PIPS: Memory effects of Statements

Fabien Coelho
François Irigoin
Pierre Jouvelot
Rémi Triolet

CRI, M&S, MINES ParisTech

Béatrice Creusillet
Ronan Keryell

HPC Project

March 2, 2016, revision r23065

Contents

Introduction

Each statement reads and writes several memory locations to retrieve stored values and to store new values. Understanding the relationship between statements and memory is one of the many keys to restructure and parallelize programs. Several analyses provide different approximations of the statement effects on memory. This document gathers the internal data structures used to represent memory locations and the effects of statements [1].

1 Imported Domains

```
import entity from "ri.newgen"

   import reference from "ri.newgen"

   import preference from "ri.newgen"

   import expression from "ri.newgen"

   import statement from "ri.newgen"

   External Psysteme
```

2 Representing Memory Locations

A first solution to represent memory locations would be to use dummy memory addresses, potentially one for each memory cell. However analyses results would be equivalent modulo a renaming of dummy addresses, making non-regression tests more dificult. Moreover, analyses results would be difficult to interpret for a human operator, and program transformations would lead to code difficult to read for an end-user. This would not be a problem in a traditional compiler. However, PIPS is a source-to-source compiler, and aims at generating user-readable code, as close as possible to the user input code. The first target language being Fortran 77, this primarily led to represent memory locations as they are represented in the internal program representation, that is to say as symbolic references on program variables. Hence a read of variable `a` is represented as an effect on reference `a`, and a read of array element `t[5]` by an effect on reference `t[5]`. `t[*]` is used to represent a set of unknown locations in array `t` or to obtain a representation independent of the memory store.

However, this representation is highly imprecise to deal with sets of array elements, which is necessary for effective dependence testing and program transformations. Polyhedral sets (also called *array regions* [3][7]) were introduced to refine the memory location representation for arrays. Internally, the `reference` domain is still used, but parameterized by dummy`phi` variables. An additional

[1] Several of the newgen domains declared in this document were originally part of the `ri.tex` file. They have been moved here because, strictly speaking, they do not deal with the internal representation of programs, and because the extension to C programs required the extension of the `cell` domain.

convex descriptor describes the relations between the `phi` variables and the program variables values in the considered memory store.

It has been chosen not to append the descriptor to the memory location representation, but to integrate it in the client domains. This seems more adapted to alias-like analyses for which we intend to use a single descriptor to describe the relationships between the sink and source memory cells representations.

For C programs, memory accesses are not solely represented by objects of the `reference` domain (see `ri.pdf`), in particular when dereferencing pointers or refering to aggregate data structures. Subsection 2.1 presents the choices made to represent memory accesses through pointers and non-recursive data structures. Subsection 2.2 introduces new domains to enable analyses of recursive data structures.

2.1 Pointers, non-recursive structs and unions

Many question arose with the introduction of C as an PIPS input language. For memory location representation, most of them were due to pointers. A discussion of the different issues encountered and the different solutions that were considered can be found in `ri_C.pdf`. As it stretches over several pages, we do not move or reproduce it here, but only present the final option.

It has been chosen to preserve the present represent representation of memory locations as symbolic `reference`s. This has the invaluable advantage of backward compatibilty for the existing analyses. But it also enables the use of existing operators and functions dealing with memory locations represented as references, and thus enables a quicker integration of new functionalities necessary do deal with C data structures.

This is not to the cost of the representation precision. Table 1 shows that memory accesses involving pointers, arrays, structs and mix of them, can be accuratly represented using symbolic references.

Unions are more difficult to track because they create an equivalence between two structures located at the same address. We intend to represent accesses as we would do it for structs, delaying the burden to disambiguate them to client analyses.

2.2 Recursive data structures : GAPs

Recursive data structures such as linked lists or trees are frequent in C programs, and being able to deal with them is essential. However, the `reference` domain is not adapted to the collective representation of sets of recursive data structures elements, because it can only describe fixed length access paths (that is to say the number of indices of the reference must be numerically known and finite).

A quick extension could have been the introduction of a special flag index meaning that up to a certain depth the path is recursive. However, some intended program transformations could not sufficiently benefit from this approach, and we have searched for a new data structure to hold the necessary information.

Experiments on real codes, and the need of a maximal compatibility with existing data structures, led us to chose an extension of Alain Deutsch's Symbolic Access Paths (SAPs) [4], which we call Generic Access Paths or GAPs. GAPs are very similar to SAPs, but for path selectors, which, instead of being limited

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 1: Representing memory locations using references

to variable names and struture field names, can additionally be any subscript expression, thus allowing the representation of elements of arrays of linked lists of arrays of linked lists...

For instance p[i]([0][. next])^(c)[. tab][j] represents the j-th element of the `tab` field of the c-th element of the linked list towards which `p[i]` points.

For backward compatibility reasons, we intend to use GAPs internally only for recursive data structures, but they could also be used for single scalar variables, stack or heap arrays, non-recursive aggregate data structures...

```
Gap = variable:entity x path_selectors

Path_selectors = path_selector*

Path_selector = expression + recursive_selector

Recursive_selector = basis:path_selectors* x
coefficient:expression
```

2.3 Cell domain

The domain `cell` is used to represent memory locations. Instead of the `reference` domain, `preference` can be used to retain a link to the actual program reference. This is mainly used for Fortran programs where all memory locations references are internally represented as objects of domain `reference`. However this does not make much sense for C programs because more complex expressions are often used to refer to memory locations. GAPs have been added for recursive data structures (see refsubsec:gaps for more details).

```
Cell = reference + preference + gap
```

3 Effects of statements and instructions

Each program variable is a unique set of memory locations. Effects can be expressed as effects on these sets. They are called *atomic* effects, because a whole data structure is seen as read or written as soon as one element is read or written. Note that indirect effects due to pointer uses are related to unknown memory locations.

The memory location representation can be refined for arrays. Certain sets of array locations are handled, for instance intervals like `A[I:J]` or even the so-called regular sections or Fortran 90 triplets, like `A[0:N:2]`, which adds a stride to the concept of interval. PIPS is able to handle polyhedral sets, called *array regions*. Extension to non-convex sets, intersections of a lattice and a polyhedron, is under investigation.

Memory effects are not always perfectly known. It is undecidable in the general case. Effects can be labelled as `MAY` if they might happen, `MUST` if they always happen, and `EXACT` if the abstract set used to represent them is equal to the dynamic set of effects. In fact, must effects are not computed because with polyhedral sets, they are either equal to the exact set or to the empty set.

Read and write effects are useful for dependence analyses. However, it is sometimes interesting to see compound statements as black boxes. In this case

we only want to known if the *initial* value of a memory location is *used* by the statement, which is called a `IN` effect, or if the memory location is only used for temporary storage. In the same way, it is important to know if the value left by a statement in a memory location is dead when leaving a statement or if it is used later by another statement execution. In the later case, it is called an `OUT` effect. Currently `IN` effects are internally represented as `READ` effects, and `OUT` effects as `WRITE` effects. The programmer must be aware of the type of effects he is dealing with.

Note that spurious effects are added in loop bodies to avoid... and/or simulate later a control dependence by a data dependence . These effects are `read` effects due to the loop bounds and increment evaluations. see PJ

Note also that `read` effects of Fortran `DO` loop indices due to the incrementations are ignored because they are never upward exposed. This is due to the *compound* nature of the PIPS `DO` construct. If it was decomposed into elementary parts, there would be no such surprising approximations. Note that read effects which might be due to bound or increment expressions as in `DO I = I, 10*I, I` must be preserved.

3.1 Effect representation

Effect = `cell x action x approximation x descriptor`

Descriptor = `convexunion:Psysteme* + convex:Psysteme + none:unit`

Domain `effect` is used to represent a read or write access to a variable or through a pointer, i.e. to abstract a reference in a statement. Statement effects are used to compute array regions, transformers and preconditions, to build use-def chains, dependence graphs, Summary Data Flow Information (SDFI), known as summary effects at the module level, and as cumulated effects at the statement level, and array regions. Proper effects, cumulated effects, summary effects and array regions are all of type `effect` but they are distinguished by pipsmake as difference resources. Proper effects, cumulated effects and summary effects are called simple effects and they do not store information in the last field, `descriptor`.

Field `cell` specifies which memory locations are accessed. In Fortran, variables, scalar or array, are accessed directly. In C, it may be a pointer specifying an indirect adressing. A `cell` is either a `reference` or a persistant[2] `preference`. Attribute `persistant` is used for so-called *reference* and *simple* effects to keep track of actual program references, that should not be modified or freed. This feature is useful for several program analyses or transformations(see the pipsmake-rc documentation).

This persistant attribute is not welcome for more advanced effects, such as regions, which use pseudo-references based on `PHI` variables. Memory allocation is even more difficult to manage when the persistant attribute is declared at the type level and not at the object level. This explains why the field `reference` was moved down and accessed now through the field `cell`.

[2]Persitant is a Newgen attribute carried by a type or by a field. It means that recursive Newgen procedures must stop there. This impacts especially the free and copy functions.

Field `action` specifies if the memory access is a read or a write, for read/write effects, and if the memory value is read from the statement store for a `in` effect or region, and if the memory value written by the statement is later read for a `out` effect or region. So, `read` and `in` effects and regions have action `read` while `write` and `out` effects and regions have action `write`.

Field `approximation` is used to specify if all the memory addresses of the reference for simple effects, or of the set of array elements defined by the `descriptor` for array regions, are or not accessed for sure. For instance, a conditional is going to generate may effects, and a sequence, exact effects[3].

Simple effects and regions may reference a global variable which is in the scope of the callee but not in the scope of the caller or a static variable of the callee declared in a `SAVE` statement. In the first case, the effect translation process from the callee to the caller must use a unique canonical name for such a variable, although the caller does not provide one. In order to define a canonical name, a module whose scope the variable belongs to is arbitrarily chosen and its name is used to prefix the variable name. There is no known trivial choice for this module. Currently, the module name of the first variable in the common variable list is used:

```
ram_section(storage_ram(entity_storage(<my_common>)))
```

Unfortunately, this name depends on the module parsing order. It would be much better to use the lexicographic order among callees, assuming that callees are known before any analysis is started, which is true, and assuming that scopes are known, which is not true because some modules may be analyzed before some other ones are parsed (see pipsmake in [8][1]).

Field `reference` can be used to specify that an effect is limited to a sub-array since a `range` can be used as subscript expression of a reference. This facility is used when the cumulated effects of a callee are translated into proper effects of the CALL site in the caller scope. For regions, field `reference` defines the accessed variable as well as pseudo-variables, known as `PHI` variables. There is one `PHI` variable per array dimension.

Field `context` only is used for effects known as *array regions*. They were defined by Rémi Triolet in [7] and extended by Béatrice Creusillet in [3].

There should not be much strict aliasing between effects in an effect list, but this is (was?) not checked and enforced. Some efforts are made when translating the summary effects of a callee into the caller's frame to avoid this problem.

3.1.1 Nature of an Effect

Action = `read:action_kind + write:action_kind`

Two different memory effects are used in Bernstein parallelization conditions [2] and in other program transformation conditions: read (a.k.a. use) and write (a.k.a. def).

To generalize use-def chains and dependence graphs to C source code, several new kinds of state mutations must be taken into account. The usual state at low level includes only the memory (a.k.a. the store), but, at C source level,

[3]Needless to say, reality is much more complex. This oversimplified statement is only written to support some intuition about may and exact information.

local variable and type declarations must also be considered before distributing loops or eliminating seemingly useless statements. So two new kinds of actions are added: `environment` for variable declarations and `type_declaration` for declarations of types, struct, union and enum. This information is carried by the `action_kind` domain.

Action_kind = store:unit + environment:unit + type_declaration:unit

`IN` regions are represented by `read` effects and `OUT` regions by `write` effects. The action kind is `store`.

3.1.2 Approximation of an Effect

Approximation = may:unit + must:unit + exact:unit

It is not always possible to determine statically if a statement guarded by control structures such as loops and tests is always executed. Thus, it is not possible to know for sure that a variable is read or written by such a compound statement. Some executions may always access it, some other ones may never access it, and some may access it or not dependeing on the statement occurence. Such effects are of the `may` kind.

Sometimes, a simple statement, such as an assignment like `J = 2`, has a known effect. Here `J` *is* written. Such `exact` effects can be used in *use/def chains* analysis to *kill* some scalar variables.

3.2 Lists of effects

The `effects` domain is a list of individual effects. Each effect can only be a read or a write and is related to only one entity, most of the time a variable entity[4], but possibly an area entity or even an entity representing a set of areas[5]. Lots of individual effects are linked to each statements, especially compound statements like `blocks`, `tests`, `loops` and `unstructured`.

Control effects, such as `STOP` or exception, are not computed. PIPS only deals with memory effects. Fortran exceptions like overflows or zero divides are considered programm errors and the error behaviors are not taken into account in PIPS analyses.

Effects and the types defined in the following subsections are not used to represent code, but to store analysis result. These types are declared in the *internal representation* for historical reasons.

Effects = effects:effect*

The next domain can be used to store summary effects of callees (comment out of date?). It's use for the ressource useful_variable_effects/regions that describe the variable and region of variable really use by the code.

entity_effects = entity->effects

The next domain can be used to store a statement to effects mapping. Should be used for proper and cumulated effects and references.

[4]In Fortran, the entity is always a variable entity.

[5]Areas and set of areas are used to express imprecise memory effects due to pointer dereferencing.

```
1 int **p, **q, *a, *b;
2 a = (int *) malloc(10*sizeof(int));
3 b = (int *) malloc(10*sizeof(int));
4 p = &a;
5 p[0][2] = 0;
6 p = &b;
7 q = p;
8 p[0][2] = 0;
9 q[0][2] = 0;
```

Listing 1: Varying paths

statement_effects = persistent statement->effects

3.2.1 Effects Classes

Effects_classes = classes:effects*

The type `effects_classes` is used to store equivalence classes of dynamic aliases, i.e. aliases created at call sites. `Effects_classes` are lists of effects, i.e. lists of lists of regions.

3.2.2 Mapping from Statements to Effects

A different mapping from reachable statements to effect lists is computed by each effect analysis. Because NewGen did not offer the *map* type construct, there is no NewGen type for these mappings. They are encoded as hash tables and used with primitives provided by the NewGen library. They are stored on and read from disk by PIPS interprocedural database manager, *pipsdbm*.

For details about effect analyses available see the Effects Section in the PIPS phase descriptions).

3.2.3 Mapping from Expressions to effects

persistant_expression_to_effects = persistent expression->effects

4 Effects and pointer analyses

In languages involving pointers, elements of symbolic references which are store-dependent are not anymore confined to array indices or polyhedra describing array indices relations with values of program variables, but include the whole memory access path description.

For instance, in Listing 1, p [0][2] on line 5 refers to a [2] whereas on line 8 it refers to b [2]: in the symbolic reference p [0][2] , p [0] is a varying part that depends on the memory store. On the contrary, p [0][2] on line 8 and q [0][2] on line 9 refer to the same memory location whereas their symbolic references are syntactically different.

Thus taking into account some kind of pointer analysis is necessary to be able to disambiguate symbolic references during effect analyses and all kind of dependence tests used throughout the parallelization processes.

Pointer analyses try to gather the *relations* that exist between memory locations, due to pointer values. May alias analyses for instance describe the possible overlaps between memory locations. Points-to analyses establish the relations existing between pointers and the memory locations they point to. We also intend to design another analysis (called *Pointer Values*) to gather the relations existing between the values of pointers as well as the memory locations represented by pointer values.

For all this kind of analyses[6] we need to know the cells whose values, addresses or locations are described, the type of information each cell represents (value, address or location), the approximation of the relation (exact, may or must) and a compulsory descriptor. It's still unclear if some kind of additional information is needed to handle casts.

```
Interpreted_cell = cell x cell_interpretation

Cell_interpretation = value_of:unit + address_of:unit
```

a `location` tag could be added for analyses describing relations between memory locations, but they are not planned for the moment.

```
Cell_relation = first:interpreted_cell x
second:interpreted_cell x approximation x descriptor

Cell_relations = list:cell_relation*

Statement_cell_relations = persistent statement->cell_relations
```

[6]Points-to analyses are currenlty under development using alternative domains described in `points_to_private.pdf`.

Annexe: NewGen Declarations – effects.newgen

—

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

-- Imported domains
-- ----------------
import entity from "ri.newgen" ;
import reference from "ri.newgen" ;
import preference from "ri.newgen" ;
import expression from "ri.newgen" ;
import statement from "ri.newgen" ;

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

-- Domains
-- -------
action_kind = store:unit + environment:unit + type_declaration:unit ;
action = read:action_kind + write:action_kind ;
approximation = may:unit + must:unit + exact:unit ;
cell_interpretation = value_of:unit + address_of:unit ;
cell = reference + preference + gap ;
cell_relation = first:interpreted_cell x second:interpreted_cell x approximation x descrip
cell_relations = list:cell_relation* ;
descriptor = convexunion:Psysteme* + convex:Psysteme + none:unit ;
effect = cell x action x approximation x descriptor ;
effects_classes = classes:effects* ;
effects = effects:effect* ;
entity_effects = entity->effects ;
gap = variable:entity x path_selectors ;
interpreted_cell = cell x cell_interpretation ;
path_selector = expression + recursive_selector ;
path_selectors = path_selector* ;
persistant_expression_to_effects = persistent expression->effects ;
recursive_selector = basis:path_selectors* x coefficient:expression ;
statement_cell_relations = persistent statement->cell_relations ;
statement_effects = persistent statement->effects ;
```

References

[1] B. Baron, *Construction flexible et cohérente pour la compilation inter-procédurale*, Rapport interne EMP-CRI-E157, juillet 1991 8

[2] A. J. Bernstein, *Analysis of Programs for Parallel Processing*, IEEE Transactions on Electronic Computers, Vol. 15, n. 5, pp. 757-763, Oct. 1966. 8

[3] B. Creusillet, *Analyses de régions de tableaux et applications*, Thèse de Doctorat, Ecole des mines de Paris, Décembre 1996. 3, 8

[4] A. Deutsch, *Interprocedural may-alias analysis for pointers: beyond k-limiting*, Conference on Programming Language Design and Implementation Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation 1994 , Orlando, Florida, United States. 4

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

[6] P. Jouvelot, R. Triolet, *NewGen User Manual*, Rapport Interne CAII ???, 1990

[7] R. Triolet, *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures*, Thèse de Docteur-Ingénieur, Université Pierre et Marie Curie, décembre 1984. 3, 8

[8] R. Triolet, *PIPSMAKE and PIPSDBM: Motivations et fonctionalités*, Rapport Interne CAII TR E/133 8

Index