Everything You Ever Wanted To Prove About Termination But Were Afraid To Try

June 19, 2014

Motivation

Does this program terminate?

```
while (x > 0) {
  x++;
}
```

Motivation

Does this program terminate?

```
while (x > 0) {
  x++;
}
```

That depends...
It *does not* terminate if the program operates on integers...
But it *does* terminate if the program operates on fixed-width words (-x decreases monotonically and is bounded by -MAXINT).

Motivation

Which of these guys terminates?

```
while (x > 0) {
  x = (x - 1) & x;
}
```

```
while (x > 0) {
  x = (x + 1) & x;
}
```

Motivation

Which of these guys terminates?

```
while (x > 0) {
  x = (x - 1) & x;
}
```

```
while (x > 0) {
  x = (x + 1) & x;
}
```

The left one does (x decreases monotonically), but the right one doesn't ($2 \to 2 \to \ldots$).

Motivation

Proving program termination has been a big deal for a while.
It'd be nice if we could automatically prove termination for programs that run on computers.

What Does a Termination Proof Look Like?

We prove that a program terminates by finding a *ranking function*.

If we have a transition relation T and a loop guard g, a ranking function R must meet the following criteria:

$$\forall x, x' \cdot g(x) \Rightarrow R(x) > 0 \wedge \tag{1}$$

$$T(x, x') \Rightarrow R(x) > R(x') \tag{2}$$

Or more generally, R must be an order-homomorphism with a well-founded co-domain (aka an ordinal).

We are therefore in the business of finding ranking functions.

Finding Ranking Functions

Traditionally, people find ranking functions under some assumptions:

- Programs are linear
- Ranking functions are linear
- Program variables are rationals

With these assumptions in hand, one can deploy a very powerful piece of linear algebra: *Farkas's lemma*.
This lemma makes light work of a ton of termination problems.

But it's not for us.

Synthesing Ranking Functions

Our approach is to use *program synthesis* to generate ranking functions.

We're going to set up a program synthesis problem that will allow us to solve the following 2nd-order satisfaction problem:

$$\exists R.\forall x, x' \cdot g(x) \Rightarrow R(x) > 0 \wedge \\ T(x, x') \Rightarrow R(x) > R(x')$$

Synthesing Ranking Functions

Our approach is to use *program synthesis* to generate ranking functions.

We're going to set up a program synthesis problem that will allow us to solve the following 2nd-order satisfaction problem:

$$\exists R.\forall x, x' \cdot g(x) \Rightarrow R(x) > 0 \wedge \\ T(x, x') \Rightarrow R(x) > R(x')$$

In other words: "Is there a ranking function for my program?"

Specifying a Ranking Function

Our program synthesiser needs a *specification*, which takes the form of a C function.

The specification function takes two arguments: a candidate program P and an input x. If P gives the correct output when fed x, our specification function returns true, otherwise it returns false.

A Specification

```
bool spec(prog_t *p, int x) {
  int y = exec(p, x);

  if (y >= x)
    return true;
  else
    return false;
}
```

A Specification

```
bool spec(prog_t *p, int x) {
  int y = exec(p, x);

  if (y >= x)
    return true;
  else
    return false;
}
```

Two of the possible programs:

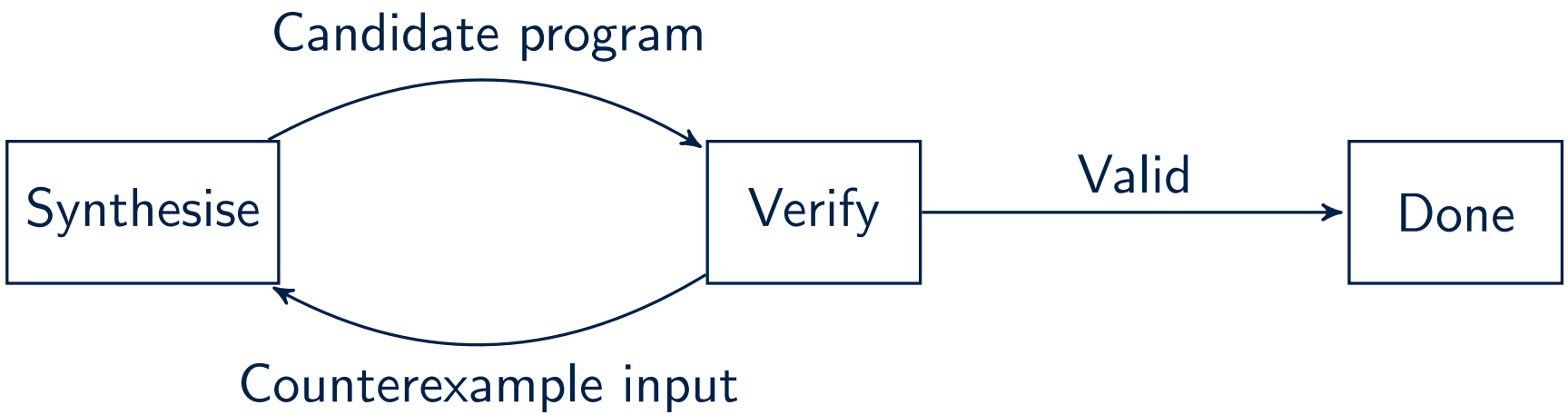
```
int f(int x) { return x; }
int f(int x) { return MAXINT; }
```

Specifying a Ranking Function

```
int spec(prog_t *p, int x) {
  int r1 = exec(p, x);

  if (g(x)) {
    if (r1 <= 0)
      return false;

    int y = body(x);
    int r2 = exec(p, y);

    if (r1 <= r2)
      return false;
  }

  return true;
}
```

Solving the Synthesis Formula with CEGIS

Solving second order formulae is hard. Instead we'll alternately solve two *first-order* formulae in a refinement loop:



First-order Synthesis

If we have some small set of test inputs $\{x_0, \ldots, x_N\}$, we can find a program that is correct on just that set:

$$\exists P.\sigma(x_0, P(x_0)) \wedge \cdots \wedge \sigma(x_N, P(x_N))$$

First-order Verification

If we have some program P that might be correct, we can check whether it is in fact correct:

$$\exists x. \neg \sigma(x, P(x))$$

If this formula is *satisfiable*, P is *not* correct and we have found an input on which it is incorrect. We can add this input to our set of test inputs and loop back to synthesising a new program.

Complexity

By far the dominant factor in how long synthesis takes is the length of the shortest correct program.

This is called the *Kolmogorov complexity* of the function being computed.

For termination, this gives us the property that our runtime doesn't depend very heavily on the size of the program we are analysing. It is instead almost entirely determined by the size of the shortest program that computes a valid ranking function.

Genetic Programming

The flip side of being so dominated by Kolmogorov complexity is that we get *really slow* beyond about 4 instructions.

It turns out that *genetic programming* helps a lot beyond this threshold.

Genetic Programming

Genetic programming is an evolutionary search strategy, which aims to find a program that is "fit" according to some metric.

It begins by generating a population of random programs. This population is subjected to *evolution* over a period of many generations.

At each generation, each individual's fitness is evaluated. The fitter an individual is, the greater its chance of breeding.
When two programs breed, their code is combined in some way (the crossover operation) and the resulting child may be randomly modified (the mutation operator). This child is copied into the next generation.

Genetic Programming Synthesis

To use genetic programming for synthesis, we need to fix a useful fitness metric.

For our purposes, we say that if a program meets the specification on n test vectors, its fitness is n.
Whenever we add a new test vector, we continue genetic programming with the most recent population (the one that produced the most recent candidate program). This is called *incremental evolution* and gives us a big speedup.

Once we've found a program that passes all the tests, we return it and continue round the CEGIS loop.

Oh, And One More Thing

Does this terminate? If so, what's the ranking function?

```
while (x > 0 && y > 0 && z > 0) {
  if (y > x) {
    y = z;
    x = nondet();
    z = x - 1;
  } else {
    z = z - 1;
    x = nondet();
    y = x - 1;
  }
}
```

Conditional Termination

$$\exists I, R \cdot \forall x, x' \cdot P(x, x') \Rightarrow \qquad I(x')$$

$$I(x) \wedge g(x) \wedge T(x, x') \Rightarrow \qquad R(x) > 0 \wedge$$

$$R(x) > R(x') \wedge$$

$$I(x')$$

Sequential Loops

$$
\begin{aligned}
\exists I_1, I_2, R \cdot \forall x, x' \cdot P(x, x') \Rightarrow & \quad I_1(x') \\
I_1(x) \wedge g_1(x) \wedge T_1(x, x') \Rightarrow & \quad I_1(x') \\
I_1(x) \wedge \neg g_1(x) \Rightarrow & \quad I_2(x) \\
I_2(x) \wedge g_2(x) \wedge T_2(x, x') \Rightarrow & \quad R(x) > 0 \wedge \\
& \quad R(x) > R(x') \wedge \\
& \quad I_2(x')
\end{aligned}
$$

Some Combinatorics

A terminating program corresponds to a *labelled partial order* on 2^k states.

Counting partial orders on a finite set turns out to be very hard (it's an open problem!). However, asymptotics are known:

$$P_n = \left(1 + O\left(\frac{1}{n}\right)\right)\left(\sum_{i=1}^{n}\sum_{j=1}^{n-1}\binom{n}{i}\binom{n-i}{j}(2^i-1)^j(2^j-1)^{n-i-j}\right)$$
$$\sim O\left(2^k\right)$$

More Combinatorics

Linear functions on bitvectors correspond to *permutations* of the natural ordering on bitvectors.

There are 2^{2k} linear functions (although only half of these permute the full space, cf. generators of the group 2^k).

Each of these functions can show that each of the *unlabelled* partial orders terminates.
The number of unlabelled partial orders over an n element set is related to the number of labelled partial orders as:

$$p_n \sim \frac{P_n}{n!}$$

The Combinatorial Payoff

Since there are P_n terminating functions, and each of the n^2 linear functions can prove that $p_n \sim \frac{P_n}{n!}$ functions terminate, we can compute the probability that a random terminating function can be proved to terminate with a linear argument:

$$\frac{2^k \times 2^k}{2^k!} \sim \frac{1}{2^k!}$$

The Combinatorial Payoff

Since there are P_n terminating functions, and each of the n^2 linear functions can prove that $p_n \sim \frac{P_n}{n!}$ functions terminate, we can compute the probability that a random terminating function can be proved to terminate with a linear argument:

$$\frac{2^k \times 2^k}{2^k!} \sim \frac{1}{2^k!}$$

This number is very small, which to some degree justifies our aim of moving beyond linear ranking functions.

Fin

Thank you!