PIPS: Generic Oriented Graph

François Irigoin
Pierre Jouvelot
Rémi Triolet

CRI, Ecole des Mines de Paris

April 30, 2024

Introduction

The `graph` type implements an oriented graph or multigraph (i.e. more than one arc may link two vertices). This data structure is not really generic, i.e. types for vertices and arcs are not imported, but types for vertex and arc *labels* are imported. They do not have to be NewGen types. If they are NewGen types, they have to be handled as non-NewGen types and explicitly dynamically declared at initialization. When labels are used, they have to be explictly casted into their effective type.

Since NewGen was not really designed to support re-entrance, unusual bugs may occur when using standard or high-level NewGen primitive such as `free` or `gen_multi_recurse()`. See NewGen documentation.

The `graph` type is used for dependence graphs and for use-def chains, with specific vertex and arc labels (see the `dg` data structures). It should have been used for control flow graphs, but another graph representation is embedded in the internal representation (see `unstructured`).

There is no `graph-util` package containing primitives for graph, such as adding a vertex, removing a vertex, adding an arc, removing an arc, printing a graph, walking a graph,.... This probably is due to the facts that the generic graph datastructure is used only twice throughout PIPS and that interesting algorithms like strongly connected component computation require extra-datastructures. These datastructures are closely linked to the concept of dependence graph because of arc levels, and they are joined to the dependence graph specific labels.

The dependence graph, defined in `dg.f.tex`, is built by the `chains` library, updated by the `ricedg` library and used by privatization transformations and by automatic parallelization (see the `transformation` and `rice` libraries). Another version of the dependence graph, the Data Flow Graph (DFG) is defined in `paf_ri.f.tex`.

Vertex and Arc Labels

External vertex_label

The `vertex_label` field points to the information attached to a vertex or node. It is not defined here and must be provided by the user.

External arc_label

The `arc_label` field points to the information attached to an arc. It is not defined here and must be provided by the user.

Graph Structure

An oriented graph is defined mathematically as a set of vertices and a set of arcs.

Graph = `vertices:vertex*`

The set of vertices is implemented as a NewGen `list`. It is not clear how sets are implemented with list as long as equality is not defined. Here, vertices only are known by their addresses and their uniqueness is easy to check. Arcs are attached to vertices and there is not set of arcs.

Vertex = `vertex_label x successors:successor*`

Each vertex is represented by an object of type `vertex`. It is identified by its address and points to its label thru the `vertex_label` field, and to a list of arcs thru the `successors` field. Quite unfortunately, arcs are named *successor*, although they only *point* to successors.

The type of a `vertex_label` is assumed not to be a NewGen type. it is application specific and must be defined somewhere else. See for instance type `dg_vertex_label` defined in file `dg.f.tex`.

Successor = `arc_label x vertex`

Each arc in the graph is implemented as an object of type `successor`. The `vertex` field contains the *effective* successor. The `vertex_label` field contains some information attached to the arc. This information depends on the application. See for instance the type `dg_arc_label` defined in file `dg.f.tex`.

Note that more than one arc may link two vertices. There is no graph primitive to add an arc. Each arc is known by its memory address. There is no direct way to find the *origin* of an arc.

There is no primitive to check if a graph data structure represent a graph or a multigraph. There is no primitive to check consistency (e.g. each vertex pointed to by an arc in the graph vertex set).

Graphs are walked with two nested loops. The outer loop is over the vertices. The innermost one is over each vertex edges, so-called `successors`.