CS 591: Formal Methods in Security and Privacy

Introduction, Class Structure, Logistics, and Objectives

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu

From the previous class

Is this code correct?

```
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Is this code correct?

```
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Something seems wrong.

Is It the name or the program?

Adding the specification

```
Precondition: x ≥ 0 and y ≥ 0
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
Postcondition: r == x + y
```

Does the program comply with the specification?

Precondition: $x \geq 0$ and $y \geq 0$

```
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

Fail to meet the specification

How about this one?

Precondition: x ≥ 0 and y ≥ 0

```
Function Add(x: int, y: int) : int
{
  r = x;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

It meets
the specification

How can we make this reasoning mathematically precise?

Formal Semantics

We need to assign a formal meaning to the different components:

```
Precondition
Program
Postcondition
```

formal semantics of specification conditions

formal semantics of programs

formal semantics of specification conditions

We also need to describe the rules which combine program and specifications.

A first example

```
FastExponentiation(n, k : Nat) : Nat
  r:=1;
  if k > 0 then
   while k > 1 do
    if even(k) then
      n := n * n;
      k := k/2;
    else
      r := n * r;
      n := n * n;
      k := (k - 1)/2;
   r := n * r;
```

Programming Language

```
c::= abort
   | skip
   | x:=e
   | c;c
   | if e then c else c
   | while e do c
```

`x, y, z, ...`	program variables
$e_1, e_2, \ldots$	expressions
$c_1, c_2, \ldots$	commands

Expressions

We want to be able to write complex programs with our language.

```
e::=  x
    | f(e₁,…,eₙ)
```

Where `f` can be any arbitrary operator.

Some expression examples

```
x+5      x mod k      x[i]      (x[i+1] mod 4)+5
```

Types

In expressions we want to be able to use “arbitrary” data types.

```
t::= b
   | T(t1,…,tn)
```

We assume a collection of base types `b` including

`Bool` `Int` `Nat` `String`

We also assume a set of type constructors `T` that we can use to build more complex types, such as:

`Bool list` `Int*Bool` `Int*String -> Bool`

Types

We also use types to guarantee that commands are well-formed.

For example, in the commands

`while e do c`   `if e then` c_1 `else` c_2

We require that `e` is of type `Bool`.

We omit the details of the type system here but you can find them in the notes by Gilles Barthe

Values

Values are atomic expressions whose semantics is self-evident and which do not need a further analysis.

For example, we have the following values

`true`   `5`   `[1,2,3,4]`   `“Hello”`

The following are not values:

`not true`   `x+5`   `[x,x+1]`   `x[1]`

We could define a grammar for values, but we prefer to leave this at the intuitive level for now.

How can we give semantics to expressions and commands?

```
FastExponentiation(n, k : Nat) : Nat
  r:=1;
  if k > 0 then
  while k > 1 do
    if even(k) then
      n := n * n;
      k := k/2;
   else
     r := n * r;
     n := n * n;
     k := (k - 1)/2;
  r := n * r;
```

Memories

We can formalize a memory as a map $\mathtt{m}$ from variables to values.

$$\mathtt{m}=[\mathtt{x}_1 \longmapsto \mathtt{v}_1, \ldots, \mathtt{x}_n \longmapsto \mathtt{v}_n]$$

We consider only maps that respect types.

We want to read the value associated to a particular variable:

$$\mathtt{m}(\mathtt{x})$$

We want to update the value associated to a particular variable:

$$\mathtt{m}[\mathtt{x}\leftarrow\mathtt{v}]$$

This is defined as

$$\mathtt{m}[\mathtt{x}\leftarrow\mathtt{v}](\mathtt{y})=\begin{cases}\mathtt{v} & \text{If} \quad \mathtt{x}=\mathtt{y} \\ \mathtt{m}(\mathtt{y}) & \text{Otherwise}\end{cases}$$

Semantics of Expressions

What is the meaning of the following expressions?

```
x+5        x mod k        x[i]        (x[i+1] mod 4)+5
```

We can give the semantics as a relation between expressions, memories and values.

```
Exp * Mem * Val
```

We will denote this relation as:

{e}$_{m}$=v

This is commonly typeset as:

$$[\![e]\!]_m = v$$

Semantics of Expressions

This is defined on the structure of expressions:

$$\{x\}_m = m(x)$$

$$\{f(e_1, \ldots, e_n)\}_m = \{f\}(\{e_1\}_m, \ldots, \{e_n\}_m)$$

where $\{f\}$ is the semantics associated with the basic operation we are considering.

Semantics of Expressions

Suppose we have a memory

$$\texttt{m=[i}\longmapsto\texttt{1,x}\longmapsto\texttt{[1,2,3],y}\longmapsto\texttt{2]}$$

That $\{\texttt{mod}\}$ is the modulo operation and $\{\texttt{+}\}$ is addition, we can derive the meaning of the following expression:

$$
\begin{aligned}
&\{\texttt{(x[i+1] mod y)+5}\}_{\texttt{m}} \\
&\quad = \{\texttt{(x[i+1] mod y)}\}_{\texttt{m}}\{\texttt{+}\}\{\texttt{5}\}_{\texttt{m}} \\
&\quad = (\{\texttt{x[i+1]}\}_{\texttt{m}}\ \{\texttt{mod}\}\ \{\texttt{y}\}_{\texttt{m}})\{\texttt{+}\}\{\texttt{5}\}_{\texttt{m}} \\
&\quad = (\{\texttt{x}\}_{\texttt{m}}[\{\texttt{i}\}_{\texttt{m}}\{\texttt{+}\}\{\texttt{1}\}_{\texttt{m}}]\ \{\texttt{mod}\}\ \{\texttt{y}\}_{\texttt{m}})\{\texttt{+}\}\{\texttt{5}\}_{\texttt{m}} \\
&\quad = (\{\texttt{x}\}_{\texttt{m}}[\texttt{1}\{\texttt{+}\}\texttt{1}]\ \{\texttt{mod}\}\ \texttt{2})\{\texttt{+}\}\texttt{5} \\
&\quad = (\{\texttt{x}\}_{\texttt{m}}[\texttt{2}]\ \{\texttt{mod}\}\ \texttt{2})\{\texttt{+}\}\texttt{5} \\
&\quad = (\texttt{2}\ \{\texttt{mod}\}\ \texttt{2})\{\texttt{+}\}\texttt{5} = \texttt{0}\ \{\texttt{+}\}\ \texttt{5} = \texttt{5}
\end{aligned}
$$

Operational vs Denotational Semantics

The style of semantics we are using is denotational, in the sense that we describe the meaning of an expression by means of the value it denotes.

A different approach, more operational in nature, would be to describe the meaning of an expression by means of the value that the expression evaluates to in an abstract machine.