PIPS: Use-Def Chains and Dependence Graph

François Irigoin
Pierre Jouvelot
Rémi Triolet
Yi-Qing Yang

CRI, MINES ParisTech

October 17, 2019

The dependence graph specific data structures, as well as auxiliary data structures used to compute it, are described in this report. The graph itself is a generic graph defined in `Newgen`graph.tex/. It is specialized into a dependence graph by using specific arc and vertex labels. The standard use-def, def-use and def-def chains are encoded with this data structure, as well as non-standard use-use chains useful for cache and locality management. The dependence graph data structure is used in libraries `chains` which computes the use-def chains, `ricedg` which refines the use-def chains into a dependence graph using various dependence tests[1], and `rice` which uses the dependence graph to compute strongly connected components according to the Allen&Kennedy algorithm [1].

The genericity is not supported explicitly by NewGen[2, 3]. Although the arc and vertex labels are NewGen objects, they are seen as *external* types when used in C code. Specific procedures are specified in calls to `gen_init_external` (see `initialize_newgen` in Library `top-level`). C modules using these labels must use explicit casts to convert the generic labels into dependence graph labels.

Auxiliary data structures are used to compute the strongly connected components (SCC) of the dependence graph. The SCC routines are not as generic as could be hoped. They take into account the graph hierarchy implied by the dependence levels of the Allen & Kennedy parallelization algorithm [1]. This explains why the SCC related data structures are declared here and not with graph data structure.

Several functions compute or print a dependence graph. See the PipsMake manual for the list of user-callable functions. Advanced users have access to more functions through the PIPS properties.

It is not possible to save efficiently the internal representation of a dependence graph onto disk. The dependence graph structure contains pointers to memory effects and indirectly to memory references included in code statements. These references which are part of the code data structure do not have *names* and would have to be duplicated in the dependence graph file. Because of the evaluation mechanism used by PipsMake manual, it is not possible to reuse a parallelized code from one PIPS session to the next one. The parallelized

[1]The dependence tests used were not all designed at Rice university but the dependence levels defined at Rice are used.

code depends on the dependence graph, and the dependence graph cannot be retrived. Hence the parallelized code, however correct, is deemed inconsistent with respect to the workspace.

Section 1 deals with the usual import of other NewGen data structures. Section 2 contains the declarations of data structure external to NewGen. Section 3 presents the arc and vertex decorations specific to the dependence graph. And the last section, Section 4, is used to describe the data structures required by the Allen&Kennedy parallelization algorithm.

1 Imported NewGen Types

The dependence graph points to statement and to memory effects (see use-def chains for both data structures).

```
Import effect from "effects.newgen"
```

The dependence graph data structure is a particular instance of the `graph` data structure.

```
Import vertex from "graph.newgen"
```

2 Data Structures External to NewGen

Generating systems are used to abstract a set of dependence arcs by dependence cones (convex and transitive closure of the dependence set) and/or dependence polyhedron (convex closure). This data structure, `generating system`, is part of the C3 Linear Algebra library.

```
External Ptsg
```

3 Arc and Vertex Labels for the Dependence Graph

3.1 DG Vertex Label

dg_vertex_label = statement:int x sccflags

This data type carries information attached to each dependence graph vertex (a.k.a. node). Graph nodes are of type `node` defined in `graph.tex`. Field `statement` contains a `statement_ordering`, i.e. a unique statement identifier (see `ri.tex`). The effective statement can be retrieved using mapping `OrderingToStatement`.

Field `sccs_flags` is used to compute strongly connected components of the dependence graph.

3.2 DG Arc Label

dg_arc_label = conflicts:conflict*

This data type contains information attached to a dependence graph arc. See Newgen data type `graph` in file `graph.tex`. Each DG arc points to the list of all conflicts due to references in either the source or the sink statements, i.e. the statements attached to the sink or the source vertex.

3.3 Dependence Conflict

conflict = persistant source:effect x persistant sink:effect x cone

A conflict is generated by two effects, one in the source statement and one in the sink statement. Simple effects are due to references in statements. More complex effects are generated for procedure calls. Different kinds of arcs (a.k.a. chains), use-def, def-use, use-use and def-def, are derived from the different kinds of effects, read and write. Note that def-def conflicts are computed to provide locality information. This information is or might be used in phase WP65.

3.4 Dependence Levels and Dependence Cone

cone = levels:int* x generating_system:Ptsg

Dependence arcs do not carry enough information to parallelize and/or transform loops [4]. The simplest loop parallelization information is called *dependence levels*. Each level correspond to one common enclosing loop. A common enclosing loop contains both source and sink statements. The outermost enclosing loop is denoted 1, and the innermost loop has the higher level. Non-loop carried dependences have level *number of common enclosing loops plus one*, which makes impossible to know if a dependence is loop carried or not if the number of common enclosing loops is unknown.

Intra-statement conflicts, i.e. conflict between two references of the same statement, are preserved, even if they are not loop-carried. They are required for instruction level scheduling (not implemented in PIPS) and for consistency across program transformations. After instruction atomization (see `pipsmake-rc.tex`), the number of dependence arc should remain the same.

Loop interchange, for instance, requires more information than dependence levels, namely *dependence direction vectors* (DDV). DDV's are not computed in PIPS because a more precise information, dependence cone and/or dependence polyhedron, is made available. These convex polyhedra are not represented by sets of affine constraints but by their generating systems because generating systems are directly useful to constraint schedules (hyperplane methods) and tilings (supernode partitioning).

The ultimate dependence information is *value* and not *location* based. PIPS contains an implementation of Pr. Feautrier's Array Data Flow Graph (DFG). The DFG is implemented with another data structure described in `paf_ri.tex`.

4 Strongly Connected Components

4.1 Flags for Strongly Connected Components

Sccflags = `enclosing_scc:scc x mark:int x dfnumber:int x lowlink:int`

This type is an auxiliary type used when computing the strongly connected components of the dependence graph. More information available in Tarjan's algorithm description?

4.2 Sets of Strongly Connected Components

Sccs = `sccs:scc*`

This data type is used to return the strongly connected components of the dependence graph. It is a simple list of SCC's.

4.3 Strongly Connected Component

Scc = `vertices:vertex* x indegree:int`

A strongly connected component is a set of vertices, represented as a list, and an integer field, `indegree`, used for their topological sort. Allen & Kennedy algorithm [1] is based on topological sorts of the dependence graph SCC's.

References

[1] J. Allen, K. Kennedy, *Automatic Translation of FORTRAN Programs to Vector Form*, TOPLAS, V. 9, n. 4, 1987 1, 4

[2] P. Jouvelot, R. Triolet, *NewGen: A Language Independent Program Generator*, Rapport Interne CAII 191, 1989 1

[3] P. Jouvelot, R. Triolet, *NewGen User Manual*, Rapport Interne CAII ???, 1990 1

[4] Y. Yang, C. Ancourt, F. Irigoin, *Minimal Data Dependence Abstractions for Loop Transformations*, International Journal of Parallel Programming, v. 23, n. 4, Aug. 1995, pp. 359-388 3