Polymorphic Predicate Abstraction

Thomas Ball
tball@microsoft.com

Todd Millstein
todd@cs.washington.edu

Sriram K. Rajamani
sriram@microsoft.com

June 17, 2002

Technical Report
MSR-TR-2001-10

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Polymorphic Predicate Abstraction

Thomas Ball
tball@microsoft.com

Todd Millstein *
todd@cs.washington.edu

Sriram K. Rajamani
sriram@microsoft.com

Microsoft Research

Abstract. Predicate abstraction is a technique for creating abstract models of software that are amenable to model checking algorithms. Because model checking algorithms have worst-case behavior that is exponential in the number of predicates in the model, it is highly desirable to reduce the number of predicates, while retaining precision. We show how polymorphism, a well-known concept in programming languages and program analysis, can be incorporated in a predicate abstraction algorithm for C programs. The use of polymorphism in predicates, via the introduction of symbolic names for values, allows us to model the effect of a large number of monomorphic predicates equivalently with a small number of polymorphic predicates. Polymorphic predicate abstraction of C programs is complicated by the presence of procedures and pointers, and our algorithm handles both constructs. We have proved that our algorithm is sound and have implemented it in the C2BP tool.

1 Introduction

Predicate abstraction [17, 10] is a technique for automatically creating a finite state system (for which a fixpoint analysis will terminate) from an infinite state system (for which a fixpoint analysis will, in general, not terminate). Under predicate abstraction, the concrete states of an infinite state system are mapped to abstract states according to their evaluation under a finite set of predicates.

The technique of predicate abstraction has been recently applied to software (rather than the transition systems usually considered in the model checking community) [29, 1, 15]. In previous work [1], we introduced an automatic predicate abstraction algorithm for C programs, as implemented in the C2BP tool [1]. Given a C program P and a set $E = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ of predicates, C2BP automatically constructs a *boolean program* abstraction $BP(P, E)$, a program that has identical control structure to P (including procedures and procedure calls) but contains only boolean variables. The program $BP(P, E)$ contains n boolean variables (*b-variables*) $V = \{b_1, b_2, \ldots, b_n\}$, where each b-variable b_i represents the predicate φ_i. Each b-variable in V has a *three-valued* domain: **false**, **true**, and $*$ (representing "don't know"). Boolean programs are amenable to model checking algorithms, including the one implemented in the BEBOP tool [3]. The combination of the C2BP and BEBOP tools can be used to discover inductive invariants in a C program that are boolean functions over the predicates in E.

The C2BP algorithm is unique in that it handles the following features of the input C program:

* Currently at the University of Washington.

- *Procedures*: The main challenge of predicate abstraction of procedure calls is in translating predicate knowledge between callers and callees in a conservative but precise manner. C2BP does this in a modular fashion: each procedure can be abstracted by C2BP given only the *interfaces* of procedures that it calls. This contrasts with other work, which inlines procedures [19]. Such an approach may lead to code explosion and does not handle programs with recursion.
- *Pointers*: Predicate abstraction is complicated by the presence of pointers in C programs. In particular, because of pointer aliasing, an update to one variable may affect the truth of predicates involving other variables. This problem is exacerbated by procedure calls, which can potentially modify the local state of the caller via pointers. We employ the results of a points-to analysis to provide conservative but precise abstraction in the presence of pointers.

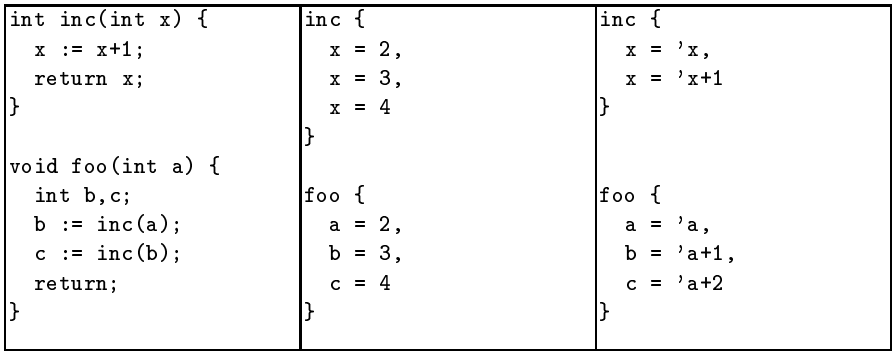
The algorithm presented in [1] supports predicates in E that are pure C boolean expressions over the variables in P. While quite expressive, this predicate language has an important shortcoming in the presence of procedures. As an example, consider the following procedure, which is the identity function on integers:

```
int id(int x) { return x; }
```

Suppose we would like the boolean program abstraction to prove that the value returned by `id` is 5 when some client `f1` invokes it. In this case, we must add the predicate $(x = 5)$ to E. Similarly, to prove that the value returned by `id` is 73 when another client `f2` invokes it, we must add the predicate $(x = 73)$ to E. In general, a separate predicate is required for each possible concrete value to be tracked.

This problem has several practical consequences. First, the number of predicates required in the abstraction of a procedure is proportional to the number of callers of the procedure. Because the abstraction process is exponential in the size of E, this requirement can become prohibitively expensive. It also somewhat undermines the modular abstraction of procedures, since a procedure may need to be re-abstracted (to incorporate new predicates) whenever a call to the procedure is added anywhere in the program. Second, the predicate language is not powerful enough for the model checker to deduce facts such as "`id` returns the same value that it is passed." Instead, the model checker can deduce only *instantiations* of this fact, given a particular value.

In this paper, we show how to use polymorphism to solve this problem. We extend the predicate language to allow the use of *symbolic constants*, which are names given to the initial values of a procedure's formal parameters, and we appropriately generalize the C2BP algorithm. A predicate containing a symbolic constant is said to be *polymorphic*. In our example above, we can use the polymorphic single predicate $(x = {}'x)$ (where ${}'x$ is the symbolic constant for x) to capture the relevant information of `id` in the resulting boolean program The predicate abstraction algorithm is modified to appropriately handle polymorphic predicates, substituting concrete values for symbolic constants during the abstraction of each procedure call (just as types are substituted for type variables at each call site of a type-polymorphic function in languages supporting parametric polymorphism). In the presence of pointers, we also support symbolic constants that refer to *dereferences* of formal parameters. The generalized algorithm is implemented in our C2BP tool.



```
int inc(int x) {
  x := x+1;
  return x;
}

void foo(int a) {
  int b,c;
  b := inc(a);
  c := inc(b);
  return;
}
```

```
inc {
  x = 2,
  x = 3,
  x = 4
}

foo {
  a = 2,
  b = 3,
  c = 4
}
```

```
inc {
  x = 'x,
  x = 'x+1
}

foo {
  a = 'a,
  b = 'a+1,
  c = 'a+2
}
```

C Program	P_{mono}	P_{poly}

Fig. 1. A simple C program and two sets of predicates, P_{mono} and P_{poly}.

```
bool,bool,bool inc(bool {x=2}, bool {x=3},
                   bool {x=4}) {
  {x=2}, {x=3}, {x=4}
      := choose(false, {x=2}|{x=3}|{x=4}),
         choose({x=2}, !{x=2}),
         choose({x=3}, !{x=3});
  return {x=2},{x=3},{x=4};
}

void foo({a=2}) {
  bool {b=3},{c=4};
  bool prm1, prm2, prm3;
  bool ret1, ret2, ret3;
  ...
  ...
  prm1 := choose({a=2}, !{a=2});
  prm2 := choose(false, {a=2});
  prm3 := choose(false, {a=2})
  ret1, ret2, ret3 := inc(prm1, prm2, prm3);
  {b=3}  := choose(ret2, !ret2);

  prm1 := choose(false, {b=3});
  prm2 := choose({b=3}, !{b=3});
  prm3 := choose(false, {b=3});
  ret1, ret2, ret3 := inc(prm1, prm2, prm3);
  {c=4}  := choose(ret3, !ret3);
  return;
}
```

Fig. 2. Boolean program abstraction created by the C2BP tool, given the C program shown in Figure 1 and predicates P_{mono}.

Section 2 informally reviews by example how C2BP performs *monomorphic* predicate abstraction for C programs with procedures and then presents the same example using polymorphic predicate abstraction. Section 3 introduces a core language used to formally explain the predicate abstraction algorithm. Section 4 presents the technical details of our algorithm, in the case when programs do not involve pointers, and Section 5 extends the algorithm to accommodate pointers. Section 6 discusses related work and Section 7 concludes the paper. Appendix B proves that the C2BP abstraction algorithm is sound.

2 Example

Figure 1 shows a simple example of a procedure `foo` calling an increment procedure `inc` twice. Consider the set of predicates P_{mono} in this figure. Three of these predicates are local to the procedure `inc` and the other three are local to the procedure `foo`. Figure 2 shows the boolean

program that C2BP produces when given the C program[1] and the predicate set P_{mono} from Figure 1. The boolean program has the same control-flow structure as the C program and contains a b-variable for each predicate input to C2BP. A b-variable that is a formal parameter is called a *b-parameter*. For example, the procedure `inc` takes three formal b-parameters, {`x=2`}, {`x=3`}, and {`x=4`}, corresponding to the three predicates in P_{mono} that mention the formal parameter x.[2] The procedure returns three b-variables that represent the updated values of the predicates about x. Procedure `foo` has a formal b-parameter {`a=2`}. It also has local b-variables for the predicates in P_{mono} involving the local variables b and c.

C2BP creates this boolean program by translating each statement s of the C program into one or more statements S' in the boolean program such that S' conservatively models the effect of s on each predicate that is in scope at the corresponding point in the boolean program. Consider the assignment statement `x := x+1` in the `inc` procedure. C2BP discovers that if the predicate $(x = 2)$ is true (false) before this statement, then the predicate $(x = 3)$ will be true (false) after. This results in the translation (as part of the parallel assignment in `inc`): {`x=3`} `:= choose( {x=2}, !{x=2} )`. The `choose` function is included in every boolean program and is defined as follows:

```
bool choose(bool pos, bool neg) {
     if (pos) return true;
     else if (neg) return false;
     else return *;
}
```

The `pos` b-parameter of the `choose` function represents sufficient conditions for the truth of a predicate, while `neg` represents sufficient conditions for the falsehood of a predicate. C2BP guarantees that `choose` is never called with both b-parameters evaluating to true. Both b-parameters may evaluate to false because the predicates being modeled are not strong enough to provide a definite answer, or because the theorem proving machinery that C2BP uses is incomplete. In this case, the `choose` function conservatively returns $*$, representing the "don't know" value.

Consider the first procedure call `b := inc(a)` in procedure `foo`. In the C program, the actual parameter passed to `inc` is the expression a, whose value is assigned to the formal parameter x of `inc`. By substituting the actual parameter a for formal x in the predicate $(x = 2)$, C2BP determines that if the predicate $(a = 2)$ is true (false) at this call then the predicate $(x = 2)$ is true (false) at the entry of `inc`, resulting in the statement: `prm1 := choose( {a=2}, !{a=2})`. The b-variable `prm1` is then sent as the actual parameter for the first b-parameter of `inc` in the boolean program. The other b-parameter values are determined similarly via assignments to the b-variables `prm2` and `prm3`. There are three return predicates in `inc`: $(x = 2)$, $(x = 3)$ and $(x = 4)$. The assignment of the return value of `inc` to variable b is an implicit assignment of the form `b := x`. C2BP deduces that the predicate $(b = 3)$ will be true (false) after the call (and the assignment of the return value to b) if the predicate $(x = 3)$ (represented by `ret2` in `foo`) is true (false) at the exit of procedure `inc`. Thus, C2BP generates the assignment {`b=3`} `:= choose( ret2, !ret2 )`.

[1] Throughout, we use ":=" instead of "=" for the assignment operator, to avoid confusion.

[2] Boolean programs permit an identifier to be of the form $\{p\}$, where p is an arbitrary string. This is useful for naming b-variables with the exact predicates they represent.

```
bool,bool inc() {
  bool {x='x}, {x='x+1};
  {x='x} := true;
  {x='x}, {x='x+1} :=
    choose(false,{x='x}|{x='x+1}),
    choose({x='x},!{x='x} );
  return {x='x}, {x='x+1};
}
```

```
void foo() {
  bool {a='a},{b='a+1},{c='a+2};
  bool ret1, ret2;
  {a='a} := true;
  ret1, ret2 := inc();
  {b='a+1} := choose({a='a}&ret2,
                     ({a='a}&!ret2)|(!{a='a}&ret2));
  ret1, ret2 := inc();
  {c='a+2} := choose({b='a+1}&ret2,
                     ({b='a+1}&!ret2)|(!{b='a+1}&ret2));
  return;
}
```

Fig. 3. Boolean program abstraction created by the C2BP tool, given the input C program shown in Figure 1 and predicates P_{poly}.

The predicates $(x = 2)$ and $(x = 3)$ are necessary so that if the value 2 is passed into `inc`, it can be determined that the return value is 3. Similarly, the two predicates $(x = 3)$ and $(x = 4)$ are necessary so that if the value 3 is passed into `inc`, it can be determined that the return value is 4. These predicates (along with the predicates local to `foo`) allow the boolean program to model the fact that if `foo` is passed the value 2, then the variable `c` will have the value 4. However, an entirely different set of predicates for `inc` would be necessary for the boolean program to glean the value of `c` if 5 were passed to `foo`, for example. Similarly, the program may contain other calls to `inc`, which can require `inc` to be re-abstracted with yet more predicates.

Polymorphic Predicate Abstraction. Although calls may differ in the values of arguments passed to a procedure, they often rely on the same underlying abstract *transfer function* of the procedure. The idea of polymorphic predicate abstraction is to employ predicates that directly model this transfer function, which can then be shared across call sites. In the example of the `inc` procedure, we wish to state that the return value of `inc` is one more than the initial value of the formal parameter x of `inc`. Symbolic constants offer a way to do this. We denote the value of x at entry to `inc` by the symbolic constant $'x$. To precisely summarize the effect of `inc` for both call sites then requires only two predicates: $x = 'x$ and $x = 'x + 1$, all local to `inc`. At each caller, C2BP binds $'x$ to the appropriate actual value to get the desired effect. The predicate set P_{poly} in Figure 1 shows such polymorphic predicates. Similar polymorphic predicates are created for `foo`, in order to make its abstraction generic with respect to its callers.

Figure 3 shows the boolean program created by C2BP when applied to the C program and the predicate set P_{poly}. This boolean program is similar to its monomorphic counterpart, but the use of symbolic constants allows it to prove much stronger properties about the original C program. In particular, the boolean program is able to model the fact that `inc` returns a value that is one greater than the original value of `x` and that variable `c` in `foo` will have a value that is two greater than the original value of `a`. In this way, all calls to `inc` and `foo` can be precisely abstracted, no matter what values are passed as arguments to those calls. Note the (perhaps) surprising effect of the translation: the procedures `inc` and `foo` have no formal b-parameters! This highlights the fact that `inc` and `foo` are truly polymorphic procedures: their transfer functions have no dependence on the contexts in which they are called.

Types:	τ	::=	**void** | **bool** | **int** | **ref** τ
Expressions:	e	::=	$c \mid x \mid e_1\ op\ e_2 \mid \&x \mid *\cdots*x$
Declaration:	d	::=	$\tau\ \ x_1, \cdots, x_n;$
Statements:	s	::=	**skip** | **goto** L | L: s
		|	**branch** $\overline{s_1}$ || $\cdots$ || $\overline{s_n}$ **end**
		|	**assume**(e)
		|	**return** $x_1, \cdots, x_n$
		|	$*x\ :=\ e$
		|	$x_1, \cdots x_n\ :=\ e_1, \cdots, e_n$
		|	$x_1, \cdots, x_m := f(e_1, \cdots, e_n)$
Statement Sequences:	$\overline{s}$	::=	$s_1; \cdots s_n;$
Procedure:	p	::=	$\tau'_1, \cdots, \tau'_m\ id\ (f_1 : \tau_1, \cdots, f_n : \tau_n)$ { $d_1 \cdots d_q\ \overline{s}$ }
Program:	g	::=	$d_1 \cdots d_m\ p_1 \cdots p_n$

Fig. 4. A core language containing references and procedures.

3 Core Language

To simplify the presentation of the central ideas in our approach, we focus our attention on a small core language containing procedures and references (but without type casts, pointer arithmetic, structures, arrays, unions, and explicit allocation and deallocation). Figure 4 presents the syntax of the core language.

The core language contains void, boolean, integer, and reference types. The booleans are *three-valued*, and we use Kleene's three-valued logic to interpret them (see Appendix A for a formal description). A procedure can return multiple values and so is annotated with a sequence of return types. The expressions include boolean and integer constants (ranging over metavariable c), variables (ranging over metavariable x), and the usual arithmetic and binary operations (ranging over metavariable *op*). We also provide two pointer expressions, the ability to create a reference to a variable and the ability to dereference a pointer variable as many times as its type allows.

The statements include the **skip** statement, **goto** statement, and labelled statements. Branches with a finite number of targets are specified with the **branch** statement, which non-deterministically selects one of its n statements (branches) to execute. The **assume** statement is the dual of the **assert** statement: the **assume** statement silently terminates execution if its expression evaluates to false. The **assume** statement, in combination with the (non-deterministic) **branch** statement, can be used to implement the common `if-then-else` and C-style `switch` statements. For example, the statement "**if** e **then** s_1 **else** s_2" is implemented as "**branch** **assume**(e); s_1; || **assume**$(!e)$; s_2; **end**".

There are two forms of assignment statements in the language. The first ($*x := e$) is an indirect assignment through a variable with reference type. For ease of exposition, we do not allow more than one dereference on the left-hand side of an assignment. This does not limit expressiveness. The second is a parallel assignment to of n expressions to n variables. Finally, the statement language contains a procedure call statement, in which the procedure can return a sequence of m values.

All variables must be declared before being used. Variables of reference type must additionally be initialized before being used. Variables of type **int** and **bool** are assigned an initial value of 0 and * ("don't know"), respectively.

4 Polymorphic Predicate Abstraction Algorithm

This section presents our polymorphic predicate abstraction algorithm for the core language without references. We are given a program P in the core language and a set $E = \{\phi_1, \cdots, \phi_n\}$ of C pure boolean expressions over the variables in P and associated symbolic constants. The goal of the algorithm is to create a boolean program abstraction $BP(P, E)$ that conservatively (yet precisely as possible) represents the effect of each statement in P on each predicate in E. The language of boolean programs is simply the language of Section 3 with integers, reference types, and all associated expressions and statements removed.[3]

After defining some preliminaries, we review the concepts of weakest preconditions and predicate strengthening that form the basis of our abstraction algorithm. We then present the syntax-directed translation of the basic program statements and then of procedure calls. The section ends with a brief example.

4.1 Preliminaries

It is useful in the following to assume that a programs are in *internal form*. A program P is in internal form if each procedure of P has the following properties:

- The procedure has a single **return** statement, and it is the last statement of the procedure.
- Each statement in the procedure has a unique label, as does each sub-statement in the cases of each **branch** statement.
- The actual parameters of each procedure call statement in the procedure are simple variables.
- The left hand side of function calls has only local variables.
- For each formal f of the procedure, there is a local variable $'f$ and an *initialization statement* $L : 'f := f$ at the beginning of the procedure's sequence of statements. The variable $'f$ is never referenced again in the procedure.

It is clear that transforming a program to internal form does not change its semantics. From now on, we assume programs are in internal form.

Each predicate ϕ_i in E will have a corresponding b-variable b_i in $BP(P, E)$. Let $V = \{b_1, \cdots, b_n\}$ and let $\mathcal{E}$ be a mapping from each b_i to the associated ϕ_i. We often extend $\mathcal{E}$ to negations, conjunctions, and disjunctions of boolean variables, in the obvious way.

Each predicate in E is annotated as being either global to $BP(P, E)$ or local to a particular procedure in $BP(P, E)$ (see Figure 1, in which predicates are local to `inc` or `foo` – there are

[3] Technically, the **choose** function given in Section 2 is not in the language of Section 3 (because the language does not contain an `if-then-else` statement). We also often employ a parallel assignment where the right-hand sides are all calls to **choose**, another violation of the language's syntax. Both of these relaxations can be straightforwardly desugared to the appropriate syntax.

no global predicates in this example), thereby determining the scope of the corresponding b-variable in $BP(P, E)$. Let $G_P = \{g_1, g_2, \dots\}$ be the global variables of the program P. A global predicate can refer only to variables in G_P. Let E_G denote the global predicates of E. $BP(P, E)$ will contain one global declaration for each of the b-variables associated with predicates in E_G.

For a procedure R, let E_R denote the subset of predicates in E that are local to R. Let $F_R = \{f_1, f_2, \dots\}$ be the formal parameters of R, and let $L_R = \{l_1, l_2, \dots\}$ be the local variables of R. We assume that the **return** statement of R has the form "**return** r", where $r \in F_R \cup L_R$.[4]

Recall that we allow local predicates of a procedure R to introduce symbolic constants corresponding to formal parameters of R. The symbolic constant for formal parameter f is denoted by $'f$.

4.2 Weakest Preconditions and Cubes

For a statement s and a predicate φ, let $WP(s, \varphi)$ denote the *weakest liberal precondition* [12, 18] of φ with respect to statement s. $WP(s, \varphi)$ is defined as the weakest predicate whose truth before s entails the truth of φ afterwards. The standard weakest precondition rule says that $WP(\mathrm{x} := \mathrm{e}, \varphi)$ is φ with all occurrences of x replaced with e, denoted $\varphi[e/x]$. For example, $WP(x := x + 1,\ x < 5) = (x + 1) < 5 = (x < 4)$. Therefore, $(x < 4)$ is true before $x := x + 1$ executes if and only if $(x < 5)$ is true afterwards.

Given a statement s, a set of predicates E, and predicate $e \in E$, it may be the case that $WP(s, e)$ is not in E. For example, suppose $E = \{(x < 5), (x = 2)\}$. We have seen that $WP(\mathtt{x} := \mathtt{x} + \mathtt{1},\ x < 5) = (x < 4)$, but the predicate $(x < 4)$ is not in E. Therefore, we use decision procedures (i.e., a theorem prover) to *strengthen* the weakest precondition to an expression over the predicates in E. In our example, we can show that $x = 2 \Rightarrow x < 4$. Therefore if $(x = 2)$ is **true** before $x := x + 1$, then $(x < 5)$ is **true** afterwards.

We formalize this strengthening of a predicate as follows. A *cube* over a set of boolean variables $V = \{b_1, \cdots, b_n\}$ is a conjunction $c_1 \wedge \ldots \wedge c_n$, where each $c_i \in \{b_i, \neg b_i\}$. For any predicate φ, a set of boolean variables V, and a function $\mathcal{E}$ that maps each variable b in V to a predicate $\mathcal{E}(b)$, let $\mathcal{F}_{V,\mathcal{E}}(\varphi)$ denote the largest disjunction of cubes c over V such that $\mathcal{E}(c)$ implies φ. The predicate $\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\varphi))$ represents the weakest predicate over E that is stronger than φ. In our example, if $V = \{b_1, b_2\}$, $\mathcal{E}(b_1) = (x < 5)$, and $\mathcal{E}(b_2) = (x = 2)$, then $\mathcal{F}_{V,\mathcal{E}}(x < 4) = (b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2) \equiv b_2$, so $\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(x < 4)) = (x = 2)$.

It will also be useful to define a corresponding weakening of a predicate to the set of predicates in E. Define $\mathcal{G}_{V,\mathcal{E}}(\varphi)$ as $\neg\mathcal{F}_{V,\mathcal{E}}(\neg\varphi)$. The predicate $\mathcal{E}(\mathcal{G}_{V,\mathcal{E}}(\varphi))$ represents the strongest predicate over E that is implied by φ.

For each cube, the implication check involves a call to a theorem prover implementing the required decision procedures. Our implementation of C2BP uses two theorem provers: Simplify [11] and Vampyre [4], both Nelson-Oppen style provers [25]. The naive computation of $\mathcal{F}_{V,\mathcal{E}}(\cdot)$ and $\mathcal{G}_{V,\mathcal{E}}(\cdot)$ require exponentially many calls to the theorem prover in the worst case. We have implemented several optimizations that make the $\mathcal{F}_{V,\mathcal{E}}(\cdot)$ and $\mathcal{G}_{V,\mathcal{E}}(\cdot)$ computations practical [1].

[4] The abstraction process can be straightforwardly extended to handle zero or multiple return values in the source program.

4.3 Statements Besides Procedure Calls and Returns

Each statement of program P (other than procedure calls) translates to one statement in $BP(P,E)$. We define the function $BP(s,V,\mathcal{E})$ to output the translation of statement s, given the set of boolean variables V and a mapping to the associated predicates being modeled. Most of the statement translations are straightforward:

$$\begin{aligned} BP(\textbf{skip},\, V,\mathcal{E}) &\equiv \textbf{skip} \\ BP(\textbf{goto L},\, V,\mathcal{E}) &\equiv \textbf{goto L} \\ BP(\textbf{L: } s,\, V,\mathcal{E}) &\equiv \textbf{L: } BP(s,V,\mathcal{E}) \\ BP(\textbf{branch } \overline{s_1} \,||\, \cdots \,||\, \overline{s_n} \textbf{ end},\, V,\mathcal{E}) &\equiv \textbf{branch } BP(\overline{s_1},V,\mathcal{E}) \,||\, \cdots \,||\, BP(\overline{s_n},V,\mathcal{E}) \textbf{ end} \end{aligned}$$

The rule for abstracting **branch** statements relies on the following natural rule for abstracting statement sequences:

$$BP(\overline{s} = s_1;\cdots s_n;,V,\mathcal{E}) \quad\equiv\quad BP(s_1,V,\mathcal{E});\ \cdots\ BP(s_n,V,\mathcal{E});$$

For a statement of the form **assume**(e), we know by the semantics of **assume** that e is true at the associated point in P, and therefore also at the corresponding point in $BP(P,E)$. However, e may not be in the set E of predicates being modeled, so the best we can do in $BP(P,E)$ is to assume the strongest predicate over expressions in E that is implied by e:

$$BP(\textbf{assume } (e),\, V,\mathcal{E}) \quad\equiv\quad \textbf{assume } (\mathcal{G}_{V,\mathcal{E}}(e))$$

For example, suppose the statement s is **assume(`x < 2`)** and the set of predicates E is $\{(x<5),(x=2)\}$. Then $BP(s,V,\mathcal{E})$ (under the obvious V and $\mathcal{E}$ interpretations) is **assume(`{x<5}&!{x=2}`)**.

Now, consider an assignment statement s of the form $x := e$.[5] The associated statement in $BP(P,E)$ must appropriately update all of the boolean variables in V that are in scope at the current program point p.[6] If $WP(s,\phi_i)$ is true at p, then b_i may be safely set to true at p in $BP(P,E)$. Similarly, if $WP(s,\neg\phi_i)$ is true at p, then b_i may be safely set to false at p in $BP(P,E)$. Because the predicate $WP(s,\phi_i)$ may not be in E, we need to weaken it to a predicate over expressions in E that implies $WP(s,\phi_i)$, and similarly for $WP(s,\neg\phi_i)$. Therefore, $BP(P,E)$ will contain the following parallel assignment statement in place of s[7]:

$$\begin{aligned} BP(x := e,V,\mathcal{E}) \quad\equiv\quad & b_1,\ldots,b_n := \\ & \texttt{choose}(\mathcal{F}_{V,\mathcal{E}}(WP(\texttt{x:=e},\varphi_1)),\mathcal{F}_{V,\mathcal{E}}(WP(\texttt{x:=e},\neg\varphi_1))), \\ & \ldots, \\ & \texttt{choose}(\mathcal{F}_{V,\mathcal{E}}(WP(\texttt{x:=e},\varphi_n)),\mathcal{F}_{V,\mathcal{E}}(WP(\texttt{x:=e},\neg\varphi_n))) \end{aligned}$$

Consider again the assignment statement `x:=x+1` and the set of predicates $E = \{(x<5),(x=2)\}$. The translation of this assignment into the boolean program is:

```
{x<5}, {x=2} := choose({x=2}, false), choose(false, {x=2});
```

[5] The abstraction process can be straightforwardly extended to handle parallel assignments in the source program.

[6] This "pointwise" updating of the boolean variables corresponds to a Cartesian approximation of the most precise Boolean abstraction possible. For details, see [2].

[7] As an optimization, we only need update those b-variables whose values may have changed as a result of the assignment [1].

4.4 Procedures, Calls, and Returns

This subsection describes the abstraction of procedures and procedure calls. When abstracting a program P, C2BP first produces the *interface* of each procedure in P, which is essentially the procedure's type signature in $BP(P, E)$. Interfaces can be determined for each procedure in isolation. Once this is done, the statements of each procedure are abstracted one-by-one; the abstraction of a call to procedure R relies on R's interface.

Procedure Interfaces Let R' be the version of procedure R in $BP(P, E)$. The interface of R' is a six-tuple $(F_R, r, S_R, E_f, E_r, B_R)$, which C2BP constructs by examining E_R and the formal parameters in procedure R in program P. The terms F_R and r were defined above. $S_R = \{'f_{i_1}, 'f_{i_2}, \dots\}$ is the set of symbolic constants used in the predicates in E_R. Let $scs(e)$ be the set of symbolic constants in predicate e. Let $vars(e)$ be the set of variables referenced in expression e.

The set E_f contains the subset of E_R that should be formal parameter predicates in $BP(P, E)$. These are the predicates in E_R that refer to variables in F_R, and possibly to globals, but do not refer to local variables of R or symbolic constants.

$$E_f = \{e \in E_R \mid vars(e) \cap F_R \neq \emptyset \wedge vars(e) \cap L_R = \emptyset \wedge scs(e) = \emptyset\}$$

The associated b-variable in V of each member of E_f will be a formal parameter in R'. The associated b-variables of all other members of E_R will be declared as local variables in R'.

The set E_r contains the subset of predicates in E_R that should be returned by R'. These are the predicates in E_R that refer to the return variable r or to symbolic constants, but not to local variables or formals of R (other than r):

$$E_r = \{e \in E_R \mid (r \in vars(e) \vee scs(e) \neq \emptyset) \wedge (vars(e) - \{r\}) \cap (L_R \cup F_R) = \emptyset\}$$

For each symbolic constant $'f_{i_j}$ in S_R, there must exist a predicate $(f_{i_j} = 'f_{i_j})$ in E_R.[8] The set $B_R \subseteq E_R$ consists of the predicates in E_R of the form $(f_{i_j} = 'f_{i_j})$. These predicates are called the *binding predicates* of R.

Translating Return Statements Now we can define the rule for abstracting **return** statements. Let $b_1, \dots, b_n$ be the associated b-variables in V for each member of E_R. Then the translation of R's **return** statement is defined as follows:

$$BP(\textbf{return } r, V, \mathcal{E}) \quad \equiv \quad \textbf{return } b_1, \dots, b_n$$

Translating Procedure Calls Consider a call $x := R(a_1, \dots, a_j)$ in some procedure Q in program P. Let $V_{Q'}$ be the b-variables in scope at Q' (the boolean program version of Q), and let $\mathcal{E}_Q$ be a map from $V_{Q'}$ to the predicates they represent. Let $(F_R, r, S_R, E_f, E_r, B_R)$ be the interface of R'. We divide the computation of $BP(x := R(a_1, \dots, a_j), V, \mathcal{E})$ into three parts:

[8] A tool could easily insert such predicates into E_R as necessary.

computation of the actual parameters, generation of the call to R', and updating of variables in the calling context.

First C2BP computes an actual value to pass to R' for each formal parameter predicate $e_i \in E_f$, assigning it to a fresh local variable prm_i. Let $e_i' = e_i[a_1/f_1, a_2/f_2, \ldots, a_j/f_j]$, where $f_1, f_2, \ldots, f_j$ are the formals from F_R. The expression e_i' represents the value of formal parameter predicate e_i in the current calling context. Therefore, if e_i' is true (false) before the call, then e_i should be passed the value true (false). As with assignment statements,[9] in general we need to strengthen e_i' to an expression over the expressions in E:

$$prm_i := \texttt{choose}(\mathcal{F}_{V_{Q'},\mathcal{E}_Q}(e_i'), \mathcal{F}_{V_{Q'},\mathcal{E}_Q}(\neg e_i')); \tag{1}$$

Generating the call to R' is straightforward: we pass all $|E_f|$ of the prm_i variables to R', and we use $|E_r|$ fresh variables to catch the return values from the call:

$$ret_1, \ldots, ret_{|E_r|} := R'(prm_1, \ldots, prm_{|E_f|}); \tag{2}$$

The most difficult part of the translation is in conservatively (but precisely) updating the b-variables in the scope of Q' that may have changed as a result of the call to R'. In particular, any predicate of Q' that mentions x (the left-hand side of the call to R in Q) or a global variable might have changed its value. Let $E_u = \{e \in E_Q \mid (x \in vars(e)) \vee (vars(e) \cap G_P \neq \emptyset)\}$ be the set of such predicates. We will update the value of each predicate in E_u, in the context of the returned predicates of R' as well as the old values of the predicates in E_Q.

First we introduce a mapping *orig* which says how to interpret each variable in scope of Q in P, after the call to R. For local variables and formal parameters y of Q, we define *orig* $(y) = y$, since their values are not affected by the call to R (as there are no reference variables, for now). For global variables y, we define *orig* $(y) = y_o$, where y_o is a fresh variable name called an *original variable*. Conceptually, we can think of y_o as caching the value of y in Q before the call to R, just as R's symbolic constants cache the original values of its formals. We extend *orig* to expressions in the usual way.

Next we define a mapping γ which says how to interpret the symbolic constants and the return value of R in the context of Q. Since a symbolic constant $'f_i$ refers to the initial value of a formal parameter f_i of R, we define $\gamma('f_i) = orig(a_i)$, where a_i is the associated actual of f_i in the call to R. For the return variable r in R, $\gamma(r) = x_{ret}$, where x_{ret} is a fresh variable. The reason for using x_{ret} instead of x is to capture the intermediate state when the call from R has returned, and before the update to x has happened. For globals g in G, $\gamma(g) = g$. We extend γ to expressions in the usual way.

Now we are ready to define the interpretation of all the boolean variables in scope of Q after the call to R. We saw above that for each return predicate e_i of E_r, there is an associated b-variable ret_i local to procedure Q'. Let $T_{Q'}$ be the set of these b-variables and let $\mathcal{E}_{RQ}$ be a map such $\mathcal{E}_{RQ}(ret_i) = e_i$. Let $\hat{V} = V_{Q'} \cup T_{Q'}$. Recall that $\mathcal{E}_Q$ maps the b-variables $V_{Q'}$ in scope in Q to predicates that they represent. Define a new map $\hat{\mathcal{E}}$ from $\hat{V}$ to predicates in the following

[9] Conceptually, the translation of parameters is akin to simulating the assignment of actuals to formals.

way:

$$\hat{\mathcal{E}}(b) = \begin{array}{ll} orig(\mathcal{E}_Q(b)), & \text{if } b \text{ is a local or formal } b\text{-variable in } V_{Q'} \\ \mathcal{E}_Q(b), & \text{if } b \text{ is a global } b\text{-variable in } V_{Q'} \\ \gamma(\mathcal{E}_{RQ}(b)), & \text{if } b \in T_{Q'} \end{array}$$

The mapping $\hat{\mathcal{E}}$ maps the b-variables in scope in Q to the predicates they represent after the call to R (but before the update to x).

Finally, we are ready to update the truth values of every b-variable b such that $\mathcal{E}(b) = e$ for some $e \in E_u$. We model this update as a conceptual assignment of x_{ret} to x after the call to R:

$$b := \texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{x} := \texttt{x}_{\texttt{ret}}, e)), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{x} := \texttt{x}_{\texttt{ret}}, \neg e))); \tag{3}$$

To summarize, $BP(x := R(a_1, \ldots, a_j), V, \mathcal{E})$ is obtained by concatenating the assignments of the form (1) for every formal parameter predicate, a call of the form (2) and assignments of the form (3) for every element in E_u.

4.5 Examples

Consider the translation of the call `b := inc(a)` in Figure 2. The b-variables V_{foo} in scope at `foo` are { {`a=2`}, {`b=3`}, {`c=4`} }. The $\mathcal{E}_{foo}$ function is obvious from the notation (e.g., $\mathcal{E}_{foo}(\{\texttt{a=2}\}) = (a = 2)$, etc.) The formal b-parameters of `inc` in the abstraction are {`x=2`}, {`x=3`} and {`x=4`}. In order to determine the actual for {`x=2`}, C2BP computes

$$\texttt{prm1} := \texttt{choose}(\mathcal{F}_{V_{foo},\mathcal{E}_{foo}}(a = 2), \mathcal{F}_{V_{foo},\mathcal{E}_{foo}}(a \neq 2))$$

where $(a = 2)$ is obtained by substituting the actual `a` for formal `x`. The values for `prm2` and `prm3` are computed in a similar manner.

Now consider the processing of return values from this call. The set of return temporaries T_{foo} is { `ret1`, `ret2`, `ret3` }. We have $\hat{V} = V_{foo} \cup T_{foo}$, $\hat{\mathcal{E}}(\texttt{ret1}) = (b_{ret} = 2)$, $\hat{\mathcal{E}}(\texttt{ret2}) = (b_{ret} = 3)$, and $\hat{\mathcal{E}}(\texttt{ret3}) = (b_{ret} = 4)$, obtained by substituting b_{ret} for y in the return predicates of `inc`. We also have $\hat{\mathcal{E}}(\{\texttt{a=2}\}) = (a = 2)$, $\hat{\mathcal{E}}(\{\texttt{b=3}\}) = (b = 3)$, and $\hat{\mathcal{E}}(\{\texttt{c=4}\}) = (c = 4)$. The new value of {`b=3`} is $\texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} = 3), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} \neq 3))$, which compiles to `choose( ret2, !ret2 )`.

Finally, consider the translation of the call `b := inc(a)` in Figure 3. The b-variables V_{foo} in scope at `foo` are { {`a='a`}, {`b='a+1`}, {`c='a+2`} }, with the obvious $\mathcal{E}_{foo}$. The `inc` function returns two predicates. The set of return temporaries T_{foo} is { `ret1, ret2` }. We have $\hat{\mathcal{E}}(\texttt{ret1}) = (b_{ret} = a)$ and $\hat{\mathcal{E}}(\texttt{ret2}) = (b_{ret} = a+1)$, obtained by substituting b_{ret} for x and a for $'x$ in the return predicates. The new value of {`b='a+1`} is $\texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} = {}'a+1), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} \neq {}'a+1))$, which compiles to

```
choose( {a='a}&ret2, ({a='a}&!ret2) | !{a='a}&ret2))
```

5 Adding Pointers

In this section, we extend the algorithm of Section 4 to handle programs with pointers. We now allow the full syntax of the language of Section 3. We also allow pointers, pointer dereferencing, and addresses of variables in the input predicates.

In addition, symbolic constants may now refer to dereferences of formal parameters. Define an *access expression* q to be a variable preceded by zero or more dereference (*) symbols. Let $var(q)$ be the underlying variable in the access expression. Consider a procedure R. Given an access expression q, where $var(q)$ is a formal parameter of R, the symbolic constant $'q$ refers to the value of q on entry to R.

These generalizations require modification to the internal form as well as the translation of assignment statements, procedure interfaces, and procedure calls.

Internal Form. We modify the definition of the internal form in two ways. First, the actual parameters to procedure calls may be locations instead of just simple variables. Second, we augment initialization statements to handle symbolic constants of dereferences to formal parameters. For each procedure, for each access expression $*^n f$ such that f is a formal of the procedure and the access expression $*^n$ is allowed by the formal's type, we require a local variable $'*^n f$. Further, we require the existence of an initialization statement $L : '*^n f := *^n {}' f$ after the initialization statement for $'f$. The variable $'*^n f$ should only appear in initialization statements.

Assignment Statements. When translating an assignment statement (either of the form `x:=e` or `*x:=e`), the standard weakest precondition computation described in Section 4.2 no longer suffices. For example, consider $WP(\texttt{x} := \texttt{5}, *y > 6)$. According to the standard definition, $WP(\texttt{x} := \texttt{5}, *y > 6) = *y > 6[5/x] = *y > 6$. This means that if $*y > 6$ is true before the execution of `x:=5`, then it is true afterwards. However, this is not the case if $*y$ and x are aliases of one another.

To handle this problem, we adapt Morris' general axiom of assignment [24]. Let metavariable l range over *locations*, which are l-valuable expressions. Consider the computation of $WP(\texttt{l} := \texttt{e}, \varphi)$, and let l' be a location mentioned in the predicate φ. Then there are two cases to consider: either l and l' are aliased, and hence a change in the contents of l will cause a change in the contents of l'; or they are not, and an assignment to l leaves l' unchanged. Define

$$l'[l \leftarrow e] = \begin{cases} e \text{ if } l \text{ and } l' \text{ are aliased;} \\ l' \text{ otherwise.} \end{cases}$$

Then the predicate $WP(\texttt{l:=e}, \varphi)$ is defined as $\varphi[l \leftarrow e]$, where $\varphi[l \leftarrow e]$ denotes the predicate obtained by syntactically substituting each location l' in φ by the expression $l'[l \leftarrow e]$. In this way, all aliases of l are replaced by e. In the absence of alias information, the weakest precondition has to consider both the case when l' is and is not aliased to l, for each l' in φ. Therefore, if φ has k locations in it, the weakest precondition will in general have 2^k syntactic disjuncts, each disjunct considering a possible alias scenario. The weakest precondition corresponding to each disjunct is obtained from φ by simultaneously substituting all locations in φ that are aliased to l with e. In our example above, $WP(\texttt{x} := \texttt{5}, *y > 6) = ((\&x = y)\&(5 > 6))|((\&x \neq y)\&(*y > 6))$. We use a may-alias analysis [9] to improve the precision of this weakest precondition computation, pruning disjuncts that represent infeasible alias scenarios.

Procedure Interfaces. For a procedure R, the components G_P, F_R, L_R, r and S_R of R's interface are defined as in Section 4.4. The definition of binding predicates B_R is augmented to require predicates of the form $(*^n\ 'f = '*^n f)$ where $'*^n f$ is in S_R. We require that if a binding predicate of the form $(*^n\ 'f = '*^n f)$ is in B_R, then there must also be a binding predicate in B_R with $'f$ on the right-hand side.

The formal parameters E_f and the return predicates E_r of R in $BP(P, E)$ are defined as before. Note that the return predicates E_r may now contain dereferences of symbolic constants (such as $*'q$) which are used to update local state of the caller modified by the call to R, as described below.

Procedure Calls. Consider a call $x := R(a_1, \ldots, a_j)$ in some procedure Q in program P. The computation of the actuals for the corresponding call to R' in $BP(P, E)$ is unchanged, as is the generation of the call to R'. However, in the presence of pointers, the set of local predicates in Q' that must be updated needs to be generalized to include predicates that can be possibly affected in the call due to pointer indirection and aliasing:

$$\begin{aligned} E_u = \{e \in E_Q \mid\ & e \text{ references an alias of } x \ \vee \\ & e \text{ references an alias of a global variable } \vee \\ & e \text{ references an alias of a (transitive) dereference of a global variable } \vee \\ & e \text{ references an alias of a (transitive) dereference of an actual parameter to } R\} \end{aligned}$$

The set E_u can be conservatively over-approximated using a may-alias analysis.

Next, the mappings *orig* and γ are extended. The mapping *orig* is generalized to handle locations. Given an access expression $*^n y$ (the value of n dereferences of y), if y is a global variable or the value of $*^n y$ may be affected by the call to R (checked conservatively by a may-alias analysis), then $orig(*^n y) = y_{o,n}$, where $y_{o,n}$ is fresh. Otherwise, $orig(*^n y) = *^n y$. Also, $orig(\&y) = \&y$. Then γ is generalized in the natural way to use the extension of *orig*. Specifically, if f is a formal parameter of R with associated actual a, then

$$\gamma(*^m\ '*^n f) \quad = \quad *^m\ \ orig(*^n a)$$

where $m \geq 0$ and $n \geq 0$.

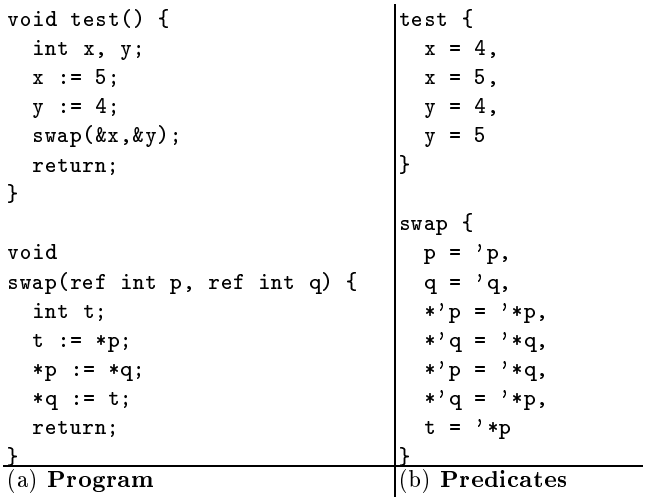
For example, consider the following simple procedure:

```
void simple(int* p) {
  *p = 8;
}
```

The symbolic constant $'*p$ refers to the value of $*p$ at the entry of the procedure. Assuming an actual parameter a to some call to `simple`, γ maps $'*p$ to $a_{o,1}$, which represents the value of $*a$ before the call. On the other hand, $*'p$ refers to the value obtained by dereferencing the original value of p. Note that unlike $'*p$, the value of $*'p$ can change through the procedure. For example, after the assignment to $*p$, $*'p$ has the value 8, while $'*p$ still has the original value of $*p$. Accordingly, γ maps $*'p$ to $*a_o$, which represents the updated value in the calling context.

Finally, the assignments used to update expressions in E_u proceed as described in Section 4, using the extended *orig* and γ mappings.

Appendix B contains the proof of soundness of the full C2BP algorithm including pointers.



```
void test() {
  int x, y;
  x := 5;
  y := 4;
  swap(&x,&y);
  return;
}

void
swap(ref int p, ref int q) {
  int t;
  t := *p;
  *p := *q;
  *q := t;
  return;
}
```

(a) **Program**

```
test {
  x = 4,
  x = 5,
  y = 4,
  y = 5
}

swap {
  p = 'p,
  q = 'q,
  *'p = '*p,
  *'q = '*q,
  *'p = '*q,
  *'q = '*p,
  t = '*p
}
```

(b) **Predicates**

Fig. 5. Polymorphic predicate abstraction with pointers

5.1 Example

Figure 5 shows an example involving pointers. The polymorphic predicates in the `swap` function allow all callers to prove that the values of $*p$ and $*q$ are swapped after the call to `swap` returns. Figure 6 shows the (simplified) boolean program output by C2BP.

The `swap` function has four binding predicates, $(p = 'p)$, $(q = 'q)$, $(*'p = '*p)$ and $(*'q = '*q)$, which are initialized to **true** at the beginning of the function as a result of abstracting the initialization statements introduced by the internal form. The translation of the three assignment statements uses the updated weakest precondition for pointers. For example, when translating the statement `*q := t`, we must consider the possibility that $*q$ is aliased to $*p$, in which case predicates involving $*p$ (and its aliases) may be affected. In this case, however, our alias analysis can easily deduce that $*p$ and $*q$ are not aliased, since the corresponding actuals are distinct variables x and y. Therefore several predicates need not be updated. For example, the weakest precondition of `*q := t` with respect to $(*'p = '*p)$ simplifies to $(*'p = '*p)$ itself given the results of the alias analysis, signifying that the predicate is indeed unaffected by the statement.

According to the definition of E_r, the predicates $(*'p = '*p), (*'q = '*q), (*'p = '*q)$, and $(*'q = '*p)$ are returned from `swap`. Consider the call to `swap` from the `test` function. Because both x and y may be modified by the call to `swap`, all four local predicates of `test` must be updated after the call. By the γ mapping, the returned predicates $\{ (*'p = '*p), (*'q = '*q), (*'p = '*q), (*'q = '*p) \}$ are mapped to the predicates $\{ (x = x_o), (y = y_o), (x = y_o), (y = x_o) \}$[10]. Further, the local predicates $\{ (x = 4), (y = 5), (x = 5), (y = 4) \}$ are mapped by γ to $\{ (x_o = 4), (y_o = 5), (x_o = 5), (y_o = 4) \}$. These mappings suffice to prove that the swap property holds. For example, we can deduce that if $(y_o = 4)$ and $(x = y_o)$ are true, then $(x = 4)$ is also true after the call to `swap`. This can be seen by the update to {`x=4`} after the call to `swap` in the abstraction of `test`.

[10] Technically, x should be $*\&x$, and similarly for y.

```
void test() {
  bool {x=4},{x=5},{y=4},{y=5};
  bool ret1,ret2,ret3,ret4;

  {x=5},{x=4} := true, false; // x := 5;
  {y=4},{y=5} := false, true; // y := 4;

  ret1,ret2,ret3,ret4 := swap();

  {x=5},{x=4},{y=5},{y=4} :=
    choose({y=5}&ret3 | {x=5}&ret1, ...),
    choose({y=4}&ret3 | {x=4}&ret1, ...),
    choose({y=5}&ret2 | {x=5}&ret4, ...),
    choose({y=4}&ret2 | {x=4}&ret4, ...);
  return;
}
```

```
bool,bool,bool,bool swap() {
  bool {p='p},{q='q};
  bool {*'p='*p},{*'q='*q},{*'p='*q},{*'q='*p};
  bool {t='*p};

  {p='p} := true;
  {q='q} := true;
  {*'p='*p} := true;
  {*'q='*q} := true;

  {t='*p} := {p='p} & {*'p='*p}; // t := *p;

  //*p := *q;
  {*'p='*q}, {*'p=*'p} :=
     choose({p='p}&{q='q}&{*'q='*q},
            {p='p}&{q='q}&!{*'q='*q}),
     choose({p='p}&{q='q}&{*'q='*p},
            {p='p}&{q='q}&!{*'q='*p});

  //*q := t;
  {*'q='*p},{*'q='*q} :=
     choose({q='q}&{t='*p},!{t='*p}), *;

  return {*'p='*p},{*'q='*q},
         {*'p='*q},{*'q='*p};
}
```

Fig. 6. Boolean program abstraction created by the C2BP tool, given the input program and predicates shown in Figure 5.

6 Related Work

Constructing abstract models of systems has been studied in several contexts [8, 7, 21]. Abstractions constructed by [14] and [20] are based on specifying transitions in the abstract system using a pattern language, or as a table of rules. Automatic abstraction support has been built into the Bandera tool set [13]. They require the user to provide finite-domain abstractions of data types. Predicate abstraction as implemented in C2BP is more general, capturing relationships among variables. Predicate abstraction was first introduced by Graf and Saidi [17]. Implementations of predicate abstraction have been reported in [10, 27, 6]. To the best of our knowledge, C2BP is the first tool to do predicate abstraction on a full scale programming language like C. The predicate abstraction tool reported in [29] does not deal with problems introduced by procedure calls, polymorphic predicates or pointers, as we have done here. Also, it is unclear if the abstractions produced by [29] are even sound in the presence of pointers and aliasing.

The use of polymorphism in programming languages [23, 5] and program analysis has a rich history. The ML programming language has a polymorphic type system that allows the definition of generic functions, whose types contain *type variables*. For example, the identity function, $I = \lambda x.x$, has type $\forall\alpha.\alpha \to \alpha$. A generic function can be safely typechecked once, no matter what types a particular calling context substitutes for the function's type variables. In our setting, we use symbolic constants to make the abstraction process for procedures generic.

Each procedure can be safely abstracted once, no matter what values a particular calling context substitutes for the procedure's symbolic constants.

A *context-sensitive* program analysis is one that analyzes each call to a given procedure Q, based on Q's calling context [28]. *Transfer functions* summarize the input-output behavior of a procedure and provide a way to avoid redundant work during context-sensitive analysis [28]. The tools C2BP and BEBOP together compute transfer functions on demand [1] (see also [30, 26]). Polymorphic predicates allow us to raise the level of abstraction of the transfer functions computed in C2BP, allowing more sharing and reuse among the different calling contexts. The use of polymorphism in transfer functions has been explored in the context of work on polymorphic points-to analysis [16]. Binding information between a calling procedure and called procedure has been discussed before in the context of program analyses [22].

7 Conclusion

We have presented an algorithm for polymorphic predicate abstraction of C programs and proven it sound. We have shown how a large number of monomorphic predicates, proportional to the number of call sites of a procedure, can often be equivalently represented by a constant number of polymorphic predicates. The main technical challenge is in capturing the effect of procedure calls on the local state of the caller, in the presence of symbolic constants and pointers. The precision of the C2BP algorithm for a subset of C has been discussed in [2]. We plan to extend this precision result for C2BP to include all constructs of C. We have used C2BP and the other SLAM tools to validate properties of Windows NT device drivers and to find invariants in several programs. More details on these experimental results can be found in [1].

Acknowledgements

We thank Rupak Majumdar for his hard work in making the C2BP tool come to life. Andreas Podelski provided a crucial insight that a symbolic constant can be thought of as a local variable to which the corresponding formal is assigned on entry to the procedure.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
2. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 268–283. Springer-Verlag, 2001.
3. 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.
4. D. Blei and et al. Vampyre: A proof generating theorem prover — http://www.eecs.berkeley.edu/~ rupak/vampyre.
5. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

7. E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL 92: Principles of Programming Languages*, pages 343–354. ACM, 1992.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
9. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.
10. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 00: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.
11. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover – http://research.compaq.com/src/esc/simplify.html.
12. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
13. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187, 2001.
14. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.
15. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02*, pages 191–202. ACM, January 2002.
16. J. S. Foster, M. Fahndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS 00: Static Analysis*, LNCS 1824, pages 175–198. Springer-Verlag, 2000.
17. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
18. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.
20. G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
21. R. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
22. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, 1992.
23. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
24. J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. D. Reidel Publishing Company, 1982.
25. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
26. 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.
27. H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.
28. 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.
29. W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *FMSP 00: Formal Methods in Software Practice*, pages 3–12. ACM, 2000.
30. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, 1995.

A Kleene's Three-Valued Logic

Figure 7 presents Kleene's interpretation for conjunction, disjunction, and negation of three-valued logic. This interpretation is used in the programs of the language presented in Section 3.

$\wedge$	**true**	**false**	$*$
true	**true**	**false**	$*$
false	**false**	**false**	**false**
$*$	$*$	**false**	$*$

$\vee$	**true**	**false**	$*$
true	**true**	**true**	**true**
false	**true**	**false**	$*$
$*$	**true**	$*$	$*$

$\neg$	
true	**false**
false	**true**
$*$	$*$

Fig. 7. Kleene's three-valued interpretation of $\wedge$, $\vee$ and $\neg$.

B Soundness

We represent a *program state* as a pair $\sigma = \langle L, \Omega \rangle$, where L is the label on the statement to be executed next and Ω is a store mapping the locations in scope at L to values. We sometimes extend Ω to expressions in the obvious way. The *initial state* of a program P is a state $\langle L, \Omega \rangle$ such that L labels the first statement of a distinguished "main" procedure and Ω maps all locations in scope to their initial values, as described in Section 3.

We say that $\langle L_1, \Omega_1 \rangle \longrightarrow \langle L_2, \Omega_2 \rangle$ if execution of the statement at L_1 with store Ω_1 produces store Ω_2 and moves to the statement labelled L_2. The semantics of the $\longrightarrow$ relation is standard. A *trace* of a program P is a sequence $\sigma_1 \longrightarrow \cdots \longrightarrow \sigma_n$ where the first state of the sequence is the initial state of P and where procedure calls and returns are properly matched: execution of a **return** statement transfers control to the statement after the most recent unmatched procedure call. Therefore a *trace* represents a valid prefix of an execution of P. Let $\longrightarrow^*$ denote the transitive closure of the $\longrightarrow$ relation. We sometimes denote traces using a sequence of states with every pair of successive states related by $\longrightarrow^*$, when the implicit intermediate states on the trace are of no importance.

Soundness intuitively means that every execution of a program has a corresponding execution in the associated boolean program abstraction. The main theorem we prove says that one execution step of a program has a corresponding sequence of execution steps in the associated boolean program abstraction. First we define a simulation relation on program states, which captures the notion that b-variables are conservative abstractions of the expressions they represent.

Definition 1. *Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the boolean program abstraction computed by* C2BP. *Let V be the b-variables in B and $\mathcal{E}$ be the mapping from V to E. We say that a state $\langle L_1, \Omega_1 \rangle$ of P* is simulated by *a state $\langle L_2, \Omega_2 \rangle$ of B if $L_1 = L_2$ and for all b-variables b in scope at L_2 we have that:*

$$(\Omega_2(b) = \textbf{true} \Rightarrow \Omega_1(\mathcal{E}(b)) = \textbf{true}) \text{ and } (\Omega_2(b) = \textbf{false} \Rightarrow \Omega_1(\mathcal{E}(b)) = \textbf{false})$$

Our theorem then says that simulation is preserved by the $\longrightarrow$ relation.

Theorem 1. *Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the boolean program abstraction computed by* C2BP, *with variables V and mapping $\mathcal{E}$ from V to E. Let $\sigma_1 \longrightarrow \cdots \longrightarrow \sigma_k \longrightarrow \sigma$ be a trace of P and $\sigma'_1 \longrightarrow^* \cdots \longrightarrow^* \sigma'_k$ be a trace of B. If for all $1 \leq i \leq k$ it is the case that σ_i is simulated by σ'_i, then there exists some σ' such that $\sigma'_1 \longrightarrow^* \cdots \longrightarrow^* \sigma'_k \longrightarrow^* \sigma'$ is a trace of B and σ is simulated by σ'.*

Proof. Since σ_k is simulated by σ'_k, we have that σ_k has the form $\langle L, \Omega_k \rangle$ and σ'_k has the form $\langle L, \Omega'_k \rangle$. Let s be the statement labelled L in P and s' be the statement labelled L in B. Case analysis of the form of s:

- $s =$ **skip**. Then $s' =$ **skip** as well. By the semantics of **skip** we have $\sigma = \langle L_1, \Omega_k \rangle$ where L_1 is the label of the statement following s. We also know that $s' =$ **skip** as well, so there exists $\sigma' = \langle L_1, \Omega'_k \rangle$ such that $\sigma'_k \longrightarrow \sigma'$. Then since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.
- $s =$ **goto** L_1. Then $s' =$ **goto** L_1 as well. By the semantics of **goto** we have $\sigma = \langle L_1, \Omega_k \rangle$ and $\sigma' = \langle L_1, \Omega'_k \rangle$, where $\sigma'_k \longrightarrow \sigma'$. Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.
- $s =$ **branch** $\overline{s_1} \,||\, \cdots \,||\, \overline{s_n}$ **end**. Then $s' =$ **branch** $BP(\overline{s_1}, V, \mathcal{E}) \,||\, \cdots \,||\, BP(\overline{s_n}, V, \mathcal{E})$ **end**. By the semantics of **branch** we have $\sigma = \langle L_1, \Omega_k \rangle$, where L_1 is the label of the first statement in one of the branch cases. Then by the semantics of **branch**, there exists some $\sigma' = \langle L_1, \Omega'_k \rangle$ such that $\sigma'_k \longrightarrow \sigma'$. Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.

- s is an assignment statement: Then $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the statement following s. By the abstraction process we know that s' is a parallel assignment, so also there exists $\sigma' = \langle L_1, \Omega' \rangle$, such that $\sigma'_k \longrightarrow \sigma'$. Now suppose $\Omega'(b)$ is **true**, for some b-variable b in scope at L_1 in B. To finish this case, we show that $\Omega(\mathcal{E}(b))$ is **true** as well. (A similar proof can be done for the case in which $\Omega'(b)$ is **false**.)
  Since $\Omega'(b)$ is **true**, we know by the abstraction of assignments that some cube in $\mathcal{F}_{V,\mathcal{E}}(WP(s, \mathcal{E}(b)))$ was **true** in Ω'_k. (Note that no cube can be in both $\mathcal{F}_{V,\mathcal{E}}(WP(s, e))$ and $\mathcal{F}_{V,\mathcal{E}}(WP(s, \neg e))$.) Let that cube be $c_1 \wedge \cdots \wedge c_m$, so for each $0 \leq r \leq m, \Omega'_k(c_r) =$ **true**. Since σ_k is simulated by σ'_k, we have that for each $0 \leq r \leq m, \Omega_k(\mathcal{E}(c_r)) =$ **true**. By the definition of $\mathcal{F}_{V,\mathcal{E}}$ we have $\mathcal{E}(c_1) \wedge \cdots \wedge \mathcal{E}(c_m) \Rightarrow WP(s, \mathcal{E}(b))$. Then by definition of WP, $\Omega(\mathcal{E}(b))$ is **true** as well.
- s is of the form **assume**(e): Then $\sigma = \langle L_1, \Omega_k \rangle$, where L_1 is the label of the statement following s. By the abstraction process we know that s' is **assume**$(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$. Suppose σ'_k can take an evaluation step to some σ'. Then by the semantics of **assume** σ' will have the form $\langle L_1, \Omega'_k \rangle$, Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.
  Therefore, we just need to prove that σ'_k can take an evaluation step to some σ', which is the case according to the semantics of **assume** if $\Omega'_k(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$ is **true** or $*$. Since $\sigma_k \longrightarrow \sigma$, we know that $\Omega_k(e)$ is **true**. Therefore $\Omega_k(\neg e)$ is **false**. By definition, $\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e)) \Rightarrow \neg e$, so also $\Omega_k(\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e)))$ is **false**. Therefore $\Omega_k(\neg\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e))) = \Omega_k(\mathcal{E}(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e)))$ is **true**. Finally, since σ_k is simulated by σ'_k we have $\Omega'_k(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$ is **true** or $*$.
- s is a procedure call of the form $v := R(a_1, a_2, \ldots, a_j)$: Then $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the first statement in procedure R and where Ω maps global variables to their current values (the same values they have in Ω_k), maps formal parameters to the values of their actuals, and maps local variables of R to their appropriate initial values. By the abstraction process, s' is a call to R', the version of R in B, preceded by assignment statements which compute the values of the actual parameters to the call. Therefore, there exists some $\sigma' = \langle L_1, \Omega' \rangle$, where $\sigma'_k \longrightarrow^* \sigma'$ and Ω' maps global variables to their current values (the same values they have in Ω'_k), maps formal parameters to the values of their actuals, and maps local variables of R' to $*$. Suppose $\Omega'(b)$ is **true**, for some b in scope. To finish this case, we show that also $\Omega(\mathcal{E}(b))$ is **true**.
  - If b is a global, then we saw above that $\Omega'(b) = \Omega'_k(b)$, so $\Omega'_k(b)$ is **true**. Then since Ω_k is simulated by Ω'_k we have that $\Omega_k(\mathcal{E}(b))$ is **true**. Since b is global, we know that $\mathcal{E}(b)$ refers only to global variables of P. Since we saw above that for each such variable x we have $\Omega(x) = \Omega_k(x)$, also $\Omega(\mathcal{E}(b)) = \Omega_k(\mathcal{E}(b))$. Therefore $\Omega(\mathcal{E}(b))$ is true.
  - If b is a formal of R', then b has the value of the associated actual parameter to the call. Then by the process for computing actuals, some cube in $\mathcal{F}_{V,\mathcal{E}}(\mathcal{E}(b)[a_1/f_1, \cdots, a_j/f_j])$ was **true** in Ω'_k, where f_i is the associated formal for actual a_i. Let that cube be $c_1 \wedge \cdots \wedge c_m$, so for each $0 \leq r \leq m, \Omega'_k(c_r) =$ **true**. Then since σ_k is simulated by σ'_k we have that for each $0 \leq r \leq m, \Omega_k(\mathcal{E}(c_r)) =$ **true**. By the definition of $\mathcal{F}_{V,\mathcal{E}}$ we have $\mathcal{E}(c_1) \wedge \cdots \wedge \mathcal{E}(c_m) \Rightarrow \mathcal{E}(b)[a_1/f_1, \cdots, a_j/f_j]$, so $\Omega_k(\mathcal{E}(b)[a_1/f_1, \cdots, a_j/f_j])$ is **true**. Therefore by the definition of Ω we also have that $\Omega(\mathcal{E}(b))$ is **true**.
  - Otherwise, b is a local variable. But then we saw above that b has the value $*$, contradicting the fact that $\Omega'(b)$ is **true**.
- s is a return statement of the form **return** r. Then $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the first statement after the last unmatched procedure call in P's trace. Statement s' is also a **return** statement. Since σ_i is simulated by σ'_i for $1 \leq i \leq k$, the two traces must agree on the last unmatched procedure call. Then there exists an appropriate $\sigma' = \langle L_1, \Omega' \rangle$ such that $\sigma'_k \longrightarrow \sigma'$.
  Let $\langle \hat{L}, \hat{\Omega} \rangle$ represent the program state before that last unmatched procedure call in P's trace. Similarly, let $\langle \hat{L}, \hat{\Omega}' \rangle$ represent the program state before computing actual parameters for the last unmatched procedure call in B's trace. Let the call in P have the form $v := R(a_1, a_2, \ldots, a_j)$, and let it appear in procedure Q. Then the call in procedure Q' of B has the form $ret_1, \ldots, ret_r := R'(prm_1, \ldots, prm_p)$, followed by updates of the b-variables associated with expressions in E_u.
  Suppose $\Omega'(b)$ is **true**, for some b in scope. To finish this case, we show that also $\Omega(\mathcal{E}(b))$ is **true**. There are three cases to consider:
  - First, suppose b is a global b-variable. Then b was in the scope of the call, and since each ret_i and each b-variable in E_u is a local variable, (and hence b is not in E_u) we have $\Omega'(b) = \Omega'_k(b)$. So we have $\Omega'_k(b)$ is **true**, and since σ_k is simulated by σ'_k also $\Omega_k(\mathcal{E}(b))$ is **true**. We know that v is not a global variable (see definition of internal form in Section 4) and $\mathcal{E}(b)$ may only refer to global variables, so $\Omega_k(\mathcal{E}(b)) = \Omega(\mathcal{E}(b))$, and therefore $\Omega(\mathcal{E}(b))$ is **true**.

- Second, suppose b is a local variable or formal parameter of Q. We have two sub-cases. First suppose that $\mathcal{E}(b) \notin E_u$, the set of expressions local to Q whose associated b-variables are updated after the call to R. Since b is a local variable or formal of Q and since each ret_i variable is fresh (and hence is not b), the value of b cannot be affected by the call to R, so $\hat{\Omega}'(b) = \Omega'(b)$ and $\hat{\Omega}'(b)$ is **true**. Then since $\langle \hat{L}, \hat{\Omega} \rangle$ is simulated by $\langle \hat{L}, \hat{\Omega}' \rangle$, we have $\hat{\Omega}(\mathcal{E}(b))$ is **true**. Since $b \notin E_u$, by the definition of E_u we have that the value of $\mathcal{E}(b)$ cannot change as a result of the call to R, so $\hat{\Omega}(\mathcal{E}(b)) = \Omega(\mathcal{E}(b))$. Therefore $\Omega(\mathcal{E}(b))$ is **true**.
- Finally, suppose $\mathcal{E}(b) \in E_u$. For this case, we conceptually consider the call to R in P to instead consist of the two statements

$$v_{ret} := R(a_1, a_2, \ldots, a_j); v := v_{ret}$$

where v_{ret} is a fresh local variable in Q. Intuitively, the point between the two statements represents the state when the call has returned but the return value has not yet been assigned to v. Additionally, just before the call we conceptually add a statement of the form $x_o := x$; for each location x in scope such that $orig(x) = x_o$, where x_o is fresh. Similarly we add a statement of the form $x_{o,n} := *^n x$ for each location $*^n x$ in scope such that $orig(*^n x) = x_{o,n}$, where $x_{o,n}$ is fresh. Clearly the semantics of the call is preserved by these modifications.
Let $\hat{\mathcal{E}}$ be the mapping from b-variables in scope of Q' and the ret_i variables to expressions, as defined in Section 4. Let Ω_r be the store at the point in (the conceptually-revised) P between the two statements above, and let Ω'_r be the store at the point in B just after the call to R', before the updates to the associated b-variables of expressions in E_u. Our strategy is to show that for all b-variables b_0 in the domain of $\hat{\mathcal{E}}$:

$$(\Omega'_r(b_0) = \textbf{true} \Rightarrow \Omega_r(\hat{\mathcal{E}}(b_0)) = \textbf{true}) \text{ and } (\Omega'_r(b_0) = \textbf{false} \Rightarrow \Omega_r(\hat{\mathcal{E}}(b_0)) = \textbf{false})$$

Suppose we can show this is the case. Then, since b (the b-variable we are currently considering in E_u) is subsequently assigned in B to $\texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{v} := \texttt{v}_{\texttt{ret}}, \hat{\mathcal{E}}(b))), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{v} := \texttt{v}_{\texttt{ret}}, \neg\hat{\mathcal{E}}(b))))$ and the current statement in P is $v := v_{ret}$, it follows from the same argument as in the case above for assignment statements that $\Omega(\mathcal{E}(b))$ is **true**.
Therefore, suppose $\Omega'_r(b_0) = \textbf{true}$, for some b_0 in the domain of $\hat{\mathcal{E}}$. According to the definition of $\hat{\mathcal{E}}$, there are three cases to consider:
  - b_0 is a global b-variable in $V_{Q'}$, and $\hat{\mathcal{E}}(b_0) = \mathcal{E}_Q(b_0) = \mathcal{E}(b_0)$. Since b_0 is global, it is in scope of R also, and since each ret_i is a local variable (and hence is not b_0), also b_0 is **true** in Ω'_k. Then since σ_k is simulated by σ'_k also $\Omega_k(\mathcal{E}(b_0))$ is **true**. We know that v_{ret} is not a global variable (see definition of internal form in Section 4) and $\mathcal{E}(b_0)$ may only refer to global variables, so $\Omega_k(\mathcal{E}(b_0)) = \Omega_r(\mathcal{E}(b_0))$, and therefore $\Omega_r(\mathcal{E}(b_0))$ is **true**.
  - b_0 is a local b-variable in $V_{Q'}$, and $\hat{\mathcal{E}}(b_0) = orig(\mathcal{E}_Q(b))$. Since b_0 is a local b-variable and is **true** in Ω'_r, it is also **true** in $\hat{\Omega}'$, since the values of locals cannot be changed by the call to R' and each ret_i variable is fresh (and hence is not b_0). Then since $\langle \hat{L}, \hat{\Omega} \rangle$ is simulated by $\langle \hat{L}, \hat{\Omega}' \rangle$, we have $\hat{\Omega}(\mathcal{E}_Q(b_0))$ is **true**. Then, by the assignments $x_o := x$ and $x_{o,n} := *^n x$ before the call, that means that $orig(\mathcal{E}_Q(b_0))$ is true in $\hat{\Omega}$. Consider a variable in $orig(\mathcal{E}_Q(b_0))$. By the definition of $orig$, all locations in $\mathcal{E}_Q(b_0)$ that can be affected by the call to R are replaced by fresh local *original* variables. Therefore no locations in $orig(\mathcal{E}_Q(b_0))$ can be affected by the call to R. Further, none of those locations can alias the ret_i variables, since those are fresh. Therefore, $orig(\mathcal{E}_Q(b_0))$ is true in Ω_r.
  - b_0 is a local of the form ret_i returned, and $\hat{\mathcal{E}}(b_0) = \gamma(\mathcal{E}_{RQ}(b_0))$. We need to prove that $\Omega_r(\gamma(\mathcal{E}_{RQ}(b))$ is **true**. Since b_0 is **true** in Ω'_r and is a returned b-variable from R', we know that b_0's counterpart b'_0, which is returned from R' in its **return** statement, is **true** in Ω'_k also. Therefore, since σ_k is simulated by σ'_k we have that $\Omega_k(\mathcal{E}_R(b'_0))$ is **true**. That is equivalent to saying that $\Omega_k(\mathcal{E}_{RQ}(b_0))$ is **true**. Therefore, we can conclude that $\Omega_r(\gamma(\mathcal{E}_{RQ}(b_0)))$ is **true** if we can show that for all locations l in the domain of γ, $\Omega_k(l) = \Omega_r(\gamma(l))$. We do a case analysis on l. We have three cases:
    - $var(l)$ is a global, so $\gamma(l) = l$. Since locations don't change values as a result of the **return** and the return value is assigned to the fresh v_{ret}, the result follows.
    - $var(l)$ is the return variable r, so $\gamma(l) = l[v_{ret}/var(l)]$. Since the return value of the call is assigned to v_{ret}, the result follows.

- $var(l)$ is a symbolic constant $'*^n f$, so $\gamma(l) = l[orig(*^n a)/var(l)]$, where a is the actual for formal f. By the internal form definition, the symbolic constant $'*^n f$ is a local of R' that is assigned the value of $*^n f$ at the entry point of the procedure, and never modified. We have two cases. First suppose $orig(*^n a) = *^n a$. By the definition of $orig$, the value of $*^n a$ is not modified by the call, so $\hat{\Omega}(orig(*^n a)) = \Omega_r(orig(*^n a))$. Since $*^n f$ at the entry point of the procedure has the same value as $*^n a$ before the call, we have $\Omega_k('*^n f) = \hat{\Omega}(orig(*^n a))$. So we have shown that $\Omega_k('*^n f) = \Omega_r(orig(*^n a))$, which also implies that $\Omega_k(l) = \Omega_r(\gamma(l))$.
  Second, suppose $orig(*^n a) = a_{o,n}$. By our conceptual revision of P, $a_{o,n}$ is assigned the value $*^n a$ just before the call. So we have $\Omega_k('*^n f) = \hat{\Omega}(orig(*^n a))$. Since $a_{o,n}$ is a local variable, it is not modified by the call, so $\hat{\Omega}(orig(*^n a)) = \Omega_r(orig(*^n a))$. So we have shown that $\Omega_k('*^n f) = \Omega_r(orig(*^n a))$, which also implies that $\Omega_k(l) = \Omega_r(\gamma(l))$.

Finally, soundness follows as a simple corollary to the above theorem:

Corollary 1. *(Soundness) Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the boolean program abstraction computed by* C2BP*. If $\sigma_1 \longrightarrow \cdots \longrightarrow \sigma_k$ is a trace of P, then there exists a trace $\sigma'_1 \longrightarrow^* \cdots \longrightarrow^* \sigma'_k$ of B such that for all $1 \leq i \leq k$, σ_i is simulated by σ'_i.*

Proof. The proof is by induction on k. For the base case, suppose $k = 1$, so the trace in P consists solely of σ_1, the initial state. Let $\sigma_1 = \langle L, \Omega \rangle$, where L is the label of the first statement in the "main" procedure. Then there is an initial state σ'_1 of B, of the form $\langle L, \Omega' \rangle$. Further, since the initial values of b-variables is $*$, all b-variables in the domain of Ω' have value $*$. Therefore, σ_1 is simulated by σ'_1 vacuously.

For the inductive case, suppose k is some $j > 1$, and assume the corollary holds for traces of P of length smaller than j. Since we're given that $\sigma_1 \longrightarrow \cdots \longrightarrow \sigma_j$ is a trace of P, so is $\sigma_1 \longrightarrow \cdots \longrightarrow \sigma_{j-1}$. Then by the inductive hypothesis there exists a trace $\sigma'_1 \longrightarrow^* \cdots \longrightarrow^* \sigma'_{j-1}$ of B such that for all $1 \leq i < j$, σ_i is simulated by σ'_i. Then by Theorem 1 there exists some σ'_j such that $\sigma'_1 \longrightarrow^* \cdots \longrightarrow^* \sigma'_j$ is a trace of B and σ_j is simulated by σ'_j, so the result follows.