Bebop: A Path-sensitive Interprocedural Dataflow Engine

Thomas Ball
Microsoft Research
One Microsoft Way
Redmond, WA 98052
tball@microsoft.com

Sriram K. Rajamani
Microsoft Research
One Microsoft Way
Redmond, WA 98052
sriram@microsoft.com

ABSTRACT

Flow-sensitive data analyses can lose precision because they assume that all paths in a control-flow graph are executable (feasible). Path-sensitive dataflow analyses can rule out infeasible paths by tracking correlations between dataflow facts. To track such correlations, in general, requires recording a set of sets of facts per statement in a program. Naive representation of such sets can lead to a very high memory consumption and running time.

We reformulate an interprocedural dataflow algorithm by Reps, Horwitz and Sagiv (based on context-free graph reachability) into a traditional interprocedural flow-sensitive dataflow algorithm. We then show how to use Binary Decision Diagrams (BDDs), a data structure from the model checking community, to turn this reformulated algorithm into an interprocedural path-sensitive dataflow analysis algorithm that tracks a set of set of facts per program statement. We have implemented this algorithm in a tool called `Bebop`.

1. INTRODUCTION

The tradeoffs between precision and efficiency in program analysis are well-known: an analysis can be made more precise by recording more detailed information at each program location, at a higher cost in space and time. Flow-insensitive analyses record global facts that hold for all statements in a program [8] but may lose precision since they do not consider the order of statement execution. Flow-sensitive analyses record facts on a per-statement basis [1], but may lose precision because they (traditionally) assume that all paths in a control-flow graph of a program are executable (feasible). Path-sensitive analyses record facts on a per-path basis, using branch correlations to eliminate infeasible paths, but must be applied carefully due to their potentially high cost [2, 5]

We present a general algorithm for *path-sensitive* interprocedural *finite* dataflow analysis. By "finite", we mean that the dataflow domain is a finite set D of facts. A flow-sensitive finite dataflow analysis keeps one set of facts per vertex in the control-flow graph. A path-sensitive finite analysis keeps a set of sets of facts per vertex in the control-flow graph, enabling it to maintain correlations between the facts (such as "if d_1 is present then d_2 is not present") and rule out infeasible paths.

We have built an interprocedural path-sensitive dataflow analysis tool called `Bebop` [4]. `Bebop` uses Binary Decision Diagrams [7] (BDDs), a data structure used in model checking tools [10], to symbolically represent dataflow transfer functions and the set of sets of facts at each vertex in the control-flow graph.

Our results are as follows:

- *Sections 2 and 3.* We show how an existing interprocedural flow-sensitive dataflow algorithm by Reps, Horwitz and Sagiv (RHS) based on context-free language reachability in graphs [11] can be reformulated as a traditional dataflow algorithm, similar to the "functional approach" of Sharir and Pnueli [13]. The RHS algorithm is applicable to *interprocedural, finite, distributive, subset problems (IFDS problems)*. These are problems that have a finite set D of dataflow facts and distributive dataflow functions (of type **set-of** $D \rightarrow$ **set-of** D). The reformulated algorithm, the SP_{rhs} algorithm, has exactly the same time complexity and precision as the RHS algorithm.

- *Section 4.* We generalize the SP_{rhs} algorithm to make it path-sensitive. Using BDDs, we lift the level of reasoning from a set of dataflow facts D to a set of sets of dataflow facts. The new algorithm permits dataflow problems having arbitrary flow functions of type **set-of** $D \rightarrow$ **set-of** D. We mention the relationship between BDDs and so called "tri-vectors" used in related work on three-valued program analysis [12].

- *Section 5.* We present *boolean programs* as a natural target language for expressing path-sensitive finite dataflow analyses and describe the tool `Bebop` that implements the path-sensitive dataflow algorithm using BDDs to represent sets of sets of facts. We encode the analysis problem of whether or not a top-level pointer in a C program is NULL or non-NULL using boolean programs.

Section 6 discusses other related work and Section 7 points to future work.

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA..

2. BACKGROUND: THE RHS ALGORITHM

In this section, we review the Reps/Horwitz/Sagiv (RHS) algorithm for performing interprocedural flow-sensitive dataflow analysis, which reduces a dataflow analysis to a reachability problem over an "exploded graph" representation.

Although the RHS algorithm is an interprocedural analysis problem, the basic insight that leads to our reformulation can be explained in the context of a single procedure program, and is applicable to the algorithm in its full generality. The reader should bear this in mind, as the algorithm presented here is simply a modified depth-first search algorithm. The material in this section is a simplified version of Sections 2 and 3 from [11].

Let P be a single-procedure program with a control-flow graph G. An instance IP of an IFDS problem is a five-tuple, $IP = \langle G, D, F, M, \sqcap \rangle$, where

- $G = \langle V_G, E_G \rangle$ is a control-flow graph consisting of vertices V_G and edges E_G;
- D is a finite set;
- $F \subseteq$ **set-of** $D \rightarrow$ **set-of** D is a set of distributive dataflow functions;
- $M : E \rightarrow F$ is a map from G's edges to dataflow functions;
- The meet operator $\sqcap$ is either set union or intersection.

As in [11], we assume (without loss of generality) that the meet operation is set union.

Let $IP = \langle G, D, F, M, \sqcap \rangle$ be an IFDS problem instance and $p = [e_1, e_2, \cdots, e_n]$ be a directed path in G. The path function M_p that defines the meaning of p under IP is simply the composition of the dataflow functions associated with the edges of p (as defined by M):

$$M_p = M(e_n) \circ M(e_{n-1}) \circ \cdots \circ M(e_1)$$

Given a CFG G with entry vertex *entry*, let $Paths(G)$ be the set of all paths in G that start with *entry*, and let $Paths(G, v)$ be paths in $Paths(G)$ that end with the vertex v. Given a vertex v in the CFG G and a dataflow fact $d \in D$, the *conditional meet-over-all-paths*[1] (*CMOP*) solution to IP is defined as follows:

$$CMOP_{\langle v,d \rangle} = \bigsqcap_{p \in Paths(G,v)} M_p(\{d\})$$

Two of the major insights behind the RHS algorithm are: (1) each function in F can be represented by a bi-partite graph (or *representation relation*) with $2(D+1)$ vertices and at most $(D+1)^2$ edges, and (2) the composition of functions in F also can be represented in this manner. The representation relation of a dataflow function f, $R_f \subseteq (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\})$, is a binary relation defined as follows:

$$\begin{array}{rcl} R_f & = & \{\langle \Lambda, \Lambda \rangle\} \\ & \cup & \{\langle \Lambda, y \rangle \mid y \in f(\emptyset)\} \\ & \cup & \{\langle x, y \rangle \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\} \end{array}$$

[1]We use the *CMOP* problem rather than the "meet-over-all-paths" problem used in [11] in order to "simulate" the demand for summary information in a single procedure setting.

```
global
PE : set-of (V_G' × V_G')
Worklist : set-of (V_G' × V_G')

procedure Propagate(e : (V_G' × V_G'))
begin
  if e ∉ PE then
    PE := PE ∪ {e}
    Worklist := Worklist ∪ {e}
  fi
end

procedure CMOP_RHS(S : set-of D)
begin
  PE := {⟨entry, d⟩ → ⟨entry, d⟩ | d ∈ S}
  Worklist := PE
  while Worklist ≠ ∅ do
    select and remove edge ⟨entry, d1⟩ → ⟨v2, d2⟩ from Worklist
    for each ⟨v3, d3⟩ such that ⟨v2, d2⟩ → ⟨v3, d3⟩ ∈ E_G' do
      Propagate(⟨entry, d1⟩ → ⟨v3, d3⟩)
    od
  od
end
```

Figure 1: The RHS algorithm, restricted to the case of a single-procedure program.

As a result of these insights, it is possible to transform the *CMOP* problem over IP into a pure graph reachability problem, as follows. Let $IP = (G : \langle V_G, E_G \rangle, D, F, M, \sqcap)$ be an IFDS problem instance. The *exploded graph* for IP, denoted by $G' = \langle V_{G'}, E_{G'} \rangle$ has vertex and edge sets:

$$\begin{array}{rcll} V_{G'} & = & V_G \times (D \cup \{\Lambda\}) & \\ E_{G'} & = & \{\langle v_1, d_1 \rangle \rightarrow \langle v_2, d_2 \rangle \mid & v_1 \rightarrow v_2 \in E_G \text{ and} \\ & & & \langle d_1, d_2 \rangle \in R_{M(v_1 \rightarrow v_2)} \quad \} \end{array}$$

The basic result from [11] is that $d_2 \in CMOP_{\langle v,d_1 \rangle}$ iff there is a path in G' from the node $\langle entry, d_1 \rangle$ to the node $\langle v, d_2 \rangle$. This is an example of a *single-source, single-sink* reachability problem, which can be solved in time linear in the size of G' (via a depth-first search). To answer the *CMOP* question for all $d \in S$ ($S \subseteq D$) and $v \in V$ requires solving a *multi-source, multi-sink* reachability problem over G', which the RHS algorithm solves using the concept of *path edges*, as shown in Figure 1. A path edge is an edge of the form $\langle entry, d_1 \rangle \rightarrow \langle v, d_2 \rangle$. Since it is assumed that d holds at *entry* for all $d \in S$, the RHS algorithm initializes the set of path edges PE to have self-loops encoding this initial information. The work list is initialized with these initial set of path edges. While the worklist is not empty, the algorithm removes a path edge $\langle entry, d_1 \rangle \rightarrow \langle v_2, d_2 \rangle$ from it. For every edge in the graph G' of the form $\langle v_2, d_2 \rangle \rightarrow \langle v_3, d_3 \rangle$, the algorithm "propagates" a new path edge $\langle entry, d_1 \rangle \rightarrow \langle v_3, d_3 \rangle$. Upon termination of the algorithm we have that

$$\forall v \in V_G,\ \forall d_1 \in S,\ \forall d_2 \in D : \\ d_2 \in CMOP_{\langle v,d_1 \rangle} \Leftrightarrow \langle entry, d_1 \rangle \rightarrow \langle v, d_2 \rangle \in PE$$

The running time of this algorithm is $O(E \times D^3)$, as the size of the exploded graph is $O(E \times D^2)$ and the algorithm is essentially performing a (simultaneous) depth-first search of this graph for each of the facts in S (which is size $O(D)$). For the interprocedural case, the running time is $O(E \times D^3)$ as well.

```
global
PE': V_G → set-of (D × D)
Worklist: V_G → set-of (D × D)

procedure Propagate(v : V_G, p : (D × D))
begin
  if p ∉ PE'(v) then
    PE'(v) := PE'(v) ∪ {p}
    Worklist(v) := Worklist(v) ∪ {p}
  fi
end

procedure CMOP_{SP_rhs}(S : set-of D)
begin
  PE'(entry) := {⟨d, d⟩ | d ∈ S}
  Worklist(entry) := PE'(entry)
  while ∃v_2 s.t Worklist(v_2) ≠ ∅ do
    select and remove ⟨d_1, d_2⟩ from Worklist(v_2)
    for each v_2 → v_3 ∈ E_G do
      for each d_3 ∈ M(v_2 → v_3)({d_2}) do
        Propagate(v_3, ⟨d_1, d_3⟩)
      od
    od
  od
end
```

Figure 2: The SP_{rhs} algorithm.

```
global
PE': V → set-of (set-of D × set-of D)
Worklist: V_G → set-of (set-of D × set-of D)

procedure Propagate(v : V_G, p : (set-of D × set-of D))
begin
  if p ∉ PE'(v) then
    PE'(v) := PE'(v) ∪ {p}
    Worklist(v) := Worklist(v) ∪ {p}
  fi
end

procedure CSMOP_{SP_rhs}(S' : set-of (set-of D))
begin
  PE'(entry) := {⟨S, S⟩ | S ∈ S'}
  Worklist(entry) := PE'(entry)
  while ∃v_2 s.t Worklist(v_2) ≠ ∅ do
    select and remove ⟨S_1, S_2⟩ from Worklist(v_2)
    for each v_2 → v_3 ∈ E_G do
      let S_3 = M(v_2 → v_3)(S_2) in
        Propagate(v_3, ⟨S_1, S_3⟩)
      ni
    od
  od
end
```

Figure 3: The SP_{rhs} algorithm, lifted to reason about the powerset of D.

3. THE SP_{rhs} ALGORITHM

Our insight is that path edges, *regardless of whether or not we are considering the intraprocedural or interprocedural version of the RHS algorithm*, always have the form

$$\langle entry, d_1 \rangle \rightarrow \langle v, d_2 \rangle$$

where *entry* is the entry vertex of a procedure P's control-flow graph and v is a vertex in P's control-flow graph. Therefore, we can represent path edges on a per procedure basis as a set of triples

$$\{\langle d_1, v, d_2 \rangle \mid \langle entry, d_1 \rangle \rightarrow \langle v, d_2 \rangle \in PE\}$$

Taking this an additional step further, we can partition this set on the basis of the second component v to get a set of pairs $PE'(v)$ for each vertex v, where

$$PE'(v) = \{\langle d_1, d_2 \rangle \mid \langle entry, d_1 \rangle \rightarrow \langle v, d_2 \rangle \in PE\}$$

As a result, it is not necessary to build the exploded supergraph explicitly in order to solve the *CMOP* problem. Rather, we can perform a traditional dataflow analysis in which each vertex v in the original control-flow graph collects a set of pairs of dataflow facts $PE'(v)$, as shown in the SP_{rhs} algorithm of Figure 2.

In the SP_{rhs} algorithm, the worklist is a map from a vertex $v \in V$ to a set of pairs of dataflow facts, representing the set of path edges associated with v that have yet to be processed. While there is a non-empty $Worklist(v_2)$, a pair of facts $\langle d_1, d_2 \rangle$ is removed from $Worklist(v_2)$. Together, the vertex v_2 and the pair $\langle d_1, d_2 \rangle$ represents the path edge $\langle entry, d_1 \rangle \rightarrow \langle v_2, d_2 \rangle$. In the RHS algorithm, there was one **for** loop that visited the successors $\langle v_3, d_3 \rangle$ of $\langle v_2, d_2 \rangle$. In the new algorithm, there are two **for** loops to achieve the same result: the outermost iterates over the successors v_3 of v_2 and the second iterates over the dataflow facts $d_3 \in M(v_2 \rightarrow v_3)(\{d_2\})$. The Propagate procedure is called with two arguments: the vertex v_3 and the pair dataflow facts $\langle d_1, d_3 \rangle$, which together represent the path edge $\langle entry, d_1 \rangle \rightarrow \langle v_3, d_3 \rangle$. The action of the Propagate procedure is as before (but parameterized with respect to vertex v).

It is straightforward to see that the SP_{rhs} algorithm is equivalent to the RHS algorithm (given our interpretation of the set of path edges PE in terms of the $PE'(v)$ sets that the SP_{rhs} algorithm computes). Furthermore, it is easy to see that the Propagate procedure is called the same number of times in the SP_{rhs} and RHS algorithms, and therefore that the SP_{rhs} algorithm has the same time complexity as the RHS algorithm.

The relationship of the SP_{rhs} algorithm to the "functional approach" of Sharir and Pnueli [13] is quite direct. Sharir and Pnueli define a set of functions $\phi_{(entry,n)}$ of type **set-of** $D \rightarrow$ **set-of** D, where *entry* is the entry vertex of a procedure and n is a vertex in the control-flow graph of the same procedure. This function represents the set of facts D_2 that hold at vertex n given that the set of facts D_1 hold at vertex *entry* (i.e., $D_2 = \phi_{(entry,n)}(D_1)$). This is exactly the function represented point-wise by the relation $PE'(n)$.

4. ADDING PATH-SENSITIVITY TO THE SP_{rhs} ALGORITHM

In this section, we show how to generalize the SP_{rhs} algorithm to solve the *conditional-subset meet-over-all-paths* problem, which is the lifting of the *CMOP* problem to apply to arbitrary subsets of D (rather than single facts of D). This allows the algorithm to track correlations between dataflow facts (elements of D), making it path-sensitive. This is useful regardless of whether or not the transfer functions in F are distributive or non-distributive. We then show how to use Binary Decision Diagrams (BDDs) to implicitly represent these sets.

4.1 The *CSMOP* Problem

Given a vertex v in the CFG G and a set $S \subseteq D$, the *conditional-subset meet-over-all-paths* (*CSMOP*) solution to IP is defined as follows:

$$CSMOP_{\langle v,S\rangle} = \bigsqcap_{p \in Paths(G,v)} M_p(S)$$

We desire to solve the *CSMOP* problem for a set S' of subsets of D and all $v \in V$. Figure 3 shows the SP_{rhs} algorithm, generalized to solve the *CSMOP* problem. Note that the algorithm is structurally identical to the algorithm given in Figure 2. We have simply lifted the domain of discourse to the powerset of D (that is, we have replaced every occurrence of "D" in a type by "**set-of** D"). Note that the second argument to the dataflow meaning function M now is a set of facts (S_2) rather than a singleton set ($\{d_2\}$). As the number of subsets of D is 2^D, the worst-case complexity of the algorithm is $O(E \times (2^D)^3)$. For the interprocedural case, the worst-case complexity is $O(E \times (2^D)^3)$.

4.2 Implementing *CSMOP* with Binary Decision Diagrams

To implement the *CSMOP* algorithm, we require a data structure to compactly store a set of pairs of subsets of D (i.e., a set of bit vectors of length $2D$) in order to represent a set of path edges (an element of **set-of** (**set-of** $D \times$ **set-of** D)). A Binary Decision Diagram [7] (BDD) is an acyclic graph data structure for implicitly representing a set of bit vectors of fixed length, and so is a good match for our problem. We give a brief review of BDDs. Consider the boolean function $f(x, y, z) = (x = y)$. This function represents the set of bit vectors $\langle x, y, z\rangle$ for which x and y are equal:

$$\{\ \langle \mathbf{0},\mathbf{0},\mathbf{0}\rangle,\ \ \langle \mathbf{0},\mathbf{0},\mathbf{1}\rangle,\ \ \langle \mathbf{1},\mathbf{1},\mathbf{0}\rangle,\ \ \langle \mathbf{1},\mathbf{1},\mathbf{1}\rangle\ \ \}$$

We can represent this function (a set of bit vectors, or set of set of facts) as a tree, as shown in Figure 4(a). The tree has 8 leaves corresponding to the eight possible valuations of the boolean variables x, y and z. Each path from the root of the tree to a leaf represents one particular valuation (the left-branch of each node represents a valuation of $\mathbf{0}$; the right-branch represents a valuation of $\mathbf{1}$). If we represent identical subtrees uniquely (via hash consing) then this tree compacts to the DAG (directed acyclic graph) shown in Figure 4(b). Note that in this DAG, the edges emanating from a node labeled z all go to the node $\mathbf{0}$ or the node $\mathbf{1}$. Therefore, the z nodes can be eliminated from the DAG with no loss of information, yielding the final DAG in Figure 4(c). This is the BDD of the function $f(x, y, z) = (x = y)$ for the variable ordering $x \rightarrow y \rightarrow z$. Given a fixed variable ordering of variables, the BDD of a boolean function is unique.

Efficient algorithms have been developed for performing set operations (and many other kinds of operations) on BDDs [7]. In the case of the $CSMOP_{SP_{rhs}}$ algorithm, every set operation in the algorithm can be implemented efficiently with BDD operations.

We must still address how a dataflow transfer function f from F is represented as a BDD. Recall that f has type **set-of** $D \rightarrow$ **set-of** D, where D is a finite set. We can represent f by its characteristic boolean function f_b:

$$f_b(D_1, D_2) = \mathbf{1} \iff f(D_1) = D_2$$

In the following section, we show how we use the syntax of boolean programs to allow an analysis developer to define transfer functions using a programming notation (rather than explicitly coding BDDs, which would be a horrendous task).

Finally, note that the use of BDDs does not change the worst-case complexity of the $CSMOP_{SP_{rhs}}$ algorithm. In the worst-case, the size of a BDD may be exponential in the number of variables ($O(D)$ in our case). We are beginning to perform experiments to see whether or not the BDDs exhibit worst-case behavior in practice.

4.3 BDDs and tri-vectors

We now briefly mention the relationship of BDDs to so-called "tri-vectors", which are vectors in which each element has one of three values ($\mathbf{1}$,$\mathbf{0}$, or $*$). Tri-vectors can be used in dataflow analysis to give precise "yes" and "no" answers, as well as "don't know".

Suppose we consider sets of bit vectors, each of length n. Both tri-vectors and BDDs may be used to represent sets of bit vectors. BDDs can represent all sets of bit vectors (there are 2^{2^n} possible such sets), whereas tri-vectors can represent a small number (3^n) of such sets. However, any set of bit vectors can be over approximated using a tri-vector. For example, with $n = 2$, the tri-vector $\langle \mathbf{0}, * \rangle$ represents the set of bit vectors $\{\langle \mathbf{0},\mathbf{0}\rangle, \langle \mathbf{0},\mathbf{1}\rangle\}$. However, the set $\{\langle \mathbf{0},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{0}\rangle\}$. cannot be directly represented as a tri-vector. It can be directly represented as a BDD. It can also be over approximated using the tri-vector $\langle *, * \rangle$.

5. BEBOP

The previous section described the basic algorithm underlying `Bebop`. In this section, we describe how `Bebop` is used through an example analysis: path-sensitive tracking of the "NULLness" of top-level pointers in C programs. Because there are many comparisons of pointers against NULL in C programs (to avoid dereferencing a NULL pointer), there is reason to expect there is a good amount of branch correlation due to such checks [6].

The input to `Bebop` is a "boolean program", a C program in which the only type available is boolean. Boolean programs have all the control-flow constructs of C (sequencing, loops, conditions, GOTOs, procedure calls). In addition, boolean programs contain parallel assignment statements and allow "$*$" (the unknown value) as the controlling predicate in a conditional statement or loop. Control non-determinism can be used to generate data non-determinism via the `unk` function:

```
bool unk()
begin
  if (*) return 1; else return 0; fi
end
```

The `unk` function implicitly represents the $*$ value. Boolean programs have global variables, local variables, and formal parameters (which, as in C, can be modified). Parameter passing is call-by-value. There are no pointers in boolean programs. The expression language of boolean program is, of course, boolean expressions.

Because boolean programs have all the control-flow constructs of C, they can faithfully represent the control-flow graph of a C program. A boolean variable in a boolean program represents a fact in the finite dataflow domain D.

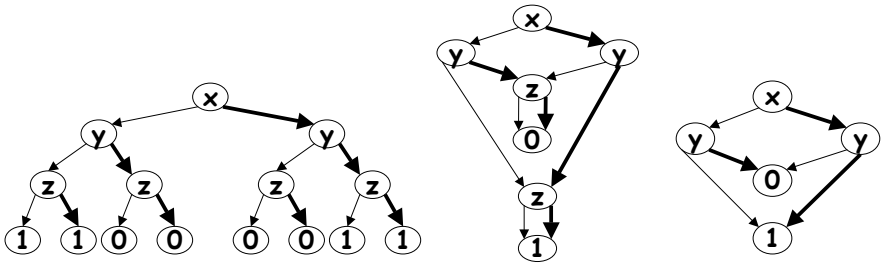


Figure 4: (a) A tree representation of the boolean function $f(x, y, z) = (x = y)$; (b) after hash consing; (c) the final BDD representation, after elimination of variable z.

Parallel assignment statements in boolean programs can be used to implement *any* transfer function of type **set-of** $D \rightarrow$ **set-of** D. In addition, an analyst can choose to use all the programming features in the boolean program language to make it easier to implement transfer functions. That is, constructs in the boolean programming language serve two purposes: to represent the control-flow structure of the C program under analysis, and to define the dataflow transfer functions.

The state of a boolean program is a pair of a program counter pc and an evaluation to all the boolean variables in scope at the pc (i.e., a bit vector). Given a boolean program B, the `Bebop` tool computes the set of reachable states for every statement in the boolean program using the algorithm given in Section 4. That is, for each vertex v_s in B's control-flow graph (corresponding to statement s), `Bebop` computes $PE'(v_s)$.

The remaining question then is how to encode an analysis problem via an automatic translation from a C program to a boolean program, which is input to `Bebop`. In general, every control-flow construct of the C program will be translated to an identical control-flow construct in the boolean program. However, the translation of assignment statements, conditionals, and procedure calls will be specific to the problem instance. We present the translation of assignments and conditionals for the NULL pointer problem.

We assume the input C program is in a simple form with no short-circuit evaluation and at most one pointer dereference per expression. In this analysis, there is one boolean variable p_b for each top-level pointer p in the C program. This boolean variable represents the predicate (`p == NULL`). The scope of a boolean variable p_b is dependent on the scope of p in the original C program: if p is a global variable, then p_b is a global variable; if p is a local variable (formal parameter) of procedure P, then p_b is a local variable (formal parameter) of procedure P_b.

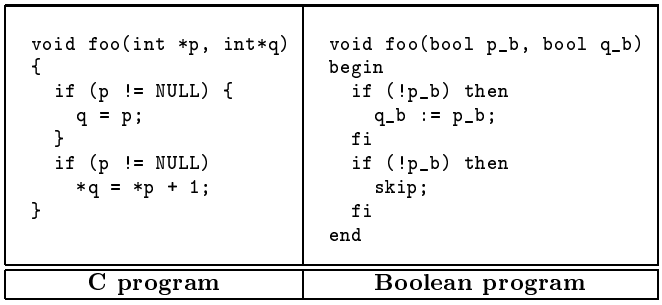
Table 1 shows the translation of assignment statements and conditional statements from the C program into the boolean program. Direct assignments to a top-level pointer p are addressed in the first four rows. The table shows the syntactic pattern in the C code, the corresponding translation into the boolean program, and the conditions under which the translation is enabled. The fourth row specifies that for all other cases of right-hand side expressions, the translation non-deterministically assigns **1** or **0** to p_b, via the `unk` function.

C statement	Transfer function	Condition
`p = NULL;`	`p_b := true;`	p a pointer
`p = &e;`	`p_b := false;`	p a pointer
`p = q;`	`p_b := q_b;`	p, q pointers
`p = e;`	`p_b := unk();`	p a pointer
`*p = NULL;`	`q_b := true;`	$ptsTo(p) = \{q\}$
`*p = &e;`	`q_b := false;`	$ptsTo(p) = \{q\}$
`*p = q;`	`r_b := q_b;`	$ptsTo(p) = \{r\}$
`*p = e;`	`q_b := unk();`	$\forall q \in ptsTo(p)$
`if (p==NULL)`	`if (p_b) then`	p a pointer
`if (p!=NULL)`	`if (!p_b) then`	p a pointer
`if (e)`	`if (*) then`	

Table 1: Translation of statements in C programs into boolean program for pointer NULLness analysis.

In the case of an indirect assignment through a pointer p, we use a points-to analysis [8] to conservatively estimate the effect of the assignment. Given a pointer p, $ptsTo(p)$ denotes the set of variables that p may point to. If $ptsTo(p)$ contains only one entry, then a strong update can take place. Otherwise (eighth row), all boolean variables representing pointers that p may point to must be given the $*$ value. Translation of conditionals is straightforward.

Figure 5 presents a simple example C program and its translation into a boolean program according to Table 1. The assignment statement `*q = *p + 1` translates to `skip` because the points-to analysis of this code shows that no variable in `foo`, including p and q, points to p or q. The `Bebop` tool will compute the abstract set of states (over $\langle p_b, q_b \rangle$) at the end of the first **if** to be $\{\langle \mathbf{0}, \mathbf{0} \rangle, \langle \mathbf{1}, * \rangle\}$. That is, either both pointers are not NULL or p is NULL and the state of q is unknown. Therefore, this analysis shows that the statement `*q = *p + 1` cannot dereference NULL. In comparison, a flow-sensitive analysis would represent the abstract set of states at the end of the first conditional to be $\{\langle *, * \rangle\}$, which would lead the analysis to report a potential NULL pointer dereference.



```
void foo(int *p, int*q)
{
  if (p != NULL) {
    q = p;
  }
  if (p != NULL)
    *q = *p + 1;
}
```

```
void foo(bool p_b, bool q_b)
begin
  if (!p_b) then
    q_b := p_b;
  fi
  if (!p_b) then
    skip;
  fi
end
```

C program | Boolean program

Figure 5: A C program and its translation into a boolean program to encode the pointer NULLness dataflow problem.

6. RELATED WORK

In previous work, we presented the interprocedural dataflow analysis algorithm underlying `Bebop` [4]. In this paper, we have explored the connection between our algorithm and the RHS algorithm, from which it was derived, in detail and have discussed the application of `Bebop` to path-sensitive dataflow problems. We have shown that the RHS algorithm can be reformulated as a traditional dataflow analysis similar to that of Sharir and Pnueli [13]. One of the key insights of the behind the RHS algorithm is the pointwise tabulation of the Sharir-Pnueli ϕ functions. Our reformulation preserves this pointwise tabulation, and hence the complexity of the RHS algorithm. Furthermore, we have described how to use our reformulation to "lift" the algorithm using BDDs to become path-sensitive.

In previous unpublished work [14], Michael Siff and Thomas Reps experimented with using BDDs in the RHS algorithm. They used BDDs to represent the entire exploded graph, which encodes both the control-flow graph and the data flow facts in a monolithic structure. They found that the BDDs grew large very quickly. Our work differs in that we use one BDD per vertex of the control-flow graph to record the set of dataflow facts for that vertex, so that BDDs are only used to represent data and not control.

Bodik and Anik [5] describe a data structure called the Value Name Graph (VNG), over which a polynomial-time flow-sensitive tri-vector dataflow analysis can be performed, yielding path-sensitive results. The VNG is analogous to the exploded graph of RHS. In the VNG, dataflow facts (names) are generated automatically by a combination of symbolic back substitution and value numbering. In the worst-case, the number of names can be exponential in the size of the program. Our approach synthesizes "names" that are boolean combinations of facts in D to achieve path-sensitive results. Boolean programs do not address the issue of tracking values as their name changes–this must be captured in the translation from C to boolean programs.

Ammons and Larus [2] use path profiling to identify "hot paths" in programs. They then duplicate code in the CFG to eliminate joins along the hot paths and perform a traditional flow-sensitive analysis on the new CFG, which yields path-sensitive information along the hot paths. In their work, an exponential blow-up can occur due to the code duplication, which must be carefully controlled.

Other related work includes systems that allow users to define program analyses and provide an underlying parameterized analysis engine [16, 15]. We focus on the problem of defining a general-purpose and parameterizable path-sensitive analysis engine. A paper in PLDI 2001 describes how to automate the process of creating boolean programs from C programs [3].

7. CONCLUSIONS

We have presented `Bebop`, a path-sensitive interprocedural dataflow analysis tool. We are beginning to perform experiments with `Bebop` and will report on these in a future paper. We expect to investigate the use of *overapproximation* techniques to help `Bebop` scale to larger programs. Clients of path-sensitive analyses may not need every ounce of path-sensitivity in order to be successful. Therefore, it makes sense to explore algorithms that provide conservative overapproximations to the *CSMOP* problem and scale to larger programs. There are two basic ideas we will pursue. First, we can easily make `Bebop` into a flow-sensitive engine through redefining the meet operator to be the Cartesian abstraction of set union. Second, we will investigate how the theory overlapping projections [9] can help obtain good overapproximations given hints about dependent and independent sets of dataflow facts.

Acknowledgements

Thanks to Manuvir Das and Manuel Fähndrich for their comments on an earlier draft of this paper.

8. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *PLDI 98: Programming Language Design and Implementation*, pages 72–84. ACM, 1998.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[4] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[5] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *POPL 98: Principles of Programming Languages*, pages 237–251. ACM, 1998.

[6] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *ESEC/FSE 97: European Software Engineering/Foundations of Software Engineering*, LNCS 1301, pages 361–377. Springer-Verlag, 1997.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[8] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[9] S. G. Govindaraju and D. L. Dill. Approximate symbolic model checking using overlapping projections. In *Electronic Notes in Theoretical Computer Science*, July 1999.

[10] K. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.

[13] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[14] M. Siff. personal communication. July 12 2000.

[15] S. W. K. Tjiang and J. L. Hennessy. Sharlit – a tool for building optimizers. In *PLDI 92: Programming Language Design and Implementation*, pages 82–93. ACM, 1992.

[16] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *PLDI 89: Programming Language Design and Implementation*, pages 1–12. ACM, 1989.