Elements of Programming Languages

Lecture 12: Imperative programming

James Cheney

University of Edinburgh

November 6, 2017

The story so far

- So far we've mostly considered *pure* computations.
- Once a variable is bound to a value, the value *never changes*.
  - that is, variables are *immutable*.
- This is **not** how most programming languages treat variables!
  - In most languages, we can *assign* new values to variables: that is, variables are *mutable* by default
- Just a few languages are completely "pure" (Haskell).
- Others strike a balance:
  - e.g. Scala distinguishes immutable (`val`) variables and mutable (`var`) variables
  - similarly `const` in Java, C

Mutable vs. immutable

- Advantages of immutability:
  - Referential transparency (substitution of equals for equals); programs easier to reason about and optimize
  - Types tell us more about what a program can/cannot do
- Advantages of mutability:
  - Some common data structures easier to implement
  - Easier to translate to machine code (in a performance-preserving way)
  - Seems closely tied to popular OOP model of "objects with hidden state and public methods"
- Today we'll consider programming with assignable variables and loops (L_{While}) and then discuss procedures and other forms of control flow

While-programs

- Let's start with a simple example: L_{While}, with *statements*

$$\begin{array}{rcl} \mathit{Stmt} \ni s & ::= & \texttt{skip} \mid s_1; s_2 \mid x := e \\ & \mid & \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \mid \texttt{while } e \texttt{ do } s \end{array}$$

- `skip` does nothing
- $s_1; s_2$ does s_1, then s_2
- $x := e$ evaluates e and **assigns** the value to x
- `if` e `then` s_1 `else` s_2 evaluates e, and evaluates s_1 or s_2 based on the result.
- `while` e `do` s tests e. If true, evaluate s and **loop**; otherwise stop.
- We typically use $\{\}$ to parenthesize statements.

A simple example: factorial again

- In Scala, mutable variables can be defined with `var`

```
var n = ...
var x = 1
while(n > 0) {
  x = n * x
  n = n-1
}
```

- In L_{While}, all variables are mutable

$$x := 1; \texttt{while}\ (n > 0)\ \texttt{do}\ \{x := n * x; n := n - 1\}$$

An interpreter for L_{While}

We will define a *pure* interpreter:

```
def exec(env: Env[Value], s: Stmt): Env[Value] =
s match {
  case Skip => env
  case Seq(s1,s2) =>
    val env1 = exec(env, s1)
    exec(env1,s2)
  case IfThenElseS(e,s1,s2) => eval(env,e) match {
    case BoolV(true) => exec(env,s1)
    case BoolV(false) => exec(env,s2)
   }
  ...
}
```


An interpreter for L_{While}

```
def exec(env: Env[Value], s: Stmt): Env[Value] =
s match {
  ...
  case WhileDo(e,s) => eval(env, e) match {
    case BoolV(true) =>
      val env1 = exec(env,s)
      exec(env1, WhileDo(e,s))
    case BoolV(false) => env
  }
  case Assign(x,e) =>
    val v = eval(env,e)
    env + (x -> v)
}
```


While-programs: evaluation

$\boxed{\sigma, s \Downarrow \sigma'}$

$$\frac{}{\sigma, \texttt{skip} \Downarrow \sigma} \qquad \frac{\sigma, s_1 \Downarrow \sigma' \quad \sigma', s_2 \Downarrow \sigma''}{\sigma, s_1; s_2 \Downarrow \sigma''}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s_1 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma'} \qquad \frac{\sigma, e \Downarrow \texttt{false} \quad \sigma, s_2 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma'}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s \Downarrow \sigma' \quad \sigma', \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma''}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma''}$$

$$\frac{\sigma, e \Downarrow \texttt{false}}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma} \qquad \frac{\sigma, e \Downarrow v}{\sigma, x := e \Downarrow \sigma[x := v]}$$

- Here, we use evaluation in context $\sigma, e \Downarrow v$ (cf. Assignment 2)

Examples

- $x := y + 1; z := 2 * x$

$$\dfrac{\dfrac{\sigma_1, y+1 \Downarrow 2}{\sigma_1, x := y+1 \Downarrow \sigma_2} \quad \dfrac{\sigma_2, 2*x \Downarrow 4}{\sigma_2, z := 2*x \Downarrow \sigma_3}}{\sigma_1, x := y+1; z := 2*x \Downarrow \sigma_3}$$

- where

$$\begin{aligned} \sigma_1 &= [y := 1] \\ \sigma_2 &= [x := 2, y := 1] \\ \sigma_3 &= [x := 2, y := 1, z := 4] \end{aligned}$$

Other control flow constructs

- We've taken "if" (with both "then" and "else" branches) and "while" to be primitive
- We can **define** some other operations in terms of these:

$$\begin{aligned} \texttt{if } e \texttt{ then } s &\iff \texttt{if } e \texttt{ then } s \texttt{ else skip} \\ \texttt{do } s \texttt{ while } e &\iff s; \texttt{while } e \texttt{ do } s \\ \texttt{for } (i \in n \ldots m) \texttt{ do } s &\iff i := n; \\ & \qquad \texttt{while } i \leq m \texttt{ do } \{ \\ & \qquad\quad s; i = i + 1 \\ & \qquad \} \end{aligned}$$

- as seen in C, Java, etc.

Procedures

- L_{While} is not a realistic language.
- Among other things, it lacks *procedures*
- Example (C/Java):

```
int fact(int n) {
  int x = 1;
  while(n > 0) {
    x = x*n;
    n = n-1;
  }
  return x;
}
```

- Procedures can be added to L_{While} (much like functions in L_{Rec})
- Rather than do this, we'll show how to combine L_{While} with L_{Rec} later.

Structured vs. unstructured programming [Non-examinable]

- All of the languages we've seen so far are *structured*
  - meaning, control flow is managed using `if`, `while`, procedures, functions, etc.
- However, low-level machine code doesn't have any of these.
- A machine-code program is just a sequence of instructions in memory
- The only control flow is branching:
  - "unconditionally go to instruction at address *n*"
  - "if some condition holds, go to instruction at address *n*"
- Similarly, "goto" statements were the main form of control flow in many early languages

"GO TO" Considered Harmful [Non-examinable]

- In a famous letter (CACM 1968), Dijkstra listed many disadvantages of "goto" and related constructs
- It allows you to write "spaghetti code", where control flow is very difficult to decipher
- For efficiency/historical reasons, many languages include such "unstructured" features:
  - "goto" — jump to a specific program location
  - "switch" statements
  - "break" and "continue" in loops
- It's important to know about these features, their pitfalls and their safe uses.

goto in C [Non-examinable]

- The C (and C++) language includes `goto`
- In C, `goto L` jumps to the statement labeled `L`
- A typical (relatively sane) use of `goto`

```
... do some stuff ...
  if (error) goto error;
... do some more stuff ...
  if (error2) goto error;
... do some more stuff...
error: .. handle the error...
```

- We'll see other, better-structured ways to do this using exceptions.

goto in C: pitfalls [Non-examinable]

- The scope of the `goto L` statement and the target `L` might be different
- for that matter, they might not even be in the same procedure!
- For example, what does this do:

```
goto L;
if(1) {
    int k = fact(3);
L:  printf("%d",k);
}
```

- Answer: `k` will be some random value!

goto: caveats [Non-examinable]

- `goto` can be used safely in C, but is best avoided unless you have a really good reason
- e.g. very high performance/systems code
- Safe use: within same procedure/scope
- Or: to jump "out" of a nested loop

goto fail [Non-examinable]

- What's wrong with this picture?

```
if (error test 1)
  goto fail;
if (error test 2)
  goto fail;
  goto fail;
if (error test 3)
  goto fail;
...
fail:  ... handle error ...
```

- (In C, braces on `if` are optional; if they're left out, only the first `goto fail` statement is conditional!)
- This led to an Apple SSL security vulnerability in 2014 (see https://gotofail.com/)

switch statements [Non-examinable]

- We've seen `case` or `match` constructs in Scala
- The `switch` statement in C, Java, etc. is similar:

```
switch (month) {
  case 1: print("January"); break;
  case 2: print("February"); break;
  ...
  default: print("unknown month"); break;
}
```

- However, typically the argument must be a base type like `int`

switch statements: gotchas [Non-examinable]

- See the `break;` statement?
- It's an important part of the control flow!
  - it says "now jump out the end of the switch statement"

```
month = 1;
switch (month) {
  case 1: print("January");
  case 2: print("February");
  ...
  default: print("unknown month");
} // prints all months!
```

- Can you think of a good reason why you would want to leave out the `break`?

Break and continue [Non-examinable]

- The `break` and `continue` statements are also allowed in loops in C/Java family languages.

```
for(i = 0; i < 10; i++) {
  if (i % 2 == 0) continue;
  if (i == 7) break;
  print(i);
}
```

- "Continue" says *Skip the rest of this iteration of the loop.*
- "Break" says *Jump to the next statement after this loop*
- This will print 135 and then exit the loop.

Labeled break and continue [Non-examinable]

- In Java, `break` and `continue` can use labels.

```
OUTER: for(i = 0; i < 10; i++) {
  INNER:  for(j = 0; j < 10; j++) {
             if (j > i) continue INNER;
             if (i == 4) break OUTER;
             print(j);
  }
}
```

- This will print `001012` and then exit the loop.
- (Labeled) break and continue accommodate some of the safe uses of `goto` without as many sharp edges

Summary

- Many real-world programming languages have:
  1. mutable state
  2. structured control flow (if/then, while, exceptions)
  3. procedures
- We've showed how to model and interpret L_{While}, a simple imperative language
- and discussed a variety of (unstructured) control flow structures, such as "goto", "switch" and "break/continue".
- Next time:
  - Small-step semantics and type soundness