15-316: Software Foundations of Security and Privacy

Lecture Notes on Automated Checkers & Memory Safety

Matt Fredrikson

Carnegie Mellon University
Lecture 6

1 Introduction & Recap

In the previous lecture we looked into proving that programs satisfy safety properties given as formulas in the first-order dynamic logic. In particular, we can write contract properties with precondition P and postconditions Q for a program α as:

$$P \rightarrow [\alpha]Q \tag{1}$$

If this formula is valid, then it means that in every state ω, if $\omega \models P$ then after all terminating runs of α starting in ω the final state $\nu \models Q$.

We then defined the semantics of dynamic logic formulas, so that we can actually prove the validity of such formulas. However, this is not always such an easy thing to do because there are an infinite number of initial states to reason about. To address this we derived a set of axioms that can be used in sequent calculus proofs to reason about the validity of DL formulas.

([:=]) $[x := e]p(x) \leftrightarrow p(e)$

([assert]) $[\texttt{assert}(Q)]P \leftrightarrow (Q \wedge P)$

([if]) $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P \leftrightarrow (Q \rightarrow [\alpha]P) \wedge (\neg Q \rightarrow [\beta]P)$

([;]) $[\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$

([unwind]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]P$

([unfold]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \rightarrow [\alpha][\texttt{while}(Q)\,\alpha]P) \wedge (\neg Q \rightarrow P)$

Each of these axioms reduces reasoning about formulas involving the box modality to reasoning about a series of simpler formulas, sometimes not involving modalities at all. Moreover, the axioms can be "implemented" as a syntactic transformation on formulas, and so automated by tools. When a sequence of syntactic transformations results in formulas that don't contain any box modalities, then the original DL validity question becomes one of validity of arithmetic formulas. It again becomes possible to automate by invoking a decision procedure for first-order arithmetic, and many good ones exist.

This is all very good, but not entirely true in all cases. In particular the axioms [unwind] and [unfold] for dealing with loops do not actually simplify matters at all. Applying either of these axioms results in an equivalent DL formula that reasons about the safety of the first iteration of the loop, and then a DL formula that is a carbon copy of the original for the remaining iterations.

The way to deal with this properly is to reason about loop invariants, introducing an axiom that requires us to prove the correctness of the invariant and that it implies the postcondition. But loop invariants cannot in general be derived automatically, and we are interested in techniques that can be automated and implemented in tools.

Another way to deal with this is to apply [unwind] or [unfold] repeatedly for a while, and simply cut the proof off at a certain point. This is equivalent to unrolling the loop some bounded number of times, and assuming that it will terminate before hitting that bound. If we can prove safety for all iterations up to the bound, then we know that any safety violation must occur on longer executions. This is not an ideal solution because we know nothing about the program's behavior past the unwinding bound, but it will certainly give us more assurance than we would have gotten by running a few test cases.

2 Bounded Model Checking

The principle behind bounded model checking is straightforward. First, pick a bound N on the execution depth of the program. This bound can refer to the total number of commands that are executed, or to the number of times loops are unrolled. We will adopt the latter convention, as it frees us from the potential awkwardness of splitting compound commands like conditionals and loops partway through their bodies.

Having fixed an upper bound on the execution depth, proceed to check the safety property $P \rightarrow [\alpha]Q$ by repeatedly applying [;], [:=], [assert], and [if], as well as the axioms from the propositinal sequent calculus when necessary. Whenever the proof reaches a point at which no further progress can be made because all box modalities contain outermost $\texttt{while}(Q)\,\alpha$ commands, then apply [unfold] and repeat the above process on the loop body α. When [unfold] has been applied to each loop N times, then replace each occurrence of $[\texttt{while}(Q)\,\alpha]P$ with P. Then proceed to close out the proof by reducing any remaining obligations to arithmetic formulas and applying the rule $\mathbb{Z}$.

The best way to understand how this works is to see it in action. Let's start off simple,

supposing that we wish to verify the following up to a bound of $N = 1$:

$$x \neq 0 \to [z := 0; \texttt{while}(y > 0)\,\{\texttt{if}(y\%2 = 1)\,\{z := z + x\}x := 2 * x; y := y/2\}]z \neq 0$$

To keep things easier to read, we will let α denote the body of the while loop. We proceed as follows.

$$\dfrac{\dfrac{\dfrac{x \neq 0, z = 0 \vdash [\texttt{while}(y > 0)\,\alpha]z \neq 0}{x \neq 0 \vdash [z := 0][\texttt{while}(y > 0)\,\alpha]z \neq 0}\,[:=]=}{x \neq 0 \vdash [z := 0; \texttt{while}(y > 0)\,\alpha]z \neq 0}\,[;]}{\vdash x \neq 0 \to [z := 0; \texttt{while}(y > 0)\,\alpha]z \neq 0}\,\to\text{R}$$

We can proceed no further in the proof without applying [unfold]. So we proceed to unfold the loop, and apply non-loop axioms afterwards until we get stuck again.

$$\dfrac{\dfrac{\dfrac{x \neq 0, z = 0, y > 0 \vdash [\alpha][\texttt{while}(y > 0)\,\alpha]z \neq 0}{x \neq 0, z = 0 \vdash y > 0 \to [\alpha][\texttt{while}(y > 0)\,\alpha]z \neq 0}\,\to\text{R} \quad \dfrac{x \neq 0, z = 0, y \leq 0 \vdash z \neq 0}{x \neq 0, z = 0 \vdash y \leq 0 \to z \neq 0}\,\to\text{R}}{x \neq 0, z = 0 \vdash (y > 0 \to [\alpha][\texttt{while}(y > 0)\,\alpha]z \neq 0) \wedge (y \leq 0 \to z \neq 0)}\,\wedge\text{R}}{x \neq 0, z = 0 \vdash [\texttt{while}(y > 0)\,\alpha]z \neq 0}\,[\text{unfold}]$$

At this point we can't help but notice that the branch of our proof with $y \leq 0$ in the assumptions has no path forward. The formula:

$$x \neq 0 \wedge z = 0 \wedge y \leq 0 \to z \neq 0 \tag{2}$$

simply is not a valid formula of arithmetic. So, we've found a bug. What's more, examining the sequent that we are unable to prove:

$$x \neq 0, z = 0, y \leq 0 \vdash z \neq 0 \tag{3}$$

we can extract more useful information about the bug. In particular the context provided by our assumptions tells us exactly what conditions of the initial state need to hold in order for the program to produce a trace that violates the safety property. So if we take any values of x, y, z that satisfy the assumptions $x \neq 0 \wedge z = 0 \wedge y \leq 0$ then we are guaranteed to "exercise" the bug. Such a set of inputs and its corresponding trace is called a *counterexample* to the safety property, and is a useful artifact of model checkers when debugging programs in practice.

Back to basic axioms. In hindsight, perhaps this is not so impressive because one of our assumptions is $z = 0$. We probably should have seen this coming, because the program begins by initializing z in this way, and will only update it when the loop body is executed. Perhaps there are more bugs to uncover if we continue with the other branch of the loop. In the following, let β denote the program $x := 2 * x; y := y/2$.

$$\dfrac{\dfrac{\overset{①}{x \neq 0, z = 0, y > 0} \vdash [\texttt{if}(y\%2 = 1)\,\{z := z + x\}]\overset{②}{[\beta][\texttt{while}(y > 0)\,\alpha]z \neq 0}}{x \neq 0, z = 0, y > 0 \vdash [\texttt{if}(y\%2 = 1)\,\{z := z + x\}; \beta][\texttt{while}(y > 0)\,\alpha]z \neq 0}\,[;]}{}\,[\text{if}]$$

The branch of the proof marked ① continues below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \neq 0, z = 0, y > 0, y\%2 = 1, z_1 = z + x, x_1 = 2 * x, y_1 = y/2 \vdash [\texttt{while}(y_1 > 0)\,\alpha]z_1 \neq 0}{x \neq 0, z = 0, y > 0, y\%2 = 1, z_1 = z + x, x_1 = 2 * x \vdash [y := y/2][\texttt{while}(y > 0)\,\alpha]z_1 \neq 0}[:=]_=}{x \neq 0, z = 0, y > 0, y\%2 = 1, z_1 = z + x \vdash [x := 2 * x][y := y/2][\texttt{while}(y > 0)\,\alpha]z_1 \neq 0}[:=]_=}{x \neq 0, z = 0, y > 0, y\%2 = 1 \vdash [z := z + x][x := 2 * x][y := y/2][\texttt{while}(y > 0)\,\alpha]z \neq 0}[:=]_=}{x \neq 0, z = 0, y > 0, y\%2 = 1 \vdash [z := z + x][\beta][\texttt{while}(y > 0)\,\alpha]z \neq 0}[;]}{x \neq 0, z = 0, y > 0 \vdash y\%2 = 1 \to [z := z + x][\beta][\texttt{while}(y > 0)\,\alpha]z \neq 0}\to R$$

And the branch of the proof marked ② continues here.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \neq 0, z = 0, y > 0, y\%2 \neq 1, x_1 = 2 * x, y_1 = y/2 \vdash [\texttt{while}(y_1 > 0)\,\alpha]z \neq 0}{x \neq 0, z = 0, y > 0, y\%2 \neq 1, x_1 = 2 * x \vdash [y := y/2][\texttt{while}(y > 0)\,\alpha]z \neq 0}[:=]_=}{x \neq 0, z = 0, y > 0, y\%2 \neq 1 \vdash [x := 2 * x][y := y/2][\texttt{while}(y > 0)\,\alpha]z \neq 0}[:=]_=}{x \neq 0, z = 0, y > 0, y\%2 \neq 1 \vdash [x := 2 * x; y := y/2][\texttt{while}(y > 0)\,\alpha]z \neq 0}[;]}{x \neq 0, z = 0, y > 0 \vdash y\%2 \neq 1 \to [x := 2 * x; y := y/2][\texttt{while}(y > 0)\,\alpha]z \neq 0}\to R$$

Now in both branches of the proof, we can't go any further without applying [unfold]. We initially set our bound to $N = 1$, and we've unrolled the loop exactly one time.

Verification conditions. So we proceed to replace the formula $[\texttt{while}(y > 0)\,\alpha]z \neq 0$ with $z \neq 0$ on both branches, yielding the sequents:

$$x \neq 0, z = 0, y > 0, y\%2 = 1, z_1 = z + x, x_1 = 2 * x, y_1 = y/2 \vdash z_1 \neq 0 \tag{4}$$

$$x \neq 0, z = 0, y > 0, y\%2 \neq 1, x_1 = 2 * x, y_1 = y/2 \vdash z \neq 0 \tag{5}$$

The sequents shown in 4 and 5 are our remaining proof obligations: if they are valid, then we can conclude that the safety property holds on the program traces corresponding to the paths that generated these obligations.

- In the case of 4, the corresponding path enters the while loop (reflected by the assumption $y > 0$ in the sequent), enters the body of the conditional (reflected by the assumption $y\%2 = 1$), and executes the remainder of the loop body stopping just before iterating again.
- In the case of 5, the corresponding path enters the while loop ($y > 0$ is still in the assumptions), skips over the body of the conditional (reflected by $y\%2 \neq 1$), and executes the rest of the loop body stopping prior to another iteration.

Observe that in both cases the proof obligations involve nothing but arithmetic. Recalling the meaning of sequents, we can derive arithmetic formulas whose validity implies the correctness of these paths.

$$x \neq 0 \land z = 0 \land y > 0 \land y\%2 = 1 \land z_1 = z + x \land x_1 = 2 * x \land y_1 = y/2 \to z_1 \neq 0 \tag{6}$$

$$x \neq 0 \land z = 0 \land y > 0 \land y\%2 \neq 1 \land x_1 = 2 * x \land y_1 = y/2 \to z \neq 0 \tag{7}$$

Equations 6 and 7 are called *verification conditions*. Recall from earlier Equation 2, the formula whose invalidity told us that the program contains a bug whenever $x \neq 0 \land z = 0 \land y \leq 0$. This was also a verification condition, corresponding to the program path where the body of the while loop is skipped over immediately leading to termination.

The primary job of a bounded model checker is to generate verification conditions for each program path within the execution depth bound. This can be done fully automatically, because there is nothing particularly difficult about applying the axioms [;], $[:=]_=$, [assert], and [if]. As each verification condition is derived, the bounded model checker consults an automated decision procedure for arithmetic. This often involves exploiting the duality between satisfiability and validity covered in the second lecture, as most decision procedures are designed to answer satisfiability queries rather than validity. But this is not a practical hurdle, as it merely involves negating the verification condition.

Other first-order theories. So far in this course we have assumed that our programs operate over "real" unbounded integers. This means that the verification conditions that we generate are formulas in the first-order theory of integer arithmetic. This theory is defined by the interpretation given to the constants (0,1,2,...), functions ($+, -, \times, \ldots$), and predicates ($\leq, =, \ldots$). We expect that claims of validity for these formulas assume the usual interpretation for such entities that we are familiar with from arithmetic over the integers.

Programs written in languages like C do not operate over such integers. Rather, they operate over machine integers that have bounded with (i.e., 32 or 64 bits), and thus can only take values from a finite set that can be represented as binary numbers of the according width. Likewise, functions like addition and subtraction have a different interpretation over the machine integers, which for example manifest when the result of an operation is too large or too small to be represented by the width of the architecture (i.e., overflow and underflow). Machine integers have additional functions such as bitwise `&` and `|`, and shift operators `>>` and `<<`.

Just as one can define a first-order theory of "real" integers, it is possible to define a first-order theory of machine integers, or perhaps floating-point decimal numbers, by assigning the appropriate interpretations to the constants, functions, and predicates pertinent to machine integers. Decision procedure developers of course realize this, and have indeed built support for such theories in widely-used tools. Bounded model checkers exploit this to faithfully model the semantics of machine arithmetic for languages like C by generating verification conditions for the first-order theory of machine integers. Importantly, *doing so does not generally entail changes to the verification condition generator itself*, as this is a purely syntactic analysis that does not depend on the interpretation of the underlying term constructors!

Let's look at a quick example involving machine integer arithmetic to get a sense of the differences that might arise when reasoning about correctness and safety. Consider the following program, which diligently checks that the denominator is non-zero before using it.

```
1 if(a>0 ∨ b>0) {
2   assert(a+b ≠ 0);
3   x := c/(a+b);
4 }
```

We can check that the safety property corresponding to the assertion holds by reasoning about the validity of $[\texttt{if}(a > 0 \lor b > 0))\,\{\texttt{assert}(a + b \neq 0); x := c/(a + b)\}]\top$, as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash a > 0 \lor b > 0 \to a + b \neq 0}{\vdash a > 0 \lor b > 0 \to [\texttt{assert}(a + b \neq 0)]\top}\,[\text{assert}]}{\vdash a > 0 \lor b > 0 \to [\texttt{assert}(a + b \neq 0)][x := c/(a + b)]\top}\,[:=]}{\vdash a > 0 \lor b > 0 \to [\texttt{assert}(a + b \neq 0); x := c/(a + b)]\top}\,[;]}{\vdash [\texttt{if}(a > 0 \lor b > 0))\,\{\texttt{assert}(a + b \neq 0); x := c/(a + b)\}]\top}\,[\text{if}]$$

The last step follows because $a + b \neq 0 \land \top$ is equivalent to $a + b \neq 0$. Now in the theory of integer arithmetic $\mathbb{Z}$, our verification condition

$$a > 0 \lor b > 0 \to a + b \neq 0 \tag{8}$$

is perfectly valid, and we could close out the proof by simply applying the rule $\mathbb{Z}$. But in the theory of 32-bit machine integer arithmetic, if a and b are unsigned then Eq. 8 is not valid. Consider the counterexample $a = 2^{32} - 1 = \texttt{0xFFFFFFFF} = 4294967295$ and $b = 1$. Then $a + b = 0$ because the result $4294967296 = 2^{32}$ is too large to fit in a 32-bit unsigned integer representation, so the result of the addition wraps around to zero.

From now on, when we want to use the theory of machine integer arithmetic in our proofs, we will use the rule $\mathbb{Z}_M$.

Unwinding assertions. From what we've seen so far, bounded model checking gives us a certain limited kind of assurance about the safety of a program. Let's break it down into cases.

1. If the bounded model checker finds a bug, it can report a counterexample. If the verification condition generator was implemented correctly and the correct theory was used by the decision procedure to discharge the proof obligation leading to the bug, then we can be sure that there is actually a bug in our program.

   How do we know this? Recall the axioms used to generate the verification condition: [;], $[:=]_=$, [assert], [if], and [unfold]. Each of these is an equivalence reducing one formula to another, so that if the resulting verification condition is valid, then the original formula is as well. Likewise, if the VC is *not* valid, then the original formula was not either. If this isn't convincing enough, then most decision procedures will produce a counterexample to the VC, that we can construct an input to the program with and actually run to observe the bug.

2. If the bounded model checker does *not* find a bug, then all that we can say with confidence is that there are no bugs on paths up to the execution depth bound.

> At least we can say this, again due to the fact that we have proved the axioms of dynamic logic to be valid equivalences. But importantly, we can't say that there are *no* safety bugs in the program when the model checker fails to find one, as there could be a bug on some path past the depth bound. Likewise, it could just as well be the case that there are indeed no such bugs on longer paths, but we shouldn't take this view without solid evidence to back it up.

The second case is unfortunate. If the model checker finds no bugs then we will probably want to follow up to see if we can convince ourselves that there are no safety violations, but how?

One simple approach that sometimes works in practice is to use an *unwinding assertion*. As the name suggests, an unwinding assertion is an assertion command that is added to the program as the bounded model checker applies the [unfold] axiom. Whenever the depth bound is reached, rather than replacing $[\texttt{while}(Q)\,\alpha]P$ with P, $[\texttt{while}(Q)\,\alpha]P$ is replaced with $[\texttt{assert}(\neg Q)]P$. This way, verification will only succeed if the program would have terminated anyway after the bound was reached because $\neg Q$ is true at that point. Let's look at a short example to illustrate the idea. We will set a depth bound of $N = 2$ in the following proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x=2,0<x,x_1=x-1\vdash[\texttt{while}(0<x_1)\,x_1:=x_1-1]x_1=0}}{x=2,0<x\vdash[x:=x-1][\texttt{while}(0<x)\,x:=x-1]x=0}\,[:=]_=}{x=2\vdash 0\le x\to[x:=x-1][\texttt{while}(0<x)\,x:=x-1]x=0}\,{\to}\text{R} \quad \mathbb{Z}_M \quad \dfrac{*}{x=2\vdash 0\ge x\to x=0}}{x=2\vdash(0\le x\to x:=x-1;[\texttt{while}(0<x)\,x:=x-1]x=0)\wedge(0\ge x\to x=0)}\,{\wedge}\text{R}}{x=2\vdash[\texttt{while}(0<x)\,x:=x-1]x=0}\,[\text{unfold}]$$

We continue with the proof below, as we have run out of space. Let P denote our assumptions so far $x = 2, 0 < x, x_1 = x - 1$. This time when we apply [unfold], we will hit the execution bound immediately, and at that point insert the unwinding assertion.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P,0<x_1,x_2=x_1-1\vdash 0\ge x_2\wedge x_2=0}{P,0<x_1,x_2=x_1-1\vdash[\texttt{assert}(0\ge x_2)]x_2=0}\,[\text{assert}]}{P,0<x_1\vdash[x_1:=x_1-1][\texttt{assert}(0\ge x_1)]x_1=0}\,[:=]_=}{P\vdash 0<x_1\to[x_1:=x_1-1][\texttt{assert}(0\ge x_1)]x_1=0}\,{\to}\text{R} \quad \mathbb{Z}_M \quad \dfrac{*}{P\vdash 0\ge x_1\to x_1=0}}{P\vdash(0<x\to[x_1:=x_1-1][\texttt{assert}(0\ge x_1)]x_1=0)\wedge(0\ge x_1\to x_1=0)}\,{\wedge}\text{R}}{P\vdash[\texttt{while}(0<x_1)\,x_1:=x_1-1]x_1=0}\,[\text{unfold}]$$

Now we have reduced the problem to machine arithmetic, leading to the verification condition

$$x = 2 \wedge 0 < x \wedge x_1 = x - 1 \wedge 0 < x_1 \wedge x_2 = x_1 - 1 \to 0 \ge x_2 \wedge x_2 = 0 \tag{9}$$

This is of course valid, and because of the unwinding condition we know that there are no paths in the program that exceed the execution depth $N = 2$. From this we conclude that the program satisfies the safety property on *all* traces.

3 Symbolic Execution

Recall that when we discussed verification conditions, we saw that the assumptions in the context of each proof obligation reflect the path covered by that proof obligation. Also worth noting is that because we used $[:=]_=$ rather than $[:=]$, moving assignments to the assumptions and renaming variables each time, the context also tracks the intermediate state rather explicitly. Consider for example the following derivations.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x=a, y=b \vdash y=b \wedge x=a}}{x=a, y=b \vdash [z:=x]y=b \wedge z=a}[:=]}{x=a, y=b \vdash [z:=x][x:=y]x=b \wedge z=a}[:=]}{x=a, y=b \vdash [z:=x][x:=y][y:=z]x=b \wedge y=a}[:=]}{x=a, y=b \vdash [z:=x; x:=y; y:=z]x=b \wedge y=a}[;],[;]$$

In this case, the verification condition is simply

$$x=a \wedge y=b \rightarrow y=b \wedge x=a \tag{10}$$

All of the information about the intermediate states that the program entered to achieve its final result is gone from the condition. On the other hand, using $[:=]_=$ to do a similar derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x=a, y=b, z_1=x, x_1=y, y_1=z_1 \vdash x_1=b \wedge y_1=a}}{x=a, y=b, z_1=x, x_1=y \vdash [y:=z_1]x_1=b \wedge y=a}[:=]_=}{x=a, y=b, z_1=x \vdash [x:=y][y:=z_1]x=b \wedge y=a}[:=]_=}{x=a, y=b \vdash [z:=x][x:=y][y:=z]x=b \wedge y=a}[:=]_=}{x=a, y=b \vdash [z:=x; x:=y; y:=z]x=b \wedge y=a}[;],[;]$$

Now the verification condition is

$$x=a \wedge y=b \wedge z_1=x \wedge x_1=y \wedge y_1=z_1 \rightarrow x_1=b \wedge y_1=a \tag{11}$$

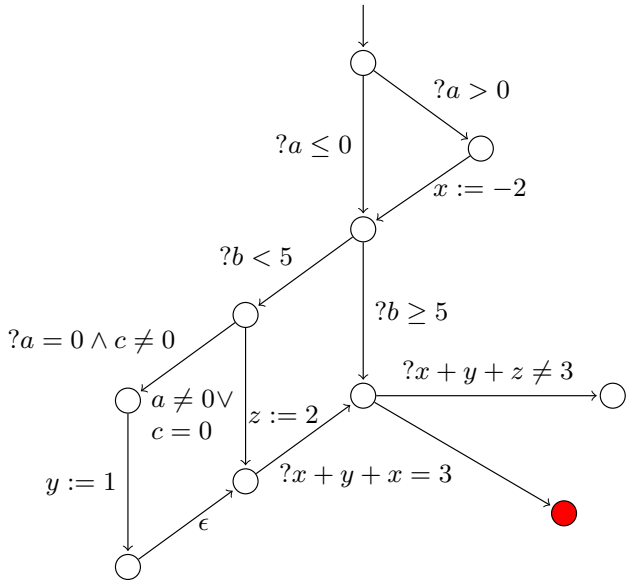
In terms of reasoning about correctness, there is no difference here. Equations 10 and 11 are equivalent. But the context tells us that the most recent "version" of y (i.e. y_1) was updated to take the most recent version of z (i.e. z_1), which was in turn updated to take the initial version of x.

Path formulas. The conjunction of the assumptions calculated in this way is called the *path formula* for the corresponding program path behind this derivation. Any feasible path through a program has a corresponding path formula that is satisfiable. *Symbolic execution* is a technique that enumerats program paths that may contain safety violations, generates their path formulas, and checks each formula for satisfiability. Whenever a path formula is satisfiable, it means that there is at least one trace that follows that path.

To facilitate enumerating path formulas, symbolic execution first constructs a *control flow graph* of the program that reflects all of the paths in the program. Consider the following program as an example.

```
1 if(a > 0) { x := -2; }
2 if(b < 5) {
3   if(a = 0 ∧ c ≠ 0) { y:= 1 }
4   z := 2;
5 }
6 assert(x+y+z ≠ 3);
```

The corresponding control flow graph is as follows.



The edges of the control flow graph are labeled to reflect the corresponding program command on that portion of the path. As shorthand, we write $?P$ to represent conditions that must hold, i.e. in place of $\texttt{assert}(P)$. The node marked in red denotes a path that violates the assertion. Edges labeled with ϵ are noops, and don't correspond to any program command. Then the task of symbolic execution is to determine if this node is reachable from an initial state from a feasible path.

Each path through the control flow graph corresponds to a verification condition, which we obtain by listing out a corresponding program for that path and then applying the relevant axioms of dynamic logic. For example, the path that first takes the right edge from the initial state, and then left edges until reaching the red node would be:

$$\vdash [?a > 0; x := -1; ?b < 5; ?a = 0 \land c \neq 0; y := 1; z := 2; ?x + y + z = 3]\top \tag{12}$$

Notice that we use the trivial postcondition $\top$ because we are merely interested in whether there are *any* traces that could follow this path. So we don't care what properties the final state may have, which is described by $\top$. Applying axioms $[:=]_=$ and [assert], we derive the following verification condition.

$$a > 0 \land x_1 = -1 \land b < 5 \land a = 0 \land c \neq 0 \land y_1 = 1 \land z_1 = 2 \land x_1 + y_1 + z_1 = 3 \tag{13}$$

This formula is not satisfiable, because $a > 0 \wedge a = 0$ is a contradiction, so there are no feasible traces that follow this path. On the other hand, if we had taken the left branch off of the initial state and then followed the same commands afterwards, it is not hard to check that we would obtain the following verification condition.

$$a \leq 0 \wedge b < 5 \wedge a = 0 \wedge c \neq 0 \wedge y_1 = 1 \wedge z_1 = 2 \wedge x + y_1 + z_1 = 3 \tag{14}$$

Equation 14 is indeed satisfiable, as evidenced by the witness $x = 0, y_1 = 1, z_1 = 2$. This means that the path is feasible, which we could show by running the program on any input with $x = 0$.

Checking invariants. Path formulas contain all of the information that we need to reason about the satisfaction of formulas over state at all points in the execution. For example, suppose we wish to check the invariant P. Part way through the derivation of

$$x = a, y = b \vdash [z := x; x := y; y := z]x = b \wedge y = a$$

we derived the sequent $x = a, y = b, z_1 = x \vdash [x := y][y := z]x = b \wedge y = a$. The context $x = a, y = b, z_1 = x$ reflects the state after the first assignment $z := x$. We need to show that at this point in the execution, P holds, i.e., we can prove validity of:

$$x = a, y = b, z_1 = x \vdash P$$

Executing the next assignment $x := y$ lead to the context $x = a, y = b, z_1 = x, x_1 = y$ and the obligation to prove:

$$x = a, y = b, z_1 = x, x_1 = y \vdash P$$

And finally, executing the last assignment $y := z$ gives the obligation:

$$x = a, y = b, z_1 = x, x_1 = y, y_1 = z_1 \vdash P$$

This is a useful tactic, and the ability to select paths from the program affords more flexibility than bounded model checking the program:

$$z := x; \texttt{assert}(P); x := y; \texttt{assert}(P); y := z; \texttt{assert}(P)$$

This flexibility is important when the program is too large to perform bounded model checking on to a sufficiently large execution depth. If we can reason efficiently that there is a subset of paths on which the safety property will definitely hold, then the remaining (hopefully much smaller or at least finite) set of paths can be enumerated with symbolic execution and discharged individually. In subsequent lectures, we will see how runtime safety checks can be used to ensure that certain paths will never violate safety, so that targeted techniques like symbolic execution can be used to reason about the rest.

4 Memory Safety

So far the programs that we have studied are not too interesting. While it is possible to write some non-trivial programs in the simple imperative language like Euclidean division, lots of interesting functionality like searching and sorting would be tedious to implement without arrays or some other form of indexed storage. So let's add a new feature to address this.

While most imperative programming languages support convenient dynamic memory allocation and access with syntax like `malloc` and `a[i]`, at the end of the day this is nothing more than syntactic sugar for managing a large integer-indexed array of values. We can add basic support for this to our language by introducing pointers, and adding an integer-indexed memory to our program state. Now terms in our language will have the following syntax.

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e} \mid \texttt{Mem}(e)$$

The term $\texttt{Mem}(e)$ denotes the value obtained by evaluating e in the current state, and accessing the memory at the corresponding index. This takes care of reading from the memory array, now we add support for updating memory by introducing a new type of program command.

$$\alpha, \beta \ ::= \ x := e \mid \texttt{Mem}(e) := \tilde{e} \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta \mid \alpha; \beta \mid \texttt{while}(Q)\, \alpha$$

The command $\texttt{Mem}(e) := \tilde{e}$ evaluates e and $\tilde{e}$ in the current state, and sets the value of memory indexed at the value of e to the value of $\tilde{e}$.

Now for the semantics. We will need to track the value of variables as we did before with a mapping from variables to values. But we will also need to track the state of the memory, which we will formalize as a partial mapping from non-negative integers to values. Real machines don't have unlimited memory, which is why the mapping is partial: we assume that the memory can hold at most U values, so the mapping is only defined on $0 \leq i \leq U$.

We will continue to denote states by ω, and write $\omega_V(x)$ to refer to the value of the variable mapping, and $\omega_M(x)$ to refer to the memory array. The semantics of terms can now be defined as follows.

Definition 1 (Semantics of terms)**.** The *semantics of a term* e in a state ω is its value $\omega[\![e]\!]$. It is defined inductively by distinguishing the shape of term e as follows:

- $\omega[\![x]\!] = \omega_V(x)$ for variable x
- $\omega[\![c]\!] = c$ for number literals c
- $\omega[\![e \odot \tilde{e}]\!] = \omega[\![e]\!] \odot \omega[\![\tilde{e}]\!]$, where $\odot \in \{+, \times\}$
- $\omega[\![\texttt{Mem}(e)]\!] = \omega_M(\omega[\![e]\!])$ if $0 \leq \omega[\![e]\!] < U$, else undefined

Adding pointers to our language has led to a complication: now terms can be undefined. Specifically, if e evaluates to a negative number, or a number larger than the maximum memory size U, then the term $\texttt{Mem}(e)$ is not defined.

This complication manifests in how we define the semantics of formulas. Because terms can now be undefined in certain states, we need to account for this in the semantics of formulas that might include terms. Whenever a term in a formula is undefined in a particular state, then the value of the formula is as well.

Definition 2 (Semantics of arithmetic formulas)**.** The DL formula P is true in state ω, written $\omega \models P$, as inductively defined by distinguishing the shape of formula P:

1. $\omega \not\models \bot$, i.e., $\bot$ is true in no states
2. $\omega \models \top$, i.e., $\top$ is true in all states
3. $\omega \models e = \tilde{e}$ iff $\omega[\![e]\!] = \omega[\![\tilde{e}]\!]$ and both terms are defined in ω.
4. $\omega \models e \leq \tilde{e}$ iff $\omega[\![e]\!] \leq \omega[\![\tilde{e}]\!]$ and both terms are defined in ω.
5. $\omega \models P \wedge Q$ iff $\omega \models P$ and $\omega \models Q$ if P and Q are defined in ω.
6. $\omega \models P \vee Q$ iff $\omega \models P$ or $\omega \models Q$ if P and Q are defined in ω.
7. $\omega \models \neg P$ iff $\omega \not\models P$ if P is defined in ω.
8. $\omega \models P \rightarrow Q$ iff $\omega \not\models P$ or $\omega \models Q$ and P and Q are defined in ω.
9. $\omega \models P \leftrightarrow Q$ iff both are true or both false and P and Q are defined in ω.

Finally, we get to the semantics of programs. Obviously we need to add a new definition for the memory update command $\texttt{Mem}(e) := \tilde{e}$. But programs may contain terms and formulas, which we now know can be undefined in some states. We define the semantics of a program with a term or formula that is undefined in a state as aborting in the next subsequent state.

First some notation. If ω_M is a memory in state ω, then we write $\omega_M\{e \mapsto \tilde{e}\}$ to denote the new memory obtained by copying ω_M, and changing its mapping at $\omega[\![e]\!]$ to map to $\omega[\![\tilde{e}]\!]$. So suppose that $\omega_M(0) = 1, \omega_M(1) = 2$. Then $\omega_M\{1 \mapsto 3\}(0) = 1$ and $(\omega_M\{1 \mapsto 3\})(1) = 3$. We can apply this update notation multiple times, so that:

$$\omega_M\{1 \mapsto 3\}\{0 \mapsto 4\}(0) = 4, \omega_M\{1 \mapsto 3\}\{0 \mapsto 4\}(1) = 3$$

We'll adopt the convention that the rightmost update to a particular index is the one that we use when looking up values. So for example,

$$\omega_M\{1 \mapsto 3\}\{1 \mapsto 4\}(1) = 4$$

Definition 3 (Trace semantics of programs)**.** The *trace semantics* $[\![\alpha]\!]$ *of a program* α is the set of all its possible traces and is defined inductively as follows:

1. $[\![x := e]\!] = \begin{array}{l}\{(\omega, \nu) \ : \ \omega[\![e]\!] \text{ is defined and } \nu = \omega \text{ except that } \nu_V(x) = \omega[\![e]\!]\} \cup \\ \{(\omega, \Lambda) \ : \ \omega[\![e]\!] \text{ is not defined}\}\end{array}$

2. $[\![\texttt{Mem}(e) := \tilde{e}]\!] = \begin{array}{l}\{(\omega, \nu) \ : \ 0 \leq \omega[\![e]\!] \leq U, \omega[\![\tilde{e}]\!] \text{ defined}, \nu_M = \omega_M\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}\} \cup \\ \{(\omega, \Lambda) \ : \ \neg(0 \leq \omega[\![e]\!] \leq U) \text{ or } \omega[\![\tilde{e}]\!] \text{ not defined}\}\end{array}$

3. $[\![\texttt{assert}(Q)]\!] = \begin{array}{l}\{(\omega) \ : \ \omega[\![e]\!] \text{ is defined and } \omega \models Q\} \cup \\ \{(\omega, \Lambda) \ : \ \omega \models Q \text{ is not defined or } \omega \not\models Q\}\end{array}$

4. $[\![\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]\!] = \begin{array}{l}\{\sigma \in [\![\alpha]\!] \ : \ \sigma_0[\![e]\!] \text{ is defined and } \sigma_0 \models Q\} \cup \\ \{\sigma \in [\![\beta]\!] \ : \ \sigma_0[\![e]\!] \text{ is defined and } \sigma_0 \not\models Q\} \cup \\ \{(\omega, \Lambda) \ : \ \omega \models Q \text{ is not defined}\}\end{array}$

5. $[\![\alpha; \beta]\!] = \{\sigma \circ \varsigma \ : \ \sigma \in [\![\alpha]\!], \varsigma \in [\![\beta]\!]\}$;
the composition of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \dots)$ and $\varsigma = (\varsigma_0, \varsigma_1, \varsigma_2, \dots)$ is

$$\sigma \circ \varsigma := \begin{cases}(\sigma_0, \dots, \sigma_n, \varsigma_1, \varsigma_2, \dots) & \text{if } \sigma \text{ terminates in } \sigma_n \text{ and } \sigma_n = \varsigma_0 \\ \sigma & \text{if } \sigma \text{ does not terminate}\end{cases}$$

6. $[\![\texttt{while}(Q)\,\alpha]\!] = \begin{array}{l}\{\sigma^{(0)} \circ \dots \circ \sigma^{(n)} \ : \ \text{for all } 0 \leq i < n\text{: } \sigma_0^{(i)} \models Q, \sigma^{(i)} \in [\![\alpha]\!], \text{ and} \\ \quad \sigma^{(n)} \text{ either doesn't terminate terminates where } \sigma_m^{(n)} \not\models Q\} \cup \\ \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \ : \ \text{for all } i \in \mathbb{N}\text{: } \sigma_0^{(i)} \models Q, \sigma^{(i)} \in [\![\alpha]\!]\} \cup \\ \{(\omega) \ : \ \omega \not\models Q\} \cup \\ \{(\omega, \Lambda) \ : \ \omega \models Q \text{ not defined}\}\end{array}$

While it may be tedious to track the presence of undefined terms and formulas through the evaluation of a program, we will see that this is central to the very definition of what memory safety means for a particular programming language.

Axioms and Proof Rules. Now we have semantics for programs with pointers and indexed memory, the next logical thing to do is find some useful axioms to help us reason about them.

Just as we had an axiom for assignment to variables, we have a similar axiom for updates to a pointer. But in the assignment axiom, we performed a syntactic substitution of the target variable in the postcondition. In this case we can readily see that looking for mere occurrences of a pointer expression will not suffice. Consider the following:

$$[x := 1; y := 1; \texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0 \tag{15}$$

After executing the first two assignments, $\texttt{Mem}(x)$ and $\texttt{Mem}(y)$ point to the same memory location. So if we tried to close out a proof like the following:

$$[:=]\frac{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash \texttt{Mem}(y) \neq 0}{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash [\texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0}$$

then we would be misled to say the least. Rather, we need to make sure that the update is reflected in any subsequent memory read to the same address, regardless of the syntactic form of the index term. Perhaps something like the following:

$$[\texttt{Mem}(e) := \tilde{e}]p(\texttt{Mem}) \leftrightarrow p(\texttt{Mem}\{e \mapsto \tilde{e}\}) \tag{16}$$

Now when we repeat the derivation from before,

$$\frac{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash \texttt{Mem}\{x \mapsto 0\}(y) \neq 0}{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash [\texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0}$$

there is no way to close out the proof because $x = y$ and $\texttt{Mem}\{x \mapsto 0\}(y) = 0$. But this proof rule isn't sound, because what if e evaluates to an out-of-bounds value? We need to add an assertion that the value of e is within the correct range. This leads to the $[*]_=$ axiom, which combines Equation 16 with the in-bounds check.

$$([*]_=) \quad [\texttt{Mem}(e) := \tilde{e}]p(\texttt{Mem}) \leftrightarrow p(\texttt{Mem}\{e \mapsto \tilde{e}\}) \wedge 0 \leq e < U$$

Axiom $[*]_=$ takes care of what to do when we update memory, but we also need a way to reason about reads from memory. If we only ever reason about programs that never update memory, then this is easy because anything we need to know about its value at particular indices is already in our assumptions. We can then work with it like we would any other set of value, essentially treatig each index like a separate constant.

But what about programs that update memory and then read from it afterwards? There are two cases to cover: reading from an index that was previously written to, and reading from one that was not. In the first case, we have some memory $\texttt{Mem}\{e \mapsto \tilde{e}\}$ and we perform an access $\texttt{Mem}\{e \mapsto \tilde{e}\}(e')$ where $e = e'$ in the current state. Then the value that is read from memory will be $\tilde{e}$. But of course we also need to make sure that we are reading from an index in the appropriate range. This is captured in the $[*]_1$ rules.

$$([*]_1) \quad \frac{\Gamma \vdash e = e' \quad \Gamma \vdash 0 \leq e < U}{\Gamma \vdash \texttt{Mem}\{e \mapsto \tilde{e}\}(e') = \tilde{e}}$$

In the case where $e \neq e'$, we use similar reasoning to conclude that $\texttt{Mem}\{e \mapsto \tilde{e}\}(e')$ takes whatever the value at index e' in Mem was *before* the update, i.e. $\texttt{Mem}(e')$. This gives us the $[*]_2$ rules.

$$([*]_2) \quad \frac{\Gamma \vdash e \neq e' \quad \Gamma \vdash 0 \leq e < U}{\Gamma \vdash \texttt{Mem}\{e \mapsto \tilde{e}\}(e') = \texttt{Mem}(e')}$$

The axiom $[*]_=$ and rules $[*]_1$, $[*]_2$ are sufficient to prove safety properties about programs with pointer operations.

Memory safety. The term *memory safety* refers to a set of properties that depends on the particulars of the language and instruction set architecture on which the program ultimately executes. But the common intent shared by all such properties is that programs satisfying memory safety never use pointers in a way that causes undefined behavior or forces the program to abort.

In our simplified language with pointers, any "bad" use of memory immediately leads to an abort on the corresponding trace, so we can define memory safety as the set of traces that do not abort due to a pointer read or write.

Definition 4 (Memory safety)**.** A program α satisfies memory safety if and only if for all $\sigma \in [\![\alpha]\!]$, whenever σ is finite and $\sigma_n = \Lambda$ then the last command executed on σ was not a pointer read or write.

One thing to notice is that when we use these axioms to prove *any* property about a program that uses pointers, we are forced to prove memory safety as well. The only case that me might forget to prove memory safety for is when a read is performed on memory without first having updated it. We can help ourselves remember to do this by replacing each command α that reads from or writes to memory in term $\texttt{Mem}(e)$ with the following composed command:

$$\texttt{assert}(0 \leq e < U); \alpha \tag{17}$$

This is a theorem that we are able to prove, in fact.

Theorem 5. *For any formula P and program α that has been rewritten according to (17), if $\Gamma \vdash [\alpha]P$, i.e. $[\alpha]P$ is provable from assumptions Γ using $[*]_=$, $[*]_1$, and $[*]_2$ in addition to other axioms of dynamic logic, then α satisfies memory safety.*

Proof. The way to prove this is by induction on the structure of proofs, just as we did to prove the soundness of the propositional sequent calculus and. This is a good exercise to complete on your own. □