Interpreter Project Wrap-up

November 27, 2018

Metacircularity

A metacircular interpreter is one that can interpret itself.

In order to make our interpreter metacircular, we would have to make some changes.

What language features do we use in defining our interpreter that we haven’t implemented?

Metacircularity

What language features do we use in defining our interpreter that we haven't implemented?

- Letrec
- (define (foo x)) short-cut
- Match (this is hard!)

Metacircularity

We can start by testing our interpreter with other, less complex programs.

We can now run most of HW2 in our interpreter, for instance!

Verification Languages

November 27, 2018

Introducing Dafny

Dafny is a programming language that supports **program verification**.

A program in Dafny has two components:

- The implementation of your algorithm, and
- A **formal specification** that describes what it means for the implementation to be correct.

The Dafny compiler checks that the implementation matches the specification for all possible inputs and outputs.

Dafny demo

I'm using an online interface to Dafny:
https://rise4fun.com/dafny

You can also install the language and run it locally if you want.

Example: the *Midpoint* method

Things to note:

- Dafny calls functions *methods*.
- Dafny has types.
- Dafny lets you name the return value.

```
method Midpoint(m : int, n :
int) returns (r : int)
{
  var delta := (n - m) / 2;
  return m + delta;
}

method Main()
{
  var r := Midpoint(1, 5);
  print r;
  print "\n";
}
```

What does it mean for *Midpoint* to be correct?

The result (*r*) should lie between *m* and *n*.

We can specify this using an *ensures* clause.

```
method Midpoint(m : int,
n : int) returns (r : int)
ensures r > m && r < n;
{
  var delta := (n - m) / 2;
  return m + delta;
}
```

The Dafny compiler signals an error

Dafny says the postcondition *may* not hold.

Dafny is not certain that the ensures clause is wrong. Instead, Dafny cannot show that it is not wrong.

So, we need to figure out if there is a genuine problem and help Dafny figure this out.

Two ways to address the problem

The code assumes that `m < n`. But, if `m > n` then delta will be negative and the result will be less than n.

We can fix this in two ways:

- We can introduce an if statement to deal with the case where `m > n`.
- We can promise Dafny that we will never call Midpoint with `m > n`.

A precondition for *Midpoint*

The requires clause specifies a precondition.

Dafny verifies that whenever some other method calls `Midpoint`, the caller will satisfy the precondition.

This should work!

```
method Midpoint(m : int,
n : int)
returns (r : int)
requires m < n;
ensures r > m && r < n;
{
  var delta := (n - m) / 2;
  return m + delta;
}
```

The Dafny compiler signals the same error

Dafny says the postcondition may not hold despite the precondition.

Unfortunately, it is exactly the same error message.

To get a little more insight, we can split the postcondition into two postconditions.

Multiple post-conditions

To get a little more insight, we can split the *ensures* clause into two clauses.

The semantics is exactly the same, but we get a more informative error.

Dafny was able to verify one postcondition:
`r < n`.
But it cannot show that `r > m` is not wrong.

Integer division

Due to integer division, `delta` may be zero:

  Midpoint(9, 10)

= 9 + ((10 - 9) / 2)
= 9 + (1 / 2)
= 9 + 0

Euclid’s division algorithm

This algorithm does division by repeated subtraction.

Note that Dafny lets a method return multiple values. Here, we return the quotient and the remainder.

```
method Euclid(m : int, n :
int) returns (q : int, r :
int)
{
  q := 0;
  r := m;
  while (r >= n)
  {
    r := r - n;
    q := q + 1;
  }
}
```

Euclid's division algorithm

This algorithm does division by repeated subtraction.

Note that Dafny lets a method return multiple values. Here, we return the quotient and the remainder.

```
method Euclid(m : int, n : 
int)
returns (q : int, r : int)
{
  q := 0;
  r := m;
  while (r > n)
  {
    r := r - n;
    q := q + 1;
  }
}
```

Totality checking

Dafny immediately complains that it cannot prove that the method always terminates.

In Dafny, methods are total: i.e., they always terminate with a return and do not loop forever or throw exceptions.

We will address this problem at the very end.

A specification for the division algorithm

What does it mean for division to be correct?

m == q * n + r
and
r < n

But, Dafny is not happy with these post-conditions. What could be wrong?

Division by zero? Negative numbers?

Let’s use preconditions to rule out negative numbers and division by zero.

Dafny still raises errors, but it no longer complains that the method may not terminate.

Dafny needs a little help.
It needs to reason about all possible program executions without actually running the loop.

Loop Invariants

We need to give Dafny a **loop invariant:** a property that is true before and after each loop iteration.

There are many different possible invariants. Finding the right invariant will help Dafny prove the postconditions.

Coming up with a loop invariant requires intuition and experience.

Loop Invariants

```
method Euclid(m : int, n : int)
returns (q : int, r : int)
requires n > 0;
requires m >= 0;
ensures m == q * n + r;
ensures r < n;
{
  q := 0;
  r := m;
  while (r > n)
  {
    r := r - n;
    q := q + 1;
  }
}
```

Loop Invariants: on loop entry

In this case, m == q * n + r, which is exactly a post-condition, is a loop invariant that does work.

Notice that initially:
q = 0, r = m, thus
 m = q * n + r
 r = 0 * n + r
 r = r

```
method Euclid(m : int, n : int)
returns (q : int, r : int)
requires n > 0;
requires m >= 0;
ensures m == q * n + r;
ensures r < n;
{
  q := 0;
  r := m;
  while (r > n)
  invariant m == q * n + r;
  {
    r := r - n;
    q := q + 1;
  }
}
```

Loop Invariants: after each iteration

$m = q * n + r$

The first statement in the loop subtracts n from r:

$m = q * n + (r - n)$

The second statement in the loop adds 1 to q:

$$\begin{aligned} m &= (q + 1) * n + (r - n) \\ &= q * n + n + r - n \\ &= q * n + r \end{aligned}$$

Therefore, it is an invariant!

```
method Euclid(m : int, n : int)
returns (q : int, r : int)
requires n > 0;
requires m >= 0;
ensures m == q * n + r;
ensures r < n;
{
  q := 0;
  r := m;
  while (r > n)
  invariant m == q * n + r;
  {
    r := r - n;
    q := q + 1;
  }
}
```

Still Broken

We still have errors!

Notice that Dafny is able to verify one post-condition:
`m == q * n + r`

But, the other post-condition: r < n
is not verifiable.

Is the program actually correct?

Almost done

The program was actually broken!

To fix it, we change the loop condition from `r > n` to `r >= n`.

Complete

As a final step, we need to help Dafny prove that the loop terminates.

We do this by specifying “`decreases r`”: `r` decreases on each loop iteration until it reaches zero.

Final version

```
method Euclid(m : int, n : int)
              returns (q : int, r : int)
requires n > 0;
requires m >= 0;
ensures m == q * n + r;
ensures r < n;
{
  q := 0;
  r := m;
  while (r >= n)
  decreases r;
  invariant m == q * n + r;
  {
    r := r - n;
    q := q + 1;
  }
}
```

Summary

Why verify software? Certain software systems are safety-critical. E.g., avionics, medical equipment, and cryptography.

- Astrée has been used to verify some of the software in Airbus jets and satellites.
- F* has been used to verify the certain cryptography libraries in Firefox.
- Coq has been used to verify parts of the BoringSSL cryptography library.

Different approaches to verification:

- Dafny: write verified code in a language designed for verification
- Astrée: write code in a (small!) subset of C, which the Astrée verifier can handle
- Coq: write code and proofs in Coq, then *extract* verified code to another language (e.g., C)