THESIS PROPOSAL: AUTOMATED SYNTHESIS FOR PROGRAM INVERSION

A Thesis
Presented to
The Academic Faculty

by

Perry H. Disdainful

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
April 2013

THESIS PROPOSAL: AUTOMATED SYNTHESIS FOR PROGRAM INVERSION

Approved by:

Professor Ignatius Arrogant,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Richard Vuduc, Advisor
School of Computer Science
Georgia Institute of Technology

Professor General Reference
School of Mathematics
Georgia Institute of Technology

Professor Ivory Insular
Department of Computer Science and Operations Research
North Dakota State University

Professor Earl Grey
School of Computer Science
Georgia Institute of Technology

Professor John Smith
School of Computer Science
Georgia Institute of Technology

Professor Jane Doe
Another Department With a Long Name
Another Institution

Date Approved: 1 July 2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Why should I provide a summary? Just read the thesis.

CHAPTER I

INTRODUCTION

Every dissertation should have an introduction. You might not realize it, but the introduction should introduce the concepts, background, and goals of the dissertation.

1.1 Concepts

This is where we talk about the concepts behind the dissertation.

1.1.1 Primary Concept

This is the primary concept.

1.1.2 Secondary Concept

This is the secondary concept.

1.1.2.1 Even more secondary

CHAPTER II

SYNTHESIS FOR PROGRAMS WITH ONLY SCALARS

2.1 Handling loop-free programs

2.1.1 Problem setup

Let the set of *target variables* be $S = \{s_1 \dots s_n\}$ with *initial values* $V = \{v_1 \dots v_n\}$, where v_i is the initial value of s_i. These variables are modified by a *target function*[1] M, producing $V' = \{v'_1 \dots v'_n\}$, the *final values* of the target variables. Our goal is generating two new functions, the *forward function* M^S_{fwd} and the *reverse function* M^S_{rvs}, so that M^S_{fwd} transfers V to V', and M^S_{rvs} transfers V' to V. We define *available values* as values which are ready to use at the beginning of M^S_{rvs}. For example, values in V' and constants are available values. We also call values in V *target values* which are values we want to restore from M^S_{rvs}.

Note that M and M^S_{fwd} have the same input and output, but M^S_{fwd} is instrumented to store control flow information and values that are later used in M^S_{rvs}. This introduces two kinds of cost that must be considered when generating the forward-reverse pair $\{M^S_{fwd}, M^S_{rvs}\}$: extra memory usage and run-time overhead.

2.1.2 Framework overview

We will first treat the inversion of loop-free code with only scalar data types, without aliasing. When such code is converted to static single assignment (SSA) form [**?**], each versioned variable is only defined once and thus there is a one-to-one correspondence between each SSA variable and a single value that it holds. We will also take advantage of the fact that loop-free code has a finite number of paths. Loops will

[1] The function here is a C/C++ function, not a function in mathematics.

```
int a, b;
void foo() {
  if (a == 0)
    a = 1;
  else {
    b = a + 10;
    a = 0;
  }
}
```

(a)

```
void foo_forward() {
  int trace = 0;
  if (a == 0) {
    trace |= 1;
    a = 1;
  }
  else {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

(b)

```
void foo_reverse() {
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else {
    a = b - 10;
    restore(b);
  }
}
```

(c)

Figure 1: (a) The original event (b) The forward event (c) The reverse event

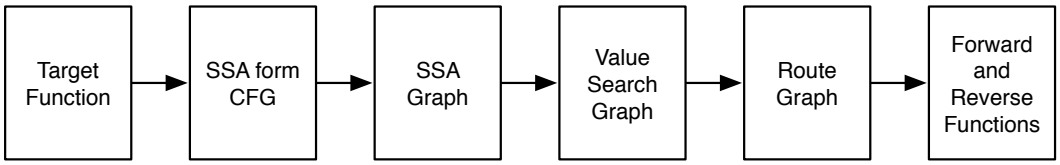

Figure 2: Overall framework of the inversion algorithm

be discussed in the next section, and non-scalar data types and aliasing will not be handled in this paper.

Given a cost measurement, for each path in the target function there should exist a best strategy to restore target values. Strategies usually vary among different paths. Therefore, the reversed function we produce should include the best strategy for each path; each path in the original function should have a corresponding path in the reverse function.

To restore target values, we will build a graph which shows equality relationships between values. We call this graph the *value search graph*, and it is built based on an SSA graph [**?**, **?**]. Then a search is performed on the value search graph to recursively find ways to recover the set of target values given the set of available values. If there is more that one way to restore a value, we choose the one with the smallest cost. The search result is a subgraph of the value search graph which we call a *route graph*. For any path, a route graph shows a specific way to recover each target value from

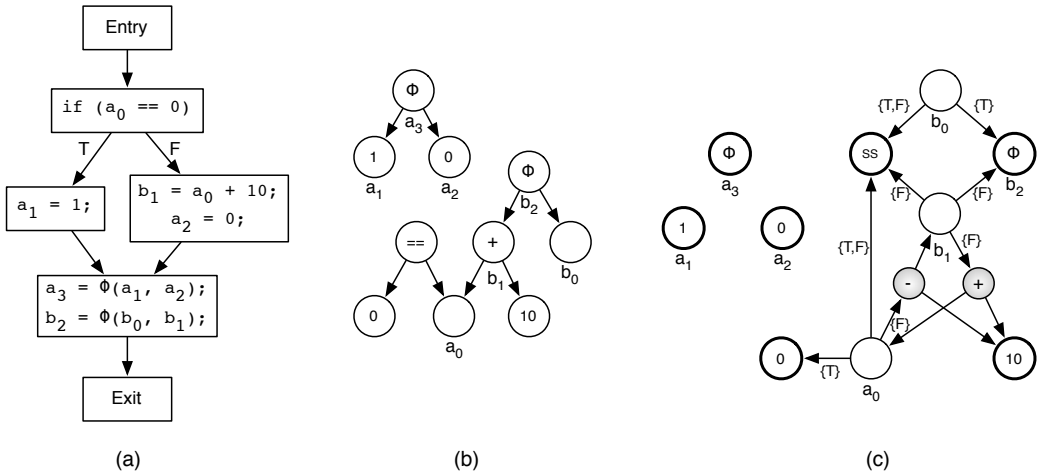

Figure 3: (a) The SSA-transformed CFG of the function in Figure 1(a) (b) The corresponding SSA graph (c) The corresponding value search graph. Nodes with bold outlines are available nodes; outgoing edges for these nodes are omitted because available nodes need not be recovered. 'SS' is the special state saving node. Edges are annotated with their CFG path set.

available values. Finally, the forward and reverse functions are built from a route graph. Figure 2 illustrates this process.

2.1.3 The value search graph (VSG)

We first build an SSA graph for the target function. An SSA graph [**?**, **?**], built based on SSA form, consists of vertices representing operators, function symbols, or ϕ functions, and directed edges connecting uses to definitions of values. It shows data dependencies between different variables. The full algorithm for building an SSA graph is presented in [**?**]. Figure 3(a)(b) show the SSA-transformed CFG and its SSA graph for the function in Figure 1(a). In this example, `a` and `b` are two target variables with initial values $\mathtt{a}_0$ and $\mathtt{b}_0$, and final values $\mathtt{a}_3$ and $\mathtt{b}_2$.

A *value search graph* enables efficient recovery of values by explicitly representing equality relationships between values. Unlike an SSA graph, operation nodes are separated from value nodes in the value search graph, since their treatment is different for recovering values. An edge connecting two value nodes u and v implies that u and

v have the same value. An edge from value node u to an operation node *op* means that u is equal to the result of evaluating *op* with its operands. To recover the value associated with node v, we can recursively search the graph starting at v.

We attach a set of CFG paths to each edge in a value search graph, meaning the edge is applicable only if one of the CFG paths in that set is selected in the original function. For operation nodes in the SSA graph, let the set of paths attached to each outgoing edge be the CFG paths for which the corresponding operation is executed. Similarly, for ϕ nodes, each reaching-definition edge should be annotated with all CFG paths for which the corresponding reaching definition reaches the ϕ function. We will describe an implementation of the path set representation later.

During the execution of the forward function, once a variable is assigned with a new value, its previous value may be destroyed and cannot be retrieved. To guarantee that a search in the value search graph can always restore a value, we introduce special *state saving edges*. The idea behind these edges is that each value may be recovered by storing it during the forward execution. Whenever a state saving edge appears in the search results, the forward function is instrumented to save the corresponding value. The path set associated with a state saving edge for a value node v is the set of all paths that include v's definition. All state saving edges point to a unique *state saving node*.

We apply the following rules to convert an SSA graph into a value search graph:

- For simple assignment $v = w$, there is a directed edge from v to w in the SSA graph. Since we can retrieve w from v, add another directed edge from w to v with the same path set.

- A ϕ node in the SSA graph has several outgoing edges connecting all its possible definitions. For each of those edges, add an opposite edge with the same path set.

- For each operation node in the SSA graph, split it into an operation node and a value node, with an edge from the value node to the new operation node. The new operation node takes over all outgoing edges, and the value node takes over all incoming edges.

- If an equality operation (`==`) is used as a branching predicate and its outcome is true, we know that the two operands are equal. Therefore, we add edges from each operand to the other, with a path set for the edge equal to the path set of the *true* CFG edge out of the branch. We add the edges analogously for a not-equal operation (`!=`), but with the path set from the *false* side of the branch.

- For every value that is not available, insert a state saving edge from the corresponding value node to the state saving node.

Lossless operations For certain operations, such as integer addition and exclusive-or, we can recover the value of an operand given the operation result and the other operand. For example, if $a = b + c$, we can recover b given a and c. For each such lossless operation, insert new operation nodes that connect its result to its operands, allowing the operands to be recovered from the result. The new nodes are added according to the following rules:

- Negation (`a = -b`) and bitwise not (`a = ~b`): the new operations are `b = -a` or `b = ~a`, respectively.

- Increment (`++a`) and decrement (`--a`): insert `--a` or `++a`, respectively.

- Integer addition (`a = b + c`) and subtraction: for addition, the new operations are `b = a - c` and `c = a - b`; analogously for subtraction.

- Bitwise exclusive-or (`a = b ^ c`): insert `b = a ^ c` and `c = a ^ b`

There are two special types of nodes in a value search graph: *target nodes* are value nodes containing target values, and *available nodes* are value nodes containing available values plus the state saving node. As an optimization, we never create any outgoing edges for an available node. Figure 3(c) shows the value search graph built for the code in Figure 1(a). The available nodes are shown with a bold outline. Since the function only has two paths, we use labels 'T' and 'F' to represent the CFG paths passing through the true and false body in the target function, respectively. The '–' operation node connecting a_0 to b_1 and the constant value '10' is generated from the '+' operation. The edge from a_0 to '0' for the path 'T' is added based on the fact that $a_0 = 0$ on that path. The 'SS' node in the graph is the state saving node, and all unavailable nodes are connected to it. From the value search graph, we can find two valid ways to restore b_0 for the path 'T': b_0 to SS node and b_0 to b_2. Obviously the second one is better since it avoids a state saving operation, and this better selection will be produced from the search algorithm described later.

2.1.4 The route graph (RG)

A *route graph* is a subgraph of a value search graph connecting all target nodes to available nodes. Each route graph represents one way to restore the target values, and there may exist many valid route graphs for the same set of target values. Edges in the route graph may have different path sets than the corresponding edges in the value search graph. For each edge e in a route graph, let $P(e)$ denote the set of CFG paths that the edge is annotated with. The following properties guarantee that the route graph properly restores all target values:

I) Let $\mathcal{U}$ be the set of all CFG paths. Then, for each target node t,

$$\bigcup_{out \in \mathrm{OutEdges}(t)} P(out) \quad = \quad \mathcal{U}$$

II) For each node n that is neither a target node nor an available node,

$$\bigcup_{out \in \text{OutEdges}(n)} P(out) = \bigcup_{in \in \text{InEdges}(n)} P(in)$$

III) For each value node n, given any two outgoing edges $n \to p$ and $n \to q$, $P(n \to p) \cap P(n \to q) = \emptyset$

IV) If e is a route graph edge and its corresponding edge in the value search graph is e', then $P(e) \subseteq P(e')$

V) For each directed cycle with edges $e_1 \ldots e_n$, $\quad \bigcup_{i=1}^{n} P(e_i) = \emptyset$

Property I specifies that each target value is recovered for every CFG path. Property II means that each value is recovered exactly for the paths for which it is needed. Property III requires that for each CFG path, there is at most one way to recover a value. Property IV requires that the set of CFG paths associated with an edge in the route graph is a subset of the CFG paths originally associated with that edge in the value search graph. Finally, property V forbids self-dependence: restoring a value cannot require that value.

Figure 4 shows three valid route graphs for the value search graph in Figure 3. Route graph 4(a) only includes state saving edges. Route graph 4(b) takes advantage of the fact that for the 'T' path the values of both $\mathtt{a}_0$ and $\mathtt{b}_0$ are known; it only uses staving for the 'F' path. Route graph 4(c) improves upon route graph 4(b) by recomputing $\mathtt{a}_0$ as $\mathtt{b}_1\mathtt{-10}$ for the CFG path 'F'; state saving is only applied to $\mathtt{b}_0$ for path 'F'.

2.1.5 Searching the value search graph

2.1.5.1 Costs in route graphs

As we have seen in Figure 4, there may be multiple valid route graphs that recover the target values, but with different overheads. In order to choose the route graph with the smallest overhead, we must define a cost metric.

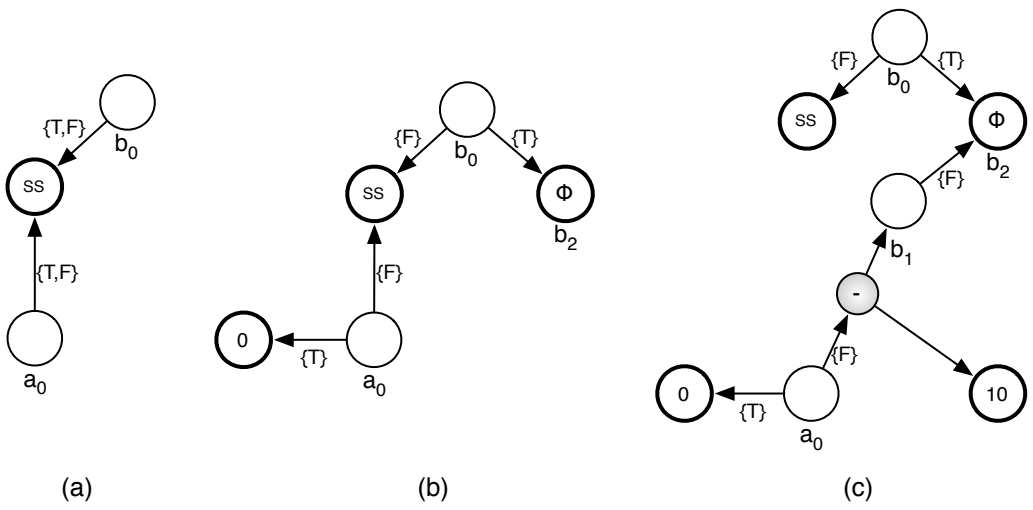

Figure 4: Three different route graphs for the target values a_0 and b_0 given the the value search graph in Figure 3(c).

Generally, there are two kinds of overhead in forward and reverse functions: execution speed and additional memory usage; we only consider the storage costs. State saving contributes the most to the overhead memory usage and it also significantly affects the running time of both forward and reverse functions. Storing the path taken during forward execution is the other factor that contributes to memory usage; this overhead is bounded and is the same for all route graphs, so we exclude it from our cost estimate. With each state saving edge in the value search graph, we associate a cost equal to the size of the value that must be saved; other edges have cost 0. The cost of a route graph for a specific CFG path is the sum of the cost of those edges whose annotated path sets include that CFG path.

In Figure 4, suppose the cost to store and restore either `a` or `b` is c, the following table shows the cost of three route graphs for each CFG path. Obviously the third route graph is the best one.

CFG path	route graph (a)	route graph (b)	route graph (c)
T	$2c$	0	0
F	$2c$	$2c$	c

We have defined the cost of a single CFG path; however, a route graph may have different costs for different CFG paths. When searching the value search graph, we would like to treat groups of CFG paths that share some edges in the route graph together, rather than performing a full search for each CFG path. For this reason, the search algorithm partitions the CFG paths into disjoint sets of paths that have equal cost and we save the cost for each set of paths independently. In our search algorithm, we denote the costs of a route graph r as r.costSet.

$$r.\text{costSet} = \{\langle P_i, c_i \rangle | P_i \text{ is a set of CFG paths and } c_i \text{ is the cost}\}$$

2.1.5.2 Search algorithm

Our search algorithm should aim to find a route graph that has the minimum cost for each path. Theoretically, however, searching for a minimal route graph is an NP-complete problem. To make the problem tractable, we apply the heuristic of finding a route graph for each target value individually; the individual route graphs are then merged into a route graph that restores all the target values. Similarly, in order to recover the value of a binary operation node, we recover each of the two operands independently and then combine the results.

The pseudocode for our heuristic search algorithm is presented in Algorithm 1. The `SearchSubRoute` function returns a route graph given a target node, the paths for which that node must be restored, and the set of value nodes visited so far. The algorithm explores all ways to recover the current node by calling itself recursively on all the nodes that are directly reachable from the current node; available nodes are the base case. Lines 5–10 handle recovering the values of operation nodes. In order to recover the value of an operation node, each of its operands must be recovered. Lines 11–14 return a trivial route graph for available nodes, with a cost of 0. The remaining body of the algorithm (lines 15–27) handles recovering a value node that is not available. Each of the out-edges of the target node may be used to recover

Algorithm 1: Searching for a route graph in a value search graph

Initial input: The search start point target, with paths $= \emptyset$, visited $= \emptyset$

```
1  SearchSubRoute(target, paths, visited)
2  begin
3      resultRoute ← ∅, subRoutes ← ∅
4      if target is an operation node then
5          foreach edge ∈ OutEdges(target) do
6              if edge.target ∈ visited then return ∅
7              newRoute ← SearchSubRoute(edge.target, paths, visited)
8              if newRoute = ∅ then return ∅
9              add edge and newRoute to resultRoute
10         return resultRoute
11     if target is available then
12         add target to resultRoute
13         add ⟨paths, 0⟩ to resultRoute.costSet
14         return resultRoute
15     foreach edge ∈ OutEdges(target) do
16         if edge.target ∈ visited then continue
17         newPaths ← edge.pathSet ∩ paths
18         if newPaths = ∅ then continue
19         newRoute ← SearchSubRoute(edge.target, newPaths, visited ∪ {target})
20         add edge with paths newPaths to newRoute
21         foreach ⟨paths, cost⟩ in newRoute.costSet do cost += edge.cost
22         foreach route in subRoutes do ChooseMinimalCosts(route, newRoute)
23         add newRoute to subRoutes
24     add target to resultRoute
25     foreach route in subRoutes do
26         if route.pathSet ≠ ∅ then add route to resultRoute
27     return resultRoute

28 ChooseMinimalCosts(route1, route2)
29 begin
30     if route1.pathSet ∩ route2.pathSet = ∅ then return
31     foreach ⟨paths1, cost1⟩ in route1.costSet do
32         foreach ⟨paths2, cost2⟩ in route2.costSet do
33             if paths1 ∩ paths2 = ∅ then continue
34             if cost1 > cost2 then
35                 paths1 ← paths1 − paths2
36                 Remove (paths1 ∩ paths2) from all edges of route1
37             else
38                 paths2 ← paths2 − paths1
39                 Remove (paths1 ∩ paths2) from all edges of route2
40     route1.pathSet = ⋃⟨paths,cost⟩∈route1.costSet paths
41     route2.pathSet = ⋃⟨paths,cost⟩∈route2.costSet paths
```

its value for the CFG paths associated with that edge; these edges are explored in the `for`-loop in lines 15–23. The variable `newPaths` on line 17 represents the set of paths that we are both interested in and are associated with the current edge. In line 19, we recursively find a route graph that recovers the target value by recovering the target of the current outgoing edge. Lines 21–22 update the cost sets of the new route graph; if it provides a lower cost for some CFG path than the solutions found so far, the partial results are modified so that each CFG path is restored with the cheapest route graph. Finally, the route graph from line 19 is added to the list of partial results (line 23). After all out-edges of the target node have been explored, the partial results are merged into a single route graph and returned (lines 24–27). Note that it is unnecessary to check whether the target node has been successfully recovered, since the state saving edge always provides a valid route graph for the node. Figure 4(c) shows the route graph produced by the algorithm when searching the value search graph from Figure 3(c).

The search algorithm enforces properties I–IV from section 2.1.4 during its execution. To make sure that the search result does not contain cycles (property V), we record which value nodes are already in the route using a set `visited` in Algorithm 1. This alone is not sufficient to guarantee that the result is acyclic, for there may be two different paths with identical cost to recover a single value node. If one way is chosen to recover a value node v during path of the search, and then later v is recovered differently for the same CFG path, a cycle may form. To prevent this situation from occurring, we always traverse out-edges in the same order of line 19 of Algorithm 1; the first route graph with the smallest cost is chosen. In addition, two paths coming from two different value nodes may also form a cycle when all costs on edges of the cycle are 0. We eliminate this possibility by replacing 0 by a small cost ε.

2.1.6 Instrumentation & Code generation

2.1.6.1 Representing CFG path sets

Our search algorithm relies on efficiently computing intersection, union, and complement of CFG path sets, as well as testing whether the set of paths is empty; for this reason we suggest implementing the set representations as bit vectors. Ball and Larus [**?**] present an path profiling method in which each path is given a number from 0 to $m - 1$, where m is the count of the CFG paths. We use their algorithm to number each path, and for each path we associate exactly one bit in the bit vector used to represent a path set.

2.1.6.2 Recording CFG paths

We need to store path information in a way that allows us to efficiently record the CFG path taken (for forward execution), and to efficiently check if the path matches a given set of CFG paths attached to a route graph edge (for reverse execution). However, if we encode each path using its path number, then examining whether a path is a member of a set is inefficient. Instead we use a bit vector to record the CFG path, in which each bit represents the outcome of a branching statement. Since this method is similar to *bit tracing*[**?**], we call this bit vector a *trace*. Note that two branches may share the same bit if they cannot appear in the same path. Thus, the number of bits required to store the path taken is equal to the largest number of branches that appear on a single CFG path. Algorithm 2 calculates bit-vector position for each branch node accordingly.

In the forward function, we use an integer as the bit vector to record all predicate results[2]. Let `trace` be the variable recording a trace, initialized to zero; then the true

[2]Potentially we could omit recording predicates that do not affect the reverse function.

Algorithm 2: Generating the bit position for each branch node.

```
foreach CFG node u in reverse topological order do
    if u is a leaf node then
        position(u) ← -1
    else if u is a branch node then
        /* u → v and u → w are its two out-going edges                    */
        position(u) ← max(position(v), position(w)) + 1
    else
        /* u → v is its out-going edge                                    */
        position(u) ← position(v)
```

edge of each branch node v is instrumented with the statement [3]

$$\texttt{trace = trace | (1 << } \textit{position}(v)\texttt{);}$$

where $\textit{position}(v)$ is calculated by Algorithm 2. The variable `trace` is stored at the end of the forward function and restored at the beginning of the reverse function. Note that we can further optimize the instrumentation by moving a trace updating operations downward through the CFG and merging them.

In the reverse function, we must test if `trace` matches the path sets that appear on route graph edges. We start with transforming each path in the set into a trace (the trace for each path can be computed by the same means as recording a trace in the forward function). Then, checking if a path set contains a path represented by `trace` is done by comparing it to each trace. Suppose a path set containing two paths is transformed into two traces `01101` and `01001`. Instead of comparing `trace` to each of them as:

```
if (trace == 01101 || trace == 01001)
```

we can simplify this predicate by using a mask `11011` on `trace`:

```
if ((trace & 11011) == 01001)
```

[3] We use several operators in C/C++ syntax here and below, which includes bitwise OR operator `|`, bitwise AND operator `&`, bitwise left shift operator `<<`, equal to operator `==`, and logical OR operator `||` .

The combined trace for `01101` and `01001` is `01×01`, where × denotes that the bit does not matter. Given a set of traces, we can combine pairs repeatedly to reduce the size of the set. This greatly reduces the complexity of the branching statements in the reverse code.

Algorithm 3 starts out with all traces corresponding to a set of CFG paths and merges them into a minimal set of traces that can be used to test membership in the set. The intuition behind Algorithm 3 is that if the traces are sorted so that bit i is the least significant bit, the traces that are identical to each other except for bit i will be adjacent. However, if we are careful we don't have to pay the full sorting cost for each bit i. If the traces are sorted when their bits are considered in the order $b_1 b_2 \ldots b_{i-1} \quad b_k b_{k-1} \ldots b_i$ and we want to sort them according to the bit order $b_1 b_2 \ldots b_{i-2} \quad b_k b_{k-1} \ldots b_{i-1}$, we need only sort each sequence of the trace for which bits 1 through $(i-2)$ are identical. For each such sequence, there are at most three sorted subsequences, indexed by bit b_{i-1}; these can be merged in linear time (similarly to mergesort). If we use a linear-time sort, such as radix sort, for the first iteration, the overall runtime of Algorithm 3 is $O(kn)$, where n is the size of the path set.

Algorithm 3: Merging a set of path traces

```
MergePathTraces(traces)
begin
    /* Each trace has k "bits", and each bit is 0, 1, or ×        */
    /* Bits are numbered ascendingly; e.g.  m = b1 b2 ... bk       */
    for i ← k down to 1 do
        /* Note:  for i = k, the bit ordering is m = b1 b2 ... bk  */
        Sort traces, where trace bits are ordered b1 b2 ... b(i-1)  bk b(k-1) ... bi
        for j ← 2 to Length(traces) do
            if traces[j − 1] and traces[j] match except for bit i then
                set bit i to × for traces[j − 1]
                delete traces[j]
```

After the merge, if we have n traces $t_1, ..., t_n$ for a path set, the resulting predicate

would be:

`if ((trace &` $mask_1$`) ==` obj_i `|| ... || (trace &` $mask_n$`) ==` obj_n)

For each trace t_i, $mask_i$ is obtained by setting all bits which are $\times$ in t_i to 0 and others to 1, and obj_i equals $mask_i$ `&` t_i.

2.1.6.3 Inserting state saving statements

The other instrumentation in the forward function are state saving statements, which are inserted according to the state saving edges in the route graph. For each state saving edge in the route graph, suppose the variable to store is *var* and the path set on this edge is P. Our task is finding one or several locations to store *var* according to the path set P, ensuring that *var* is only saved once for each CFG path in P.

To find such locations, we first compute the corresponding path traces T of P from Algorithm 3. For each trace in T, we traverse the CFG from the entry. When we reach a branch node, check the corresponding bit in the trace: fall through the true edge if the bit is 1, false edge if the bit is 0. If the bit is $\times$, the traversal forks and that bit is assigned to 0 and 1 respectively forming two new traces; and for each concretized trace the descent continues. The descent stops immediately when all bits which are not checked in the trace are $\times$. After this process, we obtain one or more locations where the descent has stopped. In each location we find a point where the definition of *var* is reachable and a state saving statement is inserted there. However, it is possible that the path set containing the paths passing through this location is larger than the one on which the state saving is needed. In this case, we guard the state saving statement with a branch whose predicate corresponds to the trace at this location.

2.1.6.4 Building a CFG for the reverse function

We build the CFG for the reverse function from a route graph; the reverse CFG is acyclic and each path in it must obey the data dependencies represented in the route graph. Each outgoing edge from a value node in the route graph will be translated to a statement in the reverse function.

There could be a large number of correct reverse CFGs for a route graph, resulting in different control flows and different numbers of branches. We choose to build a structured CFG to simplify the translation to source code. We also attempt to minimize the number of predicates in the CFG.

There are three kinds of statements that can be generated from a route graph:

- An operator node with its operands and result induces an operation statement, such as `a = b + c`.
- An edge with value nodes as both ends induces an assignment statement.
- An edge pointing to the SS node induces an value restoration statement.

The statements generated from route graph edges retain the path sets attached to the corresponding edges. We build basic blocks of statements that all share the same path sets, and insert branches so that each basic block is executed when the corresponding path is taken in the forward function. While enforcing the path set constraints ensures correct control flow, producing correct data flows depends on the order in which statements are inserted in the CFG. Note that a route graph corresponds to explicit data dependencies, and for each CFG path in the forward function it is acyclic due to property V from section 2.1.4. Hence, if we order statements in the reverse topological order of the route graph edges, dataflow dependencies are correctly maintained.

Algorithm 4 shows how to build a CFG for the reverse function. We keep a set of basic blocks, `openBlocks`, to which new statements can be appended. We

Algorithm 4: Generating a CFG for the reverse function from a route graph.

```
GenerateReverseCFG(routeGraph)
begin
    cfg ← ∅, pendingStmts ← ∅, openBlocks ← ∅, pathSetPairs ← ∅
    foreach valNode in routeGraph do valNode.pathSet ← ∅
    foreach available node availNode in routeGraph do
        BuildReadyStatements(availNode, 𝒰, pendingStmts)
    cfg.entry ← BuildBasicBlock(𝒰)
    while pendingStmts ≠ ∅ do
        if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet = b.pathSet then
            Append s to b
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet ⊂ b.pathSet then
            Append to b a branch, with the predicate generated from s.pathSet
            b1 ← BuildBasicBlock(s.pathSet)
            Append s to b1
            b2 ← BuildBasicBlock(b.pathSet − s.pathSet)
            Insert into cfg edges from b to b1 and b2 with labels true and false
            Add ⟨b1.pathSet, b2.pathSet⟩ to pathSetPairs
            openBlocks ← openBlocks − {b}
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃b1, b2 ∈ openBlocks, and ⟨b1.pathSet, b2.pathSet⟩ ∈ pathSetPairs then
            b ← BuildBasicBlock(b1.pathSet ∪ b2.pathSet)
            Insert into cfg two edges, from b1 and b2 to b
            pathSetPairs ← pathSetPairs − {⟨b1.pathSet, b2.pathSet⟩}
            openBlocks ← openBlocks − {b1, b2}
            if |openBlocks| = 1 then break
    return cfg

BuildReadyStatements(valNode, nodeAvailablePaths, pendingStmts)
begin
    valNode.pathSet ← valNode.pathSet ∪ nodeAvailablePaths
    foreach edge ∈ InEdges(valNode) do
        if edge.pathSet ⊆ valNode.pathSet then
            if edge.source is an operation node then
                Set edge to be a available for edge.source
                if all operands of edge.source are available then
                    Add to pendingStmts the statement for for edge.source, with path set
                    edge.pathSet
            else
                Add to pendingStmts the statement for edge, with path set edge.pathSet

BuildBasicBlock(pathSet) begin
    Build an empty basic block b and attach path sets pathSet to it.
    cfg ← cfg ∪ { b },    openBlocks ← openBlocks ∪ { b }
    return b
```

also maintain a set of statements, `pendingStmts`, whose data dependencies have been satisfied, but which have not yet been inserted in the CFG. Each basic block has an associated path set; these are the paths in the forward function for which the corresponding basic block in the reverse function should execute. Similarly, each statement has a set of paths from the forward function. If there is a pending statement and an open basic block whose path sets match, we simply append the statement to the basic block. When a statement is inserted into the CFG, the data dependencies of new statements may now be satisfied; we call the function `BuildReadyStatements` to generate the statements that are now valid for insertion. If there is no pending statement whose path set matches the path set of an open basic block, we must insert or join a branch in the CFG. When a branch is inserted, two new basic blocks are created and the basic block containing the branch is closed. When a branch is joined, the joined basic blocks are closed and a new open basic block is created.

Note that it is possible that the instrumentation to the forward function brings additional implicit data dependencies. For example, if stack is used for state saving the order of values popped in the reverse function should be opposite of the order of pushes in the forward function. In this case, we can order those state saving statements in `pendingStmts` according to the order in which values are pushed.

2.1.6.5 Generating code

The forward function is generated by copying the target function and adding state saving and control flow instrumentation (section 2.1.6.2). The reverse function is translated from the CFG built by Algorithm 4. Translating a structured CFG to source code is straightforward. Since each variable in the reverse CFG is in SSA form, we can use the versioned name during code generation. Because our framework generates source code that is later compiled with another compiler, the redundant variables will be optimized away; the only drawback of this approach is readability. If

readability is an issue, we can compute data dependencies in the reverse CFG and then remove versions attached to variables where this does not affect data dependencies. After version removal, we would also remove self-assignment statements such `a = a`. Figures 1(b) and 1(c) show the generated forward and reverse functions from the code in Figure 1(a).

2.2 Handling programs with loops

Unmodified, our prior method as described in Section **??** cannot handle loops for two key reasons. First, a loop results in cyclic paths in the CFG, whereas our prior analysis relies on paths being acyclic. Acyclic paths make it easy to check that the reverse program restores any desired input value no matter what path the forward program takes. Secondly, our prior VSG and RG cannot represent loop control structure. Therefore, it is simply not possible to synthesize, for example, a loop in the reverse code from the RG. Nevertheless, we *can* reuse most of the prior method by decomposing the problem suitably. In particular, we keep the basic framework of "SSA to VSG to RG." Our extension replaces SSA with a loop-enabled variant, and then extends our VSG and RG representations and algorithms to deal with cycles, thereby addressing the two aforementioned issues.

Let us first assume that each loop to be reversed is a single-entry, single-exit while loop (we will explain what is a while loop later). We explain in Section 2.2.2 how to convert other kinds of loops into this form. We also assume that each loop must terminates at run-time so that we can always get an output. Given an input while loop, there are three steps to build a VSG.

1. We temporarily collapse each while loop into a single abstract node in the CFG, thereby creating a logically loop-free CFG from which we can build a VSG by directly applying our prior method. This "transformation" is for program analysis purposes only. We denote this loop-collapsed VSG by G_P.

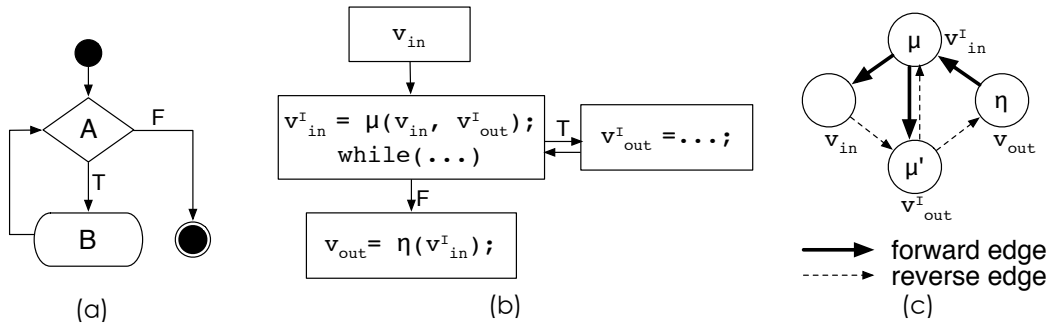

Figure 5: (a) The diagram of a while loop. (b) The CFG in loop-closed SSA form for a variable v modified in the loop. (c) Forward and reverse edges.

2. Similarly, we directly apply our prior method to build a VSG for each loop *body*, which may be treated as another loop-free program. (If the body contains nested loops, these are similarly collapsed as in Step 1 above.) Note that path information in these loop body VSGs are local to the loop body. We denote this VSG for the loop body by G_L.

3. At this point, G_P and G_L are disconnected. Therefore, we introduce new special edges to connect them, thereby resulting in a single connected VSG. These connecting edges are a new type of edge and constitute the main extension to our prior VSG in order to support loops. The new edges connect each input (or output) of a loop to the input (or output) of the loop's body. These new edges serve as markers: when we search the VSG and produce an RG containing these edges, then we know we need to synthesize a loop.

Since Steps 1 and 2 use our prior VSG construction, we need not discuss them further here. What changes is the third step, as detailed below, including new VSG searching rules and new procedures for synthesizing loops from the search result (i.e., the RG).

2.2.1 Dealing with while loops

We first consider a while loop with the diagram shown in Figure 5(a). We further assume that A has no side-effects and that there are no escapes from B. Thus, the

loop only exits from its entry.

Given such a while loop, we transform it into the *loop-closed SSA form* [**?**], illustrated in Figure 5(b). Loop-closed SSA differs from conventional loop-free SSA as follows. In conventional SSA, a special marker called a ϕ *function* is placed in the CFG at the first program point where two distinct versions (definitions) of a variable, computed along different program paths, meet. In loop-closed SSA, if a value is defined inside of a loop and used outside of it, we place a special single entry ϕ function at the *exit* of the loop. To distinguish this type of loop-specific ϕ function from a conventional ϕ function as used in loop-free programs, we denote the loop-specific form by the term η function, by convention [**?**]. Additionally, suppose a definition of a variable from outside the loop and a definition coming from a back-edge of the loop meet at a program point. Again, we create a ϕ function marker here, and to distinguish it, we refer to it as a μ function.

To see how these markers work, consider a variable v modified by a while loop; we now describe the corresponding loop-closed SSA form, which Figure 5(b) illustrates. Let v_{in} denote the input value of v before the loop executes, and v_{out} the output value of v after the loop executes. Next, let the input to the loop *body* be v^I_{in} and the output v^I_{out}. (The superscript I is intended to remind the reader that these are values associated with an *iteration* of the loop, as opposed to the values before and after the loop.) Then, v^I_{in} is defined by a μ function as $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$, and v_{out} is defined by a η function as $v_{\text{out}} = \eta(v^I_{in})$. That is, $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$ indicates the program point at which v has either the initial value before the loop executes or the value produced by some iteration of the loop; and $v_{\text{out}} = \eta(v^I_{in})$ indicates the program point at which v has the final value once the loop completes.

From this loop-closed SSA form, we wish to build a VSG that will express equality relations among the four SSA values, v_{in}, v_{out}, v^I_{in}, and v^I_{out}. This VSG result is shown in Figure 5(c). Recall that nodes in the VSG represent values, and edges

the equality relations. There are four value nodes. The nodes v_{in} and v_{out} are part of the loop-collapsed G_P, and v^I_{in} and v^I_{out} belong to the loop body's G_L. The μ and η functions indicate how to connect G_P and G_L. In particular, the three solid bold edges are associated with the dependences induced by executing the loop in the forward direction; we call these the *forward edges*, and a μ node is incident to all three. The presence of these edges make it possible to obtain v_{out} by some path passing through G_L, and simultaneously indicate that a loop is present for subsequent code generation. Similarly, the three dashed edges are *reverse edges* associated with dependences induced in the reverse direction. These edges make it possible to obtain v_{in} by some path through G_L. Note that the reverse edges form a symmetry to the forward edges. From this symmetry, we define the node incident to all three reverse edges as a μ' node. Later we will show how the search traverses these edges.

Having built the CFG, the next step is to search it, producing the RG result. Recall from Section **??** that we are given a set of target nodes whose values we wish to eventually compute from a starting set of available nodes. We search for a path from available nodes to target nodes; the subgraph representing paths is the RG, which is not necessarily unique. Our algorithm is similar to the one we have described previously [**?**], but for loops we need three additional search rules:

- During a search for a value, once a forward/reverse edge is selected, all edges in the other category cannot be chosen. This is because either a forward or a reverse loop will be built to retrieve the value.

- When the search reaches a μ or μ' node, it will be split into two sub-searches, in G_P and G_L, respectively, through the two outgoing forward or reverse edges. For example, in Figure 5(c), if the search reaches v^I_{in}, the algorithm begins two sub-searches beginning with v_{in} and v^I_{out}.

- During the search, the algorithm may form a directed cycle only in G_L; furthermore, such a cycle must contain a forward or reverse edge between a μ and μ' node. Once a cycle is formed, the search in G_L is complete.

We build a while loop as either a forward or a reverse loop. Synthesizing such a while loop consists of synthesizing its body and predicate.

2.2.1.1 Building the loop body.

The loop body in the reverse program is generated from the search result in G_L. For each variable we remove the edge between the μ and μ' nodes and hence remove the cycles, so that we can generate the loop body using our prior code generation algorithm [**?**].

2.2.1.2 Building the loop predicate.

To guarantee that the generated loop has the same iterations at runtime as the original loop, we need to build a proper loop predicate. We propose three approaches to building a correct loop predicate. To illustrate those approaches, we temporarily introduce the following loop example. We assume that the omitted statements modify neither `A[]` nor `i`.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 1:** Building the same loop predicate as that in the original loop. To build this predicate, we need to retrieve each value in the predicate. A new search is needed to acquire those values, and the search result will be combined into the RG generated above. For the example above, we can build a loop

below that has the same number of iterations as the original one. The omitted statements will be substituted by the loop body built above.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 2:** Building the loop predicate from a variable updated in the loop. Given a variable v and its four definitions: v_{in}, v^I_{in}, v^I_{out}, and v_{out}, if $v^I_{\text{in}} \neq v_{\text{out}}$ in each iteration except the last definition of v^I_{in} (which is actually v_{out}), and if we can retrieve v_{in} and v_{out} before the loop (hence we cannot retrieve them through the loop), and v^I_{out} in the loop, we can use them to build a while loop as:

$$u := v_{in};\ while(u \neq v_{out})\ \{\ /*\ update\ u\ */\ \}$$

Similarly, if $v^I_{\text{out}} \neq v_{\text{in}}$ in each iteration, and v_{in} and v_{out} can be retrieved before the loop, and v^I_{in} can be retrieved in the loop, we can use them to build a while loop as:

$$u := v_{out};\ while(u \neq v_{in})\ \{\ /*\ update\ u\ */\ \}$$

In general, it is difficult to detect all variables satisfying the properties above. However, there are some special cases. One case is that of *monotonic variables* [**?**], which are monotonically strictly increasing or decreasing in each iteration. Another is that of induction variables, which are special monotonic variables that are relatively easier to recognize. In the above example, `i` is an induction variable. Assume its final value after the loop is `i1` that is known, and then we can build the following loop with the predicate using `i`.

```
i = 0;
while (i != i1) {
    /* ... */
    i = i + 2;
}
```

- **Approach 3:** Instrumenting the original loop with a counter counting the number of iterations. The counter has the initial value zero and is incremented by one on each back edge of the loop. The final value of the counter is stored in the forward program and restored in the reverse program as the maximum value of another loop counter. This approach generally works if either of the above two approaches fail. However, it requires instrumentation (the counter), and therefore forces generation of a forward program. Below we show the instrumented loop in the forward program (left) and the generated loop in the reverse program (right) for the above example.

```
i = count = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
    count = count + 1;
}
store(count);
```

```
restore(count);
while (count > 0) {
    /* ... */
    count = count - 1;
}
```

We prioritize these approaches as follows. Applicability and state-saving cost are our main criteria. We prefer Approach 1 and 2 over 3. When either 1 or 2 apply, if no state-saving is required, we apply them. Otherwise, we try Approach 3 and choose the overall approach with the least cost.

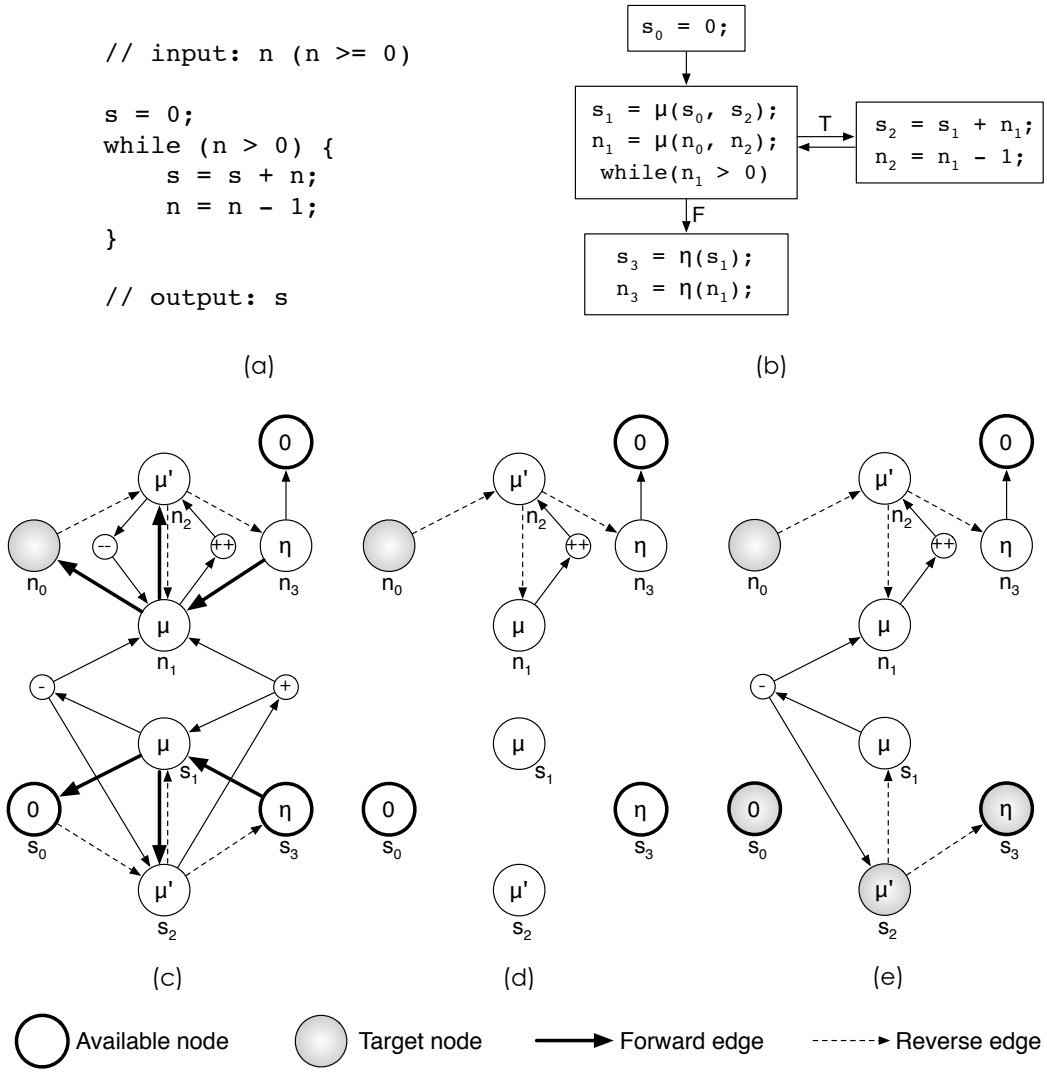

Figure 6: (a) The program of our example. (b) The CFG in loop-closed SSA form. (c) The VSG. (d) The RG for retrieving n_3. (e) The RG for retrieving n_0 and s_2.

As an example, suppose we apply this algorithm to the loop in Figure 6(a). Figure 6(b) shows its CFG in loop-closed SSA. The input is n_0 and the output s_3. Our goal is to generate a reverse program that takes s_3 as input and produces n_0. We build the VSG shown in Figure 6(c), with forward and reverse edges shown as bold and dashed edges, respectively. Note that the equality between n_3 and 0 is acquired from solving constraints, a standard compiler technique, as discussed in Section **??**.[4] The search result for value n_0 is shown in Figure 6(d), from which we can build the loop body as { `n = n + 1;` }.

Next, we build the loop predicate. In our example, because we wish to retrieve the initial value of n, we cannot use it to build the loop predicate. We can discover that s is a monotonic variable, and that both the initial and final values of s, which are 0 and s_3, respectively, are available. To get s_2, we search its value on the VSG and the search result is shown in Figure 6(e). As a result, we build the loop predicate from s and the reverse program is generated as below.

```
n = 0;
while (s != 0) {
    n = n + 1;
    s = s - n;
}
```

2.2.2 Dealing with loops other than while loops

In practice, the vast majority of loops have a single entry, which are called *natural loops* [**?**]. Loops with more than one entry are quite rare and can in fact be transformed into natural loops [**?**]. However, it is quite common that a loop has several exits. For example, in C/C++ we may exit a loop early through `break`, `return`, or `goto` statements. Nevertheless, given a non-while natural loop, we can transform

[4]For clarify, we remove the equality $n_1 = s_2 - s_1$, as this relation will not be used during the search.

it to separate the last iteration from the loop; then, the remaining iterations form a new while loop, and the last iteration will not belong to the loop and hence can considered with the control flows outside of the loop. We then process the new while loop as previously described. Note that this "transformation" is only applied to the CFG during the analysis, and not to the original program. As such, in the forward program P^+ the last iteration and other iterations of each loop continue to share the same code.

Figure 7(a) shows a loop in a CFG, with a header (node 1) and two back edges (4→1 and 5→1). There are two different exits from this loop, which are nodes 6 and 7. Figure 7(b) shows the CFG of the transformed loop. This transformation is performed as follows.

In a natural loop, only the last iteration takes the exit, and any other iteration goes back to the loop header. Therefore, if the last iteration is peeled off from the loop, this loop will turn into a while loop. To implement this transformation, we create a new branch node with an unknown predicate that returns *true* if the next iteration is not the last one and $false$ otherwise. Note that we will not build this predicate in the forward program. The new branch node turns over all in-edges of the loop header. Its *true* labeled out-edge will point to the loop header of a copy of the loop (node $1'$) with back edges but without exit edges, and all back edges are redirected to this new branch node making it a new loop header. Note that after removing exit edges it is possible that a previous branch node becomes a non-branch node (node $3'$, for example), which is fine because the removed branch edge will not be taken. Then, we can remove the (side effect free) predicates from those nodes. The edge labeled with $false$ from the new branch node will point to the original loop header (node 1) and all back edges in the original loop are removed, since the last iteration won't take the back edge. The nodes from which the exit of the program is not reachable due to the back edge removal are removed (node 4 and 5, for example).

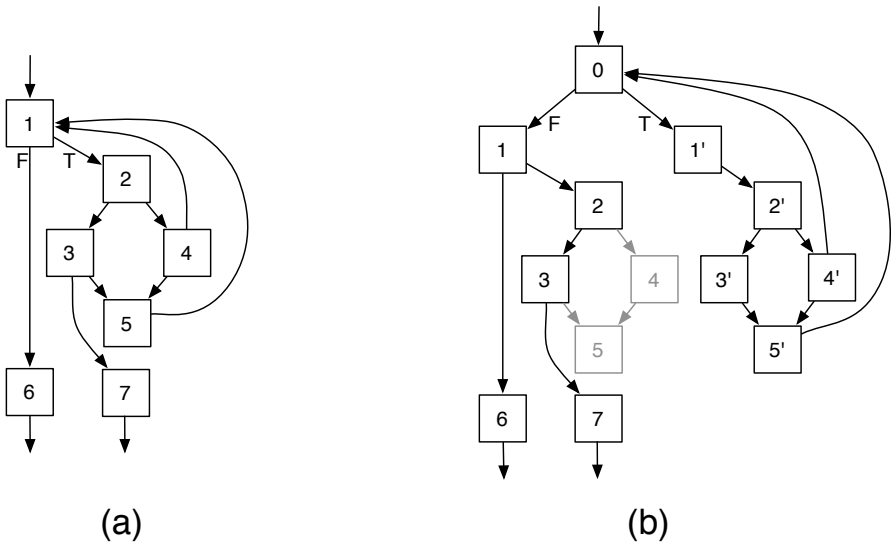

Figure 7: (a) A loop in CFG with two back edges and two exits. (b) The CFG of the transformed loop.

Again the predicate is removed from a node once it is not a branch node anymore (node 2 and 3).

After the transformation, all loops in the program become while loops and our method applies.

CHAPTER III

SYNTHESIS FOR PROGRAMS WITH ARRAYS

CHAPTER IV

CONCLUSION

APPENDIX A

SOME ANCILLARY STUFF

Ancillary material should be put in appendices, which appear just before the bibliography.

Thesis proposal: Automated synthesis for program inversion

Perry H. Disdainful

33 Pages

Directed by Professor Richard Vuduc

This is the abstract that must be turned in as hard copy to the thesis office to meet the UMI requirements. It should *not* be included when submitting your ETD. Comment out the abstract environment before submitting. It is recommended that you simply copy and paste the text you put in the summary environment into this environment. The title, your name, the page count, and your advisor's name will all be generated automatically.