Verifying Arithmetic Coding with Boogie

Darren Smith and Yunus Basagalar

Overview

We implemented and verified an arithmetic coding algorithm, using Boogie

- Arithmetic Coding
- Boogie
- Verification

Arithmetic Coding

Arithmetic Coding Outline

- What is it?
- Demo
- How it works
- What to prove about it

What is it?

- Variable-length entropy encoding used in lossless data compression
- Uses a model which is separate from the algorithm to make predictions
- Similar to Huffman encoding, but...
  - uses fractional bits
  - typically dynamic
  - slower

What is it? (cont)

- PAQ
  - won the Hutter prize
- Supported by jpeg but patent issues
  - video compression
- Difficult to implement, bugs mean your file is completely corrupt

Our Ruby Implementation

- Simplified
  - BigInt use
  - Simplified the algorithm
- predictor / encoder / decoder

coder.rb

```
string.each_byte{|c|
    ps=predictor.get_ranges()
    predictor.set_next_symbol(c)
    lo+=r*ps[c]/ps[256]
    r=r*ps[c+1]/ps[256]-r*ps[c]/ps[256]
    while r<N
        r*=2
        lo*=2
        nbits+=1
    end
}
```

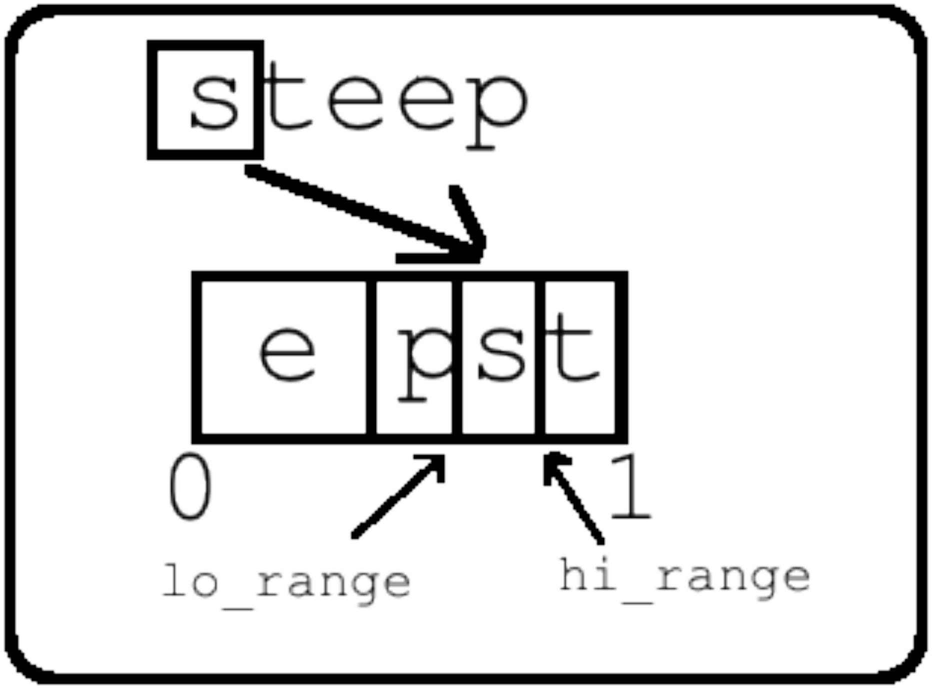
Ruby Implementation Example Use

Terminal — tcsh — 69×17 —

```
[~/src/630-Project/simplecoder]:121% ls
coder.rb        elias_gamma.rb predictor.rb   util.rb
decode          encode         test.txt
[~/src/630-Project/simplecoder]:122% head -3 test.txt
On the first Monday of the month of April, 1625, the market town
of Meung, in which the author of ROMANCE OF THE ROSE was born,
appeared to be in as perfect a state of revolution as if the
[~/src/630-Project/simplecoder]:123% encode test.txt > test.bin
[~/src/630-Project/simplecoder]:124% decode test.bin > decoded.txt
[~/src/630-Project/simplecoder]:125% diff test.txt decoded.txt
[~/src/630-Project/simplecoder]:126% tar -czf test.tgz test.txt
[~/src/630-Project/simplecoder]:127% wc -c test.*
    4178 test.bin
    4458 test.tgz
    9203 test.txt
   17839 total
[~/src/630-Project/simplecoder]:128%
```

How it Works

- Many ways to think about it (avoid patents)
  - range, fraction, reversed, variable radix
- Simplest explanation



How it Works

- To use integers, use a power of 2 for the denominator
- Each symbol can be encoded by any number within a range
- The next symbol can be encoded by any number within the range of the first symbol
- The size of the range is proportional to the probability of that symbol being next
- Good because more likely symbols reduce the range less

How it Works (cont)

- If all symbols equally likely it would be same size
- Probabilities can be stored as a header or calculated dynamically (PAQ uses neural net, ours is simpler)
- How to avoid using infinite precision
- How to encode length
- encode(str,r)=lo(str[0],r)+encode(str[1..],hi-lo)

What to Prove

- Encoding is correct
- Optimal length
- Termination
- Other
    - Probability prediction (static/dynamic)
    - Length
    - Byte conversion
    - Bijective
    - No bigint use

Boogie

Boogie Outline

- Introduction to Boogie
- Why Boogie?
- Boogie with examples
- Common pitfalls

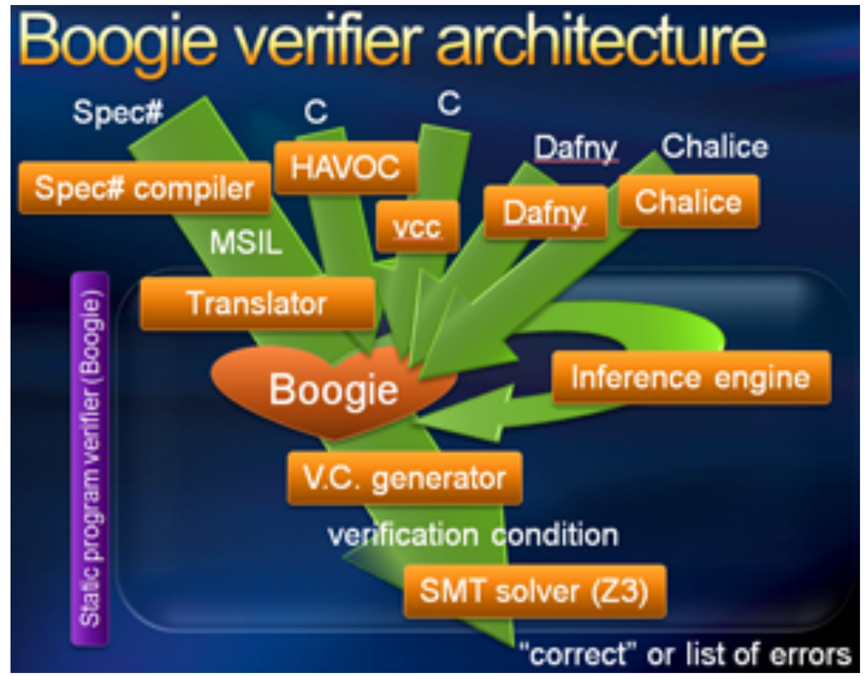
Introduction

- An intermediate verification language developed by Microsoft
  - http://research.microsoft.com/en-us/projects/boogie/

- A layer upon which verifiers can be built for other languages
  - e.g. VCC, HAVOC verification tools for C

Architecture

- The highest-level is a program
    - written with languages like C, Dafny or Boogie
- Program is translated into Boogie by corresponding compiler.
- Verification conditions are generated out of Boogie program
- These conditions are fed into a SMT solver.
- Solver proves or disproves the conditions



SMT Solver

- SMT stands for *Satisfiability Modulo Theories*
- Solves SMT problem

SMT Problem

- It is a decision problem.
- **Given:** A formula in first-order logic
  - where functions and predicate symbols interpreted with respect to background theories
- **Determine:** If the given formula is satisfiable
- **Example:** $x < 1$ and $y < 1$ and $3x + 2y > 0$
  - interpretation with respect to theory of linear real arithmetic
  - interpretation with respect to theory of linear integer arithmetic

Why Boogie?

- Based on Hoare Logic
  - Enable us to deploy what we learn in class
- Intuitive syntax
  - Ease of learning
- Z3 backend
  - A powerful SMT solver developed by Microsoft
    - http://research.microsoft.com/en-us/um/redmond/projects/z3/
  - Supports linear and nonlinear arithmetic, arrays, uninterpreted functions, quantifiers, bitvectors, datatypes

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

- Defining a new type T

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

- res is a global variable of integer type.
- m is a global constant of array type.

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

- Functions do *not* need definitions.
- Their properties can be specified with axioms.
- Only constants and functions can be used in axioms.

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

- Procedures do need implementations
- *requires* specify preconditions
- *ensures* specify postconditions
- *modifies* declares which global variables will be modified
- Procedures cannot be used in postconditions, preconditions, axioms, invariants

Overall Structure of a Boogie Program

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

- Gives implementation of an already declared procedure

A sample run

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

To verify the program with Boogie we just need to type:

./Boogie.exe example.bpl

And in this case, we will have approval of verification by Boogie

A sample run

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a < result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

To verify the program with Boogie we just need to type:

./Boogie.exe example.bpl

And in this case, we will have approval of verification by Boogie

1. b is negative because of precondition of the procedure.
2. Therefore, by using axiom we know that f(c, b) returns a positive integer.

A sample run

example.bpl

```
type T;

var result: int;
const m: [int]bool;

function f(x: T, y: int) returns(int);

axiom( forall x: T, y: int :: f(x, y) > 0 <==> y < 0 );

procedure test(a: int, b: int);
  requires b < 0;
  modifies result;
  ensures  a > result;

implementation test(a: int, b: int) {
  var c: T;

  result := a + f(c, b);
}
```

To verify the program with Boogie we just need to type:

./Boogie.exe example.bpl

And in this case, Boogie says postcondition in red does not hold.

Finding minimum with Boogie

minimum.bpl

```
const N: int;
axiom( N > 0 );
procedure findMin(arr: [int]int) returns(min: int)
  ensures ( forall i: int :: (0 <= i && i < N) ==> min <= arr[i] );
  ensures ( exists i: int :: 0 <= i && i < N && arr[i] == min );
{
  var n: int;
  min := arr[0];
  n := 1;

 while(n < N)
  invariant( n <= N );
  invariant( forall i: int :: (0 <= i && i < n) ==> min <= arr[i] );
  invariant( exists i: int :: 0 <= i && i < n && arr[i] == min );
  {
     if( arr[n] < min ) {
        min := arr[n];
     }
     n := n + 1;
  }
}
```

- First invariant is important in the verification of second postcondition

No division/modulo operation

division.bpl

```
procedure testDivision(a: int)
  requires( a > 1 );
  ensures( 1 / a <= 0 );
{ }
```

No division/modulo operation

division.bpl

```
procedure testDivision(a: int)
  requires( a > 1 );
  ensures( 1 / a <= 0 );
{ }
```

This postcondition wouldn't hold because division operator is just a syntactic sugar.

- Different programming languages have different semantics for division.
  - Ruby vs. C

No division/modulo operation

division.bpl

```
procedure testDivision(a: int)
  requires( a > 1 );
  ensures ( 1 / a <= 0 );
{ }
```

Use builtin division

Write axioms yourself

division.bpl

```
function {:builtin "div"} div(a: int, b: int) returns(int);

axiom( forall a, b: int :: div(a, b) == a / b );

procedure testDivision(a: int)
  requires( a > 1 );
  ensures( 1 / a <= 0);
{ }
```

division.bpl

```
axiom( forall x: int :: x > 1 ==> 1 / x <= 0 );

procedure testDivision(a: int)
  requires( a > 1 );
  ensures( 1 / a <= 0);
{ }
```

No subtypes and contradictions in axioms

posneg.bpl

```
type Pos = [int]int;
type Neg = [int]int;

axiom (forall i: int, A: Pos :: A[i] > 0);
axiom (forall i: int, A: Neg :: A[i] < 0);

procedure test (a: Pos, b: Neg)
  ensures a[0] > a[0] + b[0];

{ }
```

- Intuitive approach for creating subtypes
  - Pos stands for positive integers
  - Neg stands for negative integers

No subtypes and contradictions in axioms

posneg.bpl

```
type Pos = [int]int;
type Neg = [int]int;

axiom (forall i: int, A: Pos :: A[i] > 0);
axiom (forall i: int, A: Neg :: A[i] < 0);

procedure test (a: Pos, b: Neg)
  ensures a[0] > a[0] + b[0];
  ensures 1 == 2;
{ }
```

- What about postcondition with red?

No subtypes and contradictions in axioms

posneg.bpl

```
type Pos = [int]int;
type Neg = [int]int;

axiom (forall i: int, A: Pos :: A[i] > 0);
axiom (forall i: int, A: Neg :: A[i] < 0);

procedure test (a: Pos, b: Neg)
  ensures a[0] > a[0] + b[0];
  ensures 1 == 2;
{ }
```

```
axiom (forall i: int, A: [int]int :: A[i] > 0);
axiom (forall i: int, A: [int]int :: A[i] < 0);
```

```
axiom ( false );
```

- Cannot create subtypes in Boogie
- Trying so leads to a contradiction in axioms
- As a result, we can prove anything even if they are wrong

Triggers

- Many theories are decidable without quantifiers
  - e.g. theory of linear arithmetic, arrays
- However, interesting problems require quantifiers
  - This makes the problem either very slow to decide or undecidable
    - e.g. linear arithmetic and arrays with quantifiers is undecidable *(Hawblitzel and Petrank, 2009)*
- *Triggers* are the mechanism to help SMT solvers to strategically instantiate quantifiers
- Coming up with good triggers is very important but not trivial
  - with a bad trigger, you may not prove what you want even if it is correct
  - with a bad trigger, there might be too many instantiations

Triggers

bad_trigger.bpl

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom(forall x: int :: {h(x)} f(x) > 0);

procedure test(x: int)
  ensures f(x) > 0;
{ }
```

good_trigger.bpl

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom(forall x: int :: {f(x)} f(x) > 0);

procedure test(x: int)
  ensures f(x) > 0;
{ }
```

- Code pieces with red are triggers

Triggers

bad_trigger.bpl

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom(forall x: int :: {h(x)} f(x) > 0);

procedure test(x: int)
  ensures f(x) > 0;
{ }
```

- This axiom is triggered only when we encounter h(X) in the proof for some X
- Since we do not need it, we cannot use the fact that f is a positive function

good_trigger.bpl

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom(forall x: int :: {f(x)} f(x) > 0);

procedure test(x: int)
  ensures f(x) > 0;
{ }
```

Matching Loops

matching_loop.bpl (*Rustan et al. 2009)*

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom( forall x: int :: {h(x)} h(x) < h(f(x)) );
...
```

- Let us assume we encounter with h(42) somewhere in the proof
- This triggers the axiom with instantiation x = 42
  - We need to check h(42) < h(f(42))
- h(f(42)) also triggers the axiom with instantiation x = f(42)
  - We need to check h(f(42)) < h(f(f(42)))
- And so on...
- This goes into a *matching loop* and hangs forever

Matching Loops

matching_loop.bpl (*Rustan et al. 2009)*

```
function f(x: int) returns(int);
function h(x: int) returns(int);

axiom( forall x: int :: {h(f(x))} h(x) < h(f(x))
);
...
```

- A more constraining trigger would break the matching loop

Arithmetic Coding Verification

Verification Outline

- Basic idea
- Structure of proof
- Code for procedures
- Other verifications
    - termination
    - size

Basic Idea

- `encode` finds range for next symbol
  - recursively calls itself to get encoding of rest
  - adds that to beginning of range
- `decode` finds what symbol has range including x
  - recursively calls itself using the range of the symbol
- `encodef` is like a recursive invariant
  - `x/r` is the arithmetic coding of the rest of the string

Structure

global
math axioms
lemmas
lo/hi_range
lookup
encodef
encode/decode
main

Overview

simrec2.bpl

```
const nsyms: int;
const len: int;
const in: [int]int;
axiom nsyms>2;
axiom len>=0;
axiom (forall i: int :: {in[i]} i>=0 &&
    i<len ==> in[i]>=0 && in[i]<nsyms);

var out: [int]int;
```

- Use global variables
  - `in` is the input string
  - `out` is the decode result

High Level Goal

simrec2.bpl

```
procedure main() modifies out; {
    var x, range: int;

    call x, range := encode();
    call decode(x, range);
    assert (forall i: int :: i>=0 && i<len ==> out[i]==in[i]);
}
```

- Satisfying this assert would achieve our goal

Encode and Decode

simrec2.bpl

```
procedure encode() returns (x: int, range: int)
ensures x>=0 && x<range;
ensures x == encodef(0, range);
```

simrec2.bpl

```
procedure decode(x: int, range: int)
modifies out;
requires x>=0 && x<range;
requires len>=0;
requires range>0;
ensures x==encodef(0, range) ==> (forall i: int ::
    i>=0 && i<len ==> out[i]==in[i]);
```

- Encode and Decode only need to satisfy these properties

Recursive Invariant, encodef

simrec2.bpl

```
function encodef(ind: int, range: int) returns (int);
axiom (forall i,r: int :: {encodef(i, r)}
    i>=len ==> encodef(i, r) == 0);
axiom (forall i,r: int :: {encodef(i, r)}
    i>=0 && i<len ==> encodef(i, r) == lo_range(in[i], r)+encodef(i+1,
    hi_range(in[i], r)-lo_range(in[i], r)));
```

- This is the recursive definition of arithmetic coding

Encode Implementation

simrec2.bpl

```
implementation encode() returns (x: int, range: int)
{
    var fail: bool;
    range := 1;
    while (true)
    {
        call x, fail := encode_helper(0, range);
        if (!fail) { return; }
        range := range*2;
    }
}
```

Encode_helper

simrec2.bpl

```
procedure encode_helper(ind: int, range: int) returns (x: int, fail:
bool)
requires len>=0;
requires ind>=0 && ind<=len;
requires (forall i: int :: i>=0 && i<len ==> in[i]>=0 && in[i]<nsyms);
ensures (x>=0 && x<range) || fail;
ensures ind>=len ==> x==0;
ensures x == encodef(ind, range) || fail;
{
    var c, lo, hi: int;
    call range_bound_lemma();
    if (range<=0) { x, fail := 0, true; return; }
    if (ind>=len) { x, fail := 0, false; return; }
    c := in[ind];
    lo := lo_range(c, range);
    hi := hi_range(c, range);
    call x, fail := encode_helper(ind+1, hi-lo);
    x := x+lo;
}
```

Decode_helper

simrec2.bpl

```
procedure decode_helper(ind: int, range: int, x: int)
modifies out;
requires x>=0 && x<range;
requires len>=0;
requires ind>=0 && ind<=len;
requires range>0;
ensures x==encodef(ind, range) ==> (forall i: int :: i>=ind && i<len ==>
out[i]==in[i]);
{
    var c, lo, hi: int;
    call encodef_bound_lemma();
    if (ind>=len) { return; }
    call c := lookup(x, range);
    lo := lo_range(c, range);
    hi := hi_range(c, range);
    call decode_helper(ind+1, hi-lo, x-lo);
    out[ind] := c;
}
```

Lookup

simrec2.bpl

```
procedure lookup(x: int, range: int) returns(y: int)
requires x>=0 && x<range;
ensures x>=lo_range(y, range) && x<hi_range(y, range);
ensures (forall yy: int :: yy != y ==> x<lo_range(yy, range) ||
x>=hi_range(yy, range));
{
    var hi: int;
    call range_bound_lemma();
    call range_order_lemma();
    y := 0;
    while (true)
    invariant x>=lo_range(y, range);
    invariant y>=0 && y<nsyms;
    {
        hi := hi_range(y, range);
        assert(y==nsyms-1 ==> hi>x); //help it figure out invariant
        if (hi>x) { break; }
        y := y+1;
    }
}
```

Size of Encoding (manual proof)

If probabilities are accurate then the amount of information in a string is:

$$\sum_{i=string} -log_2(p(i))$$

In range encoding, final range>0

$$r \leftarrow \lfloor ps(i+1) * r \rfloor - \lfloor ps(i) * r \rfloor$$
$$r \geq \lfloor p(i) * r \rfloor$$
$$r * \prod_{i=string} p(i) > 1$$
$$bits_r = \lceil -log_2 \prod_{i=string} p(i) \rceil$$

Termination (manual proof)

- While loop in lookup
  - already ensures that 0 <= c < nsyms
- While loop in encode
  - already proven that range is minimal (finite length)
- Recursive encode/decode calls
  - executes only once per each character

Other Things to Prove

- Probability prediction (static/dynamic)
- Length
- Byte conversion
- Bijective
- No bigint use

Final Thoughts

- Arithmetic coding is pretty cool
- Boogie is simple yet powerful tool for verifying programs in a Hoare-like language

http://rise4fun.com/Boogie/

- We simplified an arithmetic coding algorithm, then proved it correct using Boogie

https://github.com/darrenks/630-Project

Questions?

References

Boogie papers

http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

http://research.microsoft.com/en-us/um/people/leino/papers/krml160.pdf

http://research.microsoft.com/en-us/um/people/leino/papers/krml186.pdf

Boogie on linux tutorial

http://www.zvonimir.info/2010/12/a-tutorial-for-running-boogie-and-z3-on-linux/

Reasoning about Comprehensions

http://www.cs.nuim.ie/research/pop/papers/rmkrml-sac09.pdf

Try Boogie Online

http://rise4fun.com/Boogie

Z3

http://rise4fun.com/Z3/tutorial/guide

Arithmetic Coding

http://michael.dipperstein.com/arithmetic/

http://www3.sympatico.ca/mt0000/biacode/biacode.html

Extra Slides

A state-of-the-art example...

- Verve developed by Microsoft
  - http://research.microsoft.com/apps/pubs/?id=122884

Verve
- It is a dependable microkernel operating system   **Motivation:**   - Type-safe languages prevent most common bugs   - However, underlying run-time systems are still susceptible to bugs   - This also makes higher level vulnerable   **Solution:** Bring type-safety to the bottom of software stack as much as possible   **Contribution:** Automatic verification thanks to Boogie/Z3   - seL4 (another verified microkernel) required 20-person years of research to prove it interactively with Isabelle/HOL

Variable versioning and boring invariants

boring_loop.bpl

```
procedure boring_loop()
{
  var A: [int]int;
  var i: int;

  A[0], i := 0, 1;

  while ( i < 5 )

  {
     A[i] := i;
     i    := i + 1;
  }

  assert(A[0] == 0);
}
```

- Loop does not touch A[0] but assertion fails. Why?

Variable versioning and boring invariants

boring_loop.bpl

```
procedure boring_loop()
{
  var A: [int]int;
  var i: int;

  A[0], i := 0, 1;

  while ( i < 5 )

  {
     A[i]  := i;
     i     := i + 1;
  }

  assert(A[0] == 0);
}
```

```
assume A@0 == A[0 := 0];
```

```
assume A@2 == A@1[ i@0 := i@0 ];
```

```
assert A@1[0] == 0;
```

- Loop does not touch A[0] but assertion still fails. Why?
- Since Boogie implicitly versions variables

Variable versioning and boring invariants

boring_loop.bpl

```
procedure boring_loop()
{
  var A: [int]int;
  var i: int;

  A[0], i := 0, 1;

  while ( i < 5 )
  invariant( A[0] == 0 );
  {
     A[i] := i;
     i    := i + 1;
  }

  assert(A[0] == 0);
}
```

```
assume A@0 == A[0 := 0];

assume A@1[0] == 0;

assume A@2 == A@1[ i@0 := i@0 ];

assert A@1[0] == 0;
```

- If invariant is explicitly written, assertion holds.
- Boring invariants!

No proof visualisation

- Non-trivial programs may require numerous invariants, axioms, preconditions, postconditions etc.
- If proof does not succeed, Boogie only tells:
  - which preconditions, postconditions and/or invariants doesn't hold
- However, it does *not* say how it reaches that conclusion.
- And you are dead in the water

Lemmas

simrec2.bpl

```
procedure range_bound_lemma()
ensures (forall r: int :: lo_range(0, r) == 0);
ensures (forall c, r: int ::
    c>=0 && c<nsyms && r>0 ==> lo_range(c, r) >= 0);
ensures (forall c, r: int ::
    c>=0 && c<nsyms && r>0 ==> hi_range(c, r) <= r);
ensures (forall r: int :: hi_range(nsyms-1, r) == r);

procedure range_order_lemma()
ensures (forall i,j,r: int ::
    i<=j ==> lo_range(i,r) <= lo_range(j,r));

procedure encodef_bound_lemma()
ensures (forall i,r: int :: r>0 ==>
    encodef(i, r)>=lo_range(in[i], r));
ensures (forall i,r: int :: r>0 ==>
    encodef(i, r)<hi_range(in[i], r));
```

Simrec2.bpl Verification

Terminal — tcsh — 80×39 — ⌘3

```
[~/src/630-Project]:929% boogie simrec2.bpl /trace
Boogie program verifier version 2.2.40408.0708, Copyright (c) 2003-2011, Microso
ft.
Parsing simrec2.bpl
Coalescing blocks...

Running abstract interpretation...
  [0.158585 s]

Verifying range_bound_lemma ...
[TRACE] Using prover: /Users/darren/src/630-Project/boogie/z3.exe
  [0.498663 s, 4 proof obligations]  verified

Verifying range_order_lemma ...
  [0.012332 s, 1 proof obligation]  verified

Verifying lookup ...
  [0.062411 s, 7 proof obligations]  verified

Verifying encodef_bound_lemma ...
  [0.024478 s, 2 proof obligations]  verified

Verifying encode_helper ...
  [0.061745 s, 6 proof obligations]  verified

Verifying decode_helper ...
  [0.195144 s, 6 proof obligations]  verified

Verifying encode ...
  [0.032742 s, 5 proof obligations]  verified

Verifying decode ...
  [0.030509 s, 5 proof obligations]  verified

Verifying main ...
  [0.024581 s, 4 proof obligations]  verified

Boogie program verifier finished with 9 verified, 0 errors
[~/src/630-Project]:930%
```

Boogie Online Tool