Test Case Generation in Prolog

1 Background: Software Testing

In practice, testing is of the upmost importance in software development, with significant amounts of code, time, and effort being devoted just to testing. For example, consider the JavaScript engines in Chrome and Firefox, which are responsible for executing JavaScript code in the browser. For both codebases, approximately half the code is devoted just to testing, which translates to several hundred thousand lines of testing code.

While this amount of testing code is impressive, these codebases are still subject to bugs. Developers aren't perfect, and they can miss important behaviors with the tests they write. As a codebase grows, later code added may invalidate the assumptions made in code written at an earlier point. People writing tests might focus on things that they intuitively think are hard to get write, while glossing over "simple" cases. In short, writing tests can be difficult, and the definition of what it means to be a "good" test can change over time.

For these reasons, various machine-aided testing techniques exist which seek to produce test cases through some automated process. Such techniques are usually employed in conjunction with normal human testing efforts, leading to overall more thorough testing. For example, Mozilla (the creator of Firefox) has developed several automated tools to produce tests for its JavaScript engine, which are run continuously in parallel with normal development efforts. These tools have collectively found literally thousands of bugs missed by normal testing efforts.

2 Enter Prolog

So what does any of this have to do with Prolog? It turns out that Prolog can be used to produce test cases with relative ease, at least compared to a number of competing techniques. What makes Prolog a good fit for this problem is that Prolog allows us to say *what* a valid test case is, as opposed to *how* to generate it. This leads to test case generators which are fairly concise, yet very expressive. That is, we can give very specific definitions regarding what "valid" means for a given problem.

2.1 Example: Grammar-Based Testing

To see this sort of testing in practice, let's consider the problem of testing an evaluator of Boolean expressions. To test such a program, we need to generate syntactially valid Boolean expressions. For our purposes, we'll use the following grammar for Boolean expressions:

$$e \in \mathit{BooleanExpression} ::= \textsf{true} \mid \textsf{false} \mid \textsf{and}(e_1, e_2) \mid \textsf{or}(e_1, e_2) \mid \textsf{not}(e)$$

In this example, valid tests are comprised of syntactically-valid sentences according to the above grammar. With this in mind, we'll write a definition of what it means to be syntactically-valid in Prolog (named `naive_generator.pl` in the course resources):

```
booleanExpression(true).
booleanExpression(false).
booleanExpression(and(E1, E2)) :-
    booleanExpression(E1),
    booleanExpression(E2).
booleanExpression(or(E1, E2)) :-
    booleanExpression(E1),
    booleanExpression(E2).
booleanExpression(not(E)) :-
    booleanExpression(E).
```

In plain English, the above code states the following:

- `true` is a Boolean expression (line 1)
- `false` is a Boolean expression (line 2)
- `and(E1, E2)` is a Boolean expression (line 3), as long as `E1` is a Boolean expression (line 4) and `E2` is a Boolean expression (line 5)
- `or(E1, E2)` is a Boolean expression (line 6), as long as `E1` is a Boolean expression (line 7) and `E2` is a Boolean expression (line 8)
- `not(E)` is a Boolean expression (line 9) as long as `E` is a Boolean expression (line 10)

As one might expect, we can use the above code to check to see if a given input is syntactically valid:

```
?- booleanExpression(and(true, false)).
true.
?- booleanExpression(or(true, not_a_boolean)).
false.
```

We can also use the above code to *generate* valid Boolean expressions, just by changing the query. In the example below, we keep pressing semicolon (`;`) to generate further solutions:

```
?- booleanExpression(E).
E = true ;
E = false ;
E = and(true, true) ;
E = and(true, false) ;
...
```

Each one of the above answers is a valid test case, which can be passed to the program under test with some additional work.

2.2 Generating All Expressions

The output from the previous query was intentionally cut-off a bit early. If we ask for more answers, an inconvenient pattern emerges:

```
E = and(true, and(true, true)) ;
E = and(true, and(true, false)) ;
E = and(true, and(true, and(true, true))) ;
E = and(true, and(true, and(true, false))) ;
E = and(true, and(true, and(true, and(true, true)))) ;
...
```

As shown above, the tests generated end up being ever more deeply-nested versions of `and`, and only the right subexpression of `and` is ever expanded. We never see expansion on the left subexpression of `and`, and we will never see an `or` or `not` AST node.

All of these aforementioned problems are artifcats of Prolog's depth-first search strategy; that is, the order in which Prolog looks for solutions. Because `and` is present in the file before `or` and `not`, Prolog always selects `and` first. Additionally, when we ask for an additional solution when `and` is involved, the last call made is always going to be the second call to `booleanExpression` in the rule for `and` (line 5). As such, it will always be this particular call for which we will choose alternative solutions, ultimately leading to ever more deeply nested `and` AST nodes on the right subexpression (`E2`).

A simple fix for this problem is to add a *bound* to the problem, which effectively constrains the depth of valid ASTs. The general idea here is that we force failure to occur if we start generating an AST which is too deep. This forced failure will eventually stop us from being able to produce more deeply nested `and` AST nodes, permitting us to expand upon alterntaive calls to `booleanExpression`.

Complete code showing this added bound is below (named `bounded_generator.pl` in the course resources):

```
 1  decBound(In, Out) :-
 2      In > 0,
 3      Out is In - 1.
 4
 5  boundedExpression(_, true).
 6  boundedExpression(_, false).
 7  boundedExpression(B1, and(E1, E2)) :-
 8      decBound(B1, B2),
 9      boundedExpression(B2, E1),
10      boundedExpression(B2, E2).
11  boundedExpression(B1, or(E1, E2)) :-
12      decBound(B1, B2),
13      boundedExpression(B2, E1),
14      boundedExpression(B2, E2).
15  boundedExpression(B1, not(E)) :-
16      decBound(B1, B2),
17      boundedExpression(B2, E).
```

Central to the above code is the addition of a `decBound` helper procedure, which takes an input number `In` and produces an output number `Out` (line 1). This checks if the input number is greater than `0` (line 2), failing if so. If this check fails, this means that the bound is exceeded, and is ultimately the source of this forced failure. If failure does not occur here (that is, the bound has *not* been exceeded), then the bound is decremented (line 3), putting the result of the decrementation in `Out`.

The `booleanExpression` procedure has been renamed to `boundedExpression`, reflecting the augmentation of a bound. The first parameter to `boundedExpression` is the current bound, and the second parameter is the AST from before. The bulk of `boundedExpression` is the same as with `booleanExpression`, but now case has been taken to call `decBound` with the current bound whenever a recursive call to `boundedExpression` is to be made. This makes sense because the depth of the AST increases at every recursive call.

An example query is shown below, which asks for valid ASTs which are at most one node deep. Note that leaf nodes do not contribute to this depth, reflected in the fact that the base cases for `boundedExpression` (lines 5-6) do not call `decBound`, and in fact, ignore the bound parameter (using `_` for this position).

```
?- boundedExpression(1, E).
E = true ;
E = false ;
E = and(true, true) ;
E = and(true, false) ;
E = and(false, true) ;
E = and(false, false) ;
E = or(true, true) ;
E = or(true, false) ;
E = or(false, true) ;
E = or(false, false) ;
E = not(true) ;
E = not(false) .
```