Elements of Programming Languages
Lecture 3: Booleans, conditionals, and types

James Cheney
University of Edinburgh
September 30, 2016

Boolean expressions

- So far we've considered only a trivial arithmetic language $L_{\text{Arith}}$
- Let's extend $L_{\text{Arith}}$ with equality tests and Boolean true/false values:

\[ e ::= \cdots \mid b \in B \mid e_1 == e_2 \]

- We write $B$ for the set of Boolean values \{true, false\}
- Basic idea: $e_1 == e_2$ should evaluate to true if $e_1$ and $e_2$ have equal values, false otherwise

What use is this?

- Examples:
  - $2 + 2 == 4$ should evaluate to true
  - $3 \times 3 + 4 \times 4 == 5 \times 5$ should evaluate to true
  - $3 \times 3 == 4 \times 7$ should evaluate to false
  - How about true == true? Or false == true?
- So far, there's not much we can do.
- We can evaluate a numerical expression for its value, or a Boolean equality expression to true or false
- We can't write an expression whose result depends on evaluating a comparison.
  - We lack an “if then else” (conditional) operation.
- We also can't “and”, “or” or negate Boolean values.

Conditionals

- Let's also add an “if then else” operation:

\[ e ::= \cdots \mid b \in B \mid e_1 == e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \]

- We define $L_{\text{If}}$ as the extension of $L_{\text{Arith}}$ with booleans, equality and conditionals.
- Examples:
  - if true then 1 else 2 should evaluate to 1
  - if 1 + 1 == 2 then 3 else 4 should evaluate to 3
  - if true then false else true should evaluate to false
- Note that if $e$ then $e_1$ else $e_2$ is the first expression that makes nontrivial “choices”: whether to evaluate the first or second case.
Booleans and Conditionals Types

Extending evaluation

- We consider the Boolean values true and false to be values:
  \[ v ::= n \in \mathbb{N} \mid b \in \mathbb{B} \]

- and we add the following evaluation rules:

  \[
  \begin{align*}
  & e_1 \downarrow v_1, e_2 \downarrow v_2 \quad e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \\
  & e_1 == e_2 \downarrow \text{true} \quad e_1 == e_2 \downarrow \text{false} \\
  & e \downarrow \text{true} \quad e_1 \downarrow v_1 \quad e \downarrow \text{false} \quad e_2 \downarrow v_2 \\
  & \text{if } e \text{ then } e_1 \text{ else } e_2 \downarrow v_1 \quad \text{if } e \text{ then } e_1 \text{ else } e_2 \downarrow v_2
  \end{align*}
  \]

Extending the interpreter

- To interpret \( L_{\text{if}} \), we need new expression forms:

  ```
  case class Bool(n: Boolean) extends Expr
  case class Eq(e1: Expr, e2:Expr) extends Expr
  case class IfThenElse(e: Expr, e1: Expr, e2: Expr) extends Expr
  ```

- and different types of values (not just Ints):

  ```
  abstract class Value
  case class NumV(n: Int) extends Value
  case class BoolV(b: Boolean) extends Value
  ```

  (Technically, we could encode booleans as integers, but in general we will want to separate out the kinds of values.)

---

// helpers

``` scala
def add(v1: Value, v2: Value): Value =
  (v1,v2) match {
    case (NumV(v1), NumV(v2)) => NumV (v1 + v2)
    case (BoolV(b1), BoolV(b2)) => BoolV(b1 == b2)
  }

def mult(v1: Value, v2: Value): Value = ...

def eval(e: Expr): Value = e match {
  // Arithmetic
  case Num(n) => NumV(n)
  case Plus(e1,e2) => add(eval(e1),eval(e2))
  case Times(e1,e2) => mult(eval(e1),eval(e2))
  ...
}
```
Aside: Other Boolean operations

- We can add Boolean and, or and not operations as follows:
  \[ e ::= \cdots | e_1 \land e_2 | e_1 \lor e_2 | \neg(e) \]
- with evaluation rules:
  \[
  \begin{array}{c|c|c}
  e_1 \downarrow v_1 & e_2 \downarrow v_2 & e_1 \land e_2 \downarrow v_1 \land_B v_2 \\
  e_1 \downarrow v_1 & e_2 \downarrow v_2 & e_1 \lor e_2 \downarrow v_1 \lor_B v_2 \\
  \end{array}
  \]
- where again, \( \land_B \) and \( \lor_B \) are the mathematical “and” and “or” operations
- These are definable in L_{If}, so we will leave them out to avoid clutter.

What else can we do?

- We can also do strange things like this:
  \[ e_1 = 1 + (2 == 3) \]
- Or this:
  \[ e_2 = \text{if } 1 \text{ then } 2 \text{ else } 3 \]
  What should these expressions evaluate to?
- There is no \( v \) such that \( e_1 \downarrow v \) or \( e_2 \downarrow v \):
  - the Totality property for L_{Arith} fails, for L_{If}!
- If we try to run the interpreter: we just get an error

One answer: Conversions

- In some languages (notably C, Java), there are built-in conversion rules
  - For example, “if an integer is needed and a boolean is available, convert \text{true} to 1 and \text{false} to 0”
  - Likewise, “if a boolean is needed and an integer is available, convert 0 to \text{false} and other values to \text{true}”
- LISP family languages have a similar convention: if we need a Boolean value, \text{nil} stands for “false” and any other value is treated as “true”
- Conversion rules are convenient but can make programs less predictable
- We will avoid them for now, but consider principled ways of providing this convenience later on.
Booleans and Conditionals

Another answer: Types

Should programs like:

\[ 1 + (2 == 3) \text{ if } 1 \text{ then } 2 \text{ else } 3 \]

even be allowed?

Idea: use a type system to define a subset of “well-formed” programs

Well-formed means (at least) that at run time:

- arguments to arithmetic operations (and equality tests) should be numeric values
- arguments to conditional tests should be Boolean values

Typing rules, informally: arithmetic

Consider an expression \( e \)

- If \( e = n \), then \( e \) has type “integer”
- If \( e = e_1 + e_2 \), then \( e_1 \) and \( e_2 \) must have type “integer”.
  If so, \( e \) has type “integer” also, else error.
- If \( e = e_1 \times e_2 \), then \( e_1 \) and \( e_2 \) must have type “integer”.
  If so, \( e \) has type “integer” also, else error.

Typing rules, informally: booleans, equality and conditionals

Consider an expression \( e \)

- If \( e = \text{true or false} \), then \( e \) has type “boolean”
- If \( e = e_1 == e_2 \), then \( e_1 \) and \( e_2 \) must have the same type.
  If so, \( e \) has type “boolean”, else error.
- If \( e = \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \), then \( e_0 \) must have type “boolean”, and \( e_1 \) and \( e_2 \) must have the same type. If so, then \( e \) has the same type as \( e_1 \) and \( e_2 \), else error.

Note 1: Equality arguments have the same (unknown) type.

Note 2: Conditional branches have the same (unknown) type. This type determines the type of the whole conditional expression.

Concise notation for typing rules

We can define the possible types using a BNF grammar, as follows:

\[
\text{Type} \ni \tau ::= \text{int} \mid \text{bool}
\]

For now, we will consider only two possible types, “integer” (int) and “boolean” (bool).

We can also use rules to describe the types of expressions:

Definition (Typing judgment \( \vdash e : \tau \))

We use the notation \( \vdash e : \tau \) to say that \( e \) is a well-formed term of type \( \tau \) (or “\( e \) has type \( \tau \)”).
Booleans and Conditionals Types

Typing rules, more formally: arithmetic

- If \( e = n \), then \( e \) has type “integer”
- If \( e = e_1 + e_2 \), then \( e_1 \) and \( e_2 \) must have type “integer”. If so, \( e \) has type “integer” also, else error.
- If \( e = e_1 \times e_2 \), then \( e_1 \) and \( e_2 \) must have type “integer”. If so, \( e \) has type “integer” also, else error.

\[ \vdash e : \tau \text{ for } L_{\text{Arith}} \]

\( n \in \mathbb{N} \)

\[ \vdash n : \text{int} \]

\[ \vdash e_1 : \text{int} \]

\[ \vdash e_2 : \text{int} \]

\[ \vdash e_1 + e_2 : \text{int} \]

\[ \vdash e_1 \times e_2 : \text{int} \]

Typing rules, more formally: equality and conditionals

We indicate that the types of subexpressions of \( == \) must be equal by using the same \( \tau \)

Similarly, we indicate that the result of a conditional has the same type as the two branches using the same \( \tau \) for all three.

\[ \vdash e_1 : \tau \]

\[ \vdash e_2 : \tau \]

\[ \vdash e_1 == e_2 : \text{bool} \]

\[ \vdash e : \text{bool} \]

\[ \vdash e_1 : \tau \]

\[ \vdash e_2 : \tau \]

\[ \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \]

Typing judgments: examples

\[ \vdash 1 : \text{int} \]

\[ \vdash 2 : \text{int} \]

\[ \vdash 1 + 2 : \text{int} \]

\[ \vdash 4 : \text{int} \]

\[ \vdash 1 + 2 == 4 : \text{bool} \]

\[ \vdash \text{if } 1 + 2 == 4 \text{ then } 42 \text{ else } 17 : \text{int} \]

Typing judgments: non-examples

But we also want some things not to typecheck:

\[ \vdash 1 == \text{true} : \tau \]

\[ \vdash \text{if } 42 \text{ then } e_1 \text{ else } e_2 : \tau \]

These judgments do not hold for any \( e_1, e_2, \tau \).
Fundamental property of typing

- The point of the typing judgment is to ensure soundness: if an expression is well-typed, then it evaluates “correctly.”
- That is, evaluation is well-behaved on well-typed programs.

**Theorem (Type soundness for L_if)**

\[
\text{if } \vdash e : \tau \text{ then } e \Downarrow v \text{ and } \vdash v : \tau.
\]

- For a language like L_if, soundness is fairly easy to prove by induction on expressions. We’ll present soundness for more realistic languages in detail later.

Static vs. dynamic typing

- Some languages proudly advertise that they are “static” or “dynamic.”
- **Static typing:**
  - not all expressions are well-formed; some sensible programs are not allowed
  - types can be used to catch errors, improve performance
- **Dynamic typing:**
  - all expressions are well-formed; any program can be run
  - type errors arise dynamically; higher overhead for tagging and checking

- These are rarely-realized extremes: most “statically” typed languages handle some errors dynamically.
- In contrast, any “dynamically” typed language can be thought of as a statically typed one with just one type.

Summary

- In this lecture we covered:
  - Boolean values, equality tests and conditionals
  - Extending the interpreter to handle them
  - Typing rules
- Next time:
  - Variables and let-binding
  - Substitution, environments and type contexts