CS162 Week 4

Kyle Dewey

Overview

- Reactive imperative programming refresher
- Rest of `newCons`
- What `atomic` is
- Implementation details

Reactive Imperative Programming Refresher

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)

newCons

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

Output:

0

1

...

10

Questions

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

Multiple Constraints #1

```
var a in
  a := 0;
  newCons {
    output a
  };
newCons {
  output a + 1
};
a := 10
```

Output:

0

1

11

10

Cyclical Constraints

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

Output:

4

Multiple Constraints #2

```
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

The Point

- There are a lot of different edge cases
- As the language designer, these should all be accounted for

atomic **Blocks**

Problem

- We need to update a variable multiple times during a loop
- The computation is not “done” until the last assignment
- We want to update only when the computation is done

Example

```
var a in
   a := 0;
   newCons {
      output a
   };
   while (a < 11) {
      a := a + 3
   };
   a := a + a // now `a` is ready
```

Output:

0

3

6

9

12

24

Hacky Solution

- Add a flag `isDone`
  - Set to `false` beforehand
  - Set to `true` when a constraint is ready
  - In the constraint, only process if `isDone` is `true`

Better Solution

- Let the language handle it
- Introduce a special `atomic` block
  - Constraints are only updated once we leave the `atomic` block
  - Instead of having multiple updates of the same constraint, only update the constraint once at the end

With atomic

```
var a in
   a := 0;
   newCons {
      output a
   };
   atomic {
      while (a < 11) {
         a := a + 3
      };
      a := a + a // now `a` is ready
   }
```

Output:
0
24

Nesting atomic

```
var a, b in
   newCons {
      output b
   };
   newCons {
      output a
   };
   atomic {
      a := 2;
      atomic {
         b := 4
      };
      a := 3
   }
```

Output:
undef
undef
3
4

Implementation Details

Evaluation Modes

- The interpreter can be in one of three modes:
  - Normal mode (normal execution)
  - Constraint mode
  - Atomic Mode
- See `domains.scala`

Constraint Mode

- Whenever the body of a `newCons` block is executed
  - First entrance of `newCons`
  - When a reactive address is updated in normal mode or constraint mode
  - When we exit all `atomic` blocks
- Stores which constraint is currently being executed (useful for preventing recursive constraints)

Atomic Mode

- Whenever we execute the body of an `atomic` **block**
  - No constraints are triggered in this mode
  - Store reactive addresses that were updated to trigger them once we leave atomic mode

Data Structures

- `Dependencies`
  - Maps reactive addresses to sets of constraints
  - See `domains.scala`
- `constraintStack`
- `atomicStack`

constraintStack

- For nested new constraints
- Records which constraint is currently active

`atomicStack`

- **For nested** `atomic` **blocks**
- Records which reactive addresses need constraint updates upon leaving the last `atomic` **block**

Tips

- **Never** execute a constraint when you are in atomic mode
- Self-recursive constraints should **never** trigger themselves
- Reactive addresses can be both **added** and **removed** via `newCons`, depending on what gets used in the `newCons`' body
  - If a previously reactive address is not used when executing ~~a constraint~~ `newCons`, the address is no longer reactive