed number of bytes.

The code structure must be a sequence of loop nests. Loops must be perfectly nested and parallel. Each loop body must be a single array assignment. The right-hand side expression must be a function call.

If the input code does not meet these conditions, a user error is generated.

This pass is used for the specification input and transformation in the XML format which can further be used by number of application as input. This function can be viewed as a XML pretty-printer of a subset of C and Fortran programs.

```
alias print_xml_code 'XML Pretty-Printer'

print_xml_code      > MODULE.xml_printed_file
    < PROGRAM.entities
```

```
    < MODULE.code
    < MODULE.complexities
    < MODULE.preconditions
    < MODULE.regions
```

This phase was developped for the DREAM-UP/Ter@ops project to generate models of functions used for automatic mapping by APOTRES []. It generates XML code like the PRINT_XML_CODE pass, but the input contains explicitly loops to scan motifs. It is useless for other purposes.

RK: gnih?
FI: to be deleted?
CA: more to say?

```
alias print_xml_code_with_explicit_motif 'XML Pretty-Printer with explicit motif'

print_xml_code_with_explicit_motif       > MODULE.xml_printed_file
        < PROGRAM.entities
    < MODULE.code
```

This pass is used in the DREAM-UP project for module specification input and transformation (?) []. This function can be viewed as a CLAIRE pretty-printer of a subset of Fortran.

```
alias print_claire_code 'Claire Pretty-Printer'

print_claire_code        > MODULE.claire_printed_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.preconditions
        < MODULE.regions
```

This pass generates CLAIRE code like the PRINT_CLAIRE_CODE pass, but the input contains explicitly loops to scan motifs.

```
alias print_claire_code_with_explicit_motif 'Claire Pretty-Printer with explicit motif'

print_claire_code_with_explicit_motif       > MODULE.claire_printed_file
        < PROGRAM.entities
        < MODULE.code
```

This pass was developped for the Ter@ops project to generate models of functions and application used for automatic mapping by SPEAR. It generates XML code. There are two versions: one using points-to analysis information and the second one without points-to analyses.

```
XML_APPLICATION_MAIN "main"
```

```
alias print_xml_application 'Spear XML Pretty-Printer'

print_xml_application                > MODULE.spear_code_file
                                     > MODULE.task_variable_changed_by
    < PROGRAM.entities
    < MODULE.code
    < MODULE.proper_effects
```

```
    < MODULE.preconditions
    < MODULE.regions
    < MODULE.dg
    < CALLEES.code
    < CALLEES.summary_effects
    < CALLEES.spear_code_file

alias print_xml_application_with_points_to 'Spear XML Pretty-Printer (with pt)'

print_xml_application_with_points_to > MODULE.spear_code_file
                                     > MODULE.task_variable_changed_by
    < PROGRAM.entities
    < MODULE.code
    < MODULE.points_to
    < MODULE.proper_effects
    < MODULE.preconditions
    < MODULE.regions
    < MODULE.dg
    < CALLEES.code
    < CALLEES.summary_effects
    < CALLEES.spear_code_file

print_xml_application_main > MODULE.spear_app_file
    < PROGRAM.entities
    < MODULE.code
    < MODULE.points_to
    < MODULE.proper_effects
    < MODULE.preconditions
    < MODULE.regions
    < MODULE.dg
    < MODULE.spear_code_file
    < MODULE.task_variable_changed_by
    < CALLEES.summary_effects
```

Chapter 11

Feautrier Methods (a.k.a. Polyhedral Method)

This part of PIPS was implemented at Centre d'Études Atomiques, Limeil-Brévannes, by Benoît de Dinechin, Arnauld Leservot and Alexis Platonoff.

Unfortunately, this part is no longer used in PIPS right now because of some typing issues in the code. To be fixed when somebody needs it.

11.1 Static Control Detection

`static_controlize` 11.1 transforms all the loops in order to have steps equal to one. Only loops with constant step different than one are normalized. Normalized loop counters are instantiated as a new kind of entity: `NLC`. This entity is forwarded in the inner statements. It also gets the structural parameters and makes new ones when it is possible ("NSP"). It detects enclosing loops, enclosing tests and the `static_control` property for each statement. Those three informations are mapped on statements. Function `static_controlize` 11.1 also modifies code (`> MODULE.code`). It is not specified here for implementation bug purpose.

The definition of a static control program is given in [21].

```
alias static_controlize 'Static Controlize'
static_controlize               > MODULE.static_control
        < PROGRAM.entities
        < MODULE.code
```

See the alias `print_code_static_control` 10.2.19 and function `print_code_static_control` 10.2.19 in Section 10.1 and so on.

11.2 Scheduling

Function `scheduling` computes a schedule, called *Base De Temps* in French, for each assignment instruction of the program. This computation is based on the Array Data Flow Graph (see [22, 23]).

The output of the scheduling is of the following form: (the statements are named in the same manner as in the array DFG)

`W:` Statement examined

`pred:` Conditions for which the following schedule is valid

`dims:` Time at which the execution of W is schedule, in function of the loop counters of the surrounding loops.

11.3 Code Generation for Affine Schedule

Function `reindexing` transforms the code using the schedule (`bdt`) and the mapping (`plc`) (see [15, 46]). The result is a new resource named `reindexed_code`.

11.4 Prettyprinters for CM Fortran

How to get a pretty-printed version of `reindexed_code` ? Two prettyprinters are available. The first one produces CM Fortran and the result is stored in a file suffixed by `.fcm`. The second one produces CRAFT Fortran and the result is stored in a file suffixed by `.craft`.

Use the polyhedric method to parallelize the code and display the reindexed code in a CMF (parallel Fortran extension from TMC, Thinking Machine Corporation) style.

Use the polyhedric method to parallelize the code and display the reindexed code in a CRAFT (parallel Fortran used on the Cray T3 serie) style.

Chapter 12

User Interface Menu Layouts

For presentation issues, it is useful to select only the features that are needed by a user and to display them in a comprehensive order. For that purpose, a layout description mechanism is used here to pick among the PIPS phases described above.

For each menu, the left part before the arrow, ->, is the menu item title and the right part is the PIPS procedure to be called when the item is selected. For the view menu (section 12.1, there is two display methods to view resources separated by a comma, the first one is the method for *wpips*, the second one is the one used in *epip*, followed by the icon to use.

Use a blank line to insert a menu separator.

12.1 View Menu

The view menu is displayed according to the following layout and methods (*wpips* method, *epip* method, icon name for the frame):

View

```
printed_file -> wpips_display_plain_file,epips-display-fortran-file,sequential
parsed_printed_file -> wpips_display_plain_file,epips-display-fortran-file,user
alias_file -> wpips_display_plain_file,epips-display-plain-file,-
graph_printed_file -> wpips_display_graph_file_display,epips-display-graph-file,-

dg_file -> wpips_display_plain_file,epips-display-plain-file,DG

adfg_file -> wpips_display_plain_file,epips-display-plain-file,-
bdt_file -> wpips_display_plain_file,epips-display-plain-file,-
plc_file -> wpips_display_plain_file,epips-display-plain-file,-

callgraph_file -> wpips_display_plain_file,epips-display-xtree-file,callgraph
dvcg_file -> wpips_display_graph_file_display,epips-display-graph-file,callgraph
icfg_file -> wpips_display_plain_file,epips-display-plain-file,ICFG
```

```
wp65_compute_file -> wpips_display_WP65_file,epips-display-distributed-file,WP65_PE
parallelprinted_file -> wpips_display_plain_file,epips-display-fortran-file,parallel

flinted_file -> wpips_display_plain_file,epips-display-plain-file,-
```

12.2 Transformation Menu

The transformation menu is displayed as here:

Transformations

```
distributer
full_unroll
unroll
loop_interchange
loop_normalize
strip_mine
loop_tiling
tiling_sequence

privatize_module
array_privatizer
declarations_privatizer

restructure_control
unspaghettify
simplify_control
simplify_control_directly
partial_eval
dead_code_elimination
stf
freeze_variables
partial_redundancy_elimination

array_bound_check_bottom_up
array_bound_check_top_down
array_bound_check_interprocedural

array_resizing_bottom_up
array_resizing_top_down

alias_check

atomizer
new_atomizer

clone
clone_substitute
clone_on_argument
```

```
clean_declarations
unsplit

static_controlize
```

At the end of this menu is added a special entry in *wpips*, the "Edit" line that allows the user to edit the original file. It is seen as a very special transformation, since the user can apply whatever transformation (s)he wants...

Chapter 13

Conclusion

New functionalities can easily be added to `PIPS`. The new pass names must be declared somewhere in this file as well as the resources required and produced. Then, `make` must be run in the `Documentation` directory and the `pipsmake` library must be recompiled and `PIPS` interfaces (PIPS, TPIPS, WPIPS) linked with the new C modules.

It is much more difficult to add a new type of resources, because `PIPS` database manager, `pipsdbm`, is not as automatized as `pipsmake`. This is explained in [28].

Chapter 14

Known Problems

1. *pipsmake* behavior may be erratic if files are accessed across a NFS network of non-synchronized workstations (see for instance UNIX `rdate` command or better NTP daemon).

2. `STOP` statements in subroutines (i.e. control effects and control dependencies) are not taken into account when parallelizing the caller.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. 48, 56, 162, 173

[2] Allen, R., and Kennedy, K. Automatic translation of fortran programs to vector form. *TOPLAS 9* (Oct. 1987), 491–542. 60, 149

[3] Amini, M., Coelho, F., Irigoin, F., and Keryell, R. Static compilation analysis for host-accelerator communication optimization. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)* (Fort Collins, Colorado, May 2011). 136

[4] Amini, M., Irigoin, F., and Keryell, R. Compilation et optimisation statique des communications hôte-accélérateur. In *Rencontres Francophones du Parallélisme (RenPar)* (Saint-Malo, France, May 2011). 136

[5] Ancourt, C., Coelho, F., and Irigoin, F. A Modular Static Analysis Approach to Affine Loop Invariants Detection. In *NSAD: 2nd International Workshop on Numerical and Symbolic Abstract Domains* (Perpignan, France, Sept. 2010), ENCTS, Elsevier. 65

[6] Ancourt, C., and Irigoin, F. Scanning polyhedra with do loops. In *PPOPP* (1991), pp. 39–50. 122

[7] ANSI. Ansi x3.9-1978. programming language fortran. Tech. rep., American National Standards Institute, 1978. Also known as ISO 1539-1980, informally known as FORTRAN 77. 108

[8] Aung, M., Horwitz, S., Joiner, R., and Reps, T. Specialization slicing. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 167–167. 190

[9] Aung, M., Horwitz, S., Joiner, R., and Reps, T. Specialization slicing. *ACM Trans. Program. Lang. Syst. 36*, 2 (June 2014), 5:1–5:67. 190

[10] Baron, B. Construction flexible et cohérente pour la compilation interprocédurale. Tech. Rep. EMP-CRI-E157, ENSMP, 1991. 1

[11] Baron, B., Irigoin, F., and Jouvelot, P. Projet pips. manuel utilisateur du paralléliseur batch. Tech. Rep. EMP-CRI-E144, ENSMP, 1991. 1

[12] Callahan, D., Cooper, K. D., and Kennedy, K. Interprocedural constant propagation. In *In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (1986), pp. 152–161. 65

[13] Carr, S. *Memory Hierarchy Management.* PhD thesis, Rice University, Sept. 1992. 196

[14] Carr, S., and Kennedy, K. Scalar replacement in the presence of conditional control flow. *Softw. Pract. Exper. 24* (January 1994), 51–77. 196

[15] Collard, J.-F. Code generation in automatic parallelizers. Tech. Rep. 93-21, LIP-IMAG, 1993. 263

[16] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1978), POPL '78, ACM, pp. 84–96. 65

[17] Creusillet, B. Automatic Task Generation on the SCMP architecture for data flow applications. http://www.par4all.org/documentation/publications, 2011. 142

[18] Creusillet, B., and Irigoin, F. Interprocedural array region analyses. In *Languages and Compilers for Parallel Computing* (Aug. 1995), no. 1033 in Lecture Notes in Computer Science, Springer-Verlag, pp. 46–60. 84, 89

[19] Creusillet, B., and Irigoin, F. Interprocedural array region analyses. *IJPP 24* (December 1996), 513–546. 89

[20] Creusillet-Apvrille, B. *Analyses de régions de tableaux et applications.* PhD thesis, École des mines de Paris, Dec. 1996. 87, 89, 93, 204

[21] Feautrier, P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20* (1991), 23–53. 10.1007/BF01407931. 262

[22] Feautrier, P. Some efficient solutions to the affine scheduling problem: I. One-dimensional time. *International Journal of Parallel Programming 21* (Oct. 1992), 313–348. 262

[23] Feautrier, P. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming 21* (1992), 389–420. 10.1007/BF01379404. 262

[24] Griebl, M., Feautrier, P., and Lengauer, C. Index set splitting. *International Journal of Parallel Programming 28* (1999), 607–631. 152

[25] Guelton, S. *Building Source-to-Source compilers for Heterogenous targets.* PhD thesis, Télécom Bretagne, 2011. 201

[26] Irigoin, F. *Partitionnement des boucles imbriquées. Une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Université Pierre et Marie Curie (Paris 6), June 1987. 60

[27] Irigoin, F. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools For Parallel Scientific Computing, CNRS-NSF* (Sept. 1992). 65

[28] Irigoin, F. Projet pips. environnement de dÃ©veloppement. Tech. Rep. E-146, CRI, École des mines de Paris, 1994. 267

[29] Irigoin, F. Detecting affine loop invariants using a modular static analysis. Tech. Rep. A-368, CRI, École des mines de Paris, 2005. 65

[30] Irigoin, F., and Ancourt, C. Final report on software caching for simulated global memory, puma esprit 2701, deliberable 6.5.1. Emp-caii-i155, École des Mines de Paris, Nov. 1991. 121

[31] Irigoin, F., and Ancourt, C. Automatic code distribution. In *CPC* (1992). 121

[32] Irigoin, F., and Ancourt, C. Compilation pour machines à mémoire répartie. In *Algorithmique Parallèle*, M. Cosnard, Nivat, and Y. Robert, Eds., École de Printemps du LITP. Masson, 1992. 121

[33] Irigoin, F., Jouvelot, P., and Triolet, R. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 5th international conference on Supercomputing* (New York, NY, USA, 1991), ICS '91, ACM, pp. 244–251. 1, 41

[34] Irigoin, F., Jouvelot, P., and Triolet, R. Pips: Internal representation of fortran code. Tech. Rep. CAI E-166, École des mines de Paris, 1992. This report is constantly updated and available on-line. 30, 48, 65

[35] Irigoin, F., and Triolet, R. Automatic do-loop partitioning for improving data locality in scientific programs. In *Vector and Parallel Processors for Scientific Computation 2* (Rome, Sept. 1987). 60

[36] Irigoin, F., and Triolet, R. Computing dependence direction vectors and dependence cones with linear systems. Tech. Rep. CAI E-94, École des mines de Paris, 1987. 60

[37] Irigoin, F., and Triolet, R. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 319–329. 60

[38] Jouvelot, P., and Gifford, D. K. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation* (New York, NY, USA, 1989), PLDI '89, ACM, pp. 218–226. 48

[39] Karr, M. Affine Relationships Among Variables of a Program. *Acta Informatica*, 133–151. 65

[40] Li, Z., and Yew, P.-C. Program parallelization with interprocedural analysis. *The Journal of Supercomputing 2* (1988), 225–244. 10.1007/BF00128178. 51

[41] Muchnick, S. S. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. 159, 160, 162, 173

[42] Nguyen, N. T. V. *Vérifications efficaces des applications scientifiques par analyse statique et instrumentation de code. Efficient and effective software verifications for scientific applications using static analysis and code instrumentation.* PhD thesis, École des mines de Paris, November 2002. 105, 107, 108, 109, 192, 193

[43] Nguyen, T. V. N., and Irigoin, F. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst. 27* (May 2005), 527–570. 105

[44] Osterweil, L. Toolpack-an experimental software development environment research project. *IEEE Transactions on Software Engineering 9* (1983), 673–685. 168

[45] Platonoff, A. Calcul des effets des procédures au moyen des régions. Emp-caii-i132, École des Mines de Paris, June 1990. 84, 89

[46] Platonoff, A. *Contribution Ã la Distribution Automatique des DonnÃ©es pour Machines Massivement ParallÃ¨les.* PhD thesis, Université Pierre et Marie Curie, Mar 1995. 263

[47] Pollicini, A. A. *Using Toolpack Software Tools*, 1st ed. Kluwer Academic Publishers, Norwell, MA, USA, 1988. 168

[48] Triolet, R. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure.* PhD thesis, Université Pierre et Marie Curie (Paris 6), June 1984. 89

[49] Triolet, R., Irigoin, F., and Feautrier, P. Direct parallelization of call statements. In *SIGPLAN Symposium on Compiler Construction* (1986), pp. 176–185. 89

[50] Triolet, R., and Jouvelot, P. Newgen : A langage-independent program generator. Tech. Rep. A-191, CRI, École des mines de Paris, 1989. 18

[51] Triolet, R., and Jouvelot, P. Newgen user manual. Tech. Rep. A-xxx, CRI, École des mines de Paris, 1990. 18

[52] Ventroux, N., and David, R. Scmp architecture: an asymmetric multiprocessor system-on-chip for dynamic applications. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies* (New York, NY, USA, 2010), IFMT '10, ACM, pp. 6:1–6:12. 141

[53] Ventroux, N., Sassolas, T., Guerre, A., Creusillet, B., and Keryell, R. Sesam/par4all: a tool for joint exploration of mpsoc architectures and dynamic dataflow code generation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools* (New York, NY, USA, 2012), RAPIDO '12, ACM, pp. 9–16. 142

[54] Wolfe, M. J. *High Performance Compilers for Parallel Computing*, 1st ed. Benjamin/Cummings, Redwood City, CA, USA, 1996. 110

[55] Yang, Y.-Q. *Tests des dépendances et transformations de programme.* PhD thesis, Université Pierre et Marie Curie (Paris 6), Nov. 1993. 60

[56] Yang, Y.-Q., Ancourt, C., and Irigoin, F. Minimal data dependence abstractions for loop transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, 1995), LCPC '94, Springer-Verlag, pp. 201–216. 60

[57] Zhou, L. *Analyse statique et dynamique de la complexité des programmes scientifiques.* PhD thesis, Université Pierre et Marie Curie (Paris 6), Sept. 1994. 83, 84, 86

[58] Zory, J. *Contributions à l'optimisation de programmes scientifiques.* PhD thesis, École des mines de Paris, Dec. 1999. 158, 178, 179

Index