CS 320 : Operational Semantics

Marco Gaboardi

MSC 116

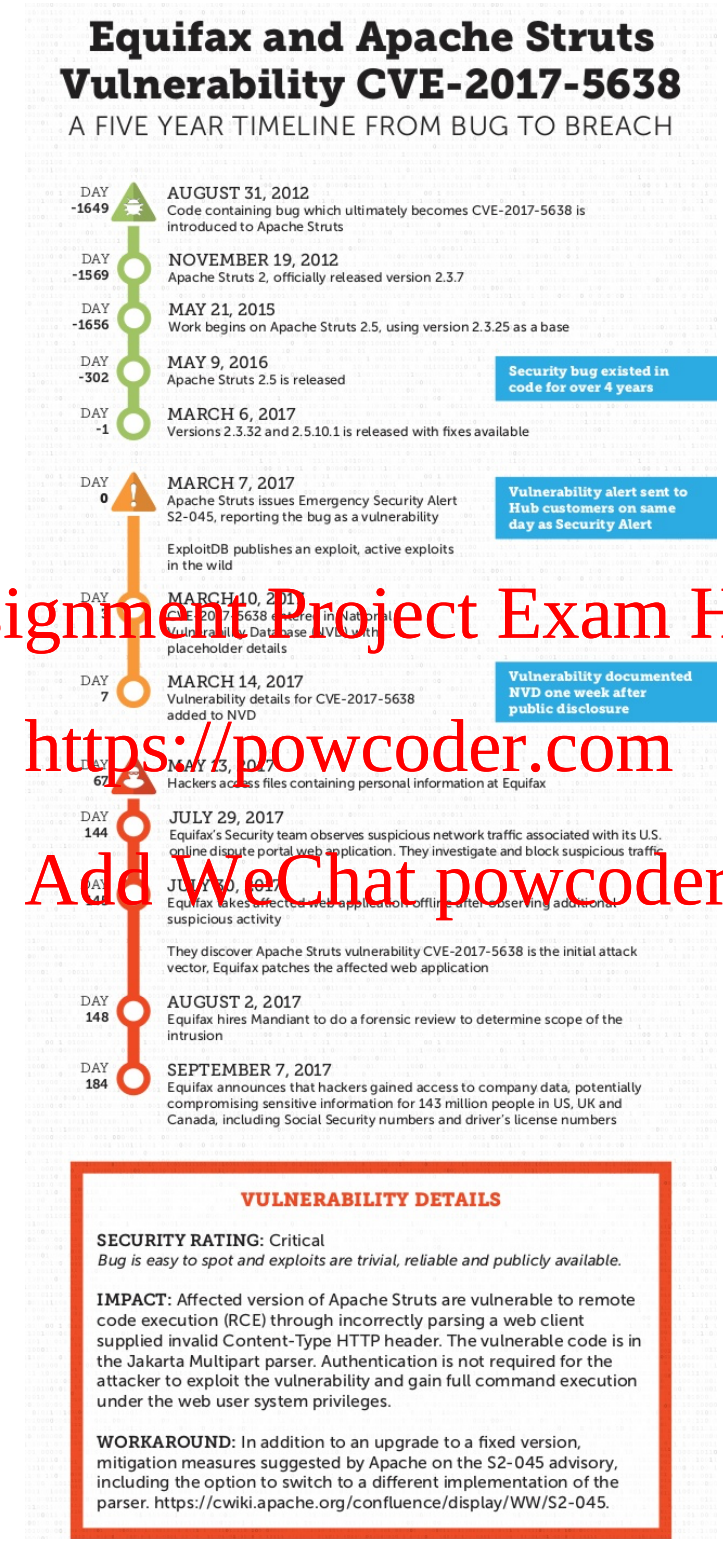
gaboardi@bu.edu

Announcements

- ***Programming Assignment #4*** is due on Friday, April 3.
- ***Grading Policy for Spring 2020***: Read the article in **BU Today**, University to Offer Students Credit/No Credit Option
- ***Programming Assignment #5*** will be posted on Friday, April 3.
- ***TopHat questions*** are included in Zoom meetings, but not graded.
- All ***Zoom meetings*** are recorded (by default), and their recordings available for download shortly after the live meetings.
- All ***Zoom links*** for CS 320 are at the bottom of the *Resources* webpage on *Piazza*.

What kind of bug?

Vulnerability in the exception handling of parsing in Apache Struts open software



Equifax agreed in paying between 570 and 700 million dollars settlement

Interesting and entertaining reading

If you want to know more about ***Apache Struts 2*** and ***Equifax***:

- Apache Struts 2 is an open-source web application framework for developing Java EE web applications.
- Equifax Inc. is one of the three largest consumer credit reporting agencies, along with Experian and TransUnion. Equifax collects and aggregates information on over 800 million individual consumers and more than 88 million businesses worldwide.

If you want to know more about the software security bug in ***Struts 2***, which was later exploited in a massive data breach by ***Equifax***:

- "Equifax couldn't find or patch vulnerable Struts implementations", The Register, October 2, 2017.
- "A series of delays and major errors led to massive Equifax breach", Ars Technica, October 2, 2017.

Another security breach caused by malfunctioning software:

- ***Zoombombing*** (if you have not heard or read about it, search for articles on the Web – many have been published since mid-March).

- Is it a ***bug*** or a ***feature*** in the Zoom software? We are still waiting for a final judgment on this ...

Are type error dangerous?

Carnegie Mellon mistakenly accepts 800 applicants, then rejects them


By Holly Yan and Katia Hetter, CNN

Updated 2:27 PM ET, Wed February 18, 2015

More from CNN


The latest on the Tru impeachment inquir

State Department U experts next up in impeachment inquir

Learning Goals for today

- To understand how the dynamic semantics of a program describes the meaning of a program.
- To understand the subtleties of different evaluation strategies.
- Digging into the concepts of binding and scope.

Language for the Interpreter (simplified)

The language for the interpreter can be described by t
grammar:

```
<const> ::= int | string

<prog>  ::= <com>;quit | <com>;<prog>

<com>   ::= push <const> | add | sub | mul | div
          | neg | rem | swap
```

We use ; here to separate commands, in the interpreter we use a newline.

- A program is a sequence of commands followed by `quit`.
- A command is one the keywords above - in the case of push this is followed by a constant.
- A (simplified) constant is either an int or a string.
- We will denote arbitrary programs with `p,p',…`

Multiple steps of Operational semantics

We have seen the reduction relation between configurations:

$$(p, S) \rightarrow (p', S')$$

We say that from the configuration (p, S) we can step (or reduce) to the configuration (p', S') in one step.

In general we are interested in (finite or infinite) sequences of reduction steps:

$$(p_1, S_1) \rightarrow (p_2, S_2) \rightarrow (p_3, S_3) \rightarrow \ldots \rightarrow (p_k, S_k)$$

Summary of some rules:

(A) (push num;p/S) → (p/num::S)

(B) (add;p/int(v_2)::int(v_1)::S) → (p/int(v_2+v_1)::S)

(C) (add;p/type(v_1)::[]) → (p/(:error:)::type(v_1)::[])

(D) (add;p\[]) → (p\(:error:)::[])

(E) (quit/S) → print(S)

(F) (add;p/notint(v)::S)→(p/(:error:)::notint(v)::S)

(G) (add;p/int(v_1)::notint(v_2)::S)
→(p/(:error:)::int(v_1)::notint(v_2)::S)

Let's give to each rule a name.

An example:

```
(push 5;push 4;add;quit/S) →
(A)  (push 4;add;quit/int(5)::S) →
(A)  (add;quit/int(4)::int(5)::S) →
(B)  (quit/int(4+5)::S) →
(E)   print(int(4+5)::S)
```

Another example:

```
(push 5;add;quit/[]) →
(A)  (add;quit/int(5)::[]) →
(C)  (quit/(:error:)::int(5)::S) →
(E) print((:error:)::int(5)::S)
```

TopHat Q1 - Q5

Another example: `div`

Informal description: The command div refers to integer division. It is a binary operator and works in the following way:

- if top two elements in the stack are integer numbers, pop the top element(y) and the nextelement(x), divide x by y, and push the result x back onto the stack y
- if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push :error: onto the stack
- if the top two elements in the stack are not all integer numbers, push them back in the same order and push :error: onto the stack
- if there is only one element in the stack, push it back and push :error: onto the stack
- if the stack is empty, push :error: onto the stack

Another example: `div`

Formal description:

$(\texttt{div};\texttt{p}/\texttt{int}(v_2)::\texttt{int}(v_1)::\texttt{S}) \rightarrow (\texttt{p}/\texttt{int}(v_2/v_1)::\texttt{S})$

$(\texttt{div};\texttt{p}/\texttt{type}(v_1)::[]) \rightarrow (\texttt{p}/(:\texttt{error}:)::\texttt{type}(v_1)::[])$

$(\texttt{div};\texttt{p}\backslash[]) \rightarrow (\texttt{p}\backslash(:\texttt{error}:)::[])$

$(\texttt{div};\texttt{p}/\texttt{notint}(v)::\texttt{S}) \rightarrow (\texttt{p}/(:\texttt{error}:)::\texttt{notint}(v)::\texttt{S})$

$(\texttt{div};\texttt{p}/\texttt{int}(v_1)::\texttt{notint}(v_2)::\texttt{S})$

$\rightarrow (\texttt{p}/(:\texttt{error}:)::\texttt{int}(v_1)::\texttt{notint}(v_2)::\texttt{S})$

Are we done?

Informal description:

if top two elements in the stack are integer numbers but y equals to 0, push them back in the same order and push :error: onto the stack

Another example: div

Formal description:

(div;p/int(v_2)::int(v_1)::S) → (p/int(v_2/v_1)::S)

(div;p/type(v_1)::[]) → (p/(:error:)::type(v_1)::[])

(div;p\[]) → (p\(:error:)::[])

(div;p/notint(v)::S) → (p/(:error:)::notint(v)::S)

(div;p/int(v_1)::notint(v_2)::S)
→(p/(:error:)::int(v_1)::notint(v_2)::S)

(div;p/int(v_2)::int(0)::S)
→ (p/(:error:)::int(v_2)::int(0)::S)

What is the problem?

Another example: `div`

This is not defined!
That is why we
want another rule!

If we have the configuration:

$$(\texttt{div;p/int}(2)\texttt{::int}(0)\texttt{::S})$$

We can apply either one of the following rules

$$(\texttt{div;p/int}(v_2)\texttt{::int}(v_1)\texttt{::S}) \rightarrow (\texttt{p/int}(v_2/v_1)\texttt{::S})$$

$$(\texttt{div;p/int}(v_2)\texttt{::int}(0)\texttt{::S}) \rightarrow (\texttt{p/(:error:)::int}(v_2)\texttt{::int}(0)\texttt{::S})$$

And obtain:

$$(\texttt{div;p/int}(2)\texttt{::int}(0)\texttt{::S}) \rightarrow \boxed{(\texttt{p/int}(2/0)\texttt{::S})}$$

$$(\texttt{div;p/int}(2)\texttt{::int}(0)\texttt{::S}) \rightarrow (\texttt{p/(:error:)::int}(2)\texttt{::int}(0)\texttt{::S})$$

Rules with hypothesis

We want some way to express a condition under which we can apply a rule.

One way to do this is by adding an hypothesis:

$$\frac{(p/S) \text{ satisfies the hypothesis}}{(p/S) \rightarrow (p'/S')}$$

Example:

$$\frac{v_1 \neq 0}{(\texttt{div};p/\texttt{int}(v_2)::\texttt{int}(v_1)::S) \rightarrow (p/\texttt{int}(v_2/v_1)::S)}$$

Another example: `div`

Formal description:

We will leave implicit the lines of the rule with no hypothesis.

$$\frac{v_1 \neq 0}{(\mathtt{div};\mathtt{p}/\mathtt{int}(v_2)::\mathtt{int}(v_1)::\mathtt{S}) \rightarrow (\mathtt{p}/\mathtt{int}(v_2/v_1)::\mathtt{S})}$$

$$\frac{}{(\mathtt{div};\mathtt{p}/\mathtt{type}(v_1)::[]) \rightarrow (\mathtt{p}/(\mathtt{:error:})::\mathtt{type}(v_1)::[])}$$

$$\frac{}{(\mathtt{div};\mathtt{p}\backslash[]) \rightarrow (\mathtt{p}\backslash(\mathtt{:error:})::[])}$$

$$\frac{}{(\mathtt{div};\mathtt{p}/\mathtt{notint}(\mathtt{v})::\mathtt{S}) \rightarrow (\mathtt{p}/(\mathtt{:error:})::\mathtt{notint}(\mathtt{v})::\mathtt{S})}$$

$$\frac{}{(\mathtt{div};\mathtt{p}/\mathtt{int}(v_1)::\mathtt{notint}(v_2)::\mathtt{S}) \rightarrow (\mathtt{p}/(\mathtt{:error:})::\mathtt{int}(v_1)::\mathtt{notint}(v_2)::\mathtt{S})}$$

$$\frac{}{(\mathtt{div};\mathtt{p}/\mathtt{int}(v_2)::\mathtt{int}(0)::\mathtt{S}) \rightarrow (\mathtt{p}/(\mathtt{:error:})::\mathtt{int}(v_2)::\mathtt{int}(0)::\mathtt{S})}$$

Language for the Interpreter (simplified)

The language for the interpreter can be described by the following grammar:

```
<const> ::= int | string
<prog>  ::= <com>;quit | <com>;<prog>
<com>   ::= push <const> | add | sub | mul | div
          | neg | rem | swap
```


What is the form of a program?

<com>;<com>;…;<com>;quit

Is this the common way we write programs?

Arithmetical expressions: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <expr> <addop> <term> | <term>
<addop>::= add | minus
<term> ::= var | val
```


What are the challenges here?

What is the form of a program?

`[[<term>(+|-)<term>](+|-)<term>]…<term>`

Do we need a stack here?

Arithmetical expressions: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <expr> <addop> <term> | <term>
<addop>::= add | minus
<term> ::= var | val
```


What are the challenges here?

What is the value of a var?

```
X add 5 add 6
```

Where is the value defined?

Operational semantics for arithmetical expressions

$$(e/m) \rightarrow (e/m)$$

Here (e/m) is a configuration where e is an expression and m is a memory. We call these pairs configurations because we think in terms of an "abstract machine".

We can think about a memory as a set of (unique) assignments of variables to values:

$$m = ((x_1=v_1), (x_2=v_2) \ldots, (x_n=v_n))$$

The memory is the environment where the variable of an expression are defined.

Arithmetical expressions: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <expr> <addop> <term> | <term>
<addop>::= add | minus
<term> ::= var | val
```


- What is the potential shape of an expression?
  - `v` — Value
  - `x` — Variable
  - `e add (n|x)` — Expression + Constant or Variable

This is recursive

An example: addition of values

What can we do when we have an expression instead of a value v_1?

$(v_1 \ \texttt{add} \ v_2/m) \rightarrow (v_1+v_2/m)$

Here we use v_1+v_2 to actually mean the result of summing the two values

An example: addition of values

$$\frac{(e/m) \rightarrow (e_1/m)}{(e \text{ add } v_2/m) \rightarrow (e_1 \text{ add } v_2/m)}$$

We can use the fact that e is recursive and hypothetical reasoning.

An example: fetching the value of a variable

$(x/m) \rightarrow ?(fetch(x,m)/m)$

How can we fetch the value of x from the environment?

We can introduce a function `fetch` taking an environment a variable and returning the value of the variable in the environment.

$fetch(x_2,((x_1=v_1),(x_2=v_2)\ldots,(x_n=v_n))=v_2$

An example: fetching the value of a variable

The variable can appear in an expression.
On the right:


```
(e add x/m) → (e add fetch(x,m)/m)
```


Or on the left:


```
(x add v/m) → (fetch(x,m) add v/m)
```

How about a value?

`(n/m)` $\not\rightarrow$ `??`

What does a number evaluate to?

This is a value, so we don't have any step

Summary of the rules:

(F) $(x/m) \rightarrow (\mathtt{fetch}(x,m)/m)$

(+A) $(e \ \mathtt{add} \ x/m) \rightarrow (e \ \mathtt{add} \ \mathtt{fetch}(x,m)/m)$

(+B) $(x \ \mathtt{add} \ v/m) \rightarrow (\mathtt{fetch}(x,m) \ \mathtt{add} \ v/m)$

(+C) $(v_1 \ \mathtt{add} \ v_2/m) \rightarrow (v_1+v_2/m)$

(+D) $$\frac{(e/m) \rightarrow (e_1/m)}{(e \ \mathtt{add} \ v_2/m) \rightarrow (e_1 \ \mathtt{add} \ v_2/m)}$$

Summary of the rules:

(-A) $(e \text{ minus } x/m) \rightarrow (e \text{ minus } \text{fetch}(x,m)/m)$

(-B) $(x \text{ minus } v/m) \rightarrow (\text{fetch}(x,m) \text{ minus } v/m)$

(-C) $(v_1 \text{ minus } v_2/m) \rightarrow (v_1 - v_2/m)$

(-D) $$\frac{(e/m) \rightarrow (e_1/m)}{(e \text{ minus } v_2/m) \rightarrow (e_1 \text{ minus } v_2/m)}$$

An example:

(3 add 5/()) → (8/())

(3 add 5 minus 6/()) → (8 minus 6/()) → (2/())

How many steps of reduction?

Another example:

Let us call m=(x=3,y=5,z=6) we have:

(x add y minus z/m) →

(x add y/m) → (x add 5/m)

(x add y minus 6/m) → (x add 5 minus 6/m) =

(x add 5/m) → (3 add 5/m)

(x add 5 minus 6/m) → (3 add 5 minus 6/m) =

(3 add 5/m) → (8/m)

(3 add 5 minus 6/m) → (8 minus 6/m) → (2/m)

Yet another example:

Let us call m=(x=3,y=5,z=6) we have:

```
(x add 4 minus z/m) →

     (x add 4/m) → (3 add 4/m)
-------------------------------------
(x add 4 minus 6/m)→(3 add 4 minus 6/m) =

     (3 add 4/m) → (8/m)
-----------------------------------
(3 add 4 minus 6/m)→(7 minus 6/m) → (1/m)
```

Variables

Variables

- Functional languages use variables as names (where the association name-value is stored in an environment).
  - We can remember the association, or read the value, but we cannot change it.
- Imperative languages are abstractions of von Neumann architecture
  - A variable abstracts the concept of memory location
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Let expression

- Let us consider this simple language for expressions

```
<expr> ::= let var= <expr> in <expr> |
           <expr> <addop> <term> | <term>
<addop>::= add | minus
<term> ::= var | val
```


- What is the semantics of a let expression?

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```


Moral: Let operates by *substituting* computed values for variables

-->

```
let y = 4      in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Rules for let

$$\frac{(e_1/m) \rightarrow (e_3/m)}{(\texttt{let } x{=}e_1 \texttt{ in } e_2/m) \rightarrow (\texttt{let } x{=}e_3 \texttt{ in } e_2/m)}$$

We can use the fact that e_1 is recursive and hypothetical reasoning.

An example: recording the value of a variable

$(\texttt{let } x=v \texttt{ in } e/m) \rightarrow ??(e/m@(x=v))$

If variables are just names for values, where shell we store the name-value association?

This is the role of the environment, storing name-value associations.

$$((x_1=v_1),(x_2=v_2)\ldots,(x_n=v_n))$$

Where we use the symbol @ to extend the environment with a new name-value association.

Example:

Let us call m= (x=3,y=5,z=6) we have:

(x add y/m) → (x add 5/m)

(let k=x add y in k minus z/m)→(let k=x add 5 in k minus z/m) =

(x add 5/m) → (3 add 5/m)

(let k=x add 5 in k minus z/m)→(let k=3 add 5 in k minus z/m) =

(3 add 5/m) → (8/m)

(let k=3 add 5 in k minus z/m)→(let k=8 in k minus z/m) →

(x add y/m) → (x add 5/m)

(k minus z/m@(k=8))→(k minus 6/m@(k=8))→(8 minus 6/m@(k=8))

→(2/m@(k=8))

TopHat Q6 - Q12