CS162 Week 3

Kyle Dewey

Overview

- Grades posted for assignment 1
- Secure information flow key implementation issues
- Reactive imperative programming

Assignment 1 Lingering Questions

Secure Information Flow Assignment

Missing File

- As written, there is a missing file: `util.scala`
- Option 1: Download zip file from the course website (under "Interpreter Code"), copy `util.scala`, and add it to the makefile
- Option 2: Remove all mentions of the pretty printer (from `util.scala`)

Adding a Field for the Label

`pc` Stack

- Define an `object` named `pc`
- It internally has a mutable stack
- There are many ways to do this, but `scala.collection.mutable.Stack` is probably the easiest

test27.not

Any questions on secure information flow?

Reactive Imperative Programming

Motivation

- An interesting language feature
- Another possible feature to add if designing a language, along with objects and higher-order functions

Citation

- Camil Demetrescu et al.: Reactive imperative programming with dataflow constraints - OOPSLA'11
- Not an easy read, and it shouldn't be necessary
- A few key details are ambiguous or missing

Reactive

- More familiar technology: spreadsheets
- The value of a cell can depend on the value in other cells
- If the value of a cell changes, all dependent cells are updated
  - As in, all cells that somehow use the changed cell's value

Imperative

- Can work with the imperative paradigm
  - Roughly, with variable/field assignment
    - When a variable/field changes, everything marked as dependent is updated
- Spreadsheets are a case of reactive **functional** programming

Marking as Dependent

- "When variable `x` changes, execute this given code"
- Explicitly associate `x` with code
- Why isn't this a great idea?

Marking as Dependent

- Better alternative: "Here is some code that is reactive"
  - Let the language figure out which variables/fields are involved
  - Let the language worry about updating the right things
- The code is called a **constraint**

What would this look like?

`newCons` Operator

- Defines **both** code and what reacts to said code

```
var a in
   a := 0;
   newCons {
      output a // `a` is reactive
   };
   while (a < 10) {
      a := a + 1 // trigger `output`
   }
```

Output:

0

1

…

10

More Interesting Example

`sanitize.not`

Implementation

- From a high level, how might we implement this in the interpreter?

```
var a in
  a := 0;
  newCons {
    output a // `a` is reactive
  };
  while (a < 10) {
    a := a + 1 // trigger `output`
  }
```

Output:

0

1

…

10

Basic Semantics

- Execute code in what `newCons` delimits
- Mark addresses used inside what `newCons` delimits as reactive
- When these are changed outside of the same `newCons`, trigger the delimited code (a.k.a, the constraint)

Questions

- Is this enough detail to implement `newCons`?
- **Is this enough detail to use `newCons`?**

Cyclical Constraints

```
var a in
  a := 0;
newCons {
   a := a + 1;
   output a
};
a := 3
```

Output:

1

4

Multiple Constraints

```
var a in
  a := 3;
newCons {
  output a
};
newCons {
  a := a + 1
};
a := 5
```

Output:
3
4
6
6

Nested Constraints

```
var a, b in
  a := 4;
  b := "";
  newCons {
    output a;
  }
  newCons {
    output b
  };
  b := b + "b"
};
a := 5;
b := "t"
```

Output:

4

<<newline>>

b

5

b

bb

5

t

tb

tb

`newCons` with Objects

- What does this output?

```
var obj in
  obj := {“foo”: 1, “bar”: 2};
  newCons {
    output obj.foo
  };
  obj.foo := 10;
  obj.bar := 20
```

Output:

```
1
10
10
```

These Slides Don't Cover...

- The `atomic` block
- Different execution modes
- Specifically how to implement in `miniJS`