

Compiling Higher-Order Languages

Lecture 24

Prof. Neula CS 164

1

Overview

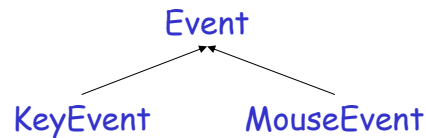
- Why higher-order languages ?
- Higher-order constructs for Cool
- Type checking
- Code generation

Prof. Neula CS 164

2

Motivating Example

- Consider a Graphical User Interface (GUI)
 - Listens to events from the user (key press, mouse move) and invokes application-specific code
- Let's write a GUI that counts the total number of events, and also prints a message for all mouse events
- We use a toolkit similar to AWT with classes:



Prof. Neula CS 164

3

User Interface Code

```
class KeyListener { key(e: KeyEvent) : Object { ... } }
class MouseListener { mouse(e: MouseEvent) : Object { ... } }

class Ui {
  klsn: KeyListener; mlsn: MouseListener; // Application-specific listeners

  setKeyListener(k: KeyListener) { klsn <- k }
  setMouseListener(m: MouseListener) { mlsn <- m }

  dispatchEvent(e: Event) { // Dispatch event to listeners
    case e of
      k: KeyEvent => klsn.key(k);
      m: MouseEvent => mlsn.mouse.(m);
    }
  }
}
```

Prof. Neula CS 164

4

Client Code

```
class Counter { count : Int <- 0; incr(x: Int) { count <- count + x } }
class Log { ...; print(x: String) { ... } }

class MyKeyListener inherits KeyListener {
  c: Counter; init(count: Counter) : MyKeyListener { c <- count; self }
  key(e: KeyEvent) { c.incr(1) }
}
class MyMouseListener inherits MouseListener {
  count: Counter; log: Log; init(c: Counter, l: Log) : MyMouseListener { ... }
  mouse(e: MouseEvent) { count.incr(1); log.print("got one"); }
}
...
let c : Counter <- new Counter in { // install the listeners into the UI
  ui.setKeyListener((new MyKeyListener).init(c));
  ui.setMouseListener((new MyMouseListener).init(c, new Log));
}
```

Adapter
classes

Prof. Neula CS 164

5

Notes

- A proliferation of classes (and code):
 - Some are really interfaces: `KeyListener`, `MouseListener`
 - Some are small "adapters": `MyKeyListener`, `MyMouseListener`
 - These classes add complexity (new namespaces, new constructors) without adding much value
- The `KeyListener` objects implement one function: `key`
- Idea: add first-class functions to the language
 - Functions can be passed as arguments
 - Expressions can evaluate to functions
 - Fields can have function values
 - As powerful as `KeyListener` objects

Prof. Neula CS 164

6

Overview

- ✓ Why higher-order languages ?
- Higher-order constructs for Cool
- Type checking
- Code generation

New Syntax for Constructing Functions

- Add new syntax for function expressions

```
fun (x: T1) : T2 { E }
```

 - An anonymous function with one argument x of type T_1 and result of type T_2 , and body E
 - Same syntax as methods
 - But can appear anywhere, not just as a features

```
let c : Counter <- new Counter in
  fun (e: KeyEvent) : Object { c.incr(1) }
```
 - The body of the function can refer to all identifiers in scope, not just to self and the attributes
 - Similar syntax to lambda from Scheme or Lisp
 - With types, fewer parentheses, and "modern Greek" keywords

New Syntax for Invoking Functions

- We add the syntax

$E_1(E_2)$

- Evaluate E_1 to a function, evaluate E_2 to a value, and invoke the function with that value

`let listener <- (fun (e: KeyEvent) ...) in ... listener(e) ...`

- How is $E_1(E_2)$ different than $E_1'.m(E_2)$?
 - E_1 must evaluate to a function that is invoked
 - E_1' must evaluate to an object with a method m . The actual code that is invoked is obtained from the dispatch table of E_1'
 - In method calls we pass both a "self" argument and actual arguments
 - In function calls we pass only the actual argument

Prof. Necula CS 164

9

Extended Cool Types

- Cool expression used to evaluate to objects
 - All types were Cool classes (we ignore SELF_TYPE today)
- We need to declare fields with function values
 - We need types for such fields
- A function is characterized by
 - The type of the arguments that it expects
 - The type of the result it yields
- We add the function type
 - $T_1 \rightarrow T_2$
 - type of functions with argument of type T_1 and result of type T_2
 - We consider only functions with one argument here
- Thus, new syntax for types: $T ::= C \mid T_1 \rightarrow T_2$

Prof. Necula CS 164

10

Function Types. Examples

KeyEvent → Object

- The type of a key listener
- One expression with this type:
`fun (e: KeyEvent) : Object { c.incr(1) }`

(KeyEvent → Object) → (KeyEvent → Object)

- The type of a function that takes a key listener and returns a key listener
- One expression with this type
`fun (lsn: KeyEvent → Object) : KeyEvent → Object {
 fun (e: KeyEvent) : Object { lsn(e); c.incr(1) }
}`
 - Wraps the listener "lsn" with counting

User Interface with Higher-Order Functions

```
class HighOrderUi {  
    klsn: KeyEvent -> Object;  
    mlsn: MouseEvent -> Object;  
    setKeyListener(k: KeyEvent -> Object) { klsn <- k }  
    setMouseListener(m: MouseEvent -> Object) { mlsn <- m }  
    dispatchEvent(e: Event) {  
        case e of  
            k: KeyEvent => klsn(k)  
            m: MouseEvent => mlsn(m)  
    }  
}
```

No need to define KeyListener and MouseListener

Client Code with Higher-Order Functions

```
class Counter { count : Int <- 0; incr(x: Int) { count <- count + x } }
class Log { ...; print(x: String) { ... } }
```

```
...
let c : Counter <- new Counter in {
  ui.setKeyListener( fun (e: KeyEvent) : Object { c.incr(1) } );
  ui.addMouseListener( fun (e: MouseEvent) : Object {
    c.incr(1); l.print("got one")
  } )
}
```

- No need to define the adapters `MyKeyListener` and `MouseListener`
- The functions can access the variables in scope

Prof. Necula CS 164

13

Side Note on Java

- Java addresses the verbosity associated with adapters in a (slightly) different way
- Java allows one to write anonymous nested classes

```
let c : Counter <- new Counter in
  ui.setKeyListener(new KeyListener {
    key(e: KeyEvent) { c.incr(1) } });
```

 - The `KeyListener { ... }` is an inner class derived from `KeyListener`
 - Advantages:
 - No need to invent the name `MyKeyListener`
 - The body of the new class can access `c` directly
- We have a more ambitious construct, that reduces verbosity even more

Prof. Necula CS 164

14

Overview

- ✓ Why higher-order languages ?
- ✓ Higher-order constructs for Cool
- Type checking
- Code generation

Typing Rules: Function Invocation

- Typing rule for function invocations

$$\frac{O, M, C \vdash e : T_1 \rightarrow T_2 \quad O, M, C \vdash e_1 : T_1}{O, M, C \vdash e(e_1) : T_2}$$

- Examples:
`let Isn: KeyEvent → Object ... in Isn(new KeyEvent)`
- This rule is not sufficient:
`let Isn : Event → Object ... in Isn(new KeyEvent)`
 - Is not well-typed
 - We must relax the typing of actual arguments, like for method calls

Typing Rules: Function Invocation

- Relaxed typing rule for function invocations

$$\frac{O, M, C \vdash e : T_1 \rightarrow T_2 \quad O, M, C \vdash e_1 : T_1' \quad T_1' \leq T_1}{O, M, C \vdash e(e_1) : T_2}$$

- The type of the actual argument must conform to the type of the function parameter

Typing Rules: Function Expression

$$\frac{O[T_1/x], M, C \vdash e : T_3 \quad T_3 \leq T_2}{O, M, C \vdash \text{fun } (x : T_1) : T_2 \{ e \} : T_1 \rightarrow T_2}$$

- Use current object environment O to check the body
 - For Cool methods we use a top-level environment, with only self and the attributes

Subtyping with Function Types

- We must also extend the \leq and **lub** operations on types
- Recall: $T \leq T'$ means that a value of type T supports all operations of values of type T'
 - $C \leq D$ when C is a subclass of D
- Can we have $T_1 \rightarrow T_2 \leq C$?
 - No, a function does not support method dispatch
- Can we have $C \leq T_1 \rightarrow T_2$?
 - No, we cannot apply an object to an argument
- Can we have $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$?
 - Yes, sometimes ...

Prof. Necula CS 164

19

When can we have $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$?

- Consider a value v of type $S_1 \rightarrow S_2$
- When can we use v in place of a value of type $T_1 \rightarrow T_2$?
- We must be able to
 - Apply v to an object of type T_1 and obtain a result of type T_2
- Ok if $T_1 \leq S_1$ and $S_2 \leq T_2$
 - Note the direction of subtyping on $T_1 \leq S_1$
- Example:
 - We had this code:
`ui.setKeyListener(fun (e: KeyEvent) : Object { c.incr(1) });`
 - Another correct variant:
`ui.setKeyListener(fun (e: Object) : Int { c.incr(1) });`
 - Because $\text{Object} \rightarrow \text{Int} \leq \text{KeyEvent} \rightarrow \text{Object}$

Prof. Necula CS 164

20

Least-Upper-Bound with Function Types

- $\text{lub}(T_1 \rightarrow T_2, C)$ is not defined
 - It is an error to return a function in a branch and an object in another
- $\text{lub}(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = \text{glb}(T_1, S_1) \rightarrow \text{lub}(T_2, S_2)$
 - $\text{glb}(T_1, S_1)$ is the greatest lower bound of T_1 and S_1
 - $\text{glb}(T_1, S_1) \leq T_1$ and $\text{glb}(T_1, S_1) \leq S_1$
- $\text{glb}(C, D) = C$ if $D \leq C$ and undefined otherwise
- $\text{glb}(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = \text{lub}(T_1, S_1) \rightarrow \text{glb}(T_2, S_2)$

Overview

- ✓ Why higher-order languages ?
- ✓ Higher-order constructs for Cool
- ✓ Type checking
- Code generation

Source-to-Source Translation

- We are adding a feature to the language
 - But we are not extending the sets of programs we can write
 - We only make it more convenient to write some programs
- We can use a new code generation strategy:
 - Instead of generating assembly language
 - We compile Cool programs with higher-order functions into Cool programs without higher-order functions.
 - Then, we can use the existing Cool compiler, debugger, runtime system, ...
- This is called source-to-source translation
 - Can be done at the AST level, after semantic analysis
- This is how Java implements inner classes
 - Backwards compatible with old virtual machines

Code Generation for Functions

- Function expression have delicate scoping issues
 - The body of the function may refer to self, attributes, local variables
- We are going to consider several cases in turn
- Case 1: Function body does not refer to self, attributes or locals
- Case 2: Function body does refer to self and attributes, but not to locals
- Case 3: The full house

Global Functions

- Case 1: Functions whose body does not refer to self, attributes, or local variables
 - ... `let f: Int → Int <- fun (x: Int) : Int { x + 1 } in f(5) ...`
- Such functions could just as well appear at top-level
 - Like functions in C or C++
- We could generate code like for a regular function
- Or, we compile this into Cool
 - We create an adapter (proxy object) for the function

```
class MyFun { apply(x: Int) : Int { x + 1 } }  
... let fa: MyFun <- new MyFun in fa.apply(5) ...
```

Prof. Necula CS 164

25

Member Functions

- Case 2: Functions that do not access local variables
 - But may access self and attributes
- ```
class Counter {
 count: Int <- 0;
 listener(): Object → Object {
 fun (x: Object): Object { count <- count + 1 }
 }
}
... let f: Object → Object = (new Counter).listener () in f(1) ...
```
- The proxy object must have access to the host Counter

Prof. Necula CS 164

26

## Translation of Member Functions

---

- Code from previous slide is translated as follows:

```
class Counter {
 count: Int <- 0;
 get_count(): Int { count }
 set_count(x: Int) { count <- x }
 listener(): CounterListener { (new CounterListener).init(self) }
}
class CounterListener {
 host: Counter;
 init(c: Counter) { host <- c }
 apply(x: Object): Object { host.set_count(host.get_count() + 1) }
}
... let fp: CounterListener <- (new Counter).listener () in fp.apply(1) ...
```

Prof. Necula CS 164

27

## Translation of Member Functions. Notes.

---

- The compiler generates the code for the adapter classes
  - The programmer does not have to write, or maintain it!
- This is how Java inner classes are compiled
- The compiler might have to add access methods for private fields
- This is a security hole in Java:
  - if your class has inner classes that access the private fields, they will be made accessible
  - Same might happen even if a subclass of your class has inner classes!
  - This could be fixed by not doing source-to-source translation

Prof. Necula CS 164

28

## Nested Functions

---

- Case 3: Functions that access local variables

```
let incr: Int <- 3 in
 let f: Int → Int <- fun (x: Int) { x + incr } in ...; f(2)
```

- The proxy object for "f" must have access to incr:

```
class MyFun {
 host: H; incr: Int;
 init(h: Host, i: Int) { host <- h; incr <- i; };
 apply(x: Int) { x + incr }
}
... let incr: Int <- 3 in
 let fp: MyFun <- (new MyFun).init(self, incr) in ...; fp.apply(2)
```

Prof. Necula CS 164

29

## Complications with Nested Functions

---

- Consider the code

```
let incr: Int <- 3 in
 let f: Int → Int <- fun (x: Int) { incr <- 4; x + incr } in
 incr <- 5; f(2)
```

- Does "f" see the "incr <- 5" ? Does code see the "incr <- 4"?

- In our compilation scheme the proxy has a copy of `incr`
  - Proxy does not see the changes to `incr` after it was created
  - Code does not see the changes to `incr` made in the proxy
  - In code above the result is  $2 + 4 = 6$

Prof. Necula CS 164

30

## Complications with Nested Functions

---

- Changes to attributes are visible because they are made in the same object
- Java inner classes can only access "final" local variables
  - Cannot be assigned-to in the code nor in the proxy
- Similar restrictions in all higher-order languages
  - Functional languages even discourage "assignable" variables

## Conclusions

---

- Higher-order functions can lead to more compact and elegant code
  - Heavily used in graphical toolkits, callbacks, iterators, ...
- We can combine the OO and functional paradigm
- Source-to-source translation is an easy way to add language features