Efficient Directionless Weakest Preconditions

Ivan Jager, David Brumley

February 2, 2010

CMU-CyLab-10-002

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Paper originally prepared November 13, 2009

Efficient Directionless Weakest Preconditions

Ivan Jager David Brumley
Carnegie Mellon University
aij@andrew.cmu.edu dbrumley@cmu.edu

Abstract

Verification condition (VC) generation is a fundamental part of many program analysis and applications, including proving program correctness, automatic test case generation, proof carrying code, and others. In all application domains, there are two critical factors for VC generation algorithms: compact final VCs and fast VC generation. Compact VCs save more than bits; empirically compact formulas are easier to reason about in subsequent steps such as VC verification [12, 18].

The theoretically most efficient algorithms for generating VCs are based upon weakest preconditions (WP). Current WP algorithms iterate over program statements *backwards* from the last statement in the program to the first, and can generate a VC that is at most $O(M^2)$ the size of a program. In practice, however, application domains that rely on VC generation often opt to use forward symbolic execution (FSE), which works in the *forward* direction from the first program statement to the last. Forward-based algorithms are attractive in practice because they easily afford optimizations such as eliminating constant expressions from VCs during generation. FSE, however, is theoretically exponentially worse than WP-based algorithms — it produces a separate VC for each program path, thus the final VC for all paths is $O(2^M)$.

We propose a new directionless weakest precondition that can be run in both the forward and backward direction. Our algorithm provides a $O(M^2)$ VC generation time and predicate size while affording optimizations that make FSE attractive in practice. We provide end-to-end proofs of correctness, size, and generation time. We then show how our approach leads to a proof of equivalence between VCs generated with WPs and FSE for typical structured programs.

1. Introduction

A fundamental primitive in program analysis is the ability to automatically generate a formula that captures the logical semantics of the a program fragment and its correctness properties. Such formulas are called a *verification condition* (VC). VCs allow us to reduce the problem of reasoning about programs to reasoning in logic.

We need algorithms that automatically and efficiently generate compact VCs. In particular, a VC that has as few redundancies as possible will be more compact and thus preferable to a VC with redundancies, e.g., $a \wedge b \wedge c$ is preferable to $(a \wedge b \wedge c) \wedge (a \wedge b)$. Compact VCs improve the performance of subsequent steps that use the VC because they need not consider the redundant parts each time they appears. For example, typical strategies employed by theorem provers will try to prove redundant VC conditions each time they appear syntactically, which can often push a theorem prover off the proverbial exponential cliff [12, 18]. In our experience, redundancies in formulas have led to situations where a theorem prover cannot even parse the formula because it is too big, while the theorem prover can easily parse and prove the more compact representation.

Application domains that benefit from fast and compact VC generation include:

1. Proving program correctness. The VC corresponds to a program satisfying a desired specification. A program is correct if the verification condition is true. Compact VCs tend to be easier to prove true in practice [12, 18].
2. Automatic test-case generation and fuzzing. Free variables in a VC correspond to program input variables, and an assignment of values to free variables which satisfy the VC correspond to inputs that execute the program successfully. Thus, solutions to VCs are valid program inputs. By successively generating and solving VCs for desired program fragments we can automatically generate input test cases of corresponding fragments (and in the process also potentially test for boundary conditions such as division by zero), such as in [6, 7, 15, 16, 22]. Thus, any improvements to VC generation time and making VCs easier to solve will in turn improve the effectiveness of automatic test case generation.
3. Proof carrying code (PCC) [20]. PCC generates a VC for program safety and proves it, similar to proving program correctness. The proof is then attached to the code. A remote party can verify the PCC code is indeed safe by recomputing the VC and verifying the attached proof works (instead of re-proving the VC itself). Faster VC generation helps both parties. Additionally compact VCs that avoid redundancy will have smaller proofs, thus are faster to check.
4. Input filter generation. Intrusion prevention/detection systems use input filters to filter out inputs that trigger known bugs and vulnerabilities. Recent work has shown that automatic VC generation can be adapted to automatically generate input filters. [4, 5, 9, 10] Faster VC generation corresponds to more quickly generating a filter, thus more quickly defending against any threats posed by the underlying bug. Compact VCs result in faster filtering, thus reducing the overall cost of the security solution.

Since these and other similar approaches scale with respect to the VC generation and verification process, one one would imagine that all applications would use the VC generation algorithm that has the best theoretical guarantees. This is not the case, however. The most efficient (theoretic) algorithm for generating compact VCs is based upon weakest preconditions, and runs in $O(M^2)$ time and generates a VC at most $O(M^2)$ the size of an M-line program [5, 12, 18]. Nonetheless, in practice many application of VC generation use forward symbolic execution, which generates a separate VC for each program path thus has an exponentially worse (theoretic) VC of size $O(2^M)$ for a program with M statements (and with many typically implementations running in $O(2^M)$ time in order to explore all paths). [1]

[1] Note that some implementations do not syntactically create a single formula for all program paths. Instead, they query a theorem prover for each

```
1.  int a = 5;
2.  int b = get_input();
3.  int c = a+a;
4.  foo(c);
5.  assert( (b+b) < 10);
```

Figure 1: An example for demonstrating the differences between predicates generated by forward symbolic execution and weakest preconditions.

Forward symbolic execution and weakest precondition algorithms differ in the direction they process program statements during VC generation. Forward symbolic execution instruments a program to return a fresh variable at each input statement (instead of an actual concrete value). The program is then "executed" by an interpreter (see S̃ 4 for a precise description). Just like normal execution, a symbolic interpreter usually executes only one path at a time. Symbolic execution builds up expressions in terms of the symbolic input variables. The built-up expressions become the VC. Thus, forward symbolic execution processes statements from the first to the last. Calculating the weakest precondition, however, is a backward computation that starts with the desired post-state, called a post-condition. Weakest precondition algorithms tell how to derive the post-condition for statement $i-1$ given statement i and the post-condition at statement i. Thus, weakest preconditions process statements last to first, which is the opposite of forward symbolic execution.

To see why processing statements first to last as with forward symbolic execution is sometimes attractive, consider generating a VC for the program shown in Figure 1. Modern implementations of forward symbolic execution, e.g., [6, 7, 9, 15–17], will realize that a is constant on line 1. They can then implement dynamic optimizations such that concrete values are calculated for line 3 and 4. The actual code executed for these computations will not be explicitly included in the final VC. Thus, modern forward symbolic execution will produce the VC (assuming no post-condition) as $(b+b) < 10$. Such tricks have proven essential in scaling applications of VC generation in many domains [7–9, 15–17]. In addition, VCs generated with forward symbolic execution are typically quantifier-free formulas, which is attractive in many application domains. For instance, the decision procedure STP [13] used in KLEE [6] does not support universally quantified VCs (i.e., STP does not fully support first order logic).

A weakest precondition-based approach, however, will generate a VC for all statements, including the code for the function `foo`, functions called by `foo`, and so on. The reason is weakest precondition works backward, thus when performing VC generation for line 4, the analysis does not yet know c is a constant. While after calculating the weakest precondition one could implement an optimization to simplify constant expressions, it does not fundamentally change the fact that such optimizations would need to be implemented in software (e.g., as an implementation of constant folding) and performed as a post-processing step. Finally, efficient weakest preconditions algorithms result in a VC with universal quantification [12, 18], which prevents many applications from using efficient weakest preconditions as a drop-in replacement for forward symbolic execution.

Contributions. In this paper we present a new algorithm for calculating weakest preconditions that can process statements in both the forward and backward direction. We call this algorithm a direc-

path's VC. Note, however, that querying on each path is equivalent to creating one formula that is the disjunction of all path formulas (see § 5).

$x := e$	Assign variable x to the value of e.
assert e	Aborts if e is not true.
assume e	Only executes when e is true.
$S_1; S_2$	Executes S_1 then S_2.
$S_1 \square S_2$	Executes either S_1 or S_2.

Table 1: The Guarded Command Language.

tionless weakest precondition (DWP). Our algorithm allows us to take advantage of optimizations that make forward symbolic execution attractive in practice while providing the better $O(M^2)$ guarantees (§ 3 Theorems 2, 4) offered by previous (backwards) weakest precondition algorithms [12, 18].

We then show equivalence between weakest preconditions and forward symbolic execution for a certain class of programs (§ 4, Theorem 9). Our algorithm highlights the deep underlying similarity between the forward symbolic and weakest precondition approaches. This equivalence means that our algorithm can be used as a "drop in" replacement for forward symbolic execution to produce (exponentially) more compact VC, assuming that in the problem setting we can compute a dynamic single assignment form of the program. The VCs for this class of programs can be quantifier-free, which makes VC verification amenable to a larger number of theorem provers. In addition, the correspondence means any improvements to either wp or FSE can be adapted to benefit the other.

2. Weakest Preconditions

2.1 The Guarded Command Language

The weakest precondition is calculated over programs written in the guarded command language (GCL). The GCL we use is shown in Table 1, and is identical to that in related work [12, 18]. Although the GCL may look simple, it is general enough that common languages as diverse as Java [12] and x86 assembly [5] can be de-sugared into GCL statements. The translation from typical programming languages into the GCL (and possible trade-offs) is addressed elsewhere, e.g., [12]. We do note, however, that translating to the GCL on the fly, or adapting the rules to suit the original programming language constructs without translating them into the GCL explicitly, is often straight-forward. Our focus is on creating a verification condition for programs already in the GCL.

A program in the GCL is written as a sequence of GCL statements. We place no limits on expressions other than they be side-effect free. Typical binary operators, unary operators, memory operations, etc., can all be used. Executing a statement in the GCL may either terminate normally or "goes wrong". An assert e statement terminates normally if e is true, and goes wrong otherwise. assume e terminates normally if e is true, else simply cannot execute. The assignment statement $x := e$ updates the program state so that the location x contains the computed value of e. $S_1; S_2$ denotes a statement sequence. Finally, we have $S_1 \square S_2$, called a "choice" statement, which denotes executing either S_1 or S_2 as allowed by the guards in S_1 and S_2.

If-Then-Else Statements. Conditional statements in other languages are modeled as choice statements in the GCL. For example, if e then S_1 else S_2 becomes:

$$(\text{assume } e; S_1) \square (\text{assume } \neg e; S_2) \quad (1)$$

As described in § 4, when showing equivalence to forward symbolic execution semantics, we require that every choice statement is of the above form. In particular, note that the same predicate e

appears on both branches of the choice. Common applications of VC generation fulfill this assumption.[6, 7, 15, 16, 20, 22].

Processing loops. Our approach, like similar previous work (e.g., [6, 12, 15, 18]), targets acyclic GCL programs. If program invariants are available, loops can also be de-sugared directly into the GCL shown in Table 1 (e.g., using algorithms described in [12]). An arbitrary program can be made acyclic by bounding the number of loop iterations considered. Bounding loop iterations turns the problem of computing such predicates into a something that can be done automatically based upon program syntax alone, albeit results are with respect to the upper loop bounds.

Forward symbolic execution also produces a predicate for only acyclic fragments of a program at a time. Thus, we also consider it restricted to acyclic programs since the confluence of all paths executed at any point in time is acyclic. Similarly, we can unroll loops for weakest preconditions as many times as a forward algorithm would execute a loop. Note bounding loop iterations is common in symbolic execution. For example, suppose n is input in:

```
while((3^n + 4^n) = 5^n) { n++; ... }
```

Forward symbolic execution would build a VC for 1 loop iteration with the branch guard $3^n+4^n = 5^n$, a VC for the 2nd iteration as $3^{n+1} + 4^{n+1} = 5^{n+1}$, and so on. At each iteration symbolic execution needs to decide whether to execute the loop once again (potentially in an effort to reach an interesting statement inside the loop), or to exit the loop and continue with the rest of the program. Providing upper bounds keeps forward symbolic execution from getting "stuck" in such potentially infinite or long-running loops.

2.2 Traditional Weakest Preconditions

The weakest precondition wp for a program P and post-condition Q is a Boolean predicate such that when $wp(P, Q)$ holds, executing P is guaranteed to terminate in a state satisfying Q. By "weakest" we mean that for all other predicates R that result in P terminating in a state satisfying Q, we have $R \Rightarrow wp(P, Q)$.

The weakest precondition is a backward, automatic, syntax driven calculation. As a result, given a GCL program we can *automatically* calculate the weakest precondition. Given a post-condition Q at statement i, we calculate the precondition for executing i such that Q will be satisfied. The calculation is driven by GCL predicate transformers, and there is one for each program statement type. The GCL predicate transformers are shown in Table 2. Most rules are self-explanatory. The rule $Q[e/x]$ indicates substituting all occurrences of x for e in Q.

Example 1. Suppose we have a language where all values and variables are 32-bit unsigned integers, and the following program where x is input:

```
if (x%2 = 1) { s := x+2; } else { s := x+3; }
```

The GCL for this program is:

$$(\mathsf{assume}\ x\%2 = 1; s := x + 2)\Box$$
$$(\mathsf{assume}\ \neg(x\%2 = 1); s := x + 3)$$

Let Q be the condition that s will not be the result of arithmetic overflow, i.e., $Q = \neg(s < x)$. The weakest precondition is calculated automatically using the rules from Table 2 as:

$$x\%2 = 1 \Rightarrow \neg((x + 2) < x) \wedge \neg(x\%2 = 1) \Rightarrow \neg((x + 3) < x)$$

This predicate is only satisfied by an assignment of values to x that result in overflow. If there is no such assignment, i.e., the predicate is unsatisfiable, then we are guaranteed the program will always terminate in a state such that overflow did not occur.

Of course overflow can happen in the example program when $x \in \{2^{32} - 3, 2^{32} - 2, 2^{32} - 1\}$. In practice, satisfiability would be decided by giving the above VC to a decision procedure.

2.3 Previous Work in Efficient Weakest Preconditions

The traditional weakest precondition predicate transformers shown in Table 2 can result in a verification condition exponential in the size of the program. The exponential blowup is due to 1) choice statements where the post-condition Q is duplicated along both branches, and 2) substitution in assignment statements. An example of the latter is:

```
wp(x = x+x; x = x+x; x=x+x, x < 10)
```

The final weakest precondition for this program will be $x+x+x+x+x+x+x+x < 10$ due to the substitution rule. The exponential blowup is not just a theoretic nuisance; exponential blowup is commonly encountered in real verification tasks. Duplication in choice statements and assignments have a negative synergistic effect, e.g., if a choice statement $S_1 \Box S_2$ preceded the above example all 8 xs would be passed to both S_1 and S_2 branches.

Flanagan and Saxe developed a weakest precondition algorithm that avoids exponential blowup during VC generation [12]. Their algorithm works on assignment-free programs, called passified programs. Their algorithm rests upon the the following equality (as pointed out by Leino [18]):

Theorem 1. For all assignment-free programs P:

$$wp(P, Q) = wp(P, \mathrm{T}) \wedge (wlp(P, \mathrm{F}) \vee Q)$$

where T, F are respectively the logical constants *true* and *false* and wlp is the *weakest liberal precondition*. The predicate transformers for the weakest liberal precondition are the same as for the weakest precondition except $wlp(\mathsf{assert}\ e, Q)$, as shown in Table 2. Unlike the semantics of the weakest precondition, the weakest liberal precondition does not require that the program terminate normally; only that if it does terminate, that Q holds. The key part of Theorem 1 is the post-condition during VC generation is always a constant.

3. Directionless Weakest Preconditions

In this section we first describe an initial directionless weakest precondition algorithm that results in a VC $O(M^4)$ in the number of program statements M. By directionless we mean it can run forward (first to last) or backward (last to first) over a program. We prove correctness, and then show how to extend the algorithm (in a way that maintains correctness) to generate VCs only $O(M^2)$ in size of the original program.

3.1 Initial Directionless Weakest Precondition Algorithm

In this section we focus on producing VCs that (given no additional optimizations) will be syntactically equal to those of Flanagan and Saxe [12] since such VCs have previously been shown more efficient to solve in practice. Unlike previous work such as Flanagan and Saxe, however, our algorithm can take advantage of forward-based optimizations, as described in the next section.

Passified programs. Our algorithm works on passified GCL programs [12, 18]. A passified GCL program is assignment-free. The high-level intuition for removing assignments is to convert them into statements about equality. For simplicity, we assume here as a pre-processing step we statically passify the program by replacing assignment statements with assume statements. In § 5, we revisit this and discuss performing passifying on-the-fly, such as needed when VC generation is performed dynamically. The algorithm to transform an GCL program into a passified program is:

1. Put P into dynamic single assignment (DSA) form P' where every variable will dynamically be assigned at most once. For

S	$wp(S, Q)$	$wlp(S, Q)$	$wp(S, \mathrm{T})$	$wlp(S, \mathrm{F})$
$x := e$	$Q[e/x]$	$Q[e/x]$	T	F
assert e	$e \wedge Q$	$e \implies Q$	e	$\neg e$
assume e	$e \implies Q$	$e \implies Q$	T	$\neg e$
$S_1; S_2$	$wp(S_1, wp(S_2, Q))$	$wlp(S_1, wlp(S_2, Q))$	$wp(S_1, wp(S_2, \mathrm{T}))$	$wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})$
$S_1 \Box S_2$	$wp(S_1, Q) \wedge wp(S_2, Q)$	$wlp(S_1, Q) \wedge wlp(S_2, Q)$	$wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{T})$	$wlp(S_1, \mathrm{F}) \wedge wlp(S_2, \mathrm{F})$

Table 2: Weakest precondition and weakest liberal precondition predicate transformers.

$$\textsc{F-ASSERT}\ \frac{}{f(\mathsf{assert}\ e) = ([e], [\neg e])} \qquad \textsc{F-ASSUME}\ \frac{}{f(\mathsf{assume}\ e) = ([e], [\mathrm{F}])}$$

$$\textsc{F-SEQ}\ \frac{f(A) = (n_A, w_A) \qquad f(B) = (n_B, w_B)}{f(A; B) = (n_a @ n_B, w_a @ w_B)}$$

$$\textsc{F-CHOICE}\ \frac{\begin{array}{ll} f(A) = (n_A, w_A) & G(n_A, w_A) = (N_A, W_A) \\ f(B) = (n_B, w_B) & G(n_B, w_B) = (N_B, W_B) \end{array}}{f(A \Box B) = ([N_A \vee N_B], [W_A \vee W_B])}$$

$$\textsc{G-BASE}\ \frac{}{G([n], [w]) = (n, w)}$$

$$\textsc{G-REC}\ \frac{G(n_s, w_s) = (N, W)}{G(([n_1]@n_s), ([w_1]@w_s)) = (n_1 \wedge N, w_1 \vee (n_1 \wedge W))}$$

$$\textsc{DWP}\ \frac{G(f(S)) = (N, W)}{\textsc{DWP}(S, Q) = \neg W \wedge (N \implies Q)}$$

Figure 2: Our basic algorithm.

acyclic programs, this can be accomplished by using the algorithm in [12], or by converting P to a static single assignment form (SSA) then removing Φ expressions. The DSA program P' is at most $O(M^2)$ the size of P, where M is the number of statements. The transformation takes $O(M^2)$ time. It is worth noting that the SSA form of a program is, in practice, usually linear in size [1, 11] since ultimately the generated VC will be linear in the size of the DSA program.

2. Replace each assignment statement of the form $x_i := e$ with assume $x_i = e$. The resulting assignment-free program is called a *passified* program.

Notation. N is used for formulas that describe the initial states from which the execution of S may terminate normally, i.e., $N = \neg wlp(S, \mathrm{F})$. W is used for formulas that describe the initial states from which the execution of S may go wrong, i.e., $W = \neg wp(S, \mathrm{T})$. We found it easier to think in terms terms of N and W than using double negatives and wp/wlp. For example, $S_1; S_2$ can go wrong when (a) S_1 goes wrong, or (b) S_1 terminates normally, but S_2 goes wrong. We write this as $W_1 \vee N_1 \wedge W_2$, which we found more straightforward than $wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F}) \vee wp(S_2, \mathrm{T})$. n and w are used for lists of said expressions, respectively. (Note that the N and W values correspond to those that would be returned by the functions of the same name in [12].)

$[a]$ denotes a list with a as its single element. $a@b$ denotes the concatenation of two lists. (a, b) denotes an ordered pair with a as the first element and b as the second. We use the following meta-syntactic variables: e for expressions, x for variables, Q for a post-condition, and S for GCL expressions. We use $\vec{x}$ be the new variables introduced by the DSA transformation.

Since we use the weakest precondition and weakest liberal precondition semantics with respect to the constants true (T) and false (F) extensively in this paper, we show their appropriate values in Table 2.

Basic Algorithm. Figure 2 shows our basic algorithm for calculating the weakest precondition of a passified program P with respect to a post-condition Q, denoted as $\textsc{dwp}(P, Q)$. We form each statement as an inference rule as this leads to a syntax-based proof of correctness and size. The algorithm can be read as a pattern-match; any time we encounter a statement below the line we perform the calculation on top of the line, returning the corresponding results.

The intuition for DWP is that we can compute sub-formulas for all the individual statements (via f) in either direction, and then later combine them into a VC for the whole sequence (via G). f returns two per-statement VCs: one for successful termination and one for going wrong. The returned VCs are prepended to a list of other VCs generated so far. G takes the two lists of VCs and constructs the appropriate wp or wlp. DWP assembles the final VC for an arbitrary post-condition similar to Theorem 1.

Note that *not* calling G on F-SEQ is not a mistake: we instead lazily call G when needed. For example, given $S_1; S_2$ we can compute the formula for S_1 and then S_2, or for S_2 and then S_1. Indeed, using DWP one could compute a VC for some first part of the program in the forward direction, and then compute a VC for the remainder of the program in the backward direction.

3.2 Proof of correctness

In order for our algorithm in Figure 2 to be correct, we must show it is equivalent to calculating the weakest precondition.

Theorem 2 (Correctness). $\forall S, Q\ \textsc{dwp}(S, Q) = wp(S, Q)$

In order to prove Theorem 2, we first show that $G(f(S))$ calculates the weakest precondition $wp(P, \mathrm{T})$ as W and $wlp(P, \mathrm{F})$ as N:

Lemma 3. If $G(f(S)) = (N, W)$ then $N = \neg wlp(S, \mathrm{F})$ and $W = \neg wp(S, \mathrm{T})$

Proof. We provide a full proof in Appendix A.1. assert and assume are straight-forward. In the case of $S_1 \Box S_2$ we simply apply the induction hypothesis to the derivation of S_1 and S_2 and show the rule correctly combines the results.

The case for sequences $S = S_1; S_2$ is important in that it makes the algorithm directionless, but requires more ingenuity to prove. When f is called on S, we do not return the pair N and W. Instead, we return a list of formulas which *later* will be put together into N and W via G. This feature allows us to calculate the formulas for a sequence in any order. We then just call G whenever we need to create the actual VC. The trick in the proof is to first show

correctness given a version of F-SEQ that calls G eagerly, and then show that correctness when maintained if called lazily. □

Proof of Theorem 2. Lemma 3 says that the sub-computations in Figure 2 compute the weakest precondition and weakest liberal precondition. In order to demonstrate correctness we just need to show DWP combines the results correctly. Let $G(f(S)) = (N, W)$. Then

$$\begin{aligned}
&\text{DWP}(S, Q) = \neg W \wedge (N \implies Q) && \text{By rule DWP.}\\
&= \neg\neg wp(S, \text{T}) \wedge (\neg wlp(S, \text{F}) \implies Q) && \text{By Lemma 3.}\\
&= wp(S, \text{T}) \wedge (wlp(S, \text{F}) \vee Q) && \text{Logic}\\
&= wp(S, Q) && \text{By Theorem 1.}
\end{aligned}$$

□

3.3 Efficient Directionless WP

The initial algorithm in Figure 2 may result in a formula quadratic in size of the passified program. Since the passified program is itself $O(M^2)$, the total formula size is $O(M^4)$ (where M is again the size of the original program). In order to see where the quadratic blowup comes from, consider the G-REC rule. G is given a list n of sub-formulas that are required for normal termination, and similarly a list w for wrong termination. The rule computes the pair $N_1 \wedge N$ and $W_1 \vee (N_1 \wedge W)$, thus duplicating N_1. The idea for N_1 in the latter is that a program may go wrong if it goes wrong at statement 1, or if it goes right on statement 1 but wrong on statement 2, and so on. This duplication becomes a real problem when there is a deeply nested series of sequences and choices.

In order to avoid duplicating the expression N_1, at a high level we give it a name and refer to the name instead of copying in the actual formula. We do this by tracking some extra state which we refer to as v and V. It is simply a set of (variable, expression) pairs corresponding to expressions that would otherwise be duplicated. For example, if e is an expression that we would be duplicating in the original rules, we instead make up a variable, say x, and use x where we would have used e and return the pair (x, e) along with the resulting formula.

The DWP needs to reflect the meaning of the variables introduced:

$$\frac{f(S) = (v, n, w) \qquad G(n, w) = (V, N, W)}{\text{DWP}(S, Q) = (\bigwedge_{(x,e)\in v\cup V} x = e) \wedge \neg W \wedge (N \implies Q)}$$

The only change to individual rules is to when we introduce $x = e$ to use x instead of duplicating the expression e. The final algorithm is shown in Figure 3.

3.4 Size and length

We will prove the size of $\text{DWP}(S, Q)$ is linear in the size of the passified program S by proving an upper bound:

Theorem 4. $|\text{DWP}(S, Q)| < 2 \cdot |S| + 9 \cdot \texttt{len}(S) + |Q|$

Note that this is a much more precise bound than found in previous work [12, 18]. Since passification may result in quadratic blowup, the final predicate will be in the worse case $O(M^2)$. Recall that our passification process (§ 3.1) does not typically incur the quadratic blowup, and is linear in practice [1, 11].

In order to prove our bounds we need formal definitions of size for both expressions and the GCL.

$$\begin{aligned}
|A \wedge B| &= |A| + |B| + 1 & |A \vee B| &= |A| + |B| + 1\\
|A \implies B| &= |A| + |B| + 1 & |A = B| &= |A| + |B| + 1\\
|\neg A| &= |A| + 1 & |x| &= 1 \text{ where } x \text{ is any variable}
\end{aligned}$$

We will leave the size definition for other expression forms up to the reader, as the reader may want to use a richer expression language inside assert and assume. The only requirement is that the size cannot be negative. We will however define the size of the passified GCL, since we will be using it in the next section.

$$\begin{aligned}
|\textsf{assert } e| &= |e| + 1 & |\textsf{assume } e| &= |e| + 1\\
|A; B| &= |A| + |B| + 1 & |A \square B| &= |A| + |B| + 1
\end{aligned}$$

For the sake of the proof, we define the size of lists of expressions as the sum of their parts:

$$|\varnothing| = 0 \qquad |[a]| = |a| \qquad |a@b| = |a| + |b|$$

We define the length of a list as the number of elements in the list without counting the size of the elements:

$$\begin{aligned}
\texttt{len}(\varnothing) &= 0 & \texttt{len}([a]) &= 1\\
\texttt{len}(a@b) &= \texttt{len}(a) + \texttt{len}(b)
\end{aligned}$$

And the length of a GCL statement as:

$$\begin{aligned}
\texttt{len}(\textsf{assert } e) &= 1 & \texttt{len}(\textsf{assume } e) &= 1\\
\texttt{len}(A; B) &= \texttt{len}(A) + \texttt{len}(B)\\
\texttt{len}(A \square B) &= \texttt{len}(A) + \texttt{len}(B) + 1
\end{aligned}$$

We also define the sequence length of a GCL statement to count the number of sequential statements (ie, without recursing inside choice statements):

$$\begin{aligned}
\texttt{seqlen}(\textsf{assert } e) &= 1 & \texttt{seqlen}(\textsf{assume } e) &= 1\\
\texttt{seqlen}(A; B) &= \texttt{seqlen}(A) + \texttt{seqlen}(B)\\
\texttt{seqlen}(A \square B) &= 1
\end{aligned}$$

3.4.1 Proof of size

Lemma 5. If $G(n, w) = (V, N, W)$ and $\texttt{len}(n) = \texttt{len}(w)$ then the following statements are true:

(i) $|N| = 2 \cdot \texttt{len}(n) - 1$
(ii) $|W| = |w| + 3(\texttt{len}(n) - 1)$
(iii) $\texttt{len}(V) = \texttt{len}(n)$
(iv) $|V| = 2 \cdot \texttt{len}(n) + |n|$

Proof. By induction. Full proofs are in Appendix A.2, but they are straightforward. □

Lemma 6. If $f(S) = (v, n, w)$ then $\texttt{len}(n) = \texttt{len}(w) = \texttt{seqlen}(S)$

In other words, for each GCL statement in a sequence, f will add one expression to each of the lists. The expressions that correspond to statements inside a choice get combined into a single expression by G.

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(S)$. The full proof appears in Appendix A.3. □

Lemma 7. If $f(S) = (v, n, w)$ then $\texttt{len}(v) = \texttt{len}(S) - \texttt{seqlen}(S)$

In other words, f will add 1 variable binding corresponding to each statement inside a choice, but not for statements in the outermost sequence. (Since we will be adding those later.) Remember $\texttt{len}(S)$ is the total number of GCL statements in S whereas $\texttt{seqlen}(S)$ is the number of statements in the outermost sequence of S. That makes their difference count only the statements that are inside a choice. Intuitively it makes sense that there are that many bindings since all the bindings that f returns get created by G which is only called inside choice statements.

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(n, w, S)$. A complete proof appears in Appendix A.4. The cases for assert and assume are straight-forward. For sequences, we need only apply the induction hypothesis (IH) and apply the rule. The case for choice statements requires more care:

$$\text{F-ASSERT}\ \frac{}{f(\mathsf{assert}\ e) = (\emptyset, [e], [\neg e])} \qquad \text{F-ASSUME}\ \frac{}{f(\mathsf{assume}\ e) = (\emptyset, [e], [false])} \qquad \text{F-SEQ}\ \frac{f(A) = (v_A, n_A, w_A) \qquad f(B) = (v_B, n_B, w_B)}{f(A; B) = ((v_A \cup v_B), n_A @ n_B, w_A @ w_B)}$$

$$\text{F-CHOICE}\ \frac{f(A) = (v_A, n_A, w_A) \qquad G(n_A, w_A) = (V_A, N_A, W_A) \qquad f(B) = (v_B, n_B, w_B) \qquad G(n_B, w_B) = (V_B, N_B, W_B)}{f(A \,\square\, B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$$

$$\text{G-BASE}\ \frac{x\ \mathsf{is\ fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)} \qquad \text{G-REC}\ \frac{x\ \mathsf{is\ fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$$

Figure 3: The final algorithm for efficient, directionless weakest preconditions.

$$\text{F-CHOICE}\ \frac{\begin{array}{ll} f(A) = (v_A, n_A, w_A) & G(n_A, w_A) = (V_A, N_A, W_A) \\ f(B) = (v_B, n_B, w_B) & G(n_B, w_B) = (V_B, N_B, W_B) \end{array}}{f(A \,\square\, B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$$

$$\begin{array}{ll}
\mathtt{len}(n_A) = \mathtt{seqlen}(A) & \text{By Lemma 6.} \\
\mathtt{len}(V_A) = \mathtt{seqlen}(A) & \text{By Lemma 5.} \\
\mathtt{len}(n_B) = \mathtt{seqlen}(B) & \text{By Lemma 6.} \\
\mathtt{len}(V_B) = \mathtt{seqlen}(B) & \text{By Lemma 5.} \\
\mathtt{len}(v_A) = \mathtt{len}(A) - \mathtt{seqlen}(A) & \text{By IH.} \\
\mathtt{len}(v_B) = \mathtt{len}(B) - \mathtt{seqlen}(B) & \text{By IH.} \\
\mathtt{len}(v_A \cup v_B \cup V_A \cup V_B) = & \\
\quad \mathtt{len}(v_A) + \mathtt{len}(v_B) + \mathtt{len}(V_A) + \mathtt{len}(V_B) & \text{Unique variables.} \\
\quad = \mathtt{len}(A) + \mathtt{len}(B) = \mathtt{len}(S) - \mathtt{seqlen}(S) &
\end{array}$$

□

Lemma 8. If $f(S) = (v, n, w)$ then $|v| + |n| + |w| \leq 2 \cdot |S| + 7(\mathtt{len}(S) - \mathtt{seqlen}(S))$

This is setting a linear upper bound on the size of everything output by f.

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(S)$. A full proof appears in § A.5. The cases for assert and assume are straight-forward. For sequences we apply the induction hypothesis. For choice, we again apply the induction hypothesis, apply Lemma 6 to argue about the lengths returned from calls to f, and then Lemma 5 for the calls to G to put everything together. □

Proof of Theorem 4 We want to prove that $|\mathrm{DWP}(S, Q)|$ is linear, so we give it a linear upper bound. Since we already proved that the sizes of all the components are linear, this is pretty straightforward.

Proof. By inspecting the sizes of the constituent parts:

$$\begin{array}{ll}
|N| = 2 \cdot \mathtt{len}(n) - 1 & \text{Lemma 5} \\
\quad = 2 \cdot \mathtt{seqlen}(S) - 1 & \text{Lemma 6} \\
|W| = |w| + 3(\mathtt{len}(n) - 1) & \text{By Lemma 5} \\
\quad = |w| + 3(\mathtt{seqlen}(S) - 1) & \text{By Lemma 6} \\
|(\bigwedge_{(x,e) \in v \cup V} x = e)| = |v \cup V| + 2\mathtt{len}(v \cup V) - 1 & \\
\quad \text{An} = \text{for every element and} \wedge \text{es between elements in } v \cup V. & \\
|v \cup V| = |v| + |V| & \text{Disjoint sets.} \\
\quad = |v| + 2\mathtt{len}(n) + |n| & \text{Lemma 5.} \\
\quad = |v| + 2\mathtt{seqlen}(S) + |n| & \text{Lemma 6.} \\
\mathtt{len}(v \cup V) = \mathtt{len}(v) + \mathtt{len}(V) & \text{Disjoint sets.} \\
\quad = \mathtt{len}(S) - \mathtt{seqlen}(S) + \mathtt{len}(V) & \text{By Lemma 7.} \\
\quad = \mathtt{len}(S) - \mathtt{seqlen}(S) + \mathtt{len}(n) & \text{By Lemma 5.} \\
\quad = \mathtt{len}(S) & \text{By Lemma 6.} \\
|(\bigwedge_{(x,e) \in v \cup V} x = e)| & \\
\quad = |v| + 2\mathtt{seqlen}(S) + |n| + 2\mathtt{len}(S) - 1 & \text{Substitution.} \\
|\mathrm{DWP}(S, Q)| & \\
\quad = |(\bigwedge_{(x,e) \in v \cup V} x = e)| + ((|W| + 1) + |N| + |Q| + 1 + 1) + 1 & \\
\quad \text{By definition of size.} & \\
\quad = |v| + 2\mathtt{seqlen}(S) + |n| + 2\mathtt{len}(S) - 1 & \\
\quad + (|w| + 3(\mathtt{seqlen}(S) - 1) + 1) & \\
\quad + (2\mathtt{seqlen}(S) - 1) + |Q| + 3 & \\
\quad = |v| + |n| + |w| + 7\mathtt{seqlen}(S) & \\
\quad + 2\mathtt{len}(S) + |Q| - 1 & \text{Simplification.} \\
\quad \leq 2 \cdot |S| + 7(\mathtt{len}(S) - \mathtt{seqlen}(S)) & \\
\quad + 7\mathtt{seqlen}(S) + 2\mathtt{len}(S) + |Q| - 1 & \text{By Lemma 8.} \\
\quad < 2 \cdot |S| + 9 \cdot \mathtt{len}(S) + |Q| & \text{Simplification}
\end{array}$$

□

4. Efficient Forward Symbolic Execution

One of our motivations for developing an efficient, directionless weakest precondition algorithm is to make forward symbolic execution efficient. In order to show DWP can act as a drop-in replacement, we first define the semantics of forward symbolic execution, and then show that our algorithm produces a semantically equivalent formula. The main benefit of using our algorithm is the exponentially smaller VCs, and depending upon the algorithm used for forward symbolic execution, shorter VC generation times.

Since forward symbolic execution is not well defined for assume statements, in this section we will restrict ourselves to programs in which the only use of assume and $\square$ are of the form assume e; $S_1 \,\square$ assume $\neg e$; S_2, where e can be an arbitrary boolean expression, and S_1 and S_2 can be arbitrary GCL statements. We will call programs of this form *deterministic* programs, since they can be evaluated efficiently on a deterministic Turing machines. In this section we will use if e then S_1 else S_2 as syntactic sugar for statements of the above form. Although particular implementations of forward symbolic execution in related work may be nuanced, the restricted language still reflects the core constructs in typical applications [2, 3, 6, 8–10, 15–17, 20, 23].

4.1 Semantics of Forward Symbolic Execution

Forward symbolic execution is an algorithm which takes the operational semantics of a language and augments them such that values at each point of execution are in terms of input variables. More precisely, values in the language become expressions. In contrast, concrete execution for the same language would typically execute the program such that values are atomic units, e.g., integers.

Since forward symbolic execution is an execution, it is natural to define the algorithm in terms of the operational semantics of the GCL. We show the operational semantics for a symbolic evaluator in Figure 4.

The semantics of symbolic execution must include the conditions under which a path is executed, called the *path predicate*. More formally, a path predicate is a predicate in terms of input variables which is satisfied by all assignments of concrete values to variables that would take the path. We use Π to denote the path predicate created so far by symbolic execution. We use σ to denote

$$\textsc{Var-Sub}\quad \frac{}{x|\sigma \Downarrow_e \sigma(x)} \qquad \textsc{Fwd-Assert}\quad \frac{e|\sigma \Downarrow_e e'}{\mathsf{assert}\ e|\langle\sigma,\Pi\rangle \Downarrow \langle\sigma, e\wedge\Pi\rangle}$$

$$\textsc{Fwd-Assign}\quad \frac{e|\sigma \Downarrow_e e' \qquad \sigma' = \sigma[x \mapsto e']}{x := e|\langle\sigma,\Pi\rangle \Downarrow \langle\sigma',\Pi\rangle}$$

$$\textsc{Fwd-Seq}\quad \frac{S_1|\langle\sigma,\Pi\rangle \Downarrow \langle\sigma_1,\Pi_1\rangle \qquad S_2|\langle\sigma_1,\Pi_1\rangle \Downarrow \langle\sigma_2,\Pi_2\rangle}{S_1;S_2|\langle\sigma,\Pi\rangle \Downarrow \langle\sigma_2,\Pi_2\rangle}$$

$$\textsc{Fwd-ITE}\quad \frac{\begin{array}{c}e|\sigma \Downarrow_e e' \\ S_1|\langle\sigma, e'\wedge\Pi\rangle \Downarrow \langle\sigma_1,\Pi_1\rangle \qquad S_2|\langle\sigma,\neg e'\wedge\Pi\rangle \Downarrow \langle\sigma_2,\Pi_2\rangle\end{array}}{\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2|\langle\sigma,\Pi\rangle \Downarrow \langle\mathsf{merge}(\sigma_1,\sigma_2),\Pi_1\vee\Pi_2\rangle}$$

Π	Path predicate
σ	Execution context, e.g., maps variables to their current symbolic expression in terms of input variables.

Figure 4: A basic forward symbolic execution interpreter built from the operational semantics of the GCL.

the current mapping of variables to their symbolic values (i.e., expressions). The rules give the semantics of the evaluation of s with current path predicate Π and state σ, denoted $\langle\sigma,\Pi\rangle$. The result of evaluating (denoted $\Downarrow$) a statement is a new evaluation context, and a (possibly new) path predicate.

Definition 4.1. Forward symbolic execution of a program P with desired post-condition Q is defined as:

$$\frac{P|\langle\sigma,\mathrm{T}\rangle \Downarrow \langle\sigma',\Pi'\rangle}{\textsc{Fwd}(P,Q) : \Pi'\wedge Q}$$

where σ maps variables to their initial symbolic values.

The evaluation rules behave as follows. We denote expression evaluation of e to e' as $e \Downarrow_e e'$. In symbolic execution an expression evaluation merely performs a substitution of used variables with symbolic variables. We show the evaluation substitution rule as VAR-SUB, but otherwise omit expression evaluation rules.

We symbolically evaluate an assignment statement $x := e$ by first evaluating e to a new expression e', and then updating our execution context σ to include the new variable/value binding. A sequence $S_1; S_2$ first evaluates S_1, then evaluates S_2. For example, if $\sigma = \{a \mapsto x_0, b \mapsto x_1\}$ where x_0 and x_1 are symbolic inputs, then $c := a+b; d := 2*c$ evaluates to $\sigma' = \sigma \cup \{c \mapsto x_0+x_1, d \mapsto 2*(x_0+x_1)\}$.

The symbolic evaluation of conditional statements evaluates both branches, and shown in the FWD-ITE rule. In contrast, during concrete execution the branch guard e is evaluated and then the single appropriate branch is selected for subsequent execution. We want the ability to execute both branches so that the formula generated captures any possible path through the program. In order to do this, we first evaluate the branch guard e to a symbolic expression e'. e' will be purely in terms of (symbolic) input variables. We then assert that e must be true along the true branch S_1 by adding it to the path predicate. The result is $e \wedge \Pi$ means that to execute statement s, everything to reach the condition in Π must be true, and finally e must be true. We similarly execute the false branch by asserting $\neg e$ to the path predicate.

Problem 1: Merging Paths One issue with forward symbolic execution is how best to handle the confluence point after executing a conditional branch. Consider a program of the form:

$$S_1; \mathsf{if}\ e\ \mathsf{then}\ S_2\ \mathsf{else}\ S_3; S_4$$

Suppose we are interested in generating a formula to reach S_4. At the branch point either S_2 or S_3 could be taken. Since either path could be taken to reach subsequent statements, it is natural to have the path predicate be the disjunction of paths at confluence points such as S_4. The FWD-ITE rule reflects this: if Π_1 is the path predicate after executing one side of a branch, and Π_2 is the predicate for the other side, then the path predicate at the confluence point is $\Pi_1 \vee \Pi_2$.

Deciding how to merge the execution contexts σ_1 and σ_2 at the confluence of two execution paths, however, is not so straightforward. We denote the *problem* of merged paths by the merge function. For example, suppose along one path we have $\sigma_1 = \{a \mapsto 2*x_0\}$ and along the other path we have $\sigma_2 = \{a \mapsto 3*x_0\}$. What value should a have at the confluence of these two paths?

In concrete execution only one of the two paths at branches will be taken, thus there is no problem with merging states. The most straight-forward solution is to mimic concrete execution during symbolic execution and "fork off" two different engines, one for each path. The final predicate will still be the disjunction of each path predicate. A forking algorithm, however, will lead to an exponentially large predicate and an exponential run time. For example, consider:

$$\mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; \mathsf{if}\ e_2\ \mathsf{then}\ S_3\ \mathsf{else}\ S_4; \mathsf{if}\ e_3\ \mathsf{then}\ S_5\ \mathsf{else}\ S_6; S_7$$

In this situation there are 8 program paths, and we will need to execute each one individually in order to create a path predicate for S_7. The final predicate will be the disjunction of all 8 paths. In general, every branch doubles the run time, and doubles the size of the final path predicate. Despite the exponential behavior, forking off the interpreter is a common solution, e.g., [6, 8, 9, 14, 16, 19, 23] and many others.

Problem 2. Formula Size is Exponential. A second issue with forward symbolic execution is the resulting formula may be exponential in size. One reason is the substitution performed by VAR-SUB, which is analogous to the substitution performed by the weakest precondition. Just as before, the program $x := x + x; x := x+x; x := x+x$ will produce a final formula that contains x 8 xs. Second, the final predicate size doubles every time we encounter a conditional branch in FWD-ITE since a copy of Π is passed along both branches.

Advantage: Mixed Execution. Given the significant performance issues with forward symbolic execution, why would it ever be used in practice?

One significant advantage is that it is relatively easy to modify a symbolic execution engine to perform a mix of symbolic and concrete execution. For example, we could alter our operational semantics such that we keep both a context for variables with concrete values σ^c, and a symbolic context for variables with values in terms of symbolic inputs σ^s. When evaluating an expression instead of merely replacing all occurrences of variables with their current symbolic formula we first try to evaluate them concretely. For example, recall given the example in Figure 1, a mixed symbolic execution engine would generate the formula $b + b < 10$, since all other statements can be evaluated concretely.

Since mixed execution takes advantage of initial constants in the program, one may wonder why getting rid of such constants is easier using forward symbolic execution. The answer is that we encounter the constant variable definitions before uses in the forward direction, but not in the backward direction. Furthermore in the forward direction we can simply execute the program concretely on a native processor and only do the symbolic execution in software. Recently several research projects in automated test case generation have implemented mixed execution in this manner, and as a

result are achieving significant code coverage, e.g., 90% on average in the Linux core-utils package [6].

One may wonder why such constants appear in programs, e.g., why were constants not removed during compilation? While the answer depends upon implementation and application domain, the "constants" are often simply just inputs we do not care about executing symbolically. For example, suppose we wish to generate test cases for the following program:

```
1.  environment = read_configuration(file);
2.  request = get_request(network);
3.  handle_request(environment, request);
```

How we handle a request is parameterized by both the URL and the web-server environment, e.g., a request for a URL "foo" may be a valid request under one configuration, but an invalid request under another.

In a particular application domain, however, we may choose to fix `environment` or `request` to have a particular concrete value. For example, suppose we want to generate test cases for `handle_request`. If we fix `environment` to have a specific concrete value (e.g., the value of the machine that the test case generation is running on), the formulas generated by symbolic execution of `handle_request` will have fewer variables, thus likely easier to solve for a decision procedure. This in turn would likely result in higher coverage for statements inside `handle_coverage` than if we left `environment` symbolic.

Adding mixed execution to a traditional backwards wp-based algorithm would be difficult. Since statements are processed in reverse order, a wp-based algorithm would not realize that lines 3 and 4 could be simplified until the very end of the calculation. At that point one could add a post-processing step to do the simplification, e.g., using constant folding. However, it would not fundamentally change the fact that the wp would have to consider all statements symbolically up to that point.

Advantage: No Universal Quantifiers A second advantage of forward symbolic execution is the generated predicates contain no universal quantifiers, while efficient weakest preconditions do. Since we typically give generated formulas to a theorem prover, we want formulas to be amenable to as many theorem provers as possible. By removing universal quantifiers we open up VC verification to a wide variety of theorem provers.

4.2 Efficient Forward Symbolic Execution

In this section we describe our algorithm for using our directionless weakest precondition as a drop-in replacement for forward symbolic execution.

Let P be a program using only the language constructs from Figure 4. Our algorithm for efficiently performing forward symbolic execution for a post-condition Q is:

Step 1. Put P DSA form. Although the straight-forward implementation of this is static (§ 3), we discuss how it could be made dynamic in § 5. Let P' be the DSA program.

Step 2. Calculate DWP(P',Q).

Forward Execution of DSA Program P'**.** In order to prove that DWP(P',Q) is correct, we first need to define what forward symbolic execution would produce. We show correctness in two steps. First, we describe a simpler form of forward symbolic execution for DSA programs. The second step is to show that forward symbolic execution on DSA programs produces a formula that is logically equivalent to our directionless weakest precondition.

First, we note that the forward symbolic execution rules in Figure 4 perform substitution because each variable name does not reflect a unique definition. For example, in $x := x + x; x = x + x$ we want any post-conditions to refer to the second x, not the first. In DSA form, however, each variable is assigned only once per path. Thus, instead of performing substitution, each variable can be referred to by name when needed instead of performing a substitution. In the above example, the DSA form is $x_1 = x_0 + x_0; x_2 = x_1 + x_1$, and the post condition can refer to the final value as x_2.

$$
\begin{aligned}
&s(S_1; S_2) = s(S_1) \wedge s(S_2) s(\textsf{assert } e) = e \\
&s(\textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2) = (e \wedge s(S_1)) \vee (\neg e \wedge s(S_2)) \\
&s(x := e) = (x = e) \\
&\textsc{fwd}'(P', Q) = s(P') \wedge Q
\end{aligned}
$$

Figure 5: Forward symbolic execution of a program in DSA form.

Thus, forward symbolic execution need not perform substitution on programs in DSA form. Further, we need not keep track of the current symbolic state: we can simply refer to the particular variable definition we need. As a result, forward symbolic execution of a DSA program eliminates the problems associated with merge. We show the simplified rules in Figure 5.

Let $\textsc{fwd}(P, Q)$ be forward symbolic execution of P such that an interpreter is forked for each execution path, i.e.:

$$\textsc{fwd}'(P, Q) = \bigvee_{\forall \text{ paths } \pi \in P} \textsc{fwd}(\pi, Q) \tag{2}$$

Then $\textsc{fwd}(P, Q)$ is the same as $\textsc{fwd}'(P', Q)$ using the rules in Figure 5. The correspondence can be shown by simple inspection of each type of program statement. In particular, note that instead of doing substitution in assignment, we simply add a variable-value definition to our logical formula that reflects the state update.

4.3 Correctness

The weakest precondition of deterministic GCL programs is the same as previously, except we must also handle conditional statements. The rule for conditionals is the same as if we had used assumes for branch guards:

$$wp(\textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2) = (e \Rightarrow wp(S_1, Q)) \wedge (\neg e \Rightarrow wp(S_2, Q))$$

Correctness states that whatever VC forward symbolic execution would calculate is logically equivalent to what the predicate transformers for weakest preconditions would calculate.

Theorem 9. For all deterministic programs P and all predicates Q: $wp(P, Q) = \textsc{fwd}(P, Q)$

This equivalence means our improvement to weakest preconditions is also an improvement to forward symbolic execution. In particular, since $\textsc{dwp}(P, Q) = wp(P, Q)$, our directionless algorithm can therefore be used as a drop-in replacement for FWD.

In order to prove this, we need to establish two things. First, recall that weakest preconditions are universally quantified, while forward symbolic execution is only existentially quantified. Thus, we will need to establish a correspondence between the two:

Lemma 10. $\forall x : (x = e \Rightarrow Q) \iff \exists x : (x = e \wedge Q)$

Proof. Full proof omitted. (This equivalence is easy to verify, eg. via automated theorem proving.) □

Note that weakest precondition is defined as:

$$
\begin{aligned}
&wp(P, Q) = wp(P, \mathrm{T}) \wedge (wlp(P, \mathrm{F}) \vee Q) \\
&= (wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F})) \vee (wp(P, \mathrm{T}) \wedge Q)
\end{aligned}
$$

The intuition behind our proof of Theorem 9 is that the first part of this disjunction is always false for deterministic programs, and the second part of the conjunction corresponds to forward symbolic execution. We prove the first part of this in the following lemma:

Lemma 11. For all deterministic programs P:

$$wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) = \mathrm{F}$$

Proof. By induction on the structure of P:

Case: $P = \mathsf{assert}\ e$

$$\begin{aligned} &wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) = e \wedge \mathrm{T} \wedge (e \Rightarrow \mathrm{F}) \\ &= e \wedge \neg e = \mathrm{F} \end{aligned}$$

Case: $P = x := e$ [2]

$$\begin{aligned} &wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ &= wp(P, \mathrm{T}) \wedge \forall x, (x = e \Rightarrow \mathrm{F}) \\ &\implies wp(P, \mathrm{T}) \wedge (e = e \Rightarrow \mathrm{F}) && (\forall \text{ elimination with } e \text{ for } x) \\ &= \mathrm{F} \end{aligned}$$

Case: $P = \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$

$$\begin{aligned} &wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ &= (e \Rightarrow wp(S_1, \mathrm{T}) \wedge \neg e \Rightarrow wp(S_2, \mathrm{T})) \\ &\quad \wedge (e \Rightarrow wlp(S_1, \mathrm{F}) \wedge \neg e \Rightarrow wlp(S_2, \mathrm{F})) \\ &= e \Rightarrow (wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F})) \\ &\quad \wedge \neg e \Rightarrow (wp(S_2, \mathrm{T}) \wedge wlp(S_2, \mathrm{F}) \\ &= e \Rightarrow \mathrm{F} \wedge \neg e \Rightarrow \mathrm{F} && \text{By IH.} \\ &= \mathrm{F} \end{aligned}$$

Case: $P = S_1; S_2$

$$\begin{aligned} &wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ &= wp(S_1, \mathrm{T}) \wedge (wlp(S_1, \mathrm{F}) \vee wp(S_2, \mathrm{F})) \\ &\quad \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) \\ &= wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F}) \vee wp(S_1, \mathrm{T}) \\ &\quad \wedge wp(S_2, \mathrm{F})) \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) \\ &= \mathrm{F} \vee wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \\ &\quad \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) && \text{By IH.} \\ &= wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \wedge wlp(S_1, \mathrm{F}) \\ &\quad \vee wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \wedge wlp(S_2, \mathrm{F}) && \text{By IH.} \\ &= \mathrm{F} \vee \mathrm{F} && \text{By IH.} \\ &= \mathrm{F} \end{aligned}$$

□

We now prove Theorem 9.

Proof of Theorem 9. By induction on the structure of P, using s from Figure 5:

Case: $P = \mathsf{assert}\ e$

$$e \wedge Q = e \wedge Q \ \therefore wp(P, Q) = s(P) \wedge Q$$

Case: $P = x := e$

$$\begin{aligned} &(1)\ wp(P, Q) = \forall x, (x = e \implies Q) \\ &(2)\ s(P) \wedge Q = (x = e) \wedge Q \\ &\quad = \exists x, (x = e \wedge Q) \\ &(1) = (2) && \text{By Lemma 10} \end{aligned}$$

Case: $P = \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$

$$\begin{aligned} &wp(P, Q) \\ &= (e \Rightarrow wp(S_1, Q)) \wedge (\neg e \Rightarrow wp(S_2, Q)) \\ &= e \wedge wp(S_1, Q) \vee \neg e \wedge wp(S_2, Q) \\ &= e \wedge s(S_1) \wedge Q \vee \neg e \wedge s(S_2) \wedge Q && \text{By IH.} \\ &= s(P) \wedge Q \end{aligned}$$

Case: $P = S_1; S_2$

$$\begin{aligned} &wp(P, Q) \\ &= wp(S_1, T) \wedge (wlp(S_1, F) \vee wp(S_2, Q)) \\ &= wp(S_1, T) \wedge (wlp(S_1, F) \vee wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) \\ &= wp(S_1, T) \wedge wlp(S_1, F) \\ &\quad \vee wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) \\ &= wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) && \text{By Lemma 11} \\ &= wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge Q && \text{By Lemma 11} \\ &= s(S_1) \wedge s(S_2,) \wedge Q && \text{By IH.} \\ &= s(P) \wedge Q \end{aligned}$$

□

[2] Since the program is in DSA form, we are simply treating assignments as assumes.

5. Discussion

Dynamic Single Assignment. Forward symbolic execution can be implemented in a way that VCs are generated at run-time, e.g., as in KLEE [6], DART [16], and CUTE [22]. Our algorithm acts as a drop-in replacement, but only when the program is first put in a dynamic single assignment form. Our assumed method for putting the program in DSA form via an SSA-like transformation, however, is static. We expect in most application domains this small amount of static analysis is acceptable, e.g., EXE is implemented on top of a compiler which already supports such transformers.

It is possible, however, to use our DWP in a purely dynamic setting by calculating the DSA form of the program on the fly. On-the-fly DSA form requires we add an additional current incarnation context to the execution engine that maps a variable name to its current DSA incarnation. Sequential statements are trivial; every assignment simply uses the context when processing expressions, and updates the context on assignment. Branches require that we merge contexts at the confluence point. For example, if in branch 1 the current incarnation of x is x_4, and along branch 2 the current incarnation is x_5, we simply need to add an assignment that $x_6 = x_4$ along the first branch and $x_6 = x_5$ along the second, and then use x_6 as the canonical identifier for x thereafter.

Equivalence between forward symbolic execution and weakest precondition. At a high level it may be unsurprising that weakest preconditions and forward symbolic execution are related. After all, both are calculating a formula that should be satisfied by inputs that cause termination in the desired state. However, we are unaware of any previous work that spells out the differences.

Our algorithm and proof show that $wp(P, \mathrm{T}) \wedge Q$ is equivalent to forward symbolic execution for typical structural programs. The proof precisely identifies the difference: they are equivalent if the program 1) acts deterministically and 2) does not use assumptions for anything other than branch guards. In particular, weakest preconditions keep track of when assumptions may not be satisfied, but forward symbolic execution does not. In order to extend forward symbolic execution to handle the full GCL, we would need to include logic in the interpreter than replicated the semantics of $wlp(P, \mathrm{F})$. Adding this logic would, in effect, result in the same semantics as our directionless weakest precondition.

Eagerly Calling Theorem Provers. Some applications of forward symbolic execution eagerly call a theorem prover when the VC for each program path is first generated. For example, an application such as automated test case generation may only care about a single path to the desired program point, and hope that eagerly calling the theorem prover will result in a satisfiable answer before all

paths are enumerated. Of course in the worse case all paths must be enumerated before finding a satisfiable one. In such a case the application would have done better by using the DWP directly.

However, the DWP can be advantageous for combining path formulas even when an application typically calls the theorem prover eagerly. For example, in if e then S_1 else S_s; S it would be naive to explore the rest of the program S when e; S_1 is initially taken, and then again consider the rest of the program S when $\neg e$; S_2 is taken. Instead, they benefit from creating a single compact VC after the if-then-else that considers all paths, which is then used when subsequently exploring S. Our DWP can be used as a replacement for previous work that try post-hoc to create compact VCs for two different path formulas [9**?**].

6. Related Work

Flanagan *et al.* [12] created the first algorithm for calculating the weakest precondition that is $O(M^2)$ in size. We use the same de-sugaring and approach to passification as proposed in that paper. They first note that by structural induction using something like Theorem 1 they can get a formula $O(M^4)$ in size, and then by renaming duplications in the wlp reduce the size to $O(M^2)$ (although the quadratic bound is argued, it is never formally proven). The actual statement of Theorem 1 was first proposed and proven by Leino [18]. We follow Brumley *et al.* [5] in forming our algorithm as a deductive system. Our motivating is to make proofs syntax-based where the induction hypotheses corresponds to the premise/sub-computation. We also use the fact that $wlp(S_1; S_2, \mathrm{F}) = wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})$ as shown [5]. Snugglebug [**?**] and Bouncer [9] perform post-hoc simplification when combining single-path formulas.

A recent survey of symbolic execution techniques and applications is provided in in [**?**]. Our formalization is motivated by recent work in automated test case generation such as [6, 8, 15–17], although not specific to only those applications. Although we restrict ourselves to symbolic execution where loops are bounded, our techniques also work when invariants are available by de-sugaring loops with invariants as in [12]. Special cases such as when loops and any loop-dependent side-effects can be written as a system of linear equations (e.g. [21]) are related and can possibly handled by a similar approach, but are not directly in the scope of this work.

7. Conclusion

We have shown the first efficient directionless weakest precondition algorithm where the order program statements are evaluated by the algorithm does not matter. We generate verification conditions at most $O(M^2)$ where M is the size of the program. In particular, our algorithm can take advantage of optimizations found in forward symbolic execution, which was previously not possible using weakest preconditions. We show equivalence between our algorithm, previous weakest precondition approaches that could only process statements last-to-first, and forward symbolic execution. The equivalence highlights a deep connection between forward symbolic execution and weakest preconditions where improvements in one algorithm will benefit the other. One implication is that we prove our algorithm can used as a drop-in replacement for forward symbolic execution engines with the benefit that VCs are exponentially smaller than existing forward symbolic execution approaches.

References

[1] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[3] D. Beyer, A. Chlipala, R. Majumdar, T. Henzinger, and R. Jhala. Generating tests from counterexamples. In *Proceedings of the ACM Conference on Software Engineering*, 2004.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *IEEE Transactions on Dependable and Secure Computing*, 5(4):224–241, Oct. 2008.

[5] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2007.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2006.

[9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the ACM Symposium on Operating System Principles*, Oct. 2007.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating System Principles*, 2005.

[11] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989.

[12] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Proceedings on the Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, Berlin, Germany, July 2007. Springer-Verlag.

[14] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

[15] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005.

[17] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2008.

[18] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the ACM Conference on Software Engineering*, pages 416–426, 2007.

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages*, 1997.

[21] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution of binary programs. In *International Symposium on Software Testing and Analysis*, 2009.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2005.

[23] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *International Symposium on Software Testing and Analysis*, 2004.

A. Full proofs

A.1 Proof of Lemma 3

Proof. We will first show equivalence with a modified version of f which eagerly calls G. We will call the modified version f'. This makes it easier to show equivalence with wp and wlp, by first proving equivalence to the intermediate formula. We start by showing that (a) if $G(f'(S)) = (N, W)$ then $N = \neg wlp(S, \mathrm{F})$ and $W = \neg wp(S, \mathrm{T})$, then we go on to show (b) $G(f(S)) = G(f'(S))$.

The inference rules for f' are the same as for f, except for F-SEQ:

$$\text{F-SEQ'}\quad \frac{f'(A) = (n_A, w_A) \qquad f'(B) = (n_B, w_B) \qquad G(n_A, W_A) = (N_A, W_B) \qquad G(n_B, w_B) = (N_B, W_B)}{f'(A; B) = ([N_a \wedge N_B], [W_a \vee (N_a \wedge W_B])}$$

We prove (a) by induction on the structure of S:

Case: $S = \mathsf{assert}\ e$

$G(f(\mathsf{assert}\ e))$
$= G([e], [\neg e])$
$= (e, \neg e)$
$= (\neg wlp(S, \mathrm{F}), \neg wp(S, \mathrm{T}))$

Case: $S = \mathsf{assume}\ e$

$G(f(\mathsf{assume}\ e))$
$= G([e], [\mathrm{F}])$
$= (e, \mathrm{F})$
$= (\neg wlp(S, \mathrm{F}), \neg wp(S, \mathrm{T}))$

Case: $S = S_1; S_2$
Let $G(f(S_1)) = (N_1, W_1)$ and $G(f(S_2)) = (N_2, W_2)$.

$(N_1, W_1) = (\neg wlp(S_1, \mathrm{F}), \neg wp(S_1, \mathrm{T}))$ By IH.
$(N_2, W_2) = (\neg wlp(S_2, \mathrm{F}), \neg wp(S_2, \mathrm{T}))$ By IH.
$G(f(S_1; S_2))$
$= G([N_1 \wedge N_2], [W_1 \vee (N_1 \wedge W_2)])$ By F-SEQ'.
$= (N_1 \wedge N_2, W_1 \vee (N_1 \wedge W_2))$ By G.
$\neg wlp(S_1; S_2, f) = \neg(wlp(S_1, \mathrm{F}) \vee wlp(S_2, f))$ By definition.
$= \neg wlp(S_1, \mathrm{F}) \wedge \neg wlp(S_2, f)$
$= N_1 \wedge N_2$
$\neg wp(S_1; S_2, t) = \neg wp(S_1, wp(S_2, t))$ By definition.
$= \neg(wp(S_1, \mathrm{T}) \wedge (wlp(S_1, f) \vee wp(S_2, \mathrm{T})))$ By Thm 1.
$= \neg wp(S_1, \mathrm{T}) \vee \neg(wlp(S_1, f) \vee wp(S_2, \mathrm{T}))$
$= \neg wp(S_1, \mathrm{T}) \vee (\neg wlp(S_1, f) \wedge \neg wp(S_2, \mathrm{T}))$
$= W_1 \vee (N_1 \wedge W_2)$
$\therefore G(f(S_1; S_2)) = (\neg wlp(S_1; S_2, \mathrm{F}), \neg wp(S_1; S_2, \mathrm{T}))$

Case: $S = S_1 \square S_2$
Let $G(f(S_1)) = (N_1, W_1)$ and $G(f(S_2)) = (N_2, W_2)$.

$G(f(S_1 \square S_2))$
$= G(N_1 \vee N_2, W_1 \vee W_2)$ By F-CHOICE.
$= (N_1 \vee N_2, W_1 \vee W_2)$
$= (\neg wlp(S_1, \mathrm{F}) \vee \neg wlp(S_2, \mathrm{F}),$
$\quad \neg wp(S_1, \mathrm{T}) \vee \neg wp(S_2, \mathrm{T}))$ By IH.
$= (\neg(wlp(S_1, \mathrm{F}) \wedge wlp(S_2, \mathrm{F})),$
$\quad \neg(wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{T})))$ Logic.
$= (\neg wlp(S, \mathrm{F}), \neg wp(S, \mathrm{T}))$ By definition.

Now we just need to show that $G(f(S)) = G(f'(S))$. Note in particular that although the value returned by f will be different in some cases we want G to give us an equivalent formula.

Let $S = S_1; S_2$. For the cases where neither S_1 or S_2 contains a sequence, we have:

$f(S_1) = f'(S_1) = ([N_1], [W_1])$
and $f(S_2) = f'(S_2) = ([N_2], [W_2])$
Then $G(f(S)) = ([N_1]@[N_2], [W_1]@[W_2])$
$= (N_1 \wedge N_2, W_1 \vee (N_1 \wedge W_2))$

$= G(f'(S))$

For the case where there is a sequence of more than 2 statements, we know that $(S_a; S_b); S_c = S_a; (S_b; S_c)$ so we will assume without loss of generality that S_1 is never itself a sequence.

Note since S_1 is not a sequence, by induction $f(S_1) = f'(S_1) = ([N_1], [W_1])$. Let $f(S_2) = (n_2, w_2)$, and $f'(S_2) = (n_2', w_2') = ([N_2], [W_2])$. Then

$f(S) = ([N_1]@n_2, [W_1]@w_2)$ By f-seq.
$f'(S) = (N_1 \wedge N_2, W_1 \vee (N_1 \wedge W_2))$ By f-seq'.
$G(n_2, w_2) = G(f'(S_2))$ By IH.
$G(f(S)) = (N_1 \wedge N_2, W_1 \vee (N_1 \wedge W_2))$ By G-rec.
$= G(f'(S))$

□

A.2 Proof of Lemma 5

Proof for part (i). By induction on the structure of the deduction $\mathcal{D}$ of $G(n, w)$:

Case: $\mathcal{D} = \dfrac{\text{G-BASE}\quad x \text{ is fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)}$

$|x| = 1$ By definition.
$= 2 \cdot \texttt{len}(n) - 1$ Since $\texttt{len}(n) = 1$

Case: $\mathcal{D} = \dfrac{\text{G-REC}\quad x \text{ is fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$

$|N_s| = 2 \cdot \texttt{len}(n_s) - 1$ By IH.
$|x \wedge N_s| = |x| + |N_s| + 1 = 2(\texttt{len}(n_s) + 1) - 1 = 2 \cdot \texttt{len}(n) - 1$

□

Proof for part (ii). By induction on the structure of the deduction $\mathcal{D}$ of $G(n, w)$:

Case: $\mathcal{D} = \dfrac{\text{G-BASE}\quad x \text{ is fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)}$

$|W| = |[W]| = |w|$ By definition
$= |w| + 3(\texttt{len}(n) - 1)$ Since $\texttt{len}(n) = 1$

Case: $\mathcal{D} = \dfrac{\text{G-REC}\quad x \text{ is fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$

$|W_s| = |w_s| + 3(\texttt{len}(n_s) - 1)$ By IH.
$|W_1 \vee (x \wedge W_s)| = |W_1| + (|x| + |W_s| + 1) + 1$ By definition.
$= |W_1| + |W_s| + 3$
$= |W_1| + |w_s| + 3(\texttt{len}(n_s) - 1) + 3$
$= |[W_1]@w_s| + 3(\texttt{len}(n_s) + 1 - 1)$
$= |w| + 3(\texttt{len}(n) - 1)$

□

Proof for part (iii). By induction on the structure of the deduction $\mathcal{D}$ of $G(n, w)$:

Case: $\mathcal{D} = \dfrac{\text{G-BASE}\quad x \text{ is fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)}$

$\texttt{len}(\{(x, N_1)\}) = 1 = \texttt{len}(n)$

Case: $\mathcal{D} = \dfrac{\text{G-REC}\quad x \text{ is fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$

$\texttt{len}(V_s) = \texttt{len}(n_s)$ — By IH.
$\texttt{len}(\{(x, N_1)\} \cup V_s) = 1 + \texttt{len}(V_s)$ — Because x is unique.
$= 1 + \texttt{len}(n_s) = \texttt{len}(n)$

□

Proof for part (iv). By induction on the structure of the deduction $\mathcal{D}$ of $G(n, w)$:

Case: $\mathcal{D} = \dfrac{\text{G-BASE}\quad x \text{ is fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)}$

$|V| = |\{(x, N_1)\}| = |x| + |N_1| + 1 = 2 + |[N_1]| = 2 \cdot \texttt{len}(n) + |n|$

Case: $\mathcal{D} = \dfrac{\text{G-REC}\quad x \text{ is fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$

$|V_s| = 2 \cdot \texttt{len}(n_s) + |n_s|$ — By IH.
$|\{(x, N_1)\} \cup V_s| = |(x, N_1)| + |V_s| = (|x| + |N_1| + 1) + 2 \cdot \texttt{len}(n_s) + |n_s| = (2 + 2 \cdot \texttt{len}(n_s)) + (|n_s| + |N_1|)$
$= 2 \cdot \texttt{len}(n) + |n|$

□

A.3 Proof of Lemma 6

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(n, w, S)$.

Case: $\mathcal{D} = \dfrac{\text{F-ASSERT}}{f(\mathsf{assert}\ e) = (\emptyset, [e], [\neg e])}$

$\texttt{len}(n) = \texttt{len}([e]) = 1$ — n has a single expression.
$= \texttt{len}([\neg e]) = \texttt{len}(w)$ — Same for w.
$= \texttt{seqlen}(S)$ — Since $\texttt{seqlen}(\mathsf{assert}\ e) = 1$

Case: $\mathcal{D} = \dfrac{\text{F-ASSUME}}{f(\mathsf{assume}\ e) = (\emptyset, [e], [false])}$

Symmetric to the previous case.

Case: $\mathcal{D} = \dfrac{\text{F-SEQ}\quad f(A) = (v_A, n_A, w_A) \qquad f(B) = (v_B, n_B, w_B)}{f(A; B) = ((v_A \cup v_B), n_A@n_B, w_A@w_B)}$

$\texttt{len}(n_A) = \texttt{len}(w_A) = \texttt{seqlen}(A)$ — By IH
$\texttt{len}(n_B) = \texttt{len}(w_B) = \texttt{seqlen}(B)$ — By IH
$\texttt{len}(n) = \texttt{len}(n_A@n_B)$
$= \texttt{len}(n_A) + \texttt{len}(N_B)$ — By definition.
$= \texttt{seqlen}(A) + \texttt{seqlen}(B)$ — Substitution.
$= \texttt{seqlen}(S)$ — By definition.

Case: $\mathcal{D} = \dfrac{\text{F-CHOICE}\quad f(A) = (v_A, n_A, w_A) \qquad G(n_A, w_A) = (V_A, N_A, W_A) \qquad f(B) = (v_B, n_B, w_B) \qquad G(n_B, w_B) = (V_B, N_B, W_B)}{f(A \,\square\, B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$

$\texttt{len}(n) = \texttt{len}([N_A \vee N_B]) = 1$ — We added one expression to n
$= \texttt{len}([W_A \vee W_B]) = \texttt{len}(w)$ — same for w
$\texttt{len}(n) = \texttt{len}(w) = \texttt{seqlen}(S)$ — Since $\texttt{seqlen}(S) = 1$

□

A.4 Proof of Lemma 7

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(n, w, S)$.

Case: $\mathcal{D} = \dfrac{\text{F-ASSERT}}{f(\mathsf{assert}\ e) = (\emptyset, [e], [\neg e])}$

$\mathtt{len}(v) = \mathtt{len}(\emptyset) = 0$
$\mathtt{len}(S) - \mathtt{seqlen}(S) = 1 - 1 = 0$

Case: $\mathcal{D} = \dfrac{\text{F-ASSUME}}{f(\mathsf{assume}\ e) = (\emptyset, [e], [false])}$

Same as previous case.

Case: $\mathcal{D} = \dfrac{\text{F-SEQ}\quad f(A) = (v_A, n_A, w_A) \qquad f(B) = (v_B, n_B, w_B)}{f(A; B) = ((v_A \cup v_B), n_A @ n_B, w_A @ w_B)}$

$\mathtt{len}(v_A) = \mathtt{len}(A) - \mathtt{seqlen}(A)$ — By IH.
$\mathtt{len}(v_B) = \mathtt{len}(B) - \mathtt{seqlen}(B)$ — By IH.
$\mathtt{len}(v_A \cup v_B) = \mathtt{len}(v_A) + \mathtt{len}(v_B)$ — Unique variables.
$= \mathtt{len}(A) - \mathtt{seqlen}(A) + \mathtt{len}(B) - \mathtt{seqlen}(B)$
$= \mathtt{len}(S) - \mathtt{seqlen}(S)$

Case: $\mathcal{D} = \dfrac{\text{F-CHOICE}\quad f(A) = (v_A, n_A, w_A) \qquad G(n_A, w_A) = (V_A, N_A, W_A) \qquad f(B) = (v_B, n_B, w_B) \qquad G(n_B, w_B) = (V_B, N_B, W_B)}{f(A \sqcap B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$

$\mathtt{len}(n_A) = \mathtt{seqlen}(A)$ — By Lemma 6.
$\mathtt{len}(V_A) = \mathtt{seqlen}(A)$ — By Lemma 5.
$\mathtt{len}(n_B) = \mathtt{seqlen}(B)$ — By Lemma 6.
$\mathtt{len}(V_B) = \mathtt{seqlen}(B)$ — By Lemma 5.
$\mathtt{len}(v_A) = \mathtt{len}(A) - \mathtt{seqlen}(A)$ — By IH.
$\mathtt{len}(v_B) = \mathtt{len}(B) - \mathtt{seqlen}(B)$ — By IH.
$\mathtt{len}(v_A \cup v_B \cup V_A \cup V_B) =$
$\mathtt{len}(v_A) + \mathtt{len}(v_B) + \mathtt{len}(V_A) + \mathtt{len}(V_B)$ — Unique variables.
$= \mathtt{len}(A) + \mathtt{len}(B) = \mathtt{len}(S) - \mathtt{seqlen}(S)$

□

A.5 Proof of Lemma 8

Proof. By induction on the structure of the deduction $\mathcal{D}$ of $f(S)$.

Case: $\mathcal{D} = \dfrac{\text{F-ASSERT}}{f(\mathsf{assert}\ e) = (\emptyset, [e], [\neg e])}$

$|n| = |[e]| = |e|$
$|w| = |[\neg e]| = |e| + 1$
$|S| = |e| + 1$
$\therefore |n| + |w| < 2 \cdot |S|$

Case: $\mathcal{D} = \dfrac{\text{F-ASSUME}}{f(\mathsf{assume}\ e) = (\emptyset, [e], [false])}$

$|n| = |[e]| = |e|$
$|w| = |[false]| = 1$
$|S| = |e| + 1$
$\therefore |n| + |w| < 2 \cdot |S|$

Case: $\mathcal{D} = \dfrac{\text{F-SEQ}\quad f(A) = (v_A, n_A, w_A) \qquad f(B) = (v_B, n_B, w_B)}{f(A; B) = ((v_A \cup v_B), n_A @ n_B, w_A @ w_B)}$

$|v_A| + |n_A| + |w_A| \leq 2 \cdot |A| + 7(\texttt{len}(A) - \texttt{seqlen}(A))$ By IH
$|v_B| + |n_B| + |w_B| \leq 2 \cdot |B| + 7(\texttt{len}(B) - \texttt{seqlen}(B))$ By IH
$|v| + |n| + |w| \leq |v_A| + |v_B| + |n_A| + |n_B| + |w_A| + |w_B|$
$\leq |v_A| + |n_A| + |w_A| + 2 \cdot |B| +$
$7(\texttt{len}(B) - \texttt{seqlen}(B))$ Substitution
$\leq 2 \cdot |A| + 7 \cdot \texttt{len}(A) - 7 \cdot \texttt{seqlen}(B) + 2 \cdot |B| +$
$7(\texttt{len}(B) - \texttt{seqlen}(B))$ Substitution
$\leq 2 \cdot |S| + 7(\texttt{len}(S) - \texttt{seqlen}(S))$
By $|A| + |B| < |A; B|$ and $\texttt{len}(A) + \texttt{len}(B) = \texttt{len}(A; B)$

Case: $\mathcal{D} =$
F-CHOICE
$$\frac{f(A) = (v_A, n_A, w_A) \qquad G(n_A, w_A) = (V_A, N_A, W_A) \qquad f(B) = (v_B, n_B, w_B) \qquad G(n_B, w_B) = (V_B, N_B, W_B)}{f(A \,\square\, B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$$

$|v_A| + |n_A| + |w_A| \leq 2 \cdot |A| + 7(\texttt{len}(A) - \texttt{seqlen}(A))$ By IH
$|V_A| + |N_A| + |W_A| = 2 \cdot \texttt{len}(n_A) + |n_A|$
$+2 \cdot \texttt{len}(n_A) - 1 + |w_A| + 3 \cdot \texttt{len}(n_A) - 3$ By Lemma 5
$= 7 \cdot \texttt{len}(n_A) + |n_A| + |w_A| - 4$ Simplification
$|v_A| + |V_A| + |N_A| + |W_A|$
$\leq 7 \cdot \texttt{len}(n_A) - 4 + 2 \cdot |A| +$
$7(\texttt{len}(A) - \texttt{seqlen}(A))$ From above two.
$= 7 \cdot \texttt{seqlen}(A) - 4 + 2 \cdot |A| +$
$7(\texttt{len}(A) - \texttt{seqlen}(A))$ By Lemma 6
$= 2 \cdot |A| + 7 \cdot \texttt{len}(A) - 4$ Simplification
Repeat above steps with B in place of A.

Then
$|v| + |n'| + |w'|$
$= |v_A| + |V_A| + |N_A| + |W_A| + |v_B| + |V_B| + |N_B| + |W_B| + 2$
$\leq 2 \cdot |A| + 7 \cdot \texttt{len}(A) + 2 \cdot |B| + 7 \cdot \texttt{len}(B) + 2 - 8$ Substitution
$= 2 \cdot |S| + 7 \cdot \texttt{len}(S) - 8$ Simplification
$< 2 \cdot |S| + 7(\texttt{len}(S) - \texttt{seqlen}(S))$ Since $\texttt{seqlen}(S) = 1$

□