Interprocedural Scalar Dataflow Analysis

Zhiyuan Li
Department of Computer Science
Purdue University, USA

Introduction

- In the discussions on dataflow analysis so far, we have assumed that the code either contains no procedure calls or such calls bear no effect on the analysis.
- In practice, “call effects” often need be taken into account
- Take reaching definition as an example:

Suppose a is a global variable in C. Whether *a* defined in d1 reaches the use in u1 depends on whether Foo() “kills” *a*. “Kill” is also known as must-mod

```
a = b + c;          (d1)
Foo( );
d = a * s;          (u1)
```

Similarly, in the following example, whether variable *a* belongs to Live_in of Foo() depends on whether Foo() “kills” *a*. If there exists a possibility that a will not be modified by Foo(), even though there exists also possibilities that *a* may be modified by Foo(), then *a* is still conservatively considered to be in Live_in of Foo().

```
Foo( );
d = a * s;              (u1)
```

In-lining

- A straightforward method for handling calls is “in-lining”, i.e. substituting the call by the body of the called routine.
- However, this method does not handle recursive calls properly.
- In-lining also increases the size of the routine to be dealt with by every dataflow analysis algorithm.
- In-lining duplicates the body of called routine for each place the routine is called.
- So, instead, we often want to analyze without in-lining.

Summary Analysis

- If Foo() does not call any other procedures, i.e. if Foo() is a "leaf" routine, then we can analyze Foo() first and see if Foo() kills *a*.
- How do we analyze this within Foo()? We need to analyze the "must-mod summary" for Foo(). Consider this flow analysis problem:

Forward propagation of mustmod_in (B) and mustmod_out (B) over all nodes B. (For simplicity assuming B contains a single instruction or statement.)

Assuming a single exit, mustmod_in(exit) will be the must-mod summary for the entire Foo() routine.

- What should be our dataflow equations?
- What should be initial value for mustmod_out (B)
- Which node should be placed on the work list first?

Call chains

- Next, let us assume Foo() calls another routine Goo(), which is a leaf routine. In order to sumarize must-mod for Foo(), we need summarize must-mod for Goo() first. Suppose a node *B* in Foo() makes a call to Goo(). How should we propagate Live_in (B) to Live_out (B)?

- Next, let us assume there is a "call chain" such that Foo() calls Goo() calls Hoo() …. calls Zoo(), where Zoo() is a leaf routine.
  - Obviously we should summarize must-mod in the reversed direction of the call chain.
  - The *call graph* of a program represents the calling relationship among its routines. The basic call graph contains only one edge from a routine Foo() to another routine Goo() even if Foo() may call Goo in multiple places, i.e. multiple *call sites*.
  - If the call graph of a program is a DAG, then we can summarize must-mod in a reversed topological sort.

Recursive Calls

- What if Foo() calls Goo() …. eventually calls Foo() again? How do we determine must-mod for each involved routine?
  - First, we need to make an assumption that recursive calls always terminate, i.e. there will be an end to a call chain at run time and the last routine called in this dynamic chain will make no further calls.

- In iterative propagation over the call graph, we will interleave intraprocedural must-mod analysis with interprocedural must-mod summary propagation:
    - Initially all must-mod summaries are assumed to include all variables
    - Every time intraprocedural must-mod is computed over a routine's control flow graph, we check to see whether the routine's must-mod summary is changed. If so, all callers of this routine must be analyzed intraprocedurally again for must-mod.

Traversal of the call graph which have cycles

- Find all strongly connected components and reduce each to a single “condensed node”, which results in a reduced graph
- Visit the reduced call graph in reversed topological order
- With each condensed node, we iterate until no must-mod information is changed.

- So far we have considered global variables only (assuming no aliases)
- Call-by-reference parameter passing introduces new difficulties
- A reference (i.e. a pointer) passed to a callee may result in the de-referenced variable (i.e. the variable pointed to) being modified.
  - Is the variable killed by the routine?
- We analyze the must-mod of the callee and see if the formal parameter belongs to must-mod. If so, the variable pointed to by the corresponding pointer argument is killed by the callee.

The Issue of Aliasing

- The same variable may be pointed to by two arguments passed to a callee.
- Within the callee, it is possible that neither of the corresponding formal parameters belong to must-mod, but if viewed as the same variable, then it belongs to must-mod.
- A global variable might also has a pointer passed to the callee.
- This is called the aliasing issue.
- Numerous algorithms have been proposed to determine which variables are aliases in the same routine.

Assuming there is no aliasing, the must mod information can still be determined by our previous mechanism after introducing a actual-formal argument mapping for each call site.

For a complete treatment of must-mod, see Meyer's paper

For a relatively comprehensive treatment of may-mod, see Cooper and Kennedy's paper