PIPS: Internal Representation of Fortran and C Code

Mehdi Amini
Fabien Coelho
Béatrice Creusillet
Serge Guelton
François Irigoin
Pierre Jouvelot
Ronan Keryell
Thi Viet Nga Nguyen
Rémi Triolet
Pierre Villalon

CRI, M&S, MINES ParisTech

August 9, 2017, revision r23412

Contents

Introduction

This document contains the high-level description of data structures used in PIPS as internal representations of programs. These data structures are declared using the NewGen Data Definition Language, which insulates them from a particular programming language. They currently are translated into C or Common Lisp declarations and basic run-time support routines, such as `make`, `free`... A basic understanding of Newgen declarations[1] is assumed throughout this document, but the notation is close enough to standard programming language to be interpreted by newcommers. For more information about NewGen see [4, 5].

This document is part of PIPS documentation and its LaTeXversion is located in `$PIPS_ROOT/src/Documentation/newgen/ri.tex` (**ri** stands for Internal Representation, *représentation interne* in French). It is used to generate a PIPS include file called `$PIPS_ROOT/include/ri.h`. The functions available to manipulate these data structures are grouped in library `$PIPS_ROOT/src/Libs/ri-util`. These functions are grouped in files according to the *main* type in their signatures. There is no partial order between types used to build the internal representation, but they are clustered in a (hopefully) meaningful way.

Some external C data structures, used by the PIPS internal representation, are described in Section 1. They key data structures `entity` (variables and functions), `code` (instructions) and `expression` are introduced in Section 2, 3 and 4. The binding of Fortran to these data structures is explained. An additional data structure, `transformer`, used for interprocedural analyses but not by the internal representation, is described in Section 5. The `transformer` data structure is used to abstract the store transformations performed by statement and procedure, as well as preconditions. This domain was part of PIPS initial main thrust, interprocedural parallelization and analyses, which explain why it is still declared with the internal representation[2]. However, separate libraries (`transformers` and `semantics`) contain the corresponding code.

The binding of Fortran to the PIPS internal representation is covered in greater details in Technical Report EMP-CAII-E105 (in French). The data structure management tool NewGen is introduced in TR EMP-CRI-A191.

1 External Data Structures

Two external data types appear in PIPS internal representation: `Pvecteur` and `Psysteme`. Other types of C3 Linear Algebra Library also appear in PIPS code: `Pcontrainte`, which is a component of `Psysteme`, `Ppolyedre` for polyhedra, `Psg` for generating systems, `Pmatrix` for matrices, and `Ppolynome`, which is used for expressing program complexity (see complexity in pipsmake-rc documentation)

[1] If you edit this document, please remember that NewGen declarations introduced by `\domain` must fit on one line in the source LaTeXfile, although they may be printed on several lines by LaTeX.

[2] It also used to be the case for domain `effect`, but its declaration has been moved in `effects.newgen`

1.1 Vector

External Pvecteur

Type `Pvecteur` is used to represent affine expressions such as `3I+2` (see also type `normalized` in Section 4) or affine constraints such as `3I + J <= 2` or `3I == J`. The representation is sparse and a special dimension, called `TCST`, is used for numerical constants. The constraints are used to build systems of equations and inequalities. Such systems are of type `Psysteme`.

An object of type `Pvecteur` is a list of pairs (c, v) where c is a numerical coefficient and v is a variable. The coefficient is an integer, strictly greater or lesser than zero, since zero components are not part of a sparse representation. For objects of type `Pvecteur` used in PIPS, the variable must be either an `entity` (see Section 2) or a special predefined variable `TCST`. Since module names are used to prefix names of regular variables, no name conflict with a program variable called `TCST` can occur.

Expressions in programs are stored as affine integer expressions, wherever possible. This affine storage does not preclude the standard storage and simply is a second representation. The consistency between the two representations is hard to maintain, especially during program transformations.

The type `Pvecteur` is imported from our Integer Linear Algebra Library, also called C3 library because its development was funded by the CNRS C3 program. See ??? for more details about this library and its content.

1.2 Set of Affine Constraints

External Psysteme

Type `Psystem` is used to store systems of affine equalities and inequalities. They are used in many areas of PIPS such as semantics analysis, region analysis, dependence testing, code generation... They appear here for the semantics analysis and are used by the `predicate` type (see Section 5).

Objects of type `Psysteme` contain six fields:

- a list of equalities,
- the number of equalities,
- a list of inequalities,
- the number of inequalities,
- the dimension of the vector space,
- a basis of the vector space.

Redundant information is stored to accelerate frequent tests and consistency must be insured carefully.

Like the `Pvecteur` type, data structure `Psysteme` is imported from our Integer Linear Algebra Library. This library contains an extensive set of functions on Psystems. See ??? for more details.

2 Entities: Variables, Functions, Operators, Constants, Labels...

Data structure `entity` is a key PIPS data structure. Entities are stored in a unique[3] hash table, the global symbol table. They can be accessed by name.

2.1 Entity

tabulated entity = `name:string x type x storage x initial:value x kind:int`

Any named object in a Fortran or C program is represented by an object of type `entity`. Such object could be a module (function or subroutine or program), a variable, a common, an operator, an intrinsic, a constant, a label, a type, a derived type (struct, union and enum), a member... Field `name` contains the global name of the object, more or less as it appears in the source code, but concatenated to a prefix string and a special one-character separator, `MODULE_SEP_STRING`. The prefix string is the name of the package defining the functional scope of the object. The package name may be a module name, the reserved name `TOP-LEVEL` for global objects, or some other reserved names for objects specific to some analysis, e.g. `*SEMANTICS*` for value names used in the semantics analysis. Note the use of star, `*`, to avoid name collision with user defined name.

To sum up, an entity name is unique whithin an application and known internally by its *global* name. It is made of a *local* name and a *module* name. The local name contains the user source name.

Several other tricks are used to store information in local names. To spot main modules which cannot be distinguished from subroutines by the typing information, their local names are prefixed by a constant character, `MAIN_PREFIX`. Local names of labels also are prefixed by one character, `LABEL_PREFIX`. Two special label entities are used and defined by two special names: `EMPTY_LABEL_NAME`, which is used for statements with no label (see Function `empty_label_p()`), and `RETURN_LABEL_NAME` which is used to define the unique return point of a module. Every `RETURN` statement is translated into a jump to this artificial return point. In the same way, PIPS-specific prefixes and separators are used to dsitinguish special entities such as struct, union, enum and their members. Finally, the scope information also is stored in the local name.

As expected, field `type` specifies the type of the object (see Section 2.2), field `storage` defines the memory allocation class (see Section 2.3) for the object (e.g. dynamic, static,...).

The last field, `initial`, contains the initial `value` of the object, if it is known. A value can be anything that makes sense. For instance, the value of a module is its `code`[4]. When the value is unknown, it is stored as unknown, not as undefined.

[3] Retrospectively, choosing a unique symbol table was a mistake because it does not scale well for medium or large size programs. However, having only one symbol table provides a uniform access to information about entities, whether they are global, local or meta variables.

[4] The field `code` does not lead to the code internal representation. The PIPS database must be queried for resource `DBR_CODE`.

Functions mostly dealing with entities are grouped in `ri-util/entity.c` and `ri-util/variable.c` for entities used to represent program variables. Functions using list to encode a small set of entities are grouped in `ri-util/arguments.c`.

Note the *tabulated* attribute. It means that NewGen keeps track implicitly of all entities allocated. All entities (and objects of other tabulated types) are accessible through a huge hash table using their names as keys.

entity_int = `entity->int`

This domain is used to map entities towards integer. Any interpretation of this integer is possible. It could be the value of a scalar integer variable, the offset of a variable in a common, the lengths of commons, etc.

entity_to_entity = `entity->entity`

This domain maps an entity to another entity.

2.2 Type

```
Type = statement:unit + area + variable + functional +
varargs:type + unknown:unit + void:qualifier* + struct:entity*
+ union:entity* + enum:entity*
```

Obviously, type `type` is used to represent the type of an entity. This type is defined as union to cover the needs of different kinds of entities. Member `statement` is used for statement labels, since a label points towards a statement. Member `area` is used for commons. Additional areas are defined as implicit common: the static and the dynamic areas associated to a module, as well as the stack and heap areas. Their specific names are defined in `ri-util.h`. The static dynamic areas contain variables of static sizes. The stack and heap areas contain variables of sizes known at run-time and variables dynamically allocated (e.g. malloc).

Member `variable` is used for all variables and symbolic constants. It also is used for formal parameters and for results of functions. Member `functional` is used for modules which are functions, subroutines and main programs. Member `varargs` is used to declare intrinsics with varying number of arguments such as `MAX`. Member `void` is used to declare the functional types of subroutines and programs.

2.2.1 Area Type

```
Area = size:int x layout:entity*
```

Type `area` is used to represent storage sections for variables such as commons or static or dynamic areas. Areas for Fortran commons are *global* objects in the current implementation. The package name used in the corresponding entity name must be `TOP-LEVEL` in Fortran, but this is not explictly enforced. Dynamic commons defined by the Fortran standard are not implemented, as is the case with most Fortran 77 compiler. Dynamic commons cannot be statically identified because a dynamic binding is used.

Field `size` is the amount of memory space expressed in bytes, or according to the Fortran standard in *character storage unit* (X3.9-1978, § 2.13), which

is required to allocate the area in memory. This space is the largest[5] space encountered in all modules of a program. At some stages, PIPS used to enforce a unique size for all declarations of a common, but this is not true in the current version which only emits a warning.

Field `layout` is the list of all variables declared allocated in the area. These variables may have been declared in different modules, where the common itself is declared. They may have been declared explicitly in a COMMON declaration or implicitly through EQUIVALENCE declarations. Their names are non-ambiguous because PIPS entity names include a package name as prefix. Their offsets in the common are stored in Type `storage` (see Section 2.3.1).

As long as no program transformation has been applied, the textual order for common declarations is preserved in the layout list. This can be used to regenerate declarations close to the programmer declarations. Note that equivalenced variables appear *after* all variables explictly declared in a common[6]. It is possible to detect the first implicit variable by checking that the increasing offsets of variables is suddenly no longer increasing. The first variable whose offset is less than or equal to the previous offset is the first variable declared in the common through an `EQUIVALENCE` statement. All variables declared in the same module and appearing beyond this one also have been declared with an `EQUIVALENCE` statement.

To provide the best possible user-friendliness, remember that programmer declarations are in fact stored as a huge string which is used by prettyprinter as long as it is consistent with the code.

2.2.2 Variable Type

```
Variable = basic x dimensions:dimension* x qualifiers:qualifier*
```

Type `variable` represents the type of usual non-functional variables. Field `basic` is the underlying scalar type, e.g. `REAL*8` or `INTEGER*4`. Field `dimensions` is a list of lower and upper bound pairs. Scalar variables are of dimension 0 and have an empty dimension list.

Each dimension is an expression, which is not always numerically expressed or known. Constant parameter can be used to build symbolic constant expressions. Formal parameters can be used to specify the dimensions of other formal parameters. A special predefined constant entity is used for arrays with no defined dimension which often are declared in libraries such as (`DIMENSION T(*)`). Its name is '*D*'[7].

2.2.3 Basic Type

```
Basic = int:int + float:int + logical:int + overloaded:unit
+ complex:int + string:value + bit:symbolic + pointer:type +
derived:entity + typedef:entity
```

[5]It may even be larger than the space required in any module if some typing information is given after the common declaration and results in smaller variables in the common. For instance variables implicitly REAL could be redeclared CHARACTER*1.

[6]This is not true for the *dynamic* area used to allocate stack variables. The bug should be fixed... soon.

[7]In `ri-util`, the unbounded dimension name is `UNBOUNDED-DIMENSION`.

Type `basic` is used to store basic type information such has `REAL` in Fortran or `int` in C. Each member includes a precision information, which can be used to derive the number of bytes or bits required to store one scalar object of this type. The precision information is numerically known for most basic types but not for `overloaded`, `string`, `bit`, `derived`, `pointer`. Note that for some C `int` types, the information `int:int` must be fixed as the int type includes `s`igned and `u`nsigned specifiers as well as the `c`har type.

For Fortran, note that no *default* type is provided. Untyped object are given the current default type when they are first encountered. They only can be typed explictly if they still have their default type when the type declaration is encountered. PIPS parser is implemented in such a way that `IMPLICIT` statements should appear as early as possible in a module declarations.

It is not clear if mapping Fortran character strings on `string:value` is the right choice. It might be better to represent them as 1-D array of one character.

2.2.4 Dimension

Dimension = `lower:expression x upper:expression x qualifiers:qualifier*`

Type `dimension` is used to represent intervals, with a lower and an upper bounds. These bounds may not be numerically known at compile time when they are used to define formal or varying length arrays.

The default lower bound is 1 in Fortran. The lower bound is always 0 in C.

The qualifiers are used in C only. They are related to the upper expression.

2.2.5 C Qualifiers

The type qualifiers are defined in the C standard and used to build declarators (Section 6.7.6). The keyword `static` is not a type qualifier according to the standard grammar rules. It can only appear in dimensions of formal array parameters when declaring a function (Section 6.7.6.2). But to simplify the implementation, we decided to consider it a type qualifier, `static_dimension`.

Qualifiers `local`, `global`, `constant` and `private` are there for handling OpenCL 1.X codes, to designate whether a pointer or array is allocated in the thread local stack or the GPU global memory. This could probably be managed through areas, although the implication are not clear. The prettyprinter ignores this by default.

Qualifier = `const:unit + restrict:unit + volatile:unit + register:unit + auto:unit + thread:unit + asm:string + static_dimension:unit + local:unit + global:unit + constant:unit + private:unit`

2.2.6 Functional Type

Functional = `parameters:parameter* x result:type`

Type `functional` is used for objects representing the explicit syntactic type of a module, function, subroutine or main program. It also is used for Fortran operators and intrinsics. Even constants have a functional type because they

are seen as 0-ary functions. This reduces the amount of coding because many Fortran constructs can be handled as (pseudo) function calls. Effects on global variables are not taken into account for typing. Field `parameters` contains the type of each formal parameter, and the in/out information. Field `result` contains the result type. Type `void` is used for subroutines and main programs.

There is no provision to represent functions or subroutines with varying number of formal parameters. This facility is not supported by Fortran for programmer-defined modules, but it is used for intrinsics such as `MINO` which expects a *list* of integer parameters, and for Fortran primitives[8] such as `WRITE` which is highly polymorphic.

Intrinsics are statically declared in libraries `bootstrap/bootstrap.c` and `effects/intrinsics.c`. There is a predicate to recognize intrinsics entities.

2.2.7 Parameter Type and Mode

Parameter = `type x mode x dummy`

Objects of type `parameter` represents type and inout information for formal parameters. The `dummy` field is used to store a dummy[9] parameter entity in C (and Fortran) function declarations, which may be different from the formal parameter name. It is required by declaration such as:

```
typedef int foo(int a, double x[a*a]);
```

When no information about a dummy parameter is available, as in `void foo(int)`, an unknown dummy is used.

Dummy = `unknown:unit + identifier:entity`

Mode = `value:unit + reference:unit`

Type `mode` is used to carry parameter passing information for formal parameters. Member `value` is used for calling by value. Member `reference` is used for calling by reference. Fortran uses calls by reference, and C calls by value.

2.3 Storage

Storage = `return:entity + ram + formal + rom:unit`

Type `storage` is used to specify where an entity is stored. There are many storage spaces, but they do not have to exist physically in the machine. Some of them would not appear in a simple compiler.

Member `return` is appropriate for Fortran and C functions. The value returned by a function is locally stored in a variable whose name is the function name. This variable can be used explictly by the Fortran programmer like any other variable in statements and expressions. In C, it is only set through the return statement. The entity accessible thru the `return` field is the corresponding function.

[8] Fortran primitives are encoded like intrinsics and called *intrinsics* in PIPS.

[9] Also known as *formal* parameter, opposed to AN *effective* parameter.

Member `ram` is only used for variables having an address in some memory space. The memory space may be linked to a module or to a common. Those may be accessed thru the `ram` field.

Member `formal` is the special space for formal parameters. Of course, they do not have their own address.

Member `rom` is used for all entities whose value cannot change. This set of entities includes modules, labels, intrinsic operators, symbolic values (defined by Fortran `PARAMETER` statement or by the semantics analysis or by the region analyses), numerical constant,...

2.3.1 RAM Storage

Ram = `function:entity x section:entity x offset:int x shared:entity*`

Type `ram` contains all information required to locate a variable in memory and to guess what its scope is. Member `function` contains the module in which a variable is declared. In Fortran, a variable scope is a module. Variables with the same name and with the same offset in the same common are two different variables. They are aliased but they are different. They have different global names (see Section 2.1).

Member `section` contains the *area* in which the variable is stored. It is an entity of type area (see Section 2.2.1). For each Fortran module, there is one area for each declared common[10]. For C and Fortran modules, several specific areas called `*STATIC*`, `*DYNAMIC*`, `*STACK*` and `*HEAP*` used for local variables. Fortran static variables are explicitly declared in a `SAVE` statement or implicitly made static by a `DATA` statement[11] unless they are explicitly declared in a `COMMON` because PIPS sets all commons as static. In other words, Fortran dynamic commons are not handled by PIPS[12]. By default, variables can be stack allocated and are called dynamic variables. When their memory footprint is numerically known at compile time, they are allocated in the `*DYNAMIC*` area. If not, they are allocated in the `*STACK*` area. In C, memory space can be allocated in the `*HEAP*` area.

Member `offset` is the variable address in its area. Addresses are allocated according to the declaration or occurence order. Increasing values starting at 0 are used. The memory unit is defined by Fortran standard and is one byte for PIPS.

Member `shared` contains a list of variables, which are statically aliased with variable whose storage is described. Static aliasing is generated by `EQUIVALENCE` statements and by multiple declarations of the same common in different procedures. Dynamic aliasing created at call sites is not taken into account. Dependence tests, use-def chain computations, semantics analysis, region analysis, and other algorithms primarily based on variable names must check aliases.

[10] Note that a module may have effects on variables beyond its scope via procedure calls and common variables.

[11] See Fortran standard Section (8-11) about SAVE and Section (9-1) about DATA.

[12] The decision not to handle dynamic commons was based on two remarks: (1) no Fortran compiler in 1988 handled dynamic commons and (2) dynamic commons are not lexically scoped which make static analyses very difficult or even impossible.

2.3.2 Formal Storage

Formal = function:entity x offset:int

Type formal defines the module related to a formal parameter through the function member and the rank of this parameter in the formal parameter list. The first parameter has rank 1, not 0.

2.4 Value

Value = code + symbolic + constant + intrinsic:unit + unknown:unit + expression + reference

Type value is used to store initial values of all kinds of entities, as long as something makes sense as initial value. Member code is used for modules. Member symbolic is used for symbolic constants, declared in Fortran by keyword PARAMETER. Member constant is used for numerical and litteral constants[13]. Their values always are stored in their entity names[14], but integer constants which are more important for automatic parallelization and code optimization also are stored in binary representation. Member intrinsic is used for entities which are language-defined, such as Fortran intrinsics, operators, IO instructions,... Member unknown is used for entities with no initial values. For instance, areas might not have any initial values because there is no sensible information to use as initial value. Also, variables which are not statically initialized by a DATA statement, (probably) have an unknown initial value.

Additional value kinds would be necessary to encode the initial value of an area, if the overloading of the unknown kind becomes a problem. Pierre Jouvelot suggested to give COMMON themselves as initial value since a common represents an address.

For C variables, the initial value can be defined by any kind of expression, hence the expression member.

To analyze C code, location entities are used to represent the address of a field in a data structure or the address of a specific array element. The initial value of a location entity is a store independent points-to reference that may represent either a field or an array element. It could be included in an expression, but a specific reference field is used to recognize a location entity, since field kind of an entity cannot be used because it cannot encode more than 64 kinds of entities.

2.4.1 Symbolic Value

Symbolic = expression x constant

Type symbolic is used to represent the declared value of a symbolic constant defined by a Fortran PARAMETER or a Pascal CONST declaration. Member expression contains the hopefully constant expression which is statically evaluated by the compiler to find the numerical initial value. This value is stored in member constant. Member expression is used to restore user-friendly declarations but has no other known use.

[13] It might also be used for variables which are initialized by a DATA statement. To be checked.

[14] Their local name is the external representation of their value as defined by the language.

2.4.2 Program Variables

Program variables are entities with specific fields, but they must also be declared.

Fortran variables are only declared in the declaration field of data structure `code`.

A C variable is supposed to be declared in one declaration statement, in its own block and in the declaration field of `code`. These three declarations have to be kept consistent, which is a pain, but each of the declarations is useful at a different time: prettyprint of source code, block exit, and function exit respectively. The consistency check is not part of Newgen.

2.4.3 Constant Value

Constant = int + float:float + logical:int + litteral:unit + call:entity + unknown:unit

Type `constant` is used to represent the numerical or non-numerical value of constant entities. Integer entities are directly flagged with their `int` value, as well as floating point and logical entities, otherwise if possible the constant is represented as an `entity`, possibly itself if the constant is within the value of an entity, and finally if no entity can be thought of a `litteral` is chosen.

For instance, a constant entity such as 123.45 has "123.45" as entity name and its initial `value` is a `constant` of type `litteral` because its value is carried by its name.

A floating point variable statically initialized, as in a `DATA` statement, with 123.45 has an initial value of type `constant` and of kind `call`, since 123.45 is a nullary function.

Logical constants are functions but they are also represented by integer 0 and 1. The `int` kind of constant is used as for integer variables.

Other values of other types, such as real and character strings, given by expressions in `PARAMETER` statements, are not cached in `constant` and are tracked as unknown value (see `value` object). These values can be obtained by evaluating the associated expression in `symbolic`.

3 Code, Statements and Instructions

3.1 Module Code

Code = declarations:entity* x decls_text:string x initializations:sequence x externs:entity* x language

Type `code` is not used to stored module bodies. The effective code body must be retrieved from the PIPS database through a call to `pipsdbm` (see [6]). There is no direct link between the symbol table and the pieces of code in order to make these data structures independent with respect to NewGen. The pieces of code can be stored and retrieved without storing and retrieving the symbol table. However, note that the symbol table, which is unique for a whole program, must be loaded before any piece of code can be loaded.

Type `code` only is used for declarations. Member `declarations` contains a list of entities in the module scope. Local variables, formal parameters and commons are in this list. It also may contain symbolic constants, operators and intrinsics declared or referenced in the module.

The order of variables in list `declarations` must be compatible with the language constraints. In general, variables or entities used in another variable or entity declaration must appear first. For instance, in Fortran, if array B is used to declare array A as A(B(1)), array B must appear first in `declarations`. The parsers reproduce the source order. If source code is legal, all the constraints should be met. But if code is generated by a PIPS phase, it is up to this piece of code and not up to the prettyprinters to obey the languague order rules. Prettyprinters may emit a user warning or a user error.

In Fortran, member `decls_text` is a copy of the declaration text. This text starts with comments placed before the module declaration and ends with the comment related to the first executable statement of the module (i.e. one too many comment is included). This text is used by default by the prettyprinter, as long as it is available, to preserve the user layout of declarations. Because of Fortran syntax, declarations are almost impossible to regenerate. When the module is deeply transformed or synthesized, the field `decls_text` is destroyed and set to the empty string[15] to force declaration generation.

In Fortran, member `initializations` contains static initialization derived from the DATA statements. They are represented as a sequence of calls to a pseudo static initialization function. In C, member `initializations` is not used for functions, but it is used for pointers to functions when they are initialized within declarations. Since pointers and arrays of pointers to functions have `code` for `value`, their initial values cannot be specified directly as `expression` in `value` as is done for other kinds of variables.

Member `externs` contains variables declared within the scope but not allocated there. They must be provided by some other module at link time.

It is sometimes useful to regenerate declarations because lazy Fortran users include every single common in every procedure.

3.2 Programming Languages

The source code can be written in three languages, Fortran, Fortran95, or C.

Language = `fortran:unit + c:unit + fortran95:unit + unknown:unit`

3.3 Callees

Callees = `callees:string*`

Type `callees` is a list of string. It was given the name `callees` by mistaken a variable for a type. It is used to store the global (?) names of subroutines and functions directly called from a piece of code. Such objects are initialized by the parser.

It is a tiny part of the call graph which is stored as a tree of strings rather than a tree of entities (it probaly was quicker to implement initially). The call

[15] A symbolic constant should be used instead of the C empty string constant "".

graph is stored implictly, using `pipsdbm`. A list `callees` is associated to each module and can be retrieved through a call to `pipsdbm`.

A set of such variables, `callers`[16], `callees`, `all`, is used by Pipsmake to schedule interprocedural analyses[6]. Pipsmake and Pipsdbm are strongly string-oriented and not entity-oriented, because it is easier to deal with disk storage and ASCII files. Some C functions in library `ri-util` have either a string or an entity type for an entity formal parameter. The NewGen hash-table for entities makes both functionally equivalent, but strings are often more of a pain to handle.

3.4 Statement

```
Statement = label:entity x number:int x ordering:int x
comments:string x instruction x declarations:entity* x
decls_text:string x extensions x synchronization
```

Type `statement` is used as a container of instructions. Methods for statements are in Library `ri-util/statement.c`. Member `label` is an entity of kind label. Such entities can be recognized by their names. See Section 2.1 for more details about name structures and handling of statements with no labels and return points. Note that statements containing a `block` or `unstructured` instruction should not have a label, as you might find out when using the prettyprinter.

Member `number` contains an external number which is not used by PIPS. This number may be used as a statement identifier for debugging purposes or for user interaction because PIPS components try to propagate it as much as possible when new code is derived. For instance, several parallel loops derived by loop distribution have the same statement number, inherited from the initial sequential loop. Desugared statements like computed GOTOs generate several simpler statements with a unique number.

The default value is `STATEMENT_NUMBER_UNDEFINED`[17]. This number could theoretically be set explictly by the user[18]. In fact it is set by the parser. The parser uses an executable statement count from the source file. Only executable Fortran statements and `FORMAT` statements are stored as statement and only them are used to define statement numbers. Statement number one is the first line of the first executable statement and thus cannot be used to retrieve the text in the source file with a standard text editor. This number is theoretically never changed by PIPS once it has been initialized by the parser. Once the code has been transformed, statement numbers may not appear in increasing order, statement number may be duplicated, for instance, after loop unrolling, and statement number may not exist at all, for instance for fully synthesized statements.

Note that parser messages are labelled by *physical* line numbers, as defined by the PIPS preprocessor and parser. These numbers may be impacted by the language.

[16] Obviously, `callers` are of type `callees`...

[17] The prettyprinter is not too strict and take any non-positive value as an undefined statement number.

[18] Columns 73 to 80 are discarded for executable statements.

Member `ordering` is a 32-bit unique statement internal identifier. It is made out of two 16-bit fields: the most-significant field is a `control` number and the least-significant one is a statement number within a structured code piece. Two statements are textually comparable if their control numbers are equal. If they are comparable, their textual order is given by the least significant 16 bits.

The ordering structure is linked to the Hierarchical Control Flow Graph (HCFG) used by PIPS. See Section 3.5.8 for information about the HCFG. It is fully managed by PIPS, with no user control, and systematically recomputed when the code structure is modified. It is used to compute the lexical ordering of statements, to label a statement with information such as effects, regions,...through hash tables on disk, to label nodes of the dependence graph with statements by reference,... Its default value is `STATEMENT_ORDERING_UNDEFINED`.

A special hash table is used for each module to convert `ordering` into `statement`. This redundant table is not part of the internal representation. It must be recomputed regularly when the code structure is changed (see function `module_reorder()`). It must be reloaded or recomputed when a different module is analyzed because only one copy of this hash-table is available within PIPS. It must also updated when the internal representation is modified because some statements are added and/or removed.

Member `ordering` is not computed for all statements reachable through the NewGen internal representation. Specifically, statements which are not reachable forwards or backwards through the control flow graph only using the entry point of an unstructured are not ordered. For instance, label free statements following a GO TO statements may have or not an ordering. Statements can be walked in at least two different ways. C macros such as `CONTROL_MAP` uses the control flow graph, whereas NewGen iterators `gen_recurse`, `gen_multi_recurse`, are more systematic, using both the entry and exit controls of unstructured. Note that fully unreachable statements, which cannot be reached backwards or forwards from the entry or the exit control of an unstructured are fully lost by the `controlizer`.

Member `comments` contains the comments associated to the statement in the source program. This string[19] is used by the prettyprinter. Comments are associated to the *next* executable statement. For statements with no comments, this member receives a special value, `empty_comments`[20], which can be tested with predicate `empty_comments_p()`. Comments associated to statements that may disappear during processing, such as CONTINUE, RETURN and GO TO, may disappear too.

Member `instruction` contains the instruction itself.

Note that if the instruction of the statement is in fact a `sequence` of other

[19]To avoid problems with static buffers, a list of strings should have been used to store comments.

[20]This special value used to be `string_undefined`, but its name carried less semantics. Empty comments could be defined either as NULL, or the null length string, "", or as the Newgen special value, `string_undefined`. The NULL string was not chosen because it is not specific enough and because it is not compatible with the UNIX string library. The null length string is compatible with the string library, but sharing would have to be carefully considered. To avoid many allocations and desallocations of one byte areas, and many storage related bugs, the Newgen solution was chosen, although it is not compatible with the UNIX library and although Newgen does not provide such a library. Objects of type `string` are not 100 % equivalent to objects of type `char *` and guards for `string_undefined` must be added.

statements (see 3.5 and **??**), the `label`, `number`, `ordering`, and `comment` should be empty. If some information of this kind is needed, it should be attached to the first statement of the sequence or to a CONTINUE (for an empty sequence) instead.

Member `declarations` is used in C to declare local variables. If the instruction is a block, the declarations are local to the block. If the instruction is a `CONTINUE`, or better if the predicate `declaration_statement_p`holds true, then the declared variables are declared till the end of the current block. The variables declared in a CONTINUE statement are also declared at the directly enclosing block level. They are also declared in the symbol table, in the `declarations` field of `code`. The prettyprinter only takes into account the `CONTINUE` or declaration statements. So implicitly declared variables and functions, such as functions returning an `int` in C, only appear at the block level. Other statements, that are neither a block nor a declaration statement, cannot have declarations. This is not enforced by Newgen. To sum up, a C program variable is declared three times in C: in the declaration field of code, in the block enclosing its declaration and in a declaration statement.

Note that local variables declared in a loop as `loop_locals` are local to the loop body. This seems redundant with `statement_declarations`, but making it obsolete would require a refactoring of the use-def chains, dependence graph and parallelization algorithms. Also, the semantics is slightly different and there are rationals[21] to preserve this local field in loops. Among them, the last value may escape the loop, the first value may be imported.

Member `decls_text` can be used to keep track to the exact text used to declare variables. This field is not currently used.

Member `extensions` is used to add various features such as pragmas or future extensions to the statements.

3.4.1 Mappings from statement to statement

`persistant_statement_to_statement` `= persistant statement -> persistant statement`

Type `persistant_statement_to_statement` is used for example in `use_def_elimination()` to store the eventual statement father of a statement. The *persistant* pragma is needed to avoid freeing the statements when the mapping is freed. See NewGen documentation in [5].

3.4.2 Mappings from statement to integer

`persistant_statement_to_int` `= persistant statement -> int`

Type `persistant_statement_to_int` is used for instance to associate line number to a statement.

3.4.3 Mappings from statement (task) to its schedule

`persistant_statement_to_cluster` `= statement:int -> number:int`

[21] FI: I do not remember them all and I do not know where it is explained.

Type `persistant_statement_to_cluster` is used in HBDSC to store the cluster where a statement is scheduled. It corresponds to sigma defined in [**?**]

3.5 Instruction

Instruction = `sequence + test + loop + whileloop + goto:statement + call + unstructured + multitest + forloop + expression`

Type `instruction` is used to represent the command associated to a statement. An instruction can either be a sequence, a test, a loop, parallel or sequential, an unconditional branch (`goto`) pointing to the branch target, an elementary command (`call`) or a whole control flow graph.

Elementary commands are used for Fortran statements and intrinsics and operators. There are `call`'s for assignments, subroutine calls, input-outputs, returns, stops, modulos, the overloaded `+`, and so on. This is detailed in Section 3.5.7.

The code of a module is either in a user-defined form or in so-called `controlized` form. In the former case, no `unstructured` instruction is allowed and explicit `goto`'s are used. In the later case, `goto`'s are forbidden and abstracted by `unstructered`. of course, a fully-structured code does not contain either `goto` or `unstructured`[22]. The user-defined form only is used by the parser and some pretty-printers. The pretty-printers are able to restore Fortran-77 `goto`'s from the `unstructured`. More on this in Section 3.5.8.

Several PIPS contributors have asked for a `while` construct. In addition, a statement in C can be any expression, not only call expression, so we have to add `expression` to `instruction`.

3.5.1 Sequence

Sequence = `statements:statement*`

Type `sequence` is self-explanatory. This is the standard sequence constructor. The empty sequence is used to represent an instruction with no effect, a `NOP`. See `empty_statement_p()`.

Note that the statement owning the sequence cannot have information such as `comment`, etc. on it. See **??**

3.5.2 Conditional (a.k.a. Test)

Test = `condition:expression x true:statement x false:statement`

Type `test` is used to represent conditional statements. Field `condition` must contain a *boolean*[23] expression to evaluate. Fields `true` and `false` contain the statement to execute if the test evaluates to true or false.

If the false branch is empty, an *empty* statement is inserted. It might be an empty sequence or a `CONTINUE` statement or... (see Function

[22] PIPS does not contain a control restructurer but it is interfaced to Toolpack.

[23] Note that expressions are untyped in PIPS internal representation. They are kept in an overloaded form because typing does not matter for parallelization. An new pass would be required to insert the conversion operators. Besides, fully typed Fortran operators would have to be added.

empty_statement_p). Of course, an empty statement *must* have an empty label (see Section 2.1).

Fortran control instructions, but DO loops with no internal exits, are decomposed into combinations of such `test` instructions and other PIPS instructions by the parser, which may add `goto` statements, and by the controlizer.

3.5.3 Switch

Multitest = controller:expression x body:statement

This is not really used in PIPS. Right now in C, the `switch/case` are replaced by `if` and `goto`.

3.5.4 DO Loop, Sequential or Parallel

Loop = index:entity x range x body:statement x label:entity x execution x locals:entity*

Type `loop` is used to represent Fortran DO loops or Pascal FOR loops. It is also used to represent C for loops, when their semantics is compatible with Fortran DO loops. Field `index` points to the loop index, an entity. Field `range` contains the lower and upper bounds, as well as the step. Field `body` points to the loop body, a unique statement which usually is a sequence. Field `label` is used for Fortran labelled DO loops. It is the label of the last statement in the loop body. In C, this field contains a redundant pointer to the label of the statement containing the loop. It has no semantics, but makes C loops easy to designate in a Fortran-compatible mode to apply loop transformations. Field `execution` specifies if the loop should be executed sequentially or concurrently. Entities in the `local` field are loop-private variables. They can be stack-allocated on body entrance and deallocated on exit. The read and write effects on these variables are not visible from outside the loop body. They can be privatized and their effects can be ignored when running the loop in parallel if each processor gets a private copy of each of them.

This field should be factored out in the `statement` type in order to declare variables local to a block, as in C. However, the two levels, statement and instruction, would make coding more difficult.

Execution = sequential:unit + parallel:unit

Type `execution` is used to specify if a loop must be executed sequentially (`sequential` or if it may be executed concurrently (`parallel`). The parser only recognizes sequential loops.

Range = lower:expression x upper:expression x increment:expression

Type `range` is used to store the loop bounds and step. The three fields are used to store the lower bound (`lower`), the upper bound (`upper`) and the step expression (`increment`). The lower and upper bound are included, ie lower¡=i¡=upper and not lower¡=i¡upper.

Expressions of type range can be used in other context. For instance, Fortran 90 triplet construct is a range. See Section **??** for details about expressions. For loops, other ranges are *not* expected in bound and step expressions.

3.5.5 While Loop

Whileloop = condition:expression x body:statement x label:entity x evaluation

Evaluation = before:unit + after:unit

Here is a while loop. It is not the `while` domain because it would interfere with C keywords. The content is similar to the `loop` domain. Possible parallel while loops are considered unimportant, hence no execution part was added. No locals are attached, because this should be rather done at the `statement` level, not within the `instruction` itself.

3.5.6 For loop

Forloop = initialization:expression x condition:expression x increment:expression x body:statement

3.5.7 Function Call

Call = function:entity x arguments:expression*

Type `call` is used to represent Fortran commands as well as user-defined function and subroutine calls in a pseudo-functional way. These pseudo-functions with side effects are very important for the PIPS internal representation since constants, operators such as + and *, intrinsics like `MOD` or `SIN`, and basic Fortran statements such as assignment `=`,`READ, WRITE, PAUSE, OPEN, CLOSE, RETURN, CALL, FORMAT`, and so on... are all encoded in the same way, like user-defined function calls. The number of arguments depends on the pseudo-function: 0 for constants, 1 or 2 for operators, and so on. Fortran keywords, operators and intrinsics are known as predefined functions. This unification of language and user defined functions is useful to reduce the size of the datastructure definition as well as the code required for many algorithms.

Type `function` points towards the entity associated with the called function. Subtype `arguments` is a list of `expression` objects which represent the actual arguments for the function.

3.5.8 Control Flow Graph (a.k.a. Unstructured)

Unstructured = entry:control x exit:control

Domain `unstructured` is used to represent unstructured parts of the code in a structured manner which as a unique statement. The entry node of the underlying CFG is in field `control`, and the unique exit node is in field `exit`. The exit node should not be modified by users of the unstructured[24]. See Figure 2. Note that the exit node may not be reachable, for instance because the program does not terminate. For instance node C7 could very well be the exit node. Note also that node C6 is not forward reachable. Like C7, C6 is reachable using the `predecessors` field in `control`. Nodes unconnected to

[24]FI: I do not understand why...

either the entry or the exit control in `unstructured` like C*, C9 and C10 are lost by the `controlizer` but they can be seen in the `user_view` representations of the program.

An `unstructured` object can be walked by function `gen_multi_recurse` and nodes C1 to C7 are visited, because the entry and exit nodes are used to perform a transitive closure. It can be walked by macro `CONTROL_MAP` and nodes C1 to C6 are visited because the undirected transitive closure starts at the entry node C1. This macro is used to compute the ordering and, if they exist, nodes such as C6 and C7 are ordered. Nodes C1 to C5 only could be visited by performing a forward transitive closure on the entry node. Transformation `unspaghettify`, which is optionally included in the `controlizer` (property `UNSPAGHETTIFY_IN_CONTROLIZER`), eliminates spurious nodes such as C6 and C7 and makes all walks equal, but for the visiting order. Note that a fourth kind of walk is implemented by the `prettyprinter`. It can bump into nodes not visited by `CONTROL_MAP`.

The hierarchical structure is induced by the recursive nature of statements. Each control node points towards a statement which can also contain an unstructured area of the code as well as structured part. Unstructured parts of the code can thus be contained as much as possible as well as be recursively decomposed.

For instance, the two DO loops in:

```
      DO 200 I = 1, N
100      CONTINUE
         DO 300 J = 1, M
            T(J) = T(J) + X
300      CONTINUE
         IF(X.GT.T(I)) GO TO 100
200   CONTINUE
```

are preserved as DO loops in spite of the GO TO statement (see Figure 1).

3.5.9 Control Flow Graph Node

Control = statement x predecessors:control* x successors:control*

Domain `control` is the type of *nodes* used to implement the CFG implied by an unstructured instruction (see Domain `unstructured`, Section 3.5.8). Each control node points towards a statement which can represent an arbitrary large piece of structured code. GOTO statements are eliminated and represented by arcs. Nodes are doubly linked. Each node points towards its successors (at most 2) and towards its predecessors. The hierarchical nature of domain `statement` is used to hide local branches from higher and lower level pieces of code. The whole unstructured area of the code is seen as a unique atomic statement from above, and is entirely ignored from under. This explains the mutual recursion between `control` and `statement` (via `instruction`).

All statements but `test`s and the exit node only have one successor. The first successor of `test` is the successor when the test condition is evaluated to true. And the other way round for the second one. The exit node (see domain `unstructured`) has no successor. The entry node as well as all other nodes may have an unlimited number of precedessors.

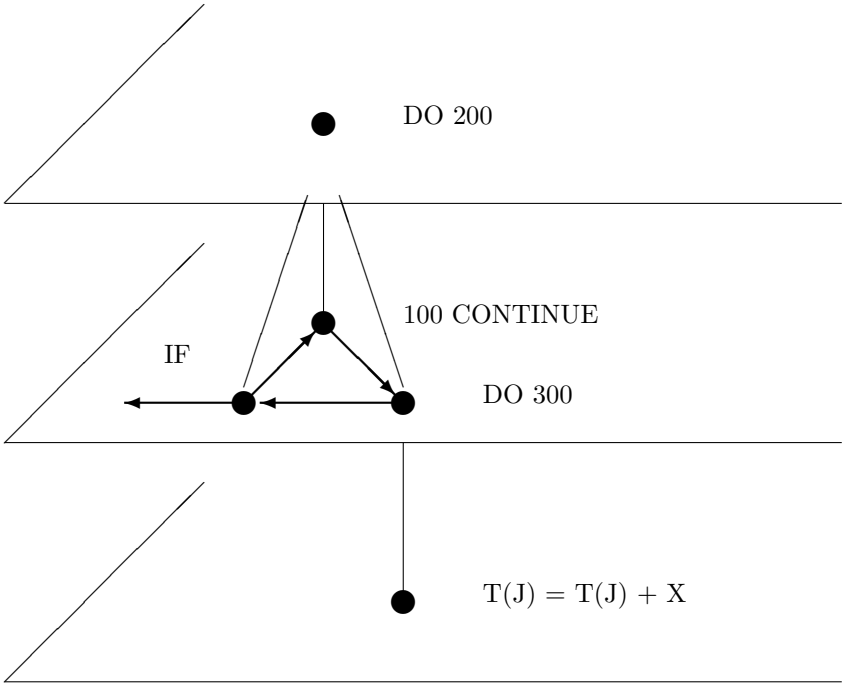

Figure 1: Hierarchical Control Flow Graph

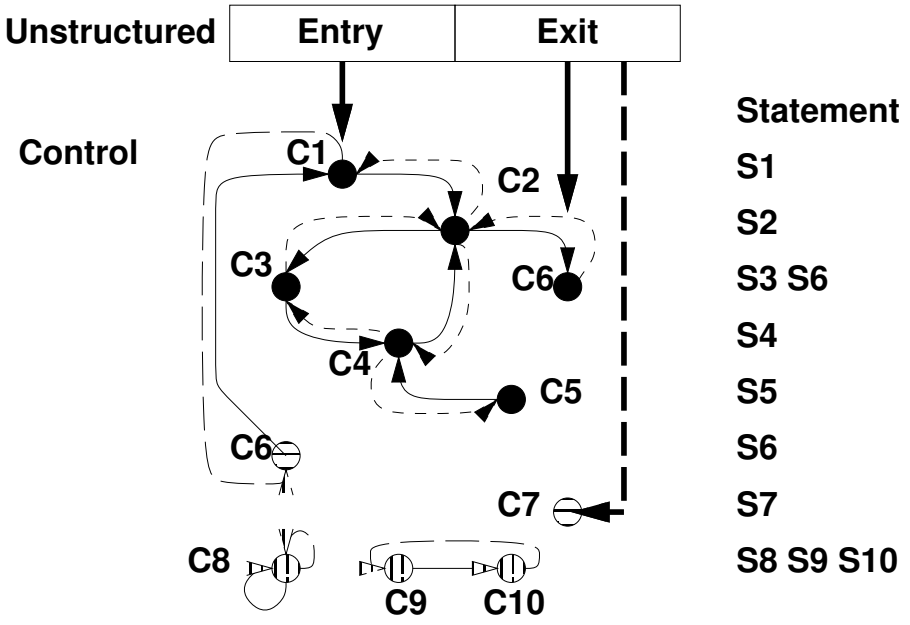

Figure 2: Control Flow Graph

The value of the exit node when it is not reachable is unclear. This is a minor problem since PIPS isn't supposed to deal with faulty programs.

Note that the two successors of a test can be identical since the two branches of a test can contain a GOTO to the same label. Hence, a node can have two identical predecessors. The lists of predecessors and successors cannot be handled like sets, in general. Use `check_control_coherency()` to make sure that the predecessor and successor lists can be interpreted as a multiset of arcs.

All reachable nodes of a CFG can be visited in a meaningless order using the `CONTROL_MAP` macro. Look for an example in library `control` because an auxiliary data structure, a block list, must be declared and freed. They can also be visited with `gen_recurse()`, in a meaningless order, but regardless of their reachability. Two more macros, `FORWARD_CONTROL_MAP` and `BACKWARD_CONTROL_MAP` are available for ordered walks.

Note that the data structure used for the CFG is obsolete. A generic structure for oriented graphs, `graph`, should be used instead so as to pool basic graph functions, e.g. search for strongly connected components.

3.5.10 Mappings between Statements and Control Nodes

Controlmap = persistant statement->control

persistant_statement_to_control = persistant statement -> persistant control

Used for example in `use_def_elimination()` to store the eventual control father of a statement in order to travel on the control graph associated to a

statement. The persistance is needed to avoid freeing the control graph when the mapping is freed.

3.6 Extensions

Extensions = extension*

Extensions are used to extend in a not too much intrusive way the internal representation of the code.

extension = pragma + unknown:unit

Extensions can be used to add pragmas to statements. Well, right now an `extension` can only be of type `pragma` but it could be something else too some days.

3.6.1 Pragma

pragma = string + expression*

Pragmas fields are used to attach `#pragma` to statements.

Pragmas can be represented as a string or as a list of expressions. The expression list is the most suitable way for pragma representation. Indeed using expressions, clasical transformations may work by side effect on pragmas too (such as variable renaming or anything else). The string representation is also provided because it is a way of handling unknown type of pragmas by simply carrying its text as it is in the source code.

The following type of pragmas are handeled as expressions:

OpenMP pragma Pips can represent OpenMP pragma as an expression, more precisely as a function call (see Section 3.5.7). The following omp clauses are supported:

- `parallel` is internally considered as function call with no parameter.
- `for` is internally considered as function call with no parameter.
- `private` is internally considered as function call that takes at least one parameter.
- `reduction` is internally considered as function call that takes at least two parameters. The first one is the reduction operator and the folowing ones are the variables that are reduced using this operator. Note that the reduction function call need a specific prettyprinter because of the `:` used between the opertor and the variables.
- `omp` is internally considered as function call with no parameter.

Then, describing a pragma as an expression is done by listing the clauses used in it. For example `#pragma omp prallel private(x,y)` will be represented by the following list:

- omp function call.

- parallel function call.
- private function call with two arguments: x and y.

Note that the `#pragma` is automatically generated by the prettyprinter and does not need to be represented in the pragma extension.

3.7 Synchronization

Synchronization = `none:unit + spawn:entity + barrier:unit + single:bool + critical:reference`

4 Expressions

Expression = `syntax x normalized`

Obviously, type `expression` is used to store expressions. Field `syntax` contains the syntactic description of the expression, as it appears in the source code. Note that parentheses may nevertheless be missing in PIPS printouts. Although they are taken into account to build the internal representation, they are not encoded and it is not possible to distinguish between redundant parentheses and omitted ones. Field `normalized` indirectly contains a secondary representation of affine integer expressions stored as `Pvecteur` (see Section 1.1).

If field `normalized` is set to value `normalized_undefined`, this implies that the PIPS function used to detect affine expressions and sub-expressions has not been called. This **does not** imply that the expression is not affine.

The `normalized` field is redundant with the `syntax` field. There is no consistency check available. When new expressions are derived from old expressions, all normalized fields should be reset. Else some normalized expressions end up with non-normalized sub-expressions, and with a non-consistent normalized form.

Expressions synthesized by program transformations, such as partial evaluation or loop interchange, should all have their `normalized` field set to `normalized_undefined`.

4.1 Abstract Tree of an Expression: Syntax

Syntax = `reference + range + call + cast + sizeofexpression + subscript + application + va_arg:sizeofexpression*`

Type `syntax` is used to represent expressions as they are defined in the program source code. A `syntax` object is either a `reference` object pointing towards an array element[25], or a `call` to a function[26], or a `range` as in loop definitions[27] and array declarations.

[25] Scalar variables are represented as 0-dimensional arrays.

[26] All operators and commands, including assignment, are encoded as *functions*. This explains why the `call` type is defined in the *Instruction* section.

[27] This explains why type `range` is defined in the *Instruction* section.

New kinds of `syntax` are added to handle the C language. They are *cast*, *sizeof*, *subscripting array* and *function application* expressions. The subscripting array expression is an extension of the reference expression, which includes other more complicated array objects such as pointer, function, structure or union member... The same extension is made to call expression, named function application, because the called function is not necessarily an entity but can be any expression that denotes the address of a function.

The field `va_arg` is added only to cope with the call `va_arg(e,t)` where `e` is an expression and `t` is a type. The domain `sizeofexpression` is reused, in spite of its name, because it combines types and expressions. The number of arguments is left unspecified as well their respective kinds so as not to add yet another domain in the internal representation for such a peculiar case of C syntax.

4.1.1 Reference

Reference = variable:entity x indices:expression*

Preference = persistant reference

Type `reference` is used to represent references to array elements[28]. Field `variable` points towards an entity representing the used or defined program variable. Field `indices` contains a list of subscript expressions[29].

4.1.2 Range

See Section 3.5.4.

4.1.3 Function Call

All operators, including assignment, are represented as function calls with side effect. See Section 3.5.7.

4.1.4 Cast

This is used to represent expressions in C such as:

```
1  ␣␣(type)␣e;
```

Cast = type x expression

4.1.5 Sizeof

This is used to represent expressions such as:

```
1  ␣␣sizeof(e);
   ␣␣sizeof(type);
```

[28]Scalar variables are represented as 0-dimensional arrays. Scalar references are special references with an empty subscript expression list. Note that arrays may also be references with an empty subscript expression list, e.g. as actual argument of a subroutine or function.

[29]The consistency of the array dimension and the list length is not checked, but in the parser. That is, there is no independent consistency checker.

Since it can be applied on a type, it cannot be represented in the RI as a classical function call to a `sizeof()` operator.

Sizeofexpression = type + expression

4.1.6 Subscript

In C, pointer expressions can be subscripted, not only arrays, so there is a subscript expression:

Subscript = array:expression x indices:expression*

4.1.7 Application

Since in C not only functions can be called but also any function pointers, there is a need to represent these function calls:

Application = function:expression x arguments:expression*

4.2 Affine Representation of an Expression

Normalized = linear:Pvecteur + complex:unit

Type `normalized` is used to check if an expression is an affine integer expression using only integer scalar variables (Field `linear`) or not (Field `complex`[30]).

Field `complex` is used if the expression is not affine, e.g. `I*J+4`, or if it is affine but contains references to non integer scalar variables, e.g. `T(I-1) + T(I) + T(I+1)`.

The `normalized` field does not exist if the expression has not yet been examined. This is an exception to NewGen data structures used in PIPS because empty pointers, `normalized_undefined` must be used. This may cause problems when using the NewGen `gen_defined_p()` consistency checker.

A C Macro, `NORMALIZE_EXPRESSION` is used to perform the normalization only if required. This macro and the underlying function is fragile. It only can be applied to fully normalized or fully non-normalized epxressions.

The normalize union may not be general enough. In some cases it would be useful to be able to encode pseudo-affine operators such as `/`, `mod`, `min` or `max` with inequalities.

5 Semantics Analysis

5.1 Transformer

Transformer = arguments:entity* x relation:predicate

Type `transformer` defines a relationship between two stores, i.e. two memory states, associated to two control points (i.e. two statements). This relation is limited by default to *integer scalar* variables in the *dynamic* scope of a module: global and static integer scalar variables are taken into account. Note that

[30] This does not mean that the expression is *complex* in the mathematical sense.

the data structure does not enforce the integer and scalar conditions and that some properties can be set to process also boolean, string and floating point scalar variables.

Variables in the list `arguments` are variables whose values *may* have changed between the two control points. Two values are denoted for each variable in the `arguments` list, the initial value and the final value. The initial values are pure values, i.e. entities specific to the semantics library. The final values are identified with the variable entities, in order to decrease the total number of pure values. Variables that do not appear in the `arguments` list *must* have the same initial and final values, by definition. This unique value is considered a final value and no value entity is allocated.

Intermediate and temporary values are also represented by entities and used internally. They are never used or displayed in prettyprinted files.

The relationship between the initial and final store is abstracted by a set of affine equations and inequations on the values.

Two kinds of transformers are used in PIPS. The first kind is linked to a statement or an expression, possibly with side effects, and is an abstraction of the corresponding command, limited to integer scalar variables and to other scalar variables according to properties. Such transformers are called *transformers*. The second kind of transformer, also associated to a statement, abstracts the relationship between the initial store of a module or of a whole program and the store just before the execution of this statement. Such a transformer si called a *precondition*.

Transformers and preconditions are computed by semantics analyses, either intra- or inter-procedurally.

5.2 Predicate

Predicate = `system:Psysteme`

Type `predicate` defines a relationship between values of integer variables and other integer entities such as `PHI` variables (see Section 1.2 for external data type Psysteme). Its meaning depends on its use. It may be an invariant predicate, always true before a statement execution (*precondition*), or a predicate linking two different points of a program (*transformer*). It may also define an array region (see `effects.newgen`).

6 Consistency

Numerous predicates should be met by consistent module and program representations.

6.1 Module consistency

Each variable always must be referenced with the same number of subscript expressions, its declared dimension, or with no subscript expressions at all (e.g. formal parameter).

No consistency check is available, beyond usual NewGen tests (see Section **??** and Reference [5]).

6.2 Program consistency

PIPS handles *constant* call graphs. It expects to find every callee in a piece of code. If library routines are used, stubs must be added.

Interprocedural consistency can be checked with the `Flinter` analysis (see [8]).

6.3 Implicit consistency

All links are not declared explictly in the internal representation, if only to break cycles between data structures. For instance, the code associated to a module is not obtained thru a pointer dereferencing but thru an explicit request to the PIPS database manager.

6.4 NewGen consistency

NewGen provides two generic consistency checkers, `gen_consistent_p()` which performs a dynamic type checking, and `gen_defined_p()` which is slightly stricted because `undefined` values (i.e. NIL pointers) are mostly prohibited.

These two type checkers are very useful when implementing program transformations.

It is possible to apply them systematically to all PIPS persitent objects by setting the proper debugging level for PIPSDBM (see [9]).

7 Disk Storage

The implicit global symbol table[31] of Section **??** is stored on or loaded from disk as a whole. It must be stored last and read first because other NewGen data structures contain pointers to it (every field of type `entity` is suchj a pointer). Pointers to entity are converted into global entity names on disk. The symbol table always is large because it contains at least all Fortran operators and intrinsics. It is stored in file `Entities` in the current workspace. See [6] for more information.

8 Conclusion

The PIPS internal representation is a relatively small set of data structures, which has very slowly increased since the project inception. Various mappings have been added. It was not possible to declare them with NewGen in 1988 and quite a few implicit mappings exist.

NewGen data types can be walked with two generic iterators, `gen_recurse()` and `gen_multi_recurse()`. These two iterators have been added to NewGen. They are not systematically used.

[31] This is implied by the `tabulated` attribute.

Annexe: NewGen Declarations – ri.newgen –

```
--   ------------------------------------------------------
--   ------------------------------------------------------
--
--    WARNING
--
--  THIS FILE HAS BEEN AUTOMATICALLY GENERATED
--
--       DO NOT MODIFY IT
--
--   ------------------------------------------------------
--   ------------------------------------------------------

-- Imported domains
-- ----------------

-- External domains
-- ----------------
external Pvecteur ;
external Psysteme ;

-- Domains
-- -------
application = function:expression x arguments:expression* ;
area = size:int x layout:entity* ;
basic = int:int + float:int + logical:int + overloaded:unit + complex:int + string:value +
callees = callees:string* ;
call = function:entity x arguments:expression* ;
cast = type x expression ;
code = declarations:entity* x decls_text:string x initializations:sequence x externs:entit
constant = int + float:float + logical:int + litteral:unit + call:entity + unknown:unit ;
controlmap = persistant statement->control ;
control = statement x predecessors:control* x successors:control* ;
dimension = lower:expression x upper:expression x qualifiers:qualifier* ;
dummy = unknown:unit + identifier:entity ;
entity_int = entity->int ;
entity_to_entity = entity->entity ;
evaluation = before:unit + after:unit ;
execution = sequential:unit + parallel:unit ;
expression = syntax x normalized ;
extension = pragma + unknown:unit ;
extensions = extension* ;
forloop = initialization:expression x condition:expression x increment:expression x body:s
formal = function:entity x offset:int ;
functional = parameters:parameter* x result:type ;
instruction = sequence + test + loop + whileloop + goto:statement + call + unstructured +
language = fortran:unit + c:unit + fortran95:unit + unknown:unit  ;
loop = index:entity x range x body:statement x label:entity x execution x locals:entity* ;
mode = value:unit + reference:unit ;
```

```
multitest = controller:expression x body:statement ;
normalized = linear:Pvecteur + complex:unit ;
parameter = type x mode x dummy ;
persistant_statement_to_cluster = statement:int -> number:int ;
persistant_statement_to_control = persistant statement -> persistant control ;
persistant_statement_to_int = persistant statement -> int ;
persistant_statement_to_statement = persistant statement -> persistant statement ;
pragma = string + expression* ;
predicate = system:Psysteme ;
preference = persistant reference ;
qualifier = const:unit + restrict:unit + volatile:unit + register:unit + auto:unit + threa
ram = function:entity x section:entity x offset:int x shared:entity* ;
range = lower:expression x upper:expression x increment:expression ;
reference = variable:entity x indices:expression* ;
sequence = statements:statement* ;
sizeofexpression = type + expression ;
statement = label:entity x number:int x ordering:int x comments:string x instruction x dec
storage = return:entity + ram + formal + rom:unit ;
subscript = array:expression x indices:expression* ;
symbolic = expression x constant ;
synchronization = none:unit + spawn:entity + barrier:unit + single:bool + critical:referen
syntax = reference + range + call + cast + sizeofexpression + subscript + application + va
tabulated entity = name:string x type x storage x initial:value x kind:int ;
test = condition:expression x true:statement x false:statement ;
transformer = arguments:entity* x relation:predicate ;
type = statement:unit + area + variable + functional + varargs:type + unknown:unit + void:
unstructured = entry:control x exit:control ;
value = code + symbolic + constant + intrinsic:unit + unknown:unit + expression + referenc
variable = basic x dimensions:dimension* x qualifiers:qualifier* ;
whileloop = condition:expression x body:statement x label:entity x evaluation ;
```

References

[1] B. Baron, *Construction flexible et cohérente pour la compilation inter-procédurale*, Rapport interne EMP-CRI-E157, juillet 1991

[2] A. J. Bernstein, *Analysis of Programs for Parallel Processing*, IEEE Transactions on Electronic Computers, Vol. 15, n. 5, pp. 757-763, Oct. 1966.

[3] B. Creusillet, *Analyses de régions de tableaux et applications*, Thèse de Docteurat, École des mines de Paris, Décembre 1996

[4] P. Jouvelot, R. Triolet, *NewGen: A Language Independent Program Generator*, Rapport Interne CAII 191, 1989 4

[5] P. Jouvelot, R. Triolet, *NewGen User Manual*, Rapport Interne CAII ???, 1990 4, 17, 28

[6] R. Triolet, *PIPSMAKE and PIPSDBM: Motivations et fonctionalités*, Rapport Interne CAII TR E/133 13, 15, 29

[7] R. Triolet, *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures*, Thèse de Docteur-Ingénieur, Université Pierre et Marie Curie, décembre 1984.

[8] R. Triolet, F. Irigoin, *PIPS High-Level Software Interface: Pipsmake* Documentation PIPS 29

[9] L. Zhou, F. Irigoin, *Properties: Low Level Tuning of PIPS*, PIPS Documentation 29

Index