Program Synthesis with TLA+

Jul 30, 2018 • Test Driven Development without the "Development" • 1543 words.

Computer programs can feel like incantations, secret spells we whisper to the electrons coursing in the metal beneath our fingertips. But they're really something much more prosaic: sequences of characters that we know how to compose and that computers know how to evaluate. They're just text.

The magic happens when we write programs, not when we run them. We can trace the evaluation of a program from keystroke to circuit. But our own thought processes, when we write, are more opaque.

That ineffability poses one central difficulty in getting computers to program themselves. If we don't understand how we program, how can we teach a computer to?

As it turns out, we don't always have to be so clever. Walking home recently, I realized that you could model what a programmer does in a state machine, and so could exploit a model checker like TLA+ to discover simple programs from specifications.

And that's exactly what Lucy, Emanuel, and I did a few weeks ago at RC.

State Machines and TLA+

A state machine isn't a literal machine. It's just a way of modeling how a system changes over time. You specify a state machine by describing an initial state of a system, and then by writing down all the rules for changing that system. For example, you might model the state of a basketball game by its score. The score can be updated by different kinds of shots: three-pointers, jumpers, free throws, etc.

TLA+ is a tool that allows you to specify a state machine, then check to see if any combination of updates can produce a new state that might satisfy a certain property. So you can ask if it's ever possible for a basketball team to have negative points (it's not). You might ask if it's possible for a team to have 42 more points than another (it is). If a state is reachable with those properties, TLA+ will find such a state and give you the steps it took to get there.

(I should note that I'm not going to get into the details of how to use TLA+. You can learn more about it from its homepage or from this tutorial.)

Programming as a State Machine

State machines require two things: an initial state and rules for updating that state. The thing I realized is that the state of a program is just the text of the program itself. And we change programs simply by adding more text to one. To model programming as a state machine, we just need to start with a blank program and specify rules for adding text to that program.

But in order to use TLA+'s model-checking capabilities, we need to test properties of our various state. This is where things get more interesting. The model checker will go through all the programs that the state machine can construct. As it generates these programs, we want to check whether each program, *when evaluated*, satisfies a given specification.

That is, if we know how to evaluate the text that is a computer program, and if we have a set of unit tests to serve as a specification for software we want to write, we can ask TLA+ to search for programs until it finds one that passes all those tests. You can think of it as "Test-Driven Development", just without the "Development".

Programming in TLA+

In order to do have TLA+ find programs for us, we first need to write a programming language that TLA+ can both generate and evaluate. TLA+ isn't the easiest language to hack in, but luckily it does have lists, which means that it's possible to write a kind of Lisp in it and to build up expressions directly in their abstract syntax tree (AST) representation.

We decided to write a tiny subset of Lisp that produced expressions that took one integer and produced one integer. The language consists of the following terms:

- The integers `1` through `4`
- A variable `x`
- The operators `+`, `-`, and `*`

We create new expressions by combining existing ones with an arithmetic operator. Each time we generate a new AST, we add it to the set of possible expressions we could use to generate more ASTs.

This is what it looks like in PlusCal, a language that compiles into TLA+. (You can also see the full code for this project on Github.)

```
variables AST = << >>,
          possibleExpr = ({<<"const", x>>: x \in 1..4}
                           \union {<<"var", "x">>});


begin
while TRUE do
    with expr1 \in possibleExpr do
        with expr2 \in possibleExpr do
            with op \in {"*", "+", "-"} do
                AST := << op, expr1, expr2 >>;
                possibleExpr := (possibleExpr \union {AST});
            end with;
        end with;
    end with;
end while;
```

Writing an evaluator

Now that we have the process for building up a Lisp expression, we need a way to evaluate it. Luckily, we're dealing with a small enough subset of Lisp that it's easy to write an evaluator. (By the way, TLA+ lists are 1-indexed, which might be jarring if you're not used to it.)

```
RECURSIVE evaluate(_, _)
evaluate(expr, x) ==
   CASE Head(expr) = "+" -> evaluate(expr[2], x) + evaluate(expr[3], x)
    [] Head(expr) = "*" -> evaluate(expr[2], x) * evaluate(expr[3], x)
    [] Head(expr) = "-" -> evaluate(expr[2], x) - evaluate(expr[3], x)
    [] Head(expr) = "const" -> expr[2]
    [] Head(expr) = "var" /\ expr[2] = "x" -> x
end define;
```

Creating the Invariant

So far, we have:

- A state machine for how to generate Lisp expressions
- An evaluator for those expressions

In order to leverage TLA+'s model checker, we still need a way to check if a given program is the one we're looking for. Here is the point where we write our "unit tests".

Before we proceed, if it's not already clear, this is clearly a hack of TLA+. I don't think it's meant to do this. And since it's a hack, there's a slight complication here.

Normally, when you model a state machine in TLA+, you want to check to see that there are states you cannot end up in. For example, you want to make sure that it's impossible for a basketball game to have negative scores. We therefore specify an *invariant*, which, when violated, TLA+ will complain about. That's really useful when you're checking for bugs, since TLA+ will surface them for you.

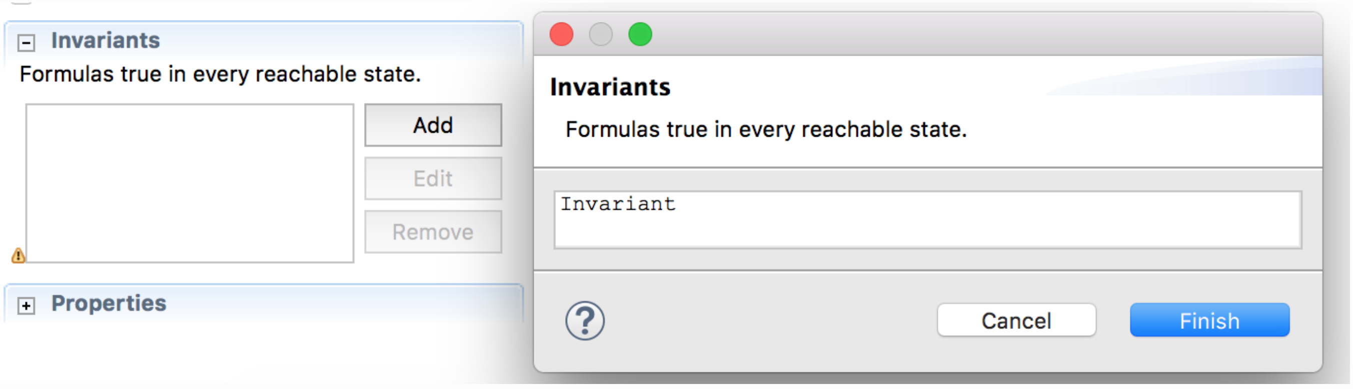
But, in our case, we *want* to trip up TLA+. We therefore express our invariant negatively, as something like "the program, when evaluated on x, y, and z, does *not* produce a, b, and c". That way, TLA+ will complain when it finds a program that *does* have that property, handing it to us on a silver platter.

Now we're ready to issue TLA+ a program synthesis challenge. Can it discover `x^2 + 1`? We might first check to see if an expression, evaluated on `0`, returns `1`. We'd write that like so:

```
Invariant == ~(length(AST) > 0 /\ evaluate(AST, 0) = 1)
```

This is analogous to writing a unit test. We expect a program evaluated on `0` to return `1`. (We also need to assume that the AST isn't empty, since our evaluator would otherwise crash on that input.)

Next, we need to alert TLA+ to check for that invariant:

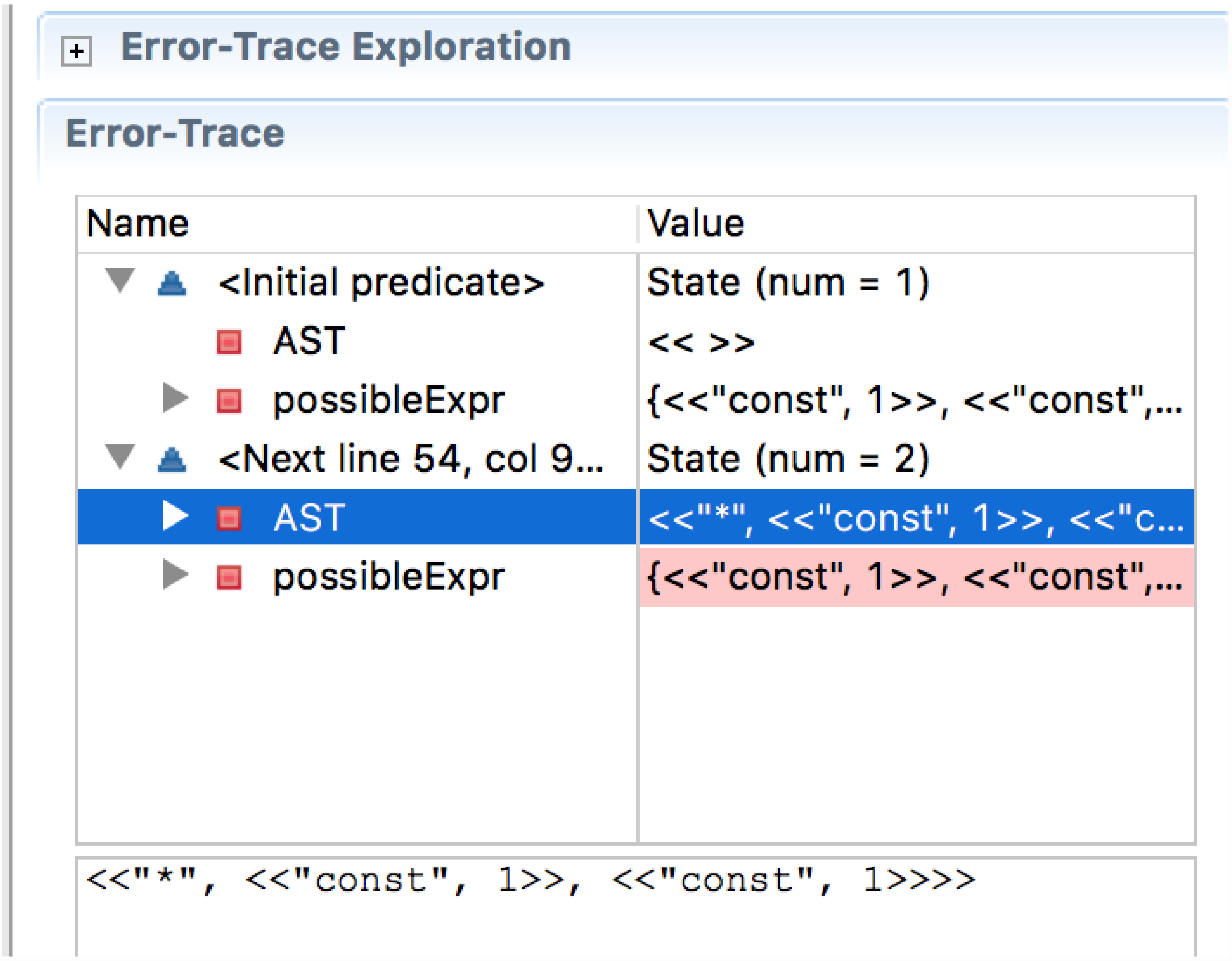

Invariants
Formulas true in every reachable state.

Invariants
Formulas true in every reachable state.

```
Invariant
```

Running the model

When we run the model, we find something surprising. TLA+ gives us `<<"*", <<"const", 1>>, <<"const", 1>>>>`.



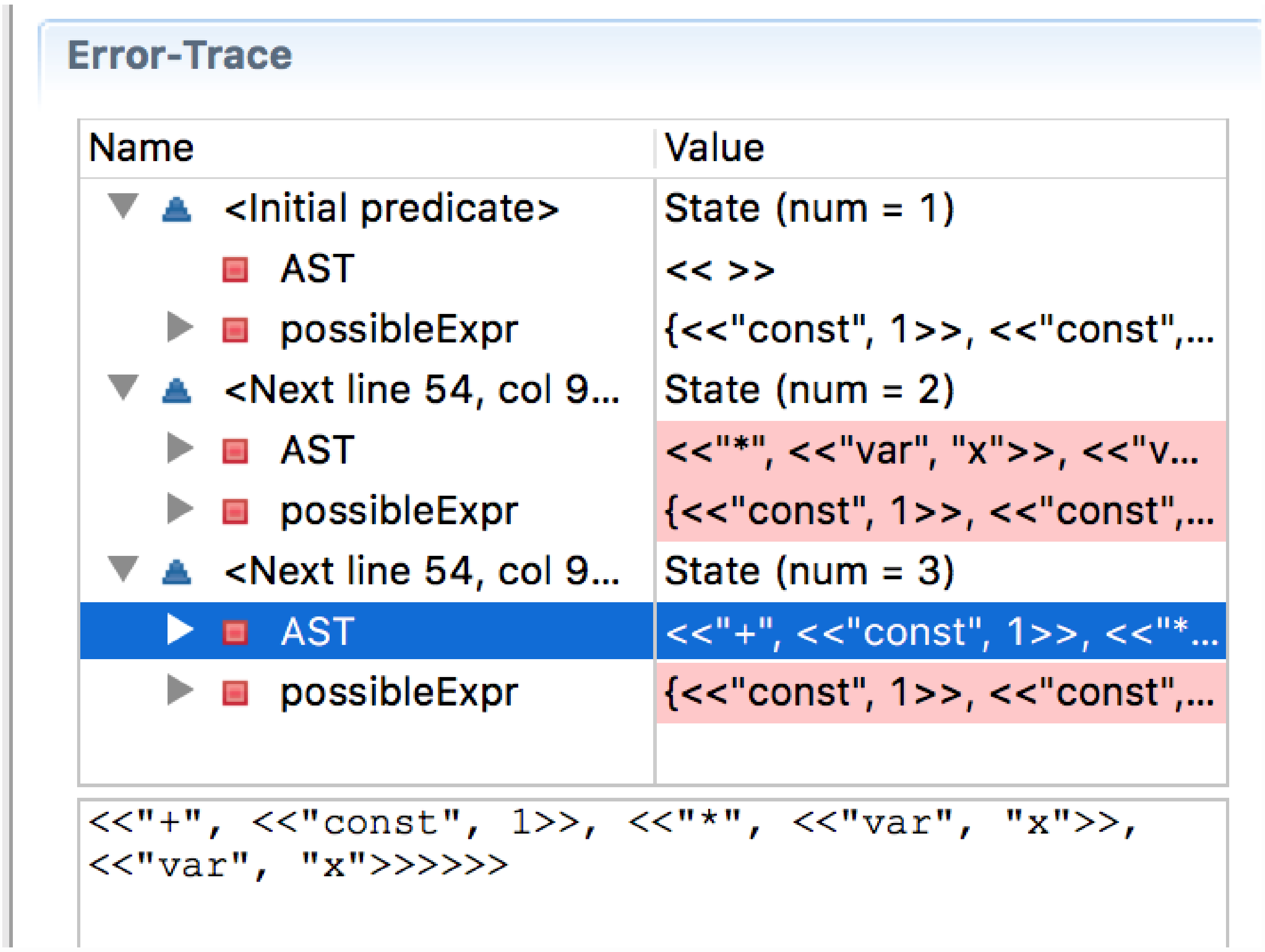
Name	Value
<Initial predicate>	State (num = 1)
AST	<< >>
possibleExpr	{<<"const", 1>>, <<"const",...
<Next line 54, col 9...	State (num = 2)
AST	<<"*", <<"const", 1>>, <<"c...
possibleExpr	{<<"const", 1>>, <<"const",...

```
<<"*", <<"const", 1>>, <<"const", 1>>>>
```

While this program technically evaluates to `1`, it's not exactly `x^2 + 1`. The problem is that we don't have enough test cases—we've underspecified. We need to add some refinements to our invariant:

```
Invariant == ~(
    /\ Len(AST) > 0
    /\ evaluate(AST, 0) = 1
    /\ evaluate(AST, 1) = 2
    /\ evaluate(AST, 2) = 5
)
```

Sure enough, now TLA+ finds `<<"+", <<"const", 1>>, <<"*", <<"var", "x">>, <<"var", "x">>>>>>`



Name	Value
▼ <Initial predicate>	State (num = 1)
AST	<< >>
▶ possibleExpr	{<<"const", 1>>, <<"const",...
▼ <Next line 54, col 9...	State (num = 2)
▶ AST	<<"*", <<"var", "x">>, <<"v...
▶ possibleExpr	{<<"const", 1>>, <<"const",...
▼ <Next line 54, col 9...	State (num = 3)
▶ AST	<<"+", <<"const", 1>>, <<"*...
▶ possibleExpr	{<<"const", 1>>, <<"const",...

```
<<"+", <<"const", 1>>, <<"*", <<"var", "x">>,
<<"var", "x">>>>>>
```

Future thoughts

You can think of programming as consisting of two parts: writing and evaluating. Evaluation is traditionally the province of computers, while the writing is the province of people.

But does it have to be this way? Usually, when I program, I'm trying to adhere to some specification, whether explicit or implicit. The creative part of a project is envisioning the finished product. Programming is often just the means to achieve it.

What if there were another way to program? What if all we had to do is imagine what we want and write down a specification to describe it? Where we could continually refine our specifications, as we did here, to hone the accuracy of our program? Why go through the laborious, often dull process of actually writing the program? What if a computer could just find it for us?

Thanks to Lucy Zhang and Emanuel Geromin for being awesome co-conspirators on this project.