Assignment #2: Data Flow Analysis

Due: Before class, February 12th

Introduction

In this assignment, you and one partner will write a dataflow analysis pass to compute reaching definitions. You will use this analysis in the next assignment when you write a dead code elimination pass.

Before starting the assignment, you should:

- Read Section 8.1 of Muchnick, Advanced Compiler Design and Implementation
- copy the files in /afs/cs.cmu.edu/academic/class/15745-s03/public/assignments/2 to your working directory. E.g.

  ```
  cp -a /afs/cs.cmu.edu/academic/class/15745-s03/public/assignments/2
  ~/localnci/assignments/
  ```

  This will create two subdirectories, `bvd` and `test`.

  - In `bvd` You will find a skeleton for the class `ReachingDefs` in `reaching_defs.{h,cpp}`. The other files in the `bvd` directory are files from the Machine SUIF distribution, you should not have to modify any of these files. You can use `gmake` in the `bvd` directory to create your version of `libbvd.so`. Your job in this assignment is to complete the missing methods in such file.
  - In `test` there is a `makefile` you can customize to create the proper MachSUIF file for testing your pass. You do not need to use this file, but you may find it useful. In case you need it and, to reduce the amount of work you have to do when you change from linux to alpha machines, you should probably define the `CODETARGET` environment variables in your `.*rc` file.

Reaching Definitions

The reaching definitions pass will compute all the definition sites that "reach" a particular program point. This is similar to the liveness pass (which you used in the peep pass you worked on in assignment 1) which computes all the variables/registers that are live at a program point. In both cases, the pass attaches a set to each program point describing the results of the particular analysis at that program point.

In order to make writing the reaching definitions pass easier we have included most of the code for two supporting classes: `DefTeller` and `DefMap`.

- Class `DefTeller` abstracts the problem of deciding how many things an instruction defines and how to identify them. It generates a unique small integer for each "definee".
- Class `DefMap` maps each definition instruction to a range of small integers. These numbers uniquely identify the definition points within the procedure being compiled. Since an instruction can potentially define more than one thing, we need a sequence of integer indices to identify the definition points within it.

Don't confuse the two different kinds of small-integer identifiers being used here. Class `DefTeller` manages integer identifiers for the quantities (registers and variables) whose definitions we care about. Class `DefMap` assigns a different number to each definition point. The bit vectors used to represent data flow are indexed by these latter (definition point) numbers, since this analyzer computes the flow of each distinct definition.

Like the Liveness analyzer (whose code you can find in `liveness.{h,cpp}`), class `ReachingDefs` defines some virtual methods of its base class, `Bvd`. These are called `find_kill_and_gen`, `kill_set`, `gen_set` and `num_slots`, and you can read about them in the BVD library document. In addition, class `ReachingDefs` provides two public methods:

- `map()` returns a pointer to a `DefMap` covering the instructions in the program under analysis.
- `def_points_for` gives you all of the definition points for a particular "definee", be it a variable, a register, or whatever. Its result is a pointer to a set of (integer) definition-point identifiers.

We provide you with the implementation of `kill_set`, `gen_set`, `map` and `num_slots`. To complete the assignment you need to implement the remaining methods plus the function `def_on_entry`, check the source code for further information.

You should not need any extra methods or data members of `ReachingDefs` to complete your implementation.

Testing Your Analyzer

To help you debug your data-flow analyzer, we provide a command called `do_rd_test`, which reads the SUIF file named on its command line, creates a `ReachingDefs` instance for each procedure that it contains, and prints the machine instructions for the procedures. It marks each definition instruction with the number assigned to it by your `ReachingDefs` solver. And it prints the numbers for definitions reaching the entry and exit of each basic block.

The input file to `do_rd_test` must be in `cfg` form. So you will have to ensure that you have run `do_il2cfg` before you run `do_rd_test`. If you use the `makefile` we supplied in the test directory, you will want to produce a `.afg` file. I.e., assuming you C source code is in the file `foo.c` you should execute: `gmake foo.afg` and then execute `do_rd_test foo.afg`.

One piece of advice: The above test is not exhaustive, for example, `do_rd_test` never invokes the method `def_points_for` of your `ReachingDefs` class. The `do_rd_test` program only outputs `def_points` that reach a particular CFG node; so you may want to design different test programs in order to ensure your implementation is correct. A good starting point is the source for `do_rd_test`, available in the class MachSUIF tree.

As in the first assignment you may need to compare the output of your analyzer with the output of the solution binary, which is available in the class MachSUIF tree.

Hand In

Please create a text file `group.txt` containing your user id and your partner's.

To hand in your solution just copy the files `reaching_defs.{h,cpp}` and `group.txt` to the appropriate hand in directory. Also, only one partner should hand in a solution.

Good luck!

Acknowledgments

Thanks to Mike Smith and Glenn Holloway who originally created this assignment; and Seth Goldstein who created the original 15-745 version.