Data and Process Abstraction in PIPS Internal Representation
Fabien Coelho, Pierre Jouvelot, Corinne Ancourt, François Irigoin

To cite this version:

Fabien Coelho, Pierre Jouvelot, Corinne Ancourt, François Irigoin. Data and Process Abstraction in PIPS Internal Representation. Troisièmes Rencontres de la Communauté Française de Compilation, Apr 2011, Dinard, France. hal-00744294

HAL Id: hal-00744294
https://hal-mines-paristech.archives-ouvertes.fr/hal-00744294
Submitted on 22 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Data and Process Abstraction in PIPS Internal Representation

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


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 [24]. 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 [3] and Open64 [8], but they are not designed for source-to-source.

The PIPS acronym stands for Parallélliseur interprocédural de programmes scientifiques (in French), that is “Interprocedural Parallelizer for Scientific Programs”. Started in 1988 and funded by DRET1, this project was intended to advance the state of the art in automatic interprocedural parallelization of legacy Fortran 77 programs [15]. From a Fortran 77 parallelizer, PIPS has evolved into an extensive, source-to-source, multi-language analysis and code transformation compiler framework [13], 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 implementation 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.
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 within a C application.

external Paysteme;
predicate = system:Paysteme;
tabulated entity = name:string x type x ...;
type = statement:unit + area + variable + ...;
reference = variable:entity x indices:expression*;
controlmap = persitent: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 lists 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 \( \text{reference} \_\text{variable}(r) \), 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 Paysteme, 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 \text{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” [3], are defined using Newgen. The domain definitions and their corresponding documentation are stored in a single \text{.\text{tex}} 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A string 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</td>
</tr>
<tr>
<td>type</td>
<td>A functional or a constant, represented by a string; this type 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</td>
</tr>
<tr>
<td>value</td>
<td>A symbolic expression, represented by a string; this value 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</td>
</tr>
</tbody>
</table>

Figure 2. Definition of entity, for PIPS symbol table elements
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:1 would be Variable 1 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 lev-
eels 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 dif-
f erent 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 4 illus-
trates 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 each symbol name (see Figure 4).

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 conver-
sions in the return expressions. Overloaded operators or func-
tions (for instance + in Fortran and C or □□□□□ in Fortran, which
must be able to deal with scalar, complex, float or double argu-
ments) 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 (runtime storage) or a standard variable (run 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, opera-
tors, 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 spe-
cial unknown values may be used and fixed later on when the infor-
mation 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.

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

**Figure 3.** Homonymous symbols in C

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

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

```c
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
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 syn-
tactic 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
controller 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 expres-
sion 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 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.
A set of hooks in the linear library used by PIPS allows the
their specific manipulation functions for serialization, copying, and
of Newgen, and are imported as
array indices. These data structures were developed independently
conditions, array regions) rely on a polyhedron representation of
efit directly from its serialization engine to ensure their persistence.
Whenever possi-
phase builds a structure that holds the expression being reduced
bedding is an historical reminder of the initial goal of PIPS, which
which scalar variables are private to the loop. This direct IR em-
ate representation: for instance, whether a DO loop is parallel and
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 preprocessor
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 reasonable 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 intermedi-
ate representation: for instance, whether a DO loop is parallel and
which scalar variables are private to the loop. This direct IR em-
bedding 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 ben-
efit directly from its serialization engine to ensure their persistence.
However, most of PIPS semantical analyses (transformers, pre-
conditions, 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.
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 prac-
tice, 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 serializa-
tion. Indeed, if one decided to store these maps as such, the New-
gen 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
persistent 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 gen-
erate or regenerate a resource, for instance the decorations com-
puted by a specific analysis phase, based on declared dependen-
cies [15] between resources, and a file storage engine, which is
a specialized database (PIPSdbm). The management of persis-
tence 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 the scripting shell, it
requests a PIPS-python bindings, or papsys PIPSAs 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 PIPS-
dbm, 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 6. 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 mod-
ule’s CODE produced by the controller. This in turn is a trans-
formation 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

Figure 5. Hierarchical Control Flow Graph

Figure 6. PIPS data management overview
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_resource, for instance db in Figure 8. A

Figure 8. Full Newgen Definition of PIPS database metadata

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

static bool loop_flt(int 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 7 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 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 7. PIPSmake rules for building the symbol table and code

5. Newgen Iterators
The Newgen runtime library provides several innovative tools, including a powerful recursion engine 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 serializing or recursive copies. It has rapidly been extended and optimized, 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 7. PIPSmake rules for building the symbol table and code
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) {
        if (i==1)
            i<<=1;
        else
            if (rand())
                i++;
    }
    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 Figure 11. The aim is to test whether adding such variables may help some semantical analyses, for instance by finding a loop invariant such as:

\[ \text{if then}_0 + \text{if else}_0 = \text{while}_0 \]

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
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 mentioned 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 Viet Nga Nguyen for their contributions to the definition and extensions of the PIPS intermediate representation.

References