SE580: Lecture 5

Overview

User-defined methods

Syntax

Dynamic semantics

Continuations

Static semantics

Implementation

Homework

User-defined methods

What is a method?

Why are methods useful?

In *imperative* languages (e.g. Pascal or C) where do methods belong?

In *class-based* languages (e.g. Java, C++ or Hobbes) where do methods belong?

In *object-based* languages (e.g. Oblique) where do methods belong?

Methods have a *self variable* (this in Java). Why?

In OO languages, method calls can be *static* or *dynamic*. What is the difference?

Syntax

Example Hobbes program:

```
class Ref {
  mutable field contents : Integer;
  method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
  method this.get () : Integer { return this.contents; }
}
object R : Ref { contents = 5; }
thread Main {
  R::Ref.set (7); // static call
  let x = R.get (); // dynamic call
}
```

How is this different from the Java equivalent? How important are the differences?

Dynamic semantics

Dynamic method calls are easy, they just become static calls!

thread a { let X = $b.m$ (V_1,...,V_n); B } object b : T {...} P

–>

thread a { let X = b::$T.m$ (V_1,...,V_n); B } object b : T {...} P

For example:

```
  class Ref {
    mutable field contents : Integer;
    method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
    method this.get () : Integer { return this.contents; }
  }
  object R : Ref { contents = 5; }
  thread Main {
    let x = R.get ();
  }
-->
  ???
```

Dynamic semantics

For static method calls, consider methods with no parameters:

```
  class Ref {
    mutable field contents : Integer;
    method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
    method this.get () : Integer { return this.contents; }
  }
  object R : Ref { contents = 5; }
  thread Main {
    let x = R::Ref.get ();
    let y = x + 1;
  }
-->
  ???
```

Dynamic semantics

Slightly trickier:

```
  class Ref {
    mutable field contents : Integer;
    method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
    method this.get () : Integer { return this.contents; }
  }
  object R : Ref { contents = 5; }
  thread Main {
    let x = R::Ref.get ();
    let y = x + 1;
    let z = R::Ref.set (y);
  }
-->
  ???
```

Dynamic semantics

Even trickier:

```
  class Ref {
    mutable field contents : Integer;
    method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
    method this.get () : Integer { return this.contents; }
   method this.inc () : Integer { let x = this.get (); let y = x+1; let z = this.set (y); return y; }
  }
  object R : Ref { contents = 5; }
  thread Main {
    let a = R.inc ();
    let b = a + a;
  }
-->
  ???
```

Dynamic semantics

Hobbes deals with this by introducing *continuations*:

```
  thread Foo {
    let X = method call;
    rest of the thread
  }
-->
  thread Foo {
    method body
    continuation (X) {
      rest of the thread
    }
  }
```

Read this as “execute the method body, then continue with the rest of the thread, binding the result to X”

Continuations

For example, what are:

```
return 37; continuation (x) { let y = x + x; }
let z = 1 + 2; return z; continuation (x) { let y = x + x; }
let z = 1 + 2; return z; continuation (x) { let y = x + x; return y; continuation (a) { let z = a + 1; } }
```

Note: multiple method calls can build up a *stack* of continuations.

Continuations

Formal definition of a continuation K:

```
Continuation ::= "(" Var ")" "{" Block "}"
```

Define B continuation K to be block where we execute B then continuation with K.

What is (return V;) continuation (X) { B }?

What is (let Y = E; B) continuation K?

What is (if (V) { B } else { C }) continuation K?

Note: we require continuations to be *closed*. What does this mean? Why?

Continuations

Note: in Hobbes, continuations are *second-class citizens*. Programmers cannot access continuations directly.

In some languages (e.g. LISP) continuations are *first-class citizens* and can be manipulated directly, for example:

```
try { throw k (57); }
catch k (x:Integer) { let y = x+x; }
```

First-class continuations are very similar to exceptions, but they are statically rather than dynamically scoped (the `catch k` statement is a binder for `k`).

Languages with first-class continuations are very powerful, but they can make writing multi-language code even harder!

Static semantics

What changes to the type system do we need for this code:

```
class Ref {
  mutable field contents : Integer;
  method this.set (x : Integer) : Void { this.contents := x; return Nothing; }
  method this.get () : Integer { return this.contents; }
 method this.inc () : Integer { let x = this.get (); let y = x+1; let z = this.set (y); return y; }
}
object R : Ref { contents = 5; }
thread Main {
  let a = R.inc ();
  let b = a + a;
}
```

Implementation

Routine changes to the parser.

What needs added to the interpreter and typechecker?

Homework

Complete the interpreter and typechecker for Hobbes (currently missing most of the cases for methods).

Same procedure as before! (But the deadline is in two weeks!)

Next week

Midterm exam.