PIPS: Extension of the Internal Representation for C

Fabien Coelho
Béatrice Creusillet
François Irigoin
Pierre Jouvelot
Ronan Keryell
Thi Viet Nga Nguyen
CRI, MINES ParisTech

March 2, 2016, revision 23065

Contents

Introduction

This document discusses the implementation choices of the high-level description of data structures used in PIPS as internal representations (RI) of C programs.

These data structures are declared using the Newgen Data Definition Language in the Abstract Syntax Tree `ri.tex` companion document and shared with the Fortran77 parser..

Here are the goals of our work:

- Enough information must be preserved to prettyprint source code from the internal representation: this is the ultimate goal we must meet.
- Whole program must be stored in the internal representation. Modules written in different languages such as Fortran and C must be stored in memory as part of one application, using the very same data structures.
- Make the internal representation compatible with the SPEC2000 CFP, SPEC1995 INT and SPEC2002 HPC as a first step, and extend it later if necessary to cover all of C ISO 99.

The reader is assumed knowledgeable in Newgen [?] and the internal representation used for Fortran [?]. Another important reference is CIL, a C Intermediate Language developed at University of California [?], which is used for comparison purposes.

The standard initially used was ISO C89, but C99 extensions must be supported too.

In Section 1, we deal with naming issues. Then in Section **??**, the memory storage of different classes of variables is presented. Types in C are much more diverse than in Fortran 77 and numerous extensions are presented in Section3. C expressions are also extended beyond Fortran 77 with features such as casts, sizeof and address-of. These issues are handled in Section 4. Finally control flow extensions with respect to Fortran 77 are covered in Section 5.

Beware that details about such or such value given in this report may be outdated. Please check `ri.newgen` and `ri-util-local.h` files for programming purposes.

Chapter 1

Naming

In C, the scope of a local variable is the block where it is declared, the scope of an external static variable is the source file where it is declared, not the module as in Fortran. So when it is necessary, we have to add all information such as the current source file, module and block to the entity name in order to locate an entity in the symbol table. Here are several objectives for naming different kinds of entities:

- As short as possible;
- As significant as possible;
- As uniform as possible;
- As efficient as possible;
- To be able to recompile after modifying a file (???);
- Faithful with respect to the original declarations ;
- Compatible with the existing internal representation (Fortran).

Before entering the details related to this naming issue, here are some recalls of the C standard [**?**].

1.1 ISO C concepts

An identifier can denote

- an object (i.e. a variable)
- a function
- a tag of a structure, union or enumeration
- a member of a structure, union or enumeration
- a typedef name
- a label name

1.1.1 Scopes of identifiers (6.2.1)

There are four kinds of scopes: function definition, file, block and function prototype, also known as function declaration.

- A label name has *function definition scope.*
- If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope.*
- If the declarator or type specifier that declares the identifier appears inside a block, the identifier has *block scope.*
- If the declarator or type specifier that declares the identifier appears inside the list of formal parameters of a function definition, the identifier has *function definition scope.*
- If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype, the identifier has *function prototype scope.*

1.1.2 Linkages of identifiers (6.2.2)

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by the *linkage* (external, internal and none).

So the concept of global entity (`TOP_LEVEL`) does not exist for C, but we can use it to refer to external linkage entities, which include functions and objects but not tags for structures, unions, enumerations and typedef names.

1.1.3 Name spaces of identifiers (6.2.3)

There are separate name spaces :

- label names
- tags of structures, unions and enumerations
- members of structures or unions
- ordinary identifiers

Members of enumerations are handled as identifiers.

1.2 File name scope

There may be functions or variables that have the same name but in physically different files, such as :

- in `Directory1/foo.c`:

  ```
  static bar()
  ```

- in `Directory2/foo.c`:

```
static bar()
```

In order to distinguish these two static functions, the two different absolute file names must be taken into account[1]. The same situation can arise with `typedef, struct, union` and `enum` types, i.e two structures with the same name in different files are different. There are several possibilities to uniquely name a source file

1. Use the absolute paths (problem with name lengths and file moves)
2. Use the relative paths (problem when copying/moving the database system)
3. Use specific naming in PIPS DBM, with a database that stores the correspondences between the actual file name (absolute or relative, FC ?) and the specific name. This is the current solution for Fortran.
4. By default: relative and an option for absolute

There may be problems related to special characters used in a C file name which can be in conflict with characters used as separators in the entity name, but only the `MODULE_SEPARATOR` is critical.

Another possible whole program compilation problem: different compilation units use different names for structurally equivalent types. CIL [?] resolves this problem by some merging phases : merge types and tags and rewrite variable declarations and function bodies.

1.3 Block scope

There are different solutions to handle the blocks:

1. Flattening like CIL [?] and then ignoring completely these block scopes, except for memory allocation and for prettyprinting.
2. Flattening the blocks and conserving the scope information somewhere to regenerate code. There must have special separators and tables to interchange between the internal names and external names. Problem for debugging? Problem for controlizer?
3. Dewey indexing for blocks (like trees). It can be associated to the above solution ?
4. Multiple symbol tables. Note: the actual solution, a unique symbol table for general "entity" may create problem when dealing with large-scale programs and prevents PIPS distribution.

We do not use flattening as in CIL because the source programs are transformed too much, which is not appropriated for a source-to-source compiler, although the separation of declaration and code makes it more easier to analyze the program. For example, in CIL, local variables in inner scopes are pulled to function scope with variable renaming like this:

[1]This may not be implemented in August 2008

```
int main() {
  int x = 6;
  {
    int x = 7;
  }
  return x;
}

int main() {
  int x__0;
  int x__1;
  {
    x__0 = 6;
    x__1 = 7
  }
  return x__0;
}
```

Impact of block scope on analyses of PIPS ? When propagating transformers or preconditions, we must take into account the block scope of variables, and pay attention to the different variables that have the same user name when prettyprinting results of analyses.

1.4 Current naming mechanism

An entity name contains the name of the object, concatenated to a prefix string and a special separator `MODULE_SEP_STRING` (":"). The prefix string is the name of the package defining the scope of the object which can be `"TOP-LEVEL"` (`TOP_LEVEL_MODULE_NAME`) or a module name.

In Fortran 77, the intrinsic `ABS` has internal name `"TOP-LEVEL:ABS"` while variable `INIT` of module `FOO` has `"FOO:INIT"` as internal name.

There are also special prefixes to distinguish between a main program, a common, a block data, an identifier or a label. For instance, `MAIN_PREFIX` ("%"), `LABEL_PREFIX` ("@"). result in

```
TOP-LEVEL:%MAIN
TOP-LEVEL:@LAB
```

Here are the characters corresponding to special separators (attention, $ can be used in the identifier name):

```
MODULE_SEP_STRING ":"
FILE_SEP_STRING "%" (compilation unit names are suffixed by "!")
BLOCK_SEP_STRING "~" (redefined as "‘")
```

So the entity name can be

- `TOP-LEVEL:name`
- or `[file%][module:][block~]name` which can be one of the following:

```
    FILE%name
    MODULE:name
    FILE%MODULE:name
    MODULE:BLOCK~name
    FILE%MODULE:BLOCK~name
```

In the current implementation, `FILE%` is (incorrectly) replaced by the compilation unit name, which is not sufficient to eliminate all anme conflicts.

The name of a label is MODULE:@label because it has function scope.

In addition, to distinguish a structure, an union or an enumerator that has the same name as other program variables, we have to add special constant characters such as `STRUCT_PREFIX, UNION_PREFIX, ENUM_PREFIX` to the name prefix. We also have to keep the name of the structure and the union in the global name of its members in order to distinguish these members with other program variables, and so a `MEMBER_SEP_STRING` is needed. It is not necessary for the enum member, because the name of a variable must be different from the name of an enumerator member[2]. A prefix for typedef `TYPEDEF_PREFIX` is necessary to distinguish a defined name and to regenerate code.

```
STRUCT_PREFIX #
UNION_PREFIX *
ENUM_PREFIX ?
TYPEDEF_PREFIX $

MEMBER_SEP_STRING ^
```

All prefixes and separators are defined in `ri-util-local.h`. They are defined twice as string and as characters. Hence they are often assumed to be of length one by PIPS programmers.

Examples:

```
struct node {};            [file%][module:][block~]#node
struct key {int node};     [file%][module:][block~]key^node
typedef int node;          [file%][module:][block~]$node
int key;                   [file%][module:][block~]key
union key2 {int node};     [file%][module:][block~]*key2
                           [file%][module:][block~]key2^node
enum hue {toto,tata};      [file%][module:][block~]?hue
                           [file%][module:][block~]toto
int hue;                   [file%][module:][block~]hue
```

A function formal parameter in a function definition does not contain any block information. A function formal parameter in a function declaration is named with a special dummy module name.

Note: C distinguishes between uppercases and lowercases while Fortran 77 considers all characters to be upper case, except in string constants.

The scope of struct, union, enum and typedef is the current compilation unit (current source file), whose name contains a special character, defined as `FILE_SEP_STRING`. Struct and union share the same space. A struct and a

[2]Enumerator members are represented as nullary functions with symbolic values, like Fortran parameters.

union cannot have the same name within one scope, which may make the above distinction between unions and structures redundant.

Chapter 2

Storage

The storage class determines the location and lifetime of the storage associated with a variable.

2.1 External variables

External variables are defined outside any function, and are thus potentially available to any function that declares it `extern`. Any function may access an external variable by referring to it by name, if the name has been declared somehow. If an external variable is to be referred to before it is defined, or if it is defined in a different source file, then an `extern` declaration is mandatory.

```
Name prefix = TOP-LEVEL:
Storage = ram
Ram_function = TOP-LEVEL-ENTITY
Ram_section = TOP-LEVEL area
```

In Fortran, the scope of a variable is the module, or in other words, a global variable is always associated to the list of entities of a module, which facilitates code regeneration.

In order not to allocate external variables several times and to prettyprint their declarations properly, they are kept in the `externs` field of the `code` data structure.

In the old versions of C, we can declare (extern is not a real declaration, since it doesn't allocate memory per se) the same variable in multiple places (files) with no problem at link time. This is probably what we had in mind when we designed this part of the RI. This is not working anymore with more recent versions of C. As it produces linking error, multiple declarations although allowed by some explicit arguments to the compilers (like `-z muldefs` in GCC) are not clean programmin practice. Hence there must be some way to know the difference between the declarations of external variable and global variables and how to regenerate the declaration related to external variables, such as:

- in `file1.c`:

  ```
  int m;
  void func1 { }
  ```

- in `file2.c`:

  ```
  int i;
  extern int m;
  void func2 { }
  ```

  in `file3.c`:

  ```
  void func3() {
    extern int m;
  }
  ```

Multiple declarations are allowed in a compilation unit only, but multiple initializations are forbidden even when they are compatible.

A source file can be considered as a module, and in the first case, the entity *m* can be added to the list of entities associated to this source file. They are named `"TOP-LEVEL:m"`. In the second and third cases there should not be any memory allocation for entity *m*. The real problem lies to differentiate between entity *i* and entity *m* for second case and hence allocate memory for `i` but not for `m`. We need to have some information in the internal representation: it is carried by the `externs` field of `code`.

2.2 Static variables

2.2.1 Internal static variables

```
int foo() {
  ...
  {
     static int i;
     ...
  }
}
```

```
Name = foo!:0`11`i, where 0`11` is just an example of block numbering
Storage = ram
Ram_function = current module or compilation unit, "foo!"
Ram_section = *STATIC* area of current module, "foo:*STATIC*"
```

The outermost block can be omitted, since in fact it is considered as the current module. It is not necessary to generate an entity for each block, we only need to number the blocks. The offset of a variable is computed from the declarations of variables in the same block. So variables with the same offset but in different blocks are different.

2.2.2 External static variables

An external static variable is defined outside of any function, and is known within the remainder of the source file in which it is declared, but not in any other file. The source file is called a compilation unit and a pseudo-function is associated to it.

In file `foo.c`

```
static int i = 0;
int f() {
}
```

we have a corresponding entity to this external static variable:

```
Name = foo!:i
Storage = ram
Ram_function = source file, "TOP-LEVEL:foo!"
Ram_section = *STATIC* area of source file , "foo!:*SSTATIC*"
```

In addition, other entities are generated for the source file and the `*STATIC*` area of this source file.

```
Source file (or compilation unit)
   name = TOP-LEVEL:source_file_name, "TOP-LEVEL:foo!"
   type = functional (parameters = NIL, result = void)
   storage = rom

Area
   name = source_file_name:*STATIC*, "foo!:*SSTATIC*"
   type = area
   storage = rom (as for all area entities)
```

The prettyprint of the external static variable is based on the list of entities associated to the source file entity. The position of the variable declaration in the source file is the position in the declaration field of the `code` data structure.

2.3 Automatic variables

They are handled like dynamic local variables in Fortran [?]. The outermost block cannot be omitted to handle conflicts with formal parameter names.

```
Name prefix =  [file%]module:[block~]
Storage = ram
Ram_function = current module
```

If the size in bytes of the object is known at compile time, we use:

```
Ram_section = *DYNAMIC* area of current module
```

Else we use

```
Ram_section = *STACK* area of current module
```

as in Fortran for varying size arrays that are not formal parameters.. The keyword `auto` is kept in variable qualifiers, presented in 3.7.

2.4 Formal variables

If they appera in a function definition, they are represented as formal variables in Fortran [?].

```
Name prefix = [file%]module:
Storage = formal
```

Note that no block information appears.

If they appear in function declaration, aka function signatures, aka function types, their name prefix is synthesized to avoid any conflict (see `DUMMY_PARAMETER_PREFIX`).

2.5 Register variables

A declaration of an identifier for an object with storage-class specifier `register` suggests that access to the object be as fast as possible and the address of any part of an object declared with register cannot be computed. A `register` declaration can only be applied to automatic variables and to the formal parameters of a function.

```
f(c,n)
register int c,n;
{
  register int i;
}
```

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

Since a formal variable can be declared with register, creating another type of storage such as `register` to store this information does not work and is not compatible with the pre-existing data structuress. Furthermore, for a source-to-source compiler, this information is not important, it is only used to regenerate the source code. So we only need to add this information some where in the type structure. A solution is presented in 3.7, which also deals with `const, volatile` and `restrict`.

Chapter 3

Type

The type system of C is much more extended than Fortran 77's which makes the backward compatibility difficult to ensure. Beside some usual data types such as int, float and char, C also has enumerated type and derived types such as array, structure, union, function and pointer. Usually, these kinds of type are added to `type`:

```
type += basic + array + pointer + struct + union + enum
array = type x dimensions
basic = int + float + ...
```

However, since PIPS Newgen internal representation has been designed for Fortran where the main data structure is array, the Newgen structure `variable` (`variable = basic x dimensions`), used to represent Fortran scalar and array variables, must be handled compatibly. Here are different possible solutions to deal with this:

1. Try to replace all the functions related to `variable` by macros, then `variable` is no more used in the new version.

   ```
   type += basic + array + pointer + struct + union + enum
   array = type x dimensions
   basic = int + float + ...
   ```

2. Keep `variable` and `array` in parallel

   ```
   type += basic + array + variable + pointer + struct + union + enum
   array = type x dimensions
   variable = basic x dimensions
   basic = int + float + ...
   ```

3. To avoid the modifications related to `variable`, which is expensive, the new types can be added to `basic`. This method is called *array-oriented*. The Newgen data structure `basic` is modified to enable the recursion among array, pointer, structure, ...

   ```
   type = statement + area + variable + functional + void + ...
   variable = basic x dimensions
   basic = int + float + logical + string + pointer + ...
   ```

However, in this method, the traversal is not always direct, i.e to access a pointer, we have to go through `variable`, ... which may create bugs with malloc, free and it is not easy for debugging.

The third solution is chosen because it requires less modifications in the actual internal representation. However its main impediment is that functional `typedef` appears as `variable` at firat, although the placement of `typedef` in the `basic` or `type` data structures does not appear clearly in the above dicussion. In particular, unlike `pointer`, `typedef` does not require the dimension field provided by `variable`.

3.1 Basic types

3.1.1 Integer type

C has different kinds of predefined integer type : int, signed int, unsigned int, short, long, long long, ... and char, signed char, unsigned char. Since `int x` and `signed int x` declarations are implementation-defined (which is found in the file stdio.h but not in SPEC 2000), they should be distinguished.

Furthermore, new types are added such as `intptr_t`, `int32_t`,...

Currently, PIPS does not have a proper mechanism to adapt to a particular architecture. At best, `#define` are used. They define a 32 bit architecture: pointers are assumed stored in 32 bits like long int.

There are different solutions to represent all this information:

1. A compact representation that only uses the basic `int` and gives different values to each kind of type:

```
char                   = 1
short_int              = 2
int                    = 4
long_int               = 6
long_long_int          = 8

unsigned_char          = 11
unsigned_short_int     = 12
unsigned_int           = 14
unsigned_long_int      = 16
unsigned_long_long_int = 18

signed_char            = 21
signed_short_int       = 22
signed_int             = 24
signed_long_int        = 26
signed_long_long_int   = 28
```

   We could use `mod(int,10)` to know the basic size and `div(int,10)` to know if the variable is unsigned, signed or not. However, this not fully compatible with the Fortran version [?] where the value of int is the number of bytes required to store one scalar object of this type. And this does not hold for `long_int`.

2. A less compact but simpler solution is to represent three different cases: unsigned, signed or not.

   `basic += int:int + signed:int + unsigned:int`

   In each case, different values are associated to short int, int, long int or long long int.

   ```
   short_int              = 1
   int                    = 2
   long_int               = 4
   long_long_int          = 8
   ```

The first solution is currently implemented. The numbering scheme can be checked in `ri-util`prettyprint.c/ and in `cyacc.y`. See also function `SizeOfarray()`.

It is not clear that merging signed and unsigned types is a good idea, especially in another data structure, `constant`.

3.1.2 Character type

1. A character variable is in fact an integer variable, so it can be associated to the basic `int`, as in the first solution of 3.1.1.

   ```
   unsigned_char          = 11
   char                   = 21
   signed_char            =  1
   ```

2. It may also be better to treat character independently.

   `basic += char:int`

   where the value of `int` is

   ```
   unsigned_char          = 11
   char                   = 21
   signed_char            = 31
   ```

3. Ambiguity between `string` and array of characters ? `string` is not used for C? Impact on semantic analysis, which does not handle arrays but handles strings optionnally?

The first solution is chosen and the basic type `string` is not used for C internal representation.

3.1.3 Bit type

We have to add a basic type `bit` to represent the integral bit fields occurred in a structure declaration.

`basic += bit:int`

```
int a:1
unsigned b:2
signed c:3
```

3.1.4 Boolean

A special type, `_Bool` is used by gcc, according to ISO standard. This type is mapped on the logical basic.

3.1.5 Varying argument lists

The gcc compiler uses a special builtin type, `va_list`. It is defined like a typedef, with not much information beyond what is needed for the prettyprinter.

3.2 Arrays

As in Fortran, we have to represent fixed size array and varying size arrays, arrays (and typedef) which are sized by expressions evaluated dynamically.

We also have to represent implictly sized arrays, whose sizes are implied by their initial values but should not be given in the declaration itself.

3.3 Pointers

Pointer can point to any variable: to a scalar variable, to an array variable, to a function, ... To represent an array of pointers and to keep the initial internal representation, a new type `pointer` is added to `basic`.

```
   basic +=  pointer:type
```

3.3.1 Pointer to integers

```
int *p1
name = p1
type = variable
   dimension = NIL
   basic = pointer
         pointer of type variable
             basic = int
             dimension = NIL.
```

with this dynamic allocation:

```
p = malloc(5*sizeof(int));
```

to store the size of the corresponding memory zone, the dimension of the pointer can be changed to :

```
int *p1
name = p1
type = variable
   dimension = NIL
   basic = pointer
         pointer of type variable
             basic = int
             dimension =5.
```

But how about most cases, as in Section 3.3.2?

Anyway, the dynamic analysis of pointers is not part of the parser. Calls to `malloc()` are not taken into account in the parser to do any memory sizing. The size of a pointer is either 32 or 64 bits, but only the 32 bit architecture is implemented (see `DEFAULT_POINTER_TYPE_SIZE` in `ri-util-local.h`, but this should be improved to chose the architecture dynamically). The first approach is used.

3.3.2 Pointer to pointer

```
char **p;
p = malloc(n*sizeof(char *));
for (i=0;i<n;i++)
    p[i] = malloc((i+1)*sizeof(char));
```

Initially, with the declaration, we have :

```
name = p
type = variable
   dimension = NIL
   basic = pointer
         pointer of type variable
             basic = pointer of type variable (basic = int/char, dimension
             = NIL)
             dimension = NIL.
```

After the first allocation, p points to an array of n pointers (attention to the scope of n):

```
             basic = pointer of type variable (basic = int/char, dimension
             = NIL)
             dimension = n.
```

How to know the memory size pointed by this each pointer? And the size is i-related ...? p[i] represents an entity or p only? The size of p[i] is lost There is a particular case where (i+1) is replaced by m (arrays of same size), we can keep this information in dimension of the second pointer.

This is left aside for future pointer analyses.

3.3.3 Pointer to an array

Declaration **int** (∗p2)[13] defines p2 as a pointer to an array of 13 integers:

```
name = p2
type = variable
   dimension = NIL
   basic = pointer
         pointer of type variable
             basic = int
             dimension = [13].
```

3.3.4 Array of pointers

int ∗a[13] is an array of 13 pointers to integers

```
name = a
type = variable
    dimension = [13]
    basic = pointer
         pointer of type variable
             basic = int
             dimension = NIL.
```

3.3.5 Function returning a pointer

char ∗f(n), f is a function that returns a pointer to a character string

```
name = f
type = functional
      result = variable
          dimension = NIL
          basic = pointer,
              pointer of type variable
                 basic = char
                 dimension = NIL.
```

3.3.6 Pointer to a function

int (∗p)()

```
name = p
type = variable
      dimension = NIL
      basic = pointer
           pointer of type functional
                parameters = undefined
                result = int.
```

Note: undefined is supposed to be avoided in PIPS internal data structure. The gen_defined_p() predicate should always return true. Unknwon or unspecified would be better. Is **void** ending up with one parameter of type **void**, different from NIL? See the prettyprinter source code?

3.4 Structure, Union and Enumerated Types

The common point between a structure, an union or an enumerated type is that each of these types has a name and a list of members. In addition, these members can be represented as entities, because they have name, type and eventually initial value. There are two possibilities to represent these new derived types:

1. Each type is represented separately:

   ```
   type +=  struct:entity* + union:entity* + enum:entity*
   ```

2. They are grouped into a composed type

   `type +=  composed`

   `composed =  members:entity* x kind`

   `kind = struct:unit + union:unit + enum:unit`

The first solution is chosen because it is simpler and more direct when we want to access a special type.

In addition, in order to be homogeneous with `pointer`, these new types can be added to `basic`:
`basic +=  struct:entity* + union:entity* + enum:entity*`
but adding them to `type` would make the traversal much shorter. We do not have to pass through `variable` each time we want to refer to an entity of struct/union/enum type.

The storage class of the struct/union/enum entities and their members is `rom`. The `initial value` of a member entity can be used (or is used?) to represent the offset in bytes of the member in the struct.

All the above discussions talk about the entity related to the struct/union/enum declaration (this template about the shape of a structure create no storage) such as

```
struct key {
  int tab[3];
  int keycount;
};
```

This entity `key` is of type `struct` and is (implictly) associated to a list of members: "#key", "key^tab", "key^keycount". Note that the special character # is not repeated in the field names.

In the declaration like **struct** key var = {{1,2,3}, 3}, the entity `var` is of type **struct** key and there are two possibilities to represent its type:

1. Associate it directly to the entity `key`
2. Associate it to a composed type which contains the name of the structure (`key`) and the list of members, etc. This solution is redundant because we have to store the same information for each variable of type **struct** key.

So to represent a variable whose type is struct/union/enum, we add to `basic` the `derived` type to point to these new types.
`basic += derived:entity`
The initial value of `var` in this case is a list of lists, which is not representable actually in Newgen. But since we know statically the size of the array `tab` in the **struct** key, we could represent this value as a normal list {1,2,3,3} and the information can be extracted when needed. A new psesudo-operator, BRACE_INTRINSIC, is added to be able to represent exactly the list of lists of expressions.

Structures, unions and enumrations that are not named receive a default internal name (see DUMMY_STRUCT_PREFIX, DUMMY_UNION_PREFIX and DUMMY_ENUM_PREFIX in ri-util-local.h).

Whithin one scope, a named structure has a unique name, even if it is defined within another structure or union.

3.4.1 Structure declaration

```
struct key {
  char *keyword;
  int keycount;
};
```

```
name = key
type = struct

     name = keyword
     type = variable
          dimension = NIL
          basic =  pointer of type variable
                  basic = char
                  dimension = NIL
     storage = rom
     initial = 8

     name = keycount
     type = variable
          dimension = NIL
          basic = int
     storage = rom
     initial = 8
```

Nga's comments: the traversal is much shorter if `pointer, array, basic` are added to `type` as in the other solution (which is more logic ...but there were too few type constructor in Fortran 77 to anticipate correctly the needs of C).

```
     name = keyword
     type = pointer of type basic = int

     name = keycount
     type = basic = int
```

3.4.2 Pointer to structures

```
  struct key *p;
```

```
name = p
type = variable
    dimension = NIL
    basic =  pointer of type variable
          basic = derived = entity key
          dimension = NIL
```

3.4.3 Array of structures

```
  struct key keytab[10];
```

```
name = keytab
type = variable
     dimension = [0:9]
     basic = derived = entity key
```

3.4.4 Self-referential structure (recursive data structure)

```
struct node {
  char word[10];
  struct node * next;
};
```

```
name = node
type = struct

      name = word
      type = variable
          basic = char
          dimension = [0:9]

      name = next
      type = variable
           dimension = NIL
           basic = pointer of type variable
                  basic = derived = entity node
                  dimension = NIL
```

3.5 Typedef

```
typedef char *STRING;
typedef int A[2][3];
typedef int (*PFI)();
typedef struct {} TREE,*TREEPTR;
typedef int f(char);
```

`STRING, A, PFI, TREE, TREEPTR` are entities with:

- Global name = `TYPEDEF_PREFIX` + local name. The `TYPEDEF_PREFIX` is used to regenerate code.
- Storage = rom
- Type = type which is named;
- Initial value = could be the type wich is named, but the ype field is alreay used for this

It is more logic if the initial value of a typedef entity is `type`. But in this case, we have to modify the `value` Newgen structure, so it is better to put the type directly in the `type` of the entity. Too bad for functional typedefs whose type is `variable` at first look. The dimensions field is not useful either.

To represent variables whose type is a `typedef` entity, we add to `basic` the `typedef` structure.

```
basic += typedef:entity
```

```
typedef struct key {...} key;
key k1;
struct key k2;
```

```
name = STRUCT_PREFIXkey
type = struct
storage = rom

name = TYPEDEF_PREFIXkey
type = variable
     dimension = NIL
     basic = derived = entity STRUCT_PREFIX:key
storage = rom

name = k1
type = variable
     dimension = NIL
     basic = typedef = entity TYPEDEF_PREFIXkey

name = k2
type = variable
     dimension = NIL
     basic = derived = entity STRUCT_PREFIX:key
```

3.6 Functional Type

Functional types have already been treated for Fortran, except for some (small) details.

```
type = functional + ...
functional = parameters:parameter* x result:type
parameter = type x mode
```

- For **extern int** f(**void**);, `parameters` is a list of one element of type **void**. The number of parameters is mislseading in this case.
- How to represent **extern int** f(); (the same for Fortran with `EXTERNAL F` ?) `parameters_undefined` creates consistency/updating problems ?
- We can add a Newgen structure for the function qualifier `inline` but it was not necessary for during the first parser development because no inlined functions appear in SPEC 2000. It is snot clear the inline should be part of the type since two functions can have the same signature and

hence the same type, but they do not have to be both inlined. Should it be a type qualifier? But qualifier a are restricted to variable types...

C supports varying argument lists and a special keyword, va_arg(), whose two arguments are a variable and a type. As for `sizeof` which takes either a type or a variable as argument, this is not representable with a standard `call`..

3.7 Type qualifiers : Const, Restrict, Volatile

```
const int *p;
void func(const a);
void h(int * const restrict p);
```

Although only a small percent of variables are declared with these qualifiers and it is expensive, we choose to create a new Newgen structure for them. Attempts to put these information in existing Newgen structures, such as a `rom` storage for local variables qualified with `const`, `shared` field of `ram` for `restrict`ed local variables are not successful because they do not handle all possible cases. For example, `rom` cannot be used for a formal variable declared `const` (which can be found in SPEC 2000 benchmarks), `shared` cannot be used for a formal variable declared `restrict`.

We can add a new field `qualifiers` for `type`, at the `variable` level. This qualifier can also contain the `register` and `auto` cases.

```
variable = basic x dimensions x qualifiers:qualifier*
qualifier = const:unit+restrict:unit+volatile:unit+register:unit+auto:unit
```

There are about 36 make_variable in PIPS source code to modify.

3.8 Conclusion

Some important information is carried by the entity name and not by the type data structure. This is the case for struct, union and enum. As a result, numerous characters become reserved and it is not always possible to use the external operator name as internal PIPS operator name. This old assumption of PIPS internal representation not longer holds.

Chapter 4

Expressions

Expressions in Fortran PIPS must be extended to handle new kinds of expressions found in the C language. Here are the Fortan 77 data structures used to represent expressions:

```
expression = syntax x normalized
syntax = reference + range + call
reference = variable:entity x indices:expression*
```

New kinds of syntax are added to handle 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 a function.

```
syntax += cast + sizeofexpression + subscript + application
```

4.1 Cast

```
cast = type x expression
```

Cast cannot be represented easily as intrinsics as in Fortran, because their number is unbounded due to the typedef mechanism.

4.2 Sizeof

```
sizeofexpression = type + expression
```

4.3 Subscript

In C, pointer expressions can be subscripted, not arrays only.

```
subscript = array:expression x indices:expression*
```

4.4 Application

C is more flexible than Fortran about functional pointers. Such pointers can be stored in data structures instead of being restricted to function call.

`application = function:expression x arguments:expression*`

For example, we have such a function call in C:

```
(*ctx->Driver.RendererString)()
```

4.5 Special calls

To represent C construct, specific calls are used

4.5.1 Member references

There are different solutions:

1. A special call expression, FIELD_MEMBER_CALL(exp1, exp2), to handle for example str [1]. fld [2]
2. `reference = variable:entity x indices:expression x offset:entity*`
   This is not sufficient because we can have in C99 foo().x where foo is a function returning a struct... That means that we need to extract fields from non l-values too.

   More explaination here !

Distinguish between `.` and `->` ? The last one can be represented by the first one and `*`. Disadvantages for type checking, program analyses, transformations?

See `ri-util-local.h`:

```
#define FIELD_OPERATOR_NAME              "."
#define POINT_TO_OPERATOR_NAME           "->"
#define DEREFERENCING_OPERATOR_NAME      "*indirection"
#define ADDRESS_OF_OPERATOR_NAME          "__address-of__"       // &

#define COMMA_OPERATOR_NAME              ","
```

4.5.2 Address of, value of

Special functional intrinsics are used for `&` and `*`.

4.5.3 Comma operator - list of expressions

f(a, (t=3,t+2),c)

Special call expression COMMA_OPERATOR: n-ary or binary call ?

The same holds for a=b=c=d.

4.5.4 Conditional expression

e1 ? e2 : e3

Special call expression based on CONDITIONAL_OPERATOR. This is the only operastor with 3 arguments.

Chapter 5

Control Flow

5.1 Module Code

```
code = declarations:entity* x decls_text:string x initializations:sequence
```

As discussed in section 2.1, external variables that are declared outside a module can be pretty-printed if the source file is considered as a module, with the declarations list and the decls_text string. This is implemented in the data structure `code` but the field `decl_text` does not seem to be initialized by the parser and is not used by the prettyprinter.

However, this does not let us know if a global variable has been declared `extern` or not. For instance, the two declarations:

```
extern int i;
```

et

```
int i;
```

outside of a function definition result both in the declaration of a global variable, top—level:i, within a compilation unit. To remember in which C file the keyword `extern` appears, a new field, `externs` is added to `code`. Note that the initialization and/or the declaration without extern can appear only once within an application. However, the sequence:

```
extern int i;
...
extern int i;
...
int i;
```

is legal, probably to simplify the design of include files.

The field `code_declarations` contains all variables declared within the function definition. Iternal PIPS entities such as memory areas are declared first, followed by formal parameters. Other variables declared within statement blocks are also listed here. So most variables appear in a `statement_declarations` field and in the `code_declarations` field of the module.

For a compilation unit, which does not really have a body, the `code_declarations` field can be used.

The field `initializations` is used to represent DATA statements in Fortran. In C, the initializer for a scalar variable is a single expression, for objects that have aggregate or union types is an initializer list, which can be not complete and by default, the remaining elements are initialized by zero (for arithmetic type) or null pointer (for pointer type).

There are two possibilities to represent this initialization information:

- Represent this initialization in the initial values of entities. New kinds of value can be added to `value` such as `expression` for scalar variables, aggregate or union types (array, structure, ...). String can be used to regenerate code, list (list of lists is not permitted in Newgen) can be used to have fine preconditions on array elements, structure member, ... But the list length is the array size ?

  Other problem: how to represent `int i = j;` ? The initial value of i is j ?

- Treat this initialization as a special kind of statements like DATA in Fortran.

- since any statement and hence any instruction can include a declaration and since ISO C99 allows mix of declarations and executable statements, represent **int** i =j as **int** i; i = j;.

The first solution is chosen and the field `initial_value` leads to a value containing the initialization expression thru `value_expression`. Sepcial expressions are built for array and structure initializations.

5.2 Statement

The Fortran 77 structure of a statement is:

`statement = label x number x ordering x comments x instruction`

Since in C, a declaration can appear in any block, not only at the beginning of a function, we have to associate variable declarations to blocks. There are two possibilities to perform this in the current internal representation: declaration can be associated to either a statement or a sequence (a block in fact). But since the true and false branches of a conditional statement as well as the body of a loop are not necessarily sequences, declarations in these statements will be lost if they are associated to sequence. So we choose to associate them to a statement, although it may be useless for some elementary statements such as call, test or loop.

Another approach would have been to consider declarations as statements (see discussion about initializations in previous section) and/or to use a NOP instruction such as Fortran `CONTINUE` to carry the declarations. This let us also preserve more comments than other approaches.

Note: we have to pay attention in coding in PIPS that we cannot refer to declarations once we have already reach the instruction because there is no upward pointer from instruction to statement.

So we suggests to modify `statement` in the following way:

`statement += declarations:entity* x decls_text:string`

instead of creating a new instruction, which could be called declaration. When a new instruction is needed, PIPS designers often use a new intrinsic rather than a new instruction because fewer PIPS source code modifications are necessary.

Member `declarations` contains a list of entities in the statement scope (all kinds of entities such as intrinsics, ... or only those that are effectively declared). Member `decls_text` could be used to regenerate source code (as for module code) if the parser initializes it.

This does not specify which statements carry the declarations. A first implementation, based on C89, used block statements only to carry declarations. All other statements had to have an empty declaration list. This is too restrictive with respect to C99, which allow declarations to appear anywhere among executable statements, and with respect to source-to-source constraints. By putting all declarations of a block in one statement, individual comments and line numbers are lost.

A second implementation is based on C nop, ";", i.e. Fortran CONTINUE. This statement has no effect and can be ignored, but for the possible initial values, by PIPS analyses. PIPS transformations must preserve it.

5.3 Instruction

The `instruction` for Fortran 77 is a union:

`sequence + test + loop + whileloop + goto + call + unstructured`

Other kinds of instructions in C such as `switch`, `for`, ... can be added to `instruction` or represented using the existing structures declared above.

In addition, a statement in C can be any expression, not only call expression, so we have to add `expression` to `instruction`. However, to make PIPS backward compatible, we try to create a call statement for each call expression. Expression statement is only used for special cases, such as cast expression.

Not data structure is added for the `switch` construct.

A new field is added to `whileloop` to indicate if the condition is evaluated before the body or after the body.

To simplify the prettyprinter, a `forloop` is added.

In each case, a decision must be made between the requirements of the prettyprinter for the source-to-source use of PIPS (the more structures the better) and the code complexity of the analyses (the fewer the structures the better).

5.3.1 Switch

There are two solutions: we can add a new kind of instruction `multitest` (the keyword `switch` cannot be used) or we can represent a switch through `if` and `goto` statements.

First solution:

`instruction += multitest multitest = controller:expression x body:statement`

The second solution is choosen:

`case` and `default` are two kinds of labeled statements, which can be treated as `goto`. Their associated labels are entities which must be unique. The initial values of these entities are constant expressions. We can add them to the declarations list of the switch statement in order to match them to the actual switch? The entity local name is the constant expression of case, and a special name for default.

The `break` statement can be treated as a `goto`.

```
switch (c) {
  case 1:
    s1;
  case 2:
    s2;
    break;
  default:
    sd;
}
```

```
  if (c==1) goto switch_xxx_case_1;
  if (c==2) goto switch_xxx_case_2;
  goto switch_xxx_default;
switch_xxx_case_1: ;
  s1;
switch_xxx_case_2: ;
  s2;
  goto switch_exit_xxx;
switch_xxx_default: ;
  sd;
switch_exit_xxx:
```

Note that this solution assume that the `default` case always appears and that it appears after all other cases. Since this is not always true, a direct syntactic translation is not possible and some post-processing is required.

How about code regeneration? It might be easier to regenerate nice code for structured if the internal control structure were based on:

```
if(c!=1)
  s1;
  goto continue1;
else if(c!=2)
continue_1:
  s2;
  goto continue2;
else if(...)
continue_2:
...
else /* default case */
continue_n:
  sd;
```

If `s1` and `s2` end with a `break`, the `goto continuex` are not reachable and the if-else-if structure is preserved.

Some pattern-matching could be tried on the resulting `untructured` and/or the `unspaghettify` option of the controlizer might be able to put things back nicely when it is possible...

5.3.2 While Loop

The "**while** (expression) statement" and "**do** statement **while** (expression)" in C can be represented together by adding a new field to distinguish if the evaluation of the controlling expression takes place before or after each execution of the loop body.

```
whileloop = condition:expression x body:statement x label:entity x evaluation
evaluation = before:unit + after:unit
```

To maximise the readability, `evaluation:bool` is not used here, as in other cases in the internal representation (`mode`, `action`, ...).

5.3.3 For Loop

There is always a trade-off between regrouping different loop structures and separating them. The first case makes program analyses more compact, with less code to write but it makes pretty-printing original code difficult. It is reverse for the second case.

We have different possibilities to consider:

1. Represent for loop as while loop
2. Represent for loop as loop (do loop in Fortran), but it is not always possible. Loops could be pretty-printed as for loops but for loops cannot always be represented as loops (several loop indexes).
3. In order to keep the initial program structures, we can treat the for loop separately from the other loops.

```
forloop = initialization:expression x condition:expression
          x incrementation:expression x body:statement
```

The ISO C standard [?] states that the initialization of a for loop may contain variable declaration, which is not the case for [?]. Such declarations could be moved in the statement containing the for, but for the time being the parser does not accept such declarations.

Initializations could be put into the very statement containing the `for` instruction and naming the block in a way to prettyprint correctly the declaration into the `for`.

5.3.4 Null statement

Null statement in C, `";"`, is treated as CONTINUE statement in Fortran. We only need to make the differences at the prettyprinter level.

5.3.5 Return statement

Return statement in C (return; or return (exp);) is treated as RETURN statement in Fortran, which is considered as nullary operator but can have 0 or 1 argument, such as the cases of STOP and PAUSE statements.

5.3.6 Break, Continue, Exit, Jump, Interruption

We can add new kinds of instruction to handle break and continue:
`instruction += break + continue`

CIL says that leaving `break` and `continue` as they are makes transformations such as code motion easier? The semantic difference between `continue` in Fortran and `continue` in C?

Another solution is to treat `break` and `continue` as `goto`, which is choosen as solution for the moment. How about code regeneration ? Some semantics can be added into the generated label names.

PIPS analyses are preserved, but the prettyprinter of unstructured must be improved to pattern-match `continue` and `break`.

Chapter 6

Memory Effects

6.1 Pointers

Pointers are a key part of C since parameters are passed by value and since recursive data structures require pointers. In the Fotran implementation, the memory effects are represented by the `effect` data structure and its fields, and by lists of such data structures:

```
effects = effects:effect* ;
effects_classes = classes:effects* ;
effect = cell x action x approximation x descriptor ;
cell = reference + preference ;
action = read:unit + write:unit ;
approximation = may:unit + must:unit + exact:unit ;
descriptor = convexunion:Psysteme* + convex:Psysteme + none:unit ;
reference = variable:entity x indices:expression* ;
preference = persistant reference ;
```

In this framework, the effect of a statement like `*p=1;` can only be expressed as a reference to a large memory entity, an area or a set of areas, as no information is locally available about `p`.

Effects are abstractions of effective memory effects. For instance `d[i]` can be captured as `d[i]` because the data structure `reference` let us do so, as `d[*]` to obtain a constant effect independent of the current store, or as an array region, with constant reference `d[phi]` and store sensitive descriptor `{phi=i}`.

We need a new abstraction to deal with indirect effects in such a way that constant pointers can be detected and taken advantage of. For instance, a C function incrementing an integer, `void inc(*int p)`, should be analyzed in such a way that the call site effect of `inc(&i)` can be derived exactly.

We need to know that `inc` performs an indirect write through `p`. This is not a great new abstraction for pointers. It is the minimum needed in a first phase.

We need to encode `p->in`, which is equivalent to `(*p).in`. Assuming that `in` is the third field of the pointed structure, this could be rewritten `(*p)[3]`.

We also need to encode `(*p)[i][j][k]` which is an array access to a formal array parameter. Pointer `p` may be a pointer to a dynamicically typed variable such as `double x[n][m]`, where `n` and `m` are formal parameters too.

We do not know if we need to keep track of pointer arrays, as in `*(p[3]):` our region framework can handle it effortlessly, but what can we do with information about pointer arrays? If our lattice is too simple, we won't be able to make a difference between `*(p[3])` and`(*p)[3]`. However, since we transform accesses to structure fields into indexing (Section 6.2), we might be interested in the post index form to keep or to retrieve information about data structures containing pointers on functions.

What is the new information we want to add? How can we add this new information in the current PIPS data structure? Many solutions are possible:

1. add nothing in the data structures and keep the information at the source code level: have different variables for direct and indirect effects;
2. define a new level of effects, like `g_effect = direct:effect + indirect:effect + ...` or, at the list level, `g_effects = direct:effects + indirect:effects + ...`
3. add new kinds of actions such as `indirect_read` or `indirect_write`;
4. re-use the `descriptor` field; this does seem to make less sense than the previous solution;
5. define a brand new pointer effect data structure, extending solution 2;
6. add a new effect field, `addressing`, in `effect`;
7. represent pointer accesses using additional effect reference dimensions ; for instance an effect on `*p` could be represented as `p[0]`;

How do we chose given the current implementation and our goals?

Separating indirect effects and direct effects at the variable level would require a huge reworking of the current code since functions should return a structure containing several lists instead of a single list. This would prevent us from giving a meaning to the effect order in an order list: it is useful to know that a write occurs before a read.

This holds whether lists are separated at the variable level (solution 1) or at the data structure level (solution 2).

Adding new actions, Solution 3, does not respect the field semantics and its link with Bernstein's conditions. The number of new actions can be great if we take into account indirect pre- and post-indexation. However, we may not need pre- and post- indexation. And it might make debugging easier with unknown action detection. Unfortunately, with only two actions, people do not use switch and default when there are only two actions. They assume that if it is not the first one, it has to be the second one.

Solution 5 would require a proper survey of most published pointer analyses. It would be nice if the old effect data structure could be mapped onto it to minimize source code modifications via macros. This could also be done later and independently, using simple pointer information gathered with the effect data structure as a starting point for more advanced analyses.

Adding a new field to effect, for instance `addressing` or `addressing_mode`, Solution 6, is nicely orthogonal, let us structure it as we want, and avoid a combination of attribute with the read and write actions. It requires source code modifications for `make_effect`. And the bugs due to a lack of access checking will not be syntactically detectable.

However indirect accesses would have to be reduced to standard accesses before the dependence test can be used and after the semantics analysis has managed to propagate pointer values and equality. The region analysis does not expect any indirect effects. The easiest way out might be to add a new phase using general effects and reducing them all to standard effects, exploiting semantics information. This would require a renaming of effects at the pipsmake and database level to avoid confusion and to force the conversion of resources.

This is the solution which was first chosen. It consisted in adding a new field `addressing_mode` (or `addressing` to avoid underscore in field names?) with three values for pre- and post-indexation:

```
effect = cell x action x approximation x descriptor x addressing;
addressing = index:unit + pre:unit + post:unit
```

All accesses were indexed by default in PIPS internal representation, so `post` and `pre` were equivalent for scalar accesses.

Mode `index` could have been called `direct_indexing` to be more homogeneous, but indexing always occured. So `index` could have been called `direct`.

The mode `preindexing` did not seem to be very useful in the short term.

But it proved to lack accuracy to handle real applications which use indirect accesses at several levels of data structures, and not only at the uppermost level. So, lastly, solution 7 was retained. Table 6.1, which is not limitative, shows that many cases of indirect accesses can be handled this way. Moreover, an obvious advantage of this approach is to unify all effects into effects on arrays which can be handled by following phases.

6.2 Data structures

PIPS was designed primarily for arrays and this shows in the reference data structure, which only support the indexed access mode. Hence, we need to map the offset accesses found with data structure references such as `c.in` onto indexed accesses.

Several possibilities come to mind for data structures if the current effect data structure is to be preserved:

1. to mimic the address computation at the byte level, assuming any field access can be interpreted as an array access of some byte elements; this does not account for bit fields, but those are not frequently used;

2. to rename fields by their ranks and to access them by indexing; this is not correct for a compiler, but the access information is preserved, which is enough for an analyzer; array fields are a natural dimension, inserted at the right place among index dimensions;

3. a variant of the previous possibility is to use field entities (and not solely their names) as effect reference indices;

4. to build new variable names located at the field address, i.e. in equivalence with the whole data structure but not with its other fields; for instance `a.b.c` would be represented by one variable of name `a-b-c` with no name conflict if dash is used inside the new name; array fields are would be

declarations	reference	effects
int a, *p;		
	a	a
	*p	p[0]
int t[N], *p, (*q)[N], *u[N], **v;		
	*t	t[0]
	t[I]	t[I]
	*p	p[0]
	p[I]	p[I]
	(*q)[I]	q[0][I]
	*u[I]	u[I][0]
	*v[I]	v[I][0]
typedef struct {		
int num;		
int tab1[N] ;		
int *tab2; } mys;		
mys a, b[N], *c, **d;		
	a.num	a[num]
	a.tab1[J]	a[tab1][J]
	a.tab2[K]	a[tab2][K]
	b[I].num	b[I][num]
	b[I].tab1[J]	b[I][tab1][J]
	b[I].tab2[K]	b[I][tab2][K]
	c->num	c[0][1]
	c->tab1[J]	c[0][tab1][J]
	c->tab2[K]	c[0][tab2][K]
	d[I]->num	d[I][0][num]
	d[I]->tab1[J]	d[I][0][tab1][J]
	d[I]->tab2[K]	d[I][0][tab2][K]

Table 6.1: Representing effects with additional dimensions

taken care of as with a regular array; for instance `a.b[i].c` would be represented by `a-b-c[i]`;

5. to build new variable names based on the offset values and to define some kind array type reflecting the real increments: an element size is associated to each dimension, so that the impact of any index is exactly reproduced; this might be a cross of the first two solutions; the offset based name may be equivalent to using the global offset as a unique last dimension instead of using one dimension per field and array traversal.

We have to analyze the impact of the choice with respect to effect computation, but also with region and semantics analyses, as well as dependence testing.

Regions are similar to effects and are not impacted. The semantics analyses is not compatible with the first option, for instance because the value of an integer will not be analyzed since four different bytes are involved. The second option requires the semantics analysis to be extended to deal with fixed array elements such as `x[1][2]`, which has been a long pending extension. The third option requires the semantics analysis to be extended to deal with equivalenced variables: currently, the analyses are restricted to non aliased variables.

The dependence test should be able to deal with all four options because it deals with regions, i.e. set of array elements, and with equivalenced variables, i.e. an aliasing test is performed before the array dependence test.

The fourth option requires improvements in memory allocation and in aliasing management. Data structures for aliasing exist because of Fortran, but they are not currently used for C. The scalar analysis of semantics can be used right away to analyze structure fields.

The effect prettyprinter is easier with option 2, unless the naming scheme is exaclty C scheme, i.e. `a.x` is equivalenced with a variable of name `a.x`. Hopefully, a dot in a variable name is not going to create implementation nightmares. But field names may conflict: they should include the structure name and `a.x` would become something like `a.a-x`.

All four schemes are minimum. They are sufficient to represent any constant address with indexing equivalent to a field access expression. And any field expression is represented in only one way.

After testing the second option for a while, we finally changed for the third one in particular because it avoids merging convex effects of different access path types. We have chosen to keep PHI variable numbers equal to their ranks in the effect reference indices list for practical reasons. Functions are provided to convert effects with references containing indices refering to field entities to their rank equivalent versions.

6.3 Unions

Union are more difficult to track. They create an equivalence between two structures located at the same address. This reminds us of Fortran equivalences, extended to data structures.

For instance, `union {int i; double x;} u;` could be interpreted as three variables called verb/u/, /u-i/ and /u-x/ located at the very same address.

Parallelism could be safely detected for arrays of unions and for unions of arrays using existing PIPS technology.

It is not yet clear how this could be extended when unions are part of more complex data structures. Can we still use a naming scheme (option 3 for data structures) and move all index information at the end of the reference? It this still a way to denote precisely the address accessed such that all analyses are based on a correct representation of locations?

If we use a naming scheme (option 3 above), we should be able to compute addresses and ranges for combination of union and struct and to build aliasing information due to unions. To be refined.

Indexing may be more difficult to deal with as its impact on address computation must be precisely reproduced when dealing with equivalenced variables.

If we use the basic byte based representation (Solution 1), union can be handled like structures and lead to automatic parallelization. But difference sets of indices will lead to array linearization.

So we have to find the best trade-off between naming, indexing and linearization for dependence testing when unions are used.

6.4 Point to Operator

The point to operator is syntactic sugar. The expression `a->x` can be replaced by `(*a).x`.

6.5 Address Expressions

Address expressions are very general as base and offset can be provided by function calls and offset by any integer arithmetic expressions. Pointers differences are allowed to compute offsets, making it difficult to identify a base pointer and an offset. Consider for instance `*(p+q-r)`: is it `p` or `q` the offset?

If we could distinguish a base pointer `p` and one offset, the address expression could be rewritten `*(p[offset])`.

But the basis may be returned by a function, as in `f()->x`.

Multiple indirections, as in `a->x->y`, `M[M[a]+x]+y` make the notions of base and offset ludicrous.

So we need a way to express fuzzy address, for instance using a special address value *anywhere* equivalent to the whole memory `M`.

6.6 Indirect Effect Normalization

When pointers value are known, indirect accesses can be converted precisely into direct accesses as in `*(&i)=1`.

Fuzzy pointer values, i.e. sets of pointer values, must be introduced to cover other cases, using if possible PIPS support for aliasing, a.k.a. equivalence in Fortran. The lattice for pointer values may be built with areas, finite sets of areas or special new areas such as the area containing all areas. PIPS uses four areas for each module: the static area for objects of constant addresses, the dynamic area for objects of constant sizes[1], the stack area for objects of unknown size and the heap area for dynamic memory allocation. Currently aliasing is only statically handled in the static and dynamic areas, using constant addresses.

The potentiel levels are:

1. at least three as for any constant propagation lattice, with *anywhere* used as soon as the constant value is unknown;

2. but it could be slightly increased by taking into account the scope of a function and the scope of its compilation unit.

3. or it could include all subsets of the area set since the latest is bounded by the number of functions and compilation units.

4. or...

Note that `free` does not let us infer that two pointers are different because both of them were malloced. Hence the different heap areas[2] are in fact a unique heap.

What aliasing assumptions should we make about formal parameters and global variables? Do we assume constant value for formal pointers and hence no aliasing between them and between local pointers?

How do we implement the lattice? By multiplying the effects in the effect list, with one effect per area possibly accessed? By adding new super areas including several subareas and using equivalence information? By limiting the number of levels in the lattice?

6.7 Putting together pointers, structures, unions and C address computations...

The general case has not been discussed yet as we are driven by the Ter@ops project and its requirements:

1. key information such as array dimensions and loop bounds are parts of structures;

2. array dimensions and hence typing are dynamic; formal arrays are likely to be accessed via pointers as in `(*p)[i]j]`;

3. ...

This does not require mixes of structures, unions and pointers to be dealt with...

[1]They are given constant addresses in this area, but they are only constant with respect to the frame pointer.

[2]A separate heap area is used for each function.

Conclusion

There is no perfect solution, especially before full implementations tell us how much each solution does really cost.

We need to keep a proper balance between information needed to prettyprint source code and unifications which streamlines the code for analyses and transformations.

This document is an evolving document. Information is not always up to date.with respect to PIPS source code, especially the `ri-util`, `c_syntax` and, to a lesser esxtent, `preprocessor` libraries. In `ri-util`, the files `ri-util-local.h`, `prettyprint.c` and `cyacc.y` are especially relevant.

The choices that have been done are in the companion file `ri.tex`

Appendix A

Inital Fortran-oriented internal representation

```
action = read:unit + write:unit ;
approximation = may:unit + must:unit + exact:unit ;
area = size:int x layout:entity* ;
basic = int:int + float:int + logical:int + overloaded:unit + complex:int + string:value ;
call = function:entity x arguments:expression* ;
callees = callees:string* ;
cell = reference + preference ;
code = declarations:entity* x decls_text:string x initializations:sequence ;
constant = int + litteral:unit + call:entity ;
control = statement x predecessors:control* x successors:control* ;
controlmap = persistant statement->control ;
descriptor = convexunion:Psysteme* + convex:Psysteme + none:unit ;
dimension = lower:expression x upper:expression ;
effect = cell x action x approximation x descriptor ;
effects = effects:effect* ;
effects_classes = classes:effects* ;
entity_effects = entity->effects ;
entity_int = entity->int ;
execution = sequential:unit + parallel:unit ;
expression = syntax x normalized ;
formal = function:entity x offset:int ;
functional = parameters:parameter* x result:type ;
instruction = sequence + test + loop + whileloop + goto:statement + call + unstructured ;
loop = index:entity x range x body:statement x label:entity x execution x locals:entity* ;
mode = value:unit + reference:unit ;
normalized = linear:Pvecteur + complex:unit ;
parameter = type x mode ;
persistant_expression_to_effects = persistant expression -> effects ;
persistant_statement_to_control = persistant statement -> persistant control ;
persistant_statement_to_int = persistant statement -> int ;
persistant_statement_to_statement = persistant statement -> persistant statement ;
predicate = system:Psysteme ;
```

```
preference = persistant reference ;
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* ;
statement = label:entity x number:int x ordering:int x comments:string x instruction ;
statement_effects = persistent statement->effects ;
storage = return:entity + ram + formal + rom:unit ;
symbolic = expression x constant ;
syntax = reference + range + call ;
tabulated entity = name:string x type x storage x initial:value ;
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 ;
variable = basic x dimensions:dimension* ;
whileloop = condition:expression x body:statement x label:entity ;
```

Appendix B

Proposed new internal representation (modified structures only)

This information may be outdated. Please check `ri.newgen` for PIPS current data structures. This information was useful initially to estimate the changes in the data structure and the impact on the existing code. It does not seem useful to maintain it now that the C parser is implemented..

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

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

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

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

syntax = reference + range + call + cast + sizeofexpression + subscript + application;

cast = type x expression ;

sizeofexpression = type + expression ;

subscript = array:expression x indices:expression* ;

application = function:expression x arguments:expression* ;

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

```
variable = basic x dimensions:dimension* x qualifiers:qualifier* ;

qualifier = const:unit + restrict:unit + volatile:unit + register:unit + auto:unit;

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

evaluation = before:unit + after:unit ;

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

Bibliography