Data and Process Abstraction in PIPS Internal Representation

Fabien Coelho, Pierre Jouvelot, Corinne Ancourt, François Irigoin

Data and Process Abstraction in PIPS Internal Representation

In proceedings, First Workshop on Intermediate Representations (WIR-1) Chamonix, France, April 2011

Technical Report MINES ParisTech A/447/CRI

Fabien Coelho Pierre Jouvelot Corinne Ancourt François Irigoin

CRI, Mathematics & Systems, MINES ParisTech
firstname.lastname@mines-paristech.fr

Abstract

PIPS, a state-of-the-art, source-to-source compilation and optimization platform, has been under development at MINES ParisTech since 1988, and its development is still running strong. Initially designed to perform automatic interprocedural parallelization of Fortran 77 programs, PIPS has been extended over the years to compile HPF (High Performance Fortran), C and Fortran 95 programs. Written in C, the PIPS framework has shown to be surprisingly resilient, and its analysis and transformation phases have been reused, adapted and extended to new targets, such as generating code for special purpose hardware accelerators, without requiring significant re-engineering of its core structure. We suggest that one of the key features that explain this adaptability is the PIPS internal representation (IR) which stores an abstract syntax tree. Although fit for source-to-source processing, PIPS IR emphasized from its origins the use of maximum abstraction over target languages' specificities and generic data structure manipulation services via the Newgen Domain Specific Language, which provides key features such as type building, automatic serialization and powerful iterators. The state of software technology has significantly advanced over the last 20 years and many of the pioneering features introduced by Newgen are nowadays present in modern programming frameworks. However, we believe that the methodology used to design PIPS IR, and presented in this paper, remains relevant today and could be put to good use in future compilation platform development projects.

1. Introduction

The number of maintained, source-to-source compilers available to the research community to implement advanced analyses and code transformations is small: First, the Rose compiler infrastructure [22] has been under development at Lawrence Livermore National Laboratory for about 10 years. The compiler is written in C++, and handles Fortran 2003 and earlier versions, as well as C and C++ code, through various front-ends. Its intermediate representation [23] emphasizes classes, and a rewriting engine is available [21]. Second, Cetus [11] is the successor to Polaris at Purdue University. It is developed in Java and supports C [20], with an emphasis on analyses and transformations for multicore targets [9]. Other open-source compilers are available, such as GCC [12], LLVM [2] and Open64 [8], but they are not designed for source-to-source.

The PIPS acronym stands for *Paralléliseur interprocédural de programmes scientifiques* (in French), that is "Interprocedural Parallelizer for Scientific Programs". Started in 1988 and funded by DRET[1], this project was intended to advance the state of the art in automatic interprocedural parallelization of legacy Fortran 77 programs [14]. From a Fortran 77 parallelizer, PIPS has evolved into an extensive, source-to-source, multi-language analysis and code transformation compiler framework [18], thanks to the extensibility and generality provided by its intermediate representation [5, 6] and modular design [15]. More than 50 people from various universities and companies have contributed to the project over the years, including 22 active developers from 5 institutions in Year 2010, who provided code in 4800 commits on PIPS subversion repositories. Open-source and distributed online under the GNU GPLv3, this sophisticated platform has been demonstrated in conferences [7]. Interested researchers can find an overview of PIPS key features in [4]. The PIPS Developer Tutorial [13] provides a smooth introduction to its key features from a developer perspective.

Now more than 20-year old, PIPS has for quite some time proven the soundness of its basic design; indeed, very few original key features have had, up to now, to be significantly challenged, despite the expansion of PIPS application domain. We suggest that one of these main reasons of such an unusual stability lies in the design of PIPS Intermediate Representation (IR), which promoted from the very beginning the use of abstraction, both in the definition and manipulation of core data structures. To support such a focus on abstraction, PIPS relies on the facilities provided by Newgen, a Domain Specific Language for data structure definition and manipulation that was designed specifically for this project. Of course, new proposals have been introduced, since the inception of the PIPS project, to ease the design and implemention of intermediate data structures of the type used in compiler suites (see for instance [19] or [10]). These new infrastructure tools are quite logically based on more abstract languages and powerful concepts. Newgen can be seen as an early, pioneering effort towards the use of more advanced DSL-based systems such as those offered in these newer tools. In fact, the success and longevity of a large project such as PIPS, inspired that this general approach, provides a useful case study for them.

[1] *Direction des recherches et études techniques*, then the French equivalent of DARPA.

We believe that, even though current software technology has significantly evolved since the inception of PIPS, the basic design approach we used back then is still valid today. This paper provides an up-to-date description of PIPS IR, some details of which, we claim, could be put to good use in future compilation platform development projects.

After this introduction, we present, in Section 2, our data definition and manipulation system: the Newgen DSL. Section 3 provides a brief overview of PIPS Internal Representation, initially targeting Fortran but now also supporting HPF and C, where we emphasize the abstraction focus used during the initial design phase. Then Section 4 describes the overall resource management in PIPS through PIPSmake for handling dependencies between phases and PIPSdbm (database) for storing computed data. Section 5 shows how Newgen powerful iterators simplify the implementation of PIPS analysis and transformation phases. We conclude in Section 6.

2. The Newgen DSL

Newgen [17] is a Domain Specific Language specialized in the definition of high-level data structure processing APIs. Beside traditional data creation, modification, access and serialization functions that systems such as IDL [24] or XDR [1] could have also provided, these APIs introduce multi-language support, dynamic type checking and higher-order iteration mechanisms. Even though the original version of Newgen was indeed designed to be used across different development languages (namely C and CommonLISP), Newgen nowadays can only be used from within a C application.

```
external Psysteme;
predicate = system:Psysteme;
tabulated entity = name:string x type x ...;
type = statement:unit + area + variable + ...;
reference = variable:entity x indices:expression*;
controlmap = persistant statement->control;
```

Figure 1. Examples of Newgen domain declarations

In practice, the developer defines new data structures using a simple syntax illustrated in Figure 1. Each equation provides the name and definition of a new user-defined datatype, also called "domain". These types are built up from basic, predefined types such as booleans, integers, floating point numbers and strings. Operators are used to build complex data types such as structures, with `x` as cross product operator, unions `+`, lists of anything `*`, arrays `[]`, sets {} and functional mappings `->`. Thus, for instance, Figure 1 states that a value r from the `reference` domain combines an `entity`, which could be accessed in C as `reference_variable(`r`)`, and a list of indices which are from the `expression` domain. External data types, that is data types unknown to Newgen, can also be used within a Newgen data structure; for such legacy data types such as `Psystem`, the developer is required to provide a set of functions to allocate, copy, free, serialize and deserialize such data.

From Newgen declarations, the Newgen compiler generates C text; this code includes `struct` declarations for each user-defined type and function definitions to create, copy, compare, test, check, update, destroy... data of these types. To limit code size explosion, these functions are in fact polymorphic, and the generated C structures include an integer to identify each Newgen data type. There structures can be walked through in a generic manner.

Once compiled, these data manipulation functions are linked to the Newgen runtime library, which contains a wide variety of functions to manipulate strings, string buffers, lists, stacks, internal hash tables used by sets, functions... This library also contains high-level introspection functions which can be used, for instance, to check that a structure is well defined or to serialize or deserialize data in a file while maintaining pointer sharing properties within structures. Finally, Newgen provides very powerful generic iterators; these innovative utilities are described in Section 5.

3. Intermediate Representation

All the key data structures required to represent user programs, also called "PIPS Internal Representation" [5], are defined using Newgen. The domain definitions and their corresponding documentation are stored in a single LaTeX file. When a new version of PIPS is created, a simple script extracts from this file the Newgen code, compiles it to generate C code which is then added to PIPS source code; the whole batch is then compiled with any C compiler such as `gcc`.

Even though PIPS was initially designed for optimizing Fortran 77 source code, a great deal of abstraction was applied from the very beginning in order to make it extensible. As time has shown, this was a prescient decision; since then, PIPS has evolved to become a much more general platform, able to deal not only with Fortran but also HPF and C source code. To illustrate the focus on high-level concepts that was sought during the initial design phase, this section addresses three points of the IR that exhibit in a clear way such a concern: (1) the symbol table; (2) code and expressions; (3) analysis results and code decorations.

3.1 Symbol Table

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

type = statement:unit + area + variable +
  functional + varargs:type + unknown:unit +
  void:qualifier* + struct:entity* +
  union:entity* + enum:entity*;
variable = basic x dimensions:dimension* x
  qualifiers:qualifier*;
basic = int:int + float:int + logical:int +
  overloaded:unit + complex:int + string:value +
  bit:symbolic + pointer:type + derived:entity +
  typedef:entity;
dimension = lower:expression x upper:expression;
qualifier = const:unit + restrict:unit +
   volatile:unit + register:unit + auto:unit;
functional = parameters:parameter* x result:type;

storage = return:entity + ram + formal + rom:unit;

value = code + symbolic + constant +
  intrinsic:unit + unknown:unit + expression;
```

Figure 2. Definition of `entity`, for PIPS symbol table elements

All symbols in user programs are represented in PIPS IR as values of the `entity` domain (see an extract in Figure 2). The Newgen keyword `tabulated` specifies that all such symbols are kept in a table, which is global. It also implicitely create an indexing structure using a hash table, so that all entities can be retrieved quickly based on their name. This fact may seem to be, and sometimes is, a significant constraint when performing analyses, but it is at least partly mandatory to have a global table. Indeed, interprocedural analyses need to store information about symbols not necessarily visible from within the scope of a routine; this is the case, for instance, when having to represent the side effect on a static variable mutated in a function call. Another approach could be to manage global symbols only for those objects that need to be treated as such, but this would add a lot of complexity to the compiler to choose which symbols need to be global and to manage both local and global symbols.

The symbol table stores anything in user source code that has a `name` feature, represented by a string; this `name` is also used by the Newgen API as a key to retrieve the information (e.g., a `type`) associated to the corresponding entity. Abstraction led to the inclusion in the symbol table of not only user local or global variables, but also of functions, intrinsics, operators, even constant

values (integer or strings for instance) which are seen as 0-ary functions.

In order to have a unique name for all these symbols, a notion of name space is used when constructing the string name. For instance `FOO:I` would be Variable `I` within Fortran Subroutine `FOO`. Additional character prefixes are used to distinguish name spaces in Fortran for labels, commons, program declarations or block data objects. Special names are also introduced for particular objects manipulated by various analysis phases, for instance to identify different kinds of memory classes such as stack, heap, static and dynamic segments.

More prefixes were added when the IR needed to be extended to handle C and Fortran 95. First, as Fortran is case insensitive, names needed to be normalized by switching them to upper case, but as C is not this cannot be done across the board. Second, four scope levels are used in C to deal with possibly homonymous but distinct objects: a function definition scope, similar to Fortran above; a file scope (two static functions of the same name can be defined in different files); a block scope to deal with homonymous variables in different blocks of the same function; a function prototype scope to handle argument names in a prototype declaration. Figure 3 illustrates some of these scopes with a homonymous function, structure, field and argument which are all legal in C. Moreover, more special characters are also used as magic numbers to distinguish between different kinds of objects, for instance a `struct` from a `union` from an `enum` from a `typedef`, some of which may all have the same name.

The global symbol table associates a type, a storage and an initial value to a each symbol name (see Figure 2):

Type. The `type` of a symbol may be used to differentiate between a variable, a function or operator, an area (for Fortran commons), a label in the code or a data construct such as C structs, unions or enums. A set of basic type allows to use the target language types such as *logicals*, *ints* of differing size and pointers. But complex types may also reference other types; for instance a variable has a type, while a function type includes its return type and the types of its arguments. The return type of a function result in `functional` can be either determined dynamically, when needed, by a typechecking phase or precomputed by a phase that inserts the necessary casts to explicit type conversions in the return expressions. Overloaded operators or functions (for instance `+` in Fortran and C or `COS` in Fortran, which must be able to deal with scalar, complex, float or double arguments) use the special *overloaded* type.

Storage. The second information associated to a symbol is its `storage`. It is used to distinguish a parameter (`formal` storage) from a constant (`rom` storage) or a standard variable (`ram` storage).

Value. The third and last information is a `value`. This may be an initialization expression for a variable, while, for a function, this field contains its internal declarations.

The PIPS symbol table is thus the single global reference for anything with a name in the compiler, including intrinsics, operators, labels or constants. Note that the definition of a symbol can be partial when encountering a partial declaration (say `int foo()` which, in C, does not declare the function argument's type), so special *unknown* values may be used and fixed later on when the information is eventually available. A set of utility functions is provided to search for the relevant description from a local name found in a file at parse time, depending on the context.

```
struct X { int X; };
extern void X(int X);
```

Figure 3. Homonymous symbols in C

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

instruction = sequence + test + loop +
  whileloop + goto:statement + call +
  unstructured + multitest + forloop +
  expression;

sequence = statements:statement* ;

test = condition:expression x true:statement x
  false:statement;

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

whileloop = condition:expression x
  body:statement x label:entity x evaluation;

call = function:entity x arguments:expression*;

unstructured = entry:control x exit:control;
control = statement x predecessors:control* x
  successors:control*;

forloop = initialization:expression x
  condition:expression x increment:expression x
  body:statement;

expression = syntax x normalized;
syntax = reference + range + call + cast +
  sizeofexpression + subscript + application +
  va_arg:sizeofexpression*;
reference = variable:entity x
  indices:expression*;
```

Figure 4. Partial Newgen definition of PIPS abstract syntax tree

3.2 Code and Expressions

The second key Newgen domain in PIPS IR is `statement`, which deals with the representation of the source code; this is a recursive[2] domain, which includes a sub-domain dedicated to expressions (an extract of the Newgen definition for PIPS `statement` is presented in Figure 4). Newgen domains are available for all traditional syntactic statements such as sequences, tests, function calls, loops and unstructured code. The `goto` field in `instruction` is temporarily used by the parser, but does not appear anymore once the code controlizer phase has changed non-structured code fragments into `unstructured` graphs (see below). Interestingly, notice that, again for abstraction purposes, there are no assignments per se; they are seen as predefined function calls, where the left-hand side expression is supposed to be passed by reference (this is indicated in the function type of the assignment `=`).

PIPS manages the source as an abstract syntax tree, so as to be able to regenerate source code as close as possible to what the user had input. Thus, for instance, great care is taken to keep track of user comments so that they can be regenerated where needed. Also, PIPS supports three[3] kinds of loops: one for the Fortran DO loop, one for the C `while` loop, with the condition evaluated before the loop body (standard) or after (`do...while`), and one for the C `for` loop; although sugarized equivalents of a while loop, these variants are kept as such for regeneration purposes. However, since

[2] Elements of the `statement` domain reappear in sequences, test branches, loop bodies...

[3] Some special looping constructs, for instance Fortran "implied DO" loops used in I/O statements, are represented by special function calls.

```
      do i=1, n
100      continue
         do j=1, m
            t(j) = t(j) + x
         enddo
         if (t(i).lt.x) goto 100
      enddo
```

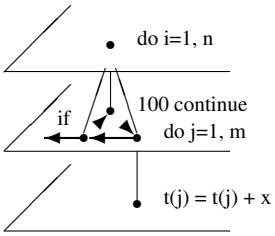

Figure 5. Hierarchical Control Flow Graph

semantical analyses are developed on top of these representations, the specific loop variant used must be trustworthy in the semantical sense: if a fixed range loop is provided in a DO loop, it must really be such a loop and there should be no other exit points.

In order to ensure this trustworthiness property, one of the first transformations performed by PIPS after parsing user code is to desugarize loops with internal exit points and translate them as unstructured code, represented by a hierarchical control flow graph (CFG). However, PIPS strives to keep the control flow as structured as possible, since unstructured flow graphs usually induce drastic approximations in subsequent analysis phases. This is illustrated in Figure 5, where the two Fortran DO loops will be kept as DO loops even though there is a goto in between them, since both external and internal loops do not have other exit points. The end result will be a DO loop that contains an unstructured graph that contains a DO loop.

Although PIPS symbol table is global, pieces of code can be stored independently, one function definition at a time, without any need to keep them in memory but for special transformations such as function inlining.

As PIPS internal representation is very generic, and as most of Fortran semantics is included in C, we developed a prettyprinter phase which generates C code from a Fortran program based in PIPS intermediate representation. The phase was developed in very little time to achieve quite reasonnable results on simple codes.

3.3 Decorations

An important data requirement for a compiler-analyzer is to be able to associate information, typically the result of analysis phases, to points in source code. In PIPS, some relevant semantical information is stored directly within the data structures of the intermediate representation: for instance, whether a DO loop is parallel and which scalar variables are private to the loop. This direct IR embedding is an historical reminder of the initial goal of PIPS, which was to parallelize loops.

Nowadays, such on-the-side information, also called "decoration", is based on additional data structures and links to the code segment they correspond to. For instance, the reduction detection phase builds a structure that holds the expression being reduced with which operator stored in which variable... Whenever possible, these data structures are expressed with Newgen, so as to benefit directly from its serialization engine to ensure their persistence.

However, most of PIPS semantical analyses (transformers, preconditions, array regions) rely on a polyhedron representation of constraints on the integer variables of the program, mostly used as array indices. These data structures were developed independently of Newgen, and are imported as *external* data types by providing their specific manipulation functions for serialization, copying, and so on. A set of hooks in the linear library used by PIPS allows the use of PIPS symbols, defined with Newgen, as variable names in the mathematical library, so that any cyclic dependence is resolved.

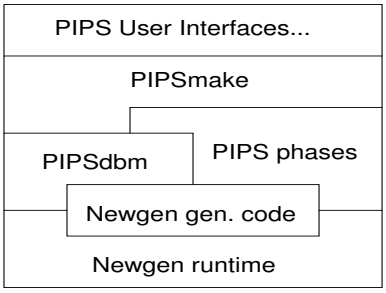

Figure 6. PIPS data management overview

Finally, the decoration must be associated with some points of the analyzed program. From a programming perspective, these are functional mappings, which are available in Newgen. In practice, such mappings can be implemented as hash tables linking the source of the information, for instance a particular statement in the code, to a descriptor that holds the result of the analysis. The biggest challenge raised by such decorations is their serialization. Indeed, if one decided to store these maps as such, the Newgen serialization engine would recursively traverse the whole data structure, which includes the code fragments to which decorations are associated. If these decorations have to be stored on file, the code does not: one does not want a new entire copy of the program source code for each kind of decoration stored on file. Not only would this be a waste of disk space, this would not do the job: when reading such data back into memory, it would be quite difficult to link together the various decorations that correspond to the same code fragment. To alleviate such a difficulty, Newgen provides the `persistant` keyword, which indicates to the serialization engine to stop there its recursive traversal. PIPS then relies on the additional notion of "statement number" to provide a unique reference to each source code point.

4. PIPS Data Management

The management of data structures in PIPS is performed by a generic make-like engine (PIPSmake), which decides when to generate or regenerate a resource, for instance the decorations computed by a specific analysis phase, based on declared dependencies [15] between resources, and a file storage engine, which is a specialized database (PIPSdbm). The management of persistence by the database relies on Newgen serialization features. The database keeps track of all the computed resources associated to an analyzed code, whether present in memory or saved in files.

Figure 6 illustrates the whole demand-driven process. When a resource is required from a user interface, such as `tpips` scripting shell, `pyps` PIPS-python bindings, or `paws` *PIPS As a Web Service*, PIPSmake checks with PIPSdbm whether the resource is already available. If not, PIPSmake recursively checks for the resources that are needed to compute this target resource, thanks to declared dependencies, and then calls the PIPS phase that produces the expected result. This phase will get the resources it needs from PIPSdbm, build the new resource with the help of Newgen generated functions and Newgen runtime, and put the result into PIPSdbm. When closing PIPS, PIPSdbm serializes resources into files in a dedicated directory, thanks to Newgen serialization capabilities.

A slightly simplified extract of the rules used by PIPSmake to generate resources is shown in Figure 7. Starting from the last rule, if the user requires the prettyprinted (`PRINTED_FILE`) version of a module (a function), the `print_code` phase will be launched by PIPSmake. This requires the program's `ENTITIES` and the module's `CODE` produced by the `controlizer`. This in turn is a transformation of the `PARSED_CODE` (raw AST) output by the `parser` phase. And so on up to the `initializer` and `bootstrap` phases, which imports the source files given by the user and generate the

initial symbol table. All these resources expressed in the requirements (<) and productions (>) of the derivation rules. They are computed on demand and the actual resources are stored in PIPSdbm.

PIPSdbm uses data structures managed with Newgen, as shown in Figure 8. A `db_symbol` can be either a "module" name, such as the name of a function for which analysis results are stored, or a special name denoting the results of a particular global analysis, say the initial preconditions of a full program. To each such symbol is associated, via `db_resources`, the resources linked to it, which are again a mapping, the `db_symbol` describing now the kind of resource, for instance *parsed code* or *preconditions*, and its status. A resource is represented by a generic pointer, of type `db_void`, since any type of resource may be stored and we do not want to have to change the PIPSdbm metadata definition each time a new kind of resource is added. Its logical status can be specified as: only in memory (`loaded`), only in file (`stored`) or available in both, once loaded and when not modified. The last status (`required`) is used internally when computing phase dependencies, with the help of logical timestamps (`time`) and possibly actual file times.

After saving all the program's data computed by PIPS, the database serializes its metadata in a file as its last operation and, on reopening the database, its status is automatically imported back in memory. Such a "Save World" capability is a key asset of the PIPS platform when performing expensive analyses on very large programs, as this allow to checkpoint the current state of the analysis process so that a later failure, even a "core dump" of a subsequent analysis phase, can be rolled back where it started.

One special feature of the Newgen serialization technique is that the correspondance between domain names and type numbers is explicitely stored. Minor changes to a data structure definition, for instance adding a new possible field at the end of an union (+) or new independent types for a specific phase, can be performed without compromising the ability to deserialize successfully a structure saved before such a design change.

5. Newgen Iterators

The Newgen runtime library provides several innovative tools, including a powerful recursion engine [16] which is very convenient when gathering information and performing syntax-directed transformations on a Newgen data structure. The recursion engine was initially designed for internal use within Newgen to implement the functions that had to recursively walk through data, such as serialization or recursive copies. It has rapidly been extended and optimized to be usable directly by the developer on its own data structures.

The most generic version of the engine is the *contextual multi domain* recursion `gen_context_multi_recurse` function, which

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

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

bootstrap   > PROGRAM.entities

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

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

print_code  > MODULE.printed_file
    < PROGRAM.entities
    < MODULE.code
```

Figure 7. PIPSmake rules for building the symbol table and code

```
tabulated db_symbol = name:string;
db_resources = db_symbol -> db_owned_resources;
db_owned_resources = db_symbol -> db_resource;
external db_void;
db_resource = pointer:db_void x db_status x
    time:int x file_time:int;
db_status = loaded:unit + stored:unit +
     required:unit + loaded_and_stored:unit;
```

Figure 8. Full Newgen Definition of PIPS database metadata

```
typedef struct {
  entity var;  bool is_index;
} ctx;

static bool loop_flt(loop l, ctx * c) {
  if (loop_index(l)==c->var &&
      gen_get_ancestor(test_domain, l)!=NULL) {
    c->is_index = true;
    gen_recurse_stop(NULL);
  }
  return true;
}

bool var_is_index_in_test(statement s, entity v) {
  ctx cs = { v, false };
  gen_context_multi_recurse(s, &cs,
     loop_domain, loop_flt, gen_null,
     NULL);
  return cs.is_index;
}
```

Figure 9. Is a variable used as a DO loop index within a test?

is illustrated in Figures 9 and 10. When using this facility, the developer must provide a NULL-terminated, variable-length list of arguments. The first two are the root Newgen data structure down from which the recursive traversal will be performed and a context data structure. Then, as many argument triplets as necessary can be passed as arguments: each represents a domain tag, a *filter* function and a *rewrite* function. Whenever a domain tag is provided, this indicates that all Newgen data structures belonging to such a domain will be visited during the traversal; the corresponding filter function is applied when going downwards and returns a boolean indicating whether the recursion must continue below that node, and the rewrite function is applied when going upwards and if the filter returned true, and can be used to modify the visited structure as needed. This walk-through process is optimized so as to only recurse in structures that contain data from the domains to be visited, thanks to a transitive closure computed on the domain occurrence graph induced by the Newgen definitions; this optimization typically reduces the number of visited nodes by half. Cyclic and shared data structures are also properly dealt with and visited only once.

Figure 9 illustrates a simple but full implementation of the function `var_is_index_in_test` that tells whether Variable `v` is used, inside Statement `s`, as a DO loop index that appears within a test. The context structure `cs`, of type `ctx`, passes around to all the functions called during the recursion both the variable of interest and the boolean `is_index` indicating whether a compatible instance has been found. The Newgen-provided convenient `gen_get_ancestors` function queries the current recursion stack to know about enclosing data structures, while `gen_null` is a nop function. The whole recursion is interrupted as soon as a compatible variable is found, using the exception-like `gen_recurse_stop` function to abort further traversal.

Figure 10 shows a code transformations algorithm which may modify two domains. The function `subs_var` intends to replace

```
typedef struct {
  entity from, to;
} ctx;

static void loop_rwt(loop l, ctx * c) {
  if (loop_index(l)==c->from)
    loop_index(l) = c->to;
}

static void ref_rwt(reference r, ctx * c) {
  if (reference_variable(r)==c->from)
    reference_variable(r) = c->to;
}

void subs_var(statement s, entity from, entity to) {
  ctx cs = { from, to };
  gen_context_multi_recurse(s, &cs,
    loop_domain, gen_true, loop_rwt,
    reference_domain, gen_true, ref_rwt,
    NULL);
}
```

Figure 10. Variable substitution for Fortran

```
int compute(int n) {
  int i = 1;
  while (i<n) {
    i<<=1;
    if (rand()) i++;
  }
  return i;
}
```

```
int compute(int n)
{
  int i = 1;
  int _if_then_0 = 0,
      _if_else_0 = 0,
      _while_0 = 0;
  while (i<n) {
    _while_0 = _while_0+1;
    i <<= 1;
    if (rand()) {
      _if_then_0 = _if_then_0+1;
      i++;
    }
    else
      _if_else_0 = _if_else_0+1;
  }
  return i;
}
```

Figure 11. Initial (left) and instrumented (right) code

every occurrence of Variable `from` by another variable, `to`. Since variables in a program only occur as loop indices or references, only these two domains need to be managed; all others are simply recursively traversed. The Newgen function `gen_true` always returns true.

The *Add Control Counters* transformation instruments a piece of code with local integer counters on test and loop control structures, as illustrated in Figures 11. The aim is to test whether adding such variables may help some semantical analyses, for instance by finding a loop invariant such as:

$$\texttt{\_if\_then\_0} + \texttt{\_if\_else\_0} = \texttt{\_while\_0}$$

in the code. Figure 12 shows an extract from PIPS implementation of this program transformation. The code has been stripped of its includes and of some comments, and expurged of half a dozen instructions or expressions to fit the column length, but otherwise all is there. Three support functions are used to create a variable with a prefix (`create_counter`), to generate an incrementation statement (`make_increment_statement`) and to perform the actual insertion on a given statement (`add_counter`) using the previous two functions. The Newgen iterator is called from the `add_counters` function with a small context that holds the current module. The transformation is applied on each target control structure with a rewrite function which provides the counter variable prefix to be used in its underlying statements. The last function (`add_control_counters`) is called by PIPSmake to actually

```
// (c) 1989-2011 MINES ParisTech
// This file is part of PIPS. PIPS is free software...
// See the GNU General Public License for more details.
#include "..."

// generate: var = var + 1
statement make_increment_statement(entity var) {
  return make_assign_statement(...);
}

// create a new integer local variable in module
entity create_counter(entity module, string name) {
  return ...;
}

// Add Control Counter recursion context
typedef struct { entity module; } acc_ctx;

// add a new counter at entry of statement "s"
void add_counter(acc_ctx * c, string name, statement s)
{
  entity counter = create_counter(c->module, name);
  insert_statement(s,
    make_increment_statement(counter), true);
}

void test_rwt(test t, acc_ctx * c) {
  add_counter(c, "if_then", test_true(t));
  add_counter(c, "if_else", test_false(t));
}

void loop_rwt(loop l, acc_ctx * c) {
  add_counter(c, "do", loop_body(l));
}

void whileloop_rwt(whileloop w, acc_ctx * c) {
  add_counter(c, "while", whileloop_body(w));
}

void forloop_rwt(forloop f, acc_ctx * c) {
  add_counter(c, "for", forloop_body(f));
}

// add control counter instrumentation
void add_counters(entity module, statement root)
{
  acc_ctx c = { module };
  gen_context_multi_recurse
    (root, &c,
     test_domain, gen_true, test_rwt,
     loop_domain, gen_true, loop_rwt,
     whileloop_domain, gen_true, whileloop_rwt,
     forloop_domain, gen_true, forloop_rwt,
     NULL);
}

// PASS: instrument with control structure counters
bool add_control_counters(string module_name) {
  // get resources from database
  entity module = module_name_to_entity(module_name);
  statement stat = (statement)
    db_get_memory_resource(DBR_CODE, module_name, true);
  set_current_module_entity(module);
  set_current_module_statement(stat);

  // do the job!
  add_counters(module, stat);

  // put updated code back in database
  DB_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, stat);
  reset_current_module_entity();
  reset_current_module_statement();
  return true;
}
```

Figure 12. Instrument code with counters on control structures

perform the transformation on a module: it loads the needed data structures from PIPSdbm, performs the transformation on the module's code and stores the result back in the database.

With the Newgen recursion engine, the developer can implement simple tests and transformations of the data structures very simply, with supporting functions to explore the context upwards from the current point in the recursive descent, and to stop or even fully abort recursion.

6. Conclusion

Innovative abstraction techniques of data and processes lie at the core of the Intermediate Representation used within the PIPS optimizing compiler. The Newgen Domain Specific Language supports both: domain definitions provide powerful APIs to manipulate data, while the `gen_*_recurse` family of traversal functionals abstract typical syntax-directed program transformation algorithms. This methodology has proven to be sound and flexible over the past two decades, a time period during which PIPS successfully evolved from a narrowly-focused Fortran parallelizer to a multi-language, multi-analysis optimization platform able to deal with Fortran 77, Fortran 95, HPF and C.

If some of the features provided by Newgen are now directly supported by modern object-oriented languages such as Python or Java (e.g., serialization and cloning), its generic traversal routines remain quite innovative. Using Newgen as is might not be optimal if one were envisioning the development of a brand new optimizing platform today, but we believe that the approach methodology pioneered in PIPS and Newgen remain sound, and that applying it in such a project would significantly increase its success.

A web interface [3] is available to navigate through PIPS internal representation online, so that interested readers can get a quick understanding of the look and feel of the PIPS IR of a given program.

Acknowledgements

As mentionned in the introduction, PIPS current development is the direct result of the work of more than 50 people over 20 years. Special thanks are due to Rémi Triolet and Thi Viêt Nga Nguyen for their contributions to the definition and extensions of the PIPS intermediate representation.

References

[1] External data representation standard. ARPA NIC RFC, June 1987.

[2] The LLVM Compiler Infrastructure. `http://llvm.org/`, Since 2002.

[3] Mehdi Amini. PIPS Internal Representation Web Navigator. `http://www.pips4u.org/ir-navigator/`, April 2010.

[4] Mehdi Amini, Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Béatrice Creusillet, and Serge Guelton. PIPS Is not (just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[5] Mehdi Amini, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, Thi Viet Nga Nguyen, Rémi Triolet, and Pierre Villalon. PIPS: Internal Representation of Fortran and C Code. Technical report, MINES ParisTech, CRI, Maths & Systems, 1988–2011. `http://www.cri.ensmp.fr/pips/newgen/ri.htdoc`.

[6] Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Thi Viet Nga Nguyen. PIPS: Extension of the Internal Representation for C. Technical report, MINES ParisTech, CRI, Maths & Systems, 2003–2010. `http://www.cri.ensmp.fr/pips/newgen/ri_C.htdoc`.

[7] Fabien Coelho and Ronan Keryell. Demonstrations of PIPS and HPFC. PRS booth at Supercomputing Conference, San Diego, USA, December 1995.

[8] Computer Architecture and Parallel Systems Laboratory. Open64 Compiler. `http://www.open64.net/`, Since 2006.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *IEEE Computer*, 42(12):36–42, December 2009.

[10] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35 – 61, 2004. Annotated Terms (ATerms).

[11] Rudolf Eigenmann and Samuel Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for C Programs. `http://cetus.ecn.purdue.edu/`, Since 2003. Purdue University, IN, USA.

[12] David et al. Henkel-Wallace. GCC: GNU Compiler Collection. `http://www.gnu.org/software/gcc/`, August Since 1997. Fortran, C, C++ and other compilers.

[13] François Irigoin, Serge Guelton, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS: An Interprocedural Extensible Source-to-Source Compiler Infrastructure for Code/Application Transformations and Instrumentations. Tutorial at PPoPP 2010 on `http://pips4u.org/`, January 2010. ACM Sigplan Symposium, Bengalore, India.

[14] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *Conference on Supercomputing*. ACM, June 1991.

[15] François Irigoin, Rémi Triolet, and many others. PIPS High-Level Software Interface Pipsmake configuration. Technical report, MINES ParisTech, CRI, Maths & Systems, 1988-2011. `http://www.cri.ensmp.fr/pips/pipsmake-rc.htdoc/`.

[16] Pierre Jouvelot and Babak Dehbonei. Recursive pattern matching on concrete data types. *ACM Sigplan Notices*, 24(11):84–91, November 1989.

[17] Pierre Jouvelot and Rémi Triolet. Newgen: A language independent program generator. Technical Report CRI/A-191, MINES ParisTech, July 1989.

[18] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report CRI/A/289, MINES ParisTech, April 1996.

[19] Ralf Lammel, Eelco Visser, and Joost Visser. The essence of strategic programming. *Draft*, 15, 2002.

[20] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003 2003.

[21] D. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, KY, USA, August 2001.

[22] Daniel J. Quinlan. Rose source-to-source compiler infrastructure. `www.rosecompiler.org`, Since 2001. Lawrence Livermore National Laboratory (LLNL).

[23] Daniel J. Quinlan and B. Philip. ROSETTA: The Compile-Time Recognition of Object-Oriented Library Abstractions and Their Use Within User Applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2001.

[24] R. Snodgrass. *The Interface Description Language*. Computer Science Press, 1989.