Elements of Programming Languages

Lecture 3: Booleans, conditionals, and types

James Cheney

University of Edinburgh

September 28, 2023

Boolean expressions

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

$$e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2$$

- We write $\mathbb{B}$ for the set of Boolean values $\{\mathtt{true}, \mathtt{false}\}$
- 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:

$$e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$$

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

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:

$\boxed{e \Downarrow v}$ for L_{If}

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v}{e_1 == e_2 \Downarrow \mathtt{true}} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \neq v_2}{e_1 == e_2 \Downarrow \mathtt{false}}$$

$$\frac{e \Downarrow \mathtt{true} \quad e_1 \Downarrow v_1}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_1} \qquad \frac{e \Downarrow \mathtt{false} \quad e_2 \Downarrow v_2}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_2}$$

Extending the interpreter

- To interpret L_{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.)

Extending the interpreter

```
// helpers
def add(v1: Value, v2: Value): Value =
      (v1,v2) match {
        case (NumV(v1), NumV(v2)) => NumV (v1 + v2)
      }
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))
   ... }
```

Extending the interpreter

```
// helper
def eq(v1: Value, v2: Value): Value = (v1,v2) match {
    case (NumV(n1), NumV(n2)) => BoolV(n1 == n2)
    case (BoolV(b1), BoolV(b2)) => BoolV(b1 == b2)
}
def eval(e: Expr): Value = e match {
  ...
  case Bool(b) => BoolV(b)
  case Eq(e1,e2) => eq (eval(e1), eval(e2))
  case IfThenElse(e,e1,e2) => eval(e) match {
    case BoolV(true) => eval(e1)
    case BoolV(false) => eval(e2)
  }
}
```

Aside: Other Boolean operations

- We can add Boolean and, or and not operations as follows:

$$e ::= \cdots \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg(e)$$

- with evaluation rules:

$\boxed{e \Downarrow v}$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \wedge e_2 \Downarrow v_1 \wedge_{\mathbb{B}} v_2} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \vee e_2 \Downarrow v_1 \vee_{\mathbb{B}} v_2}$$

- where again, $\wedge_{\mathbb{B}}$ and $\vee_{\mathbb{B}}$ are the mathematical "and" and "or" operations
- These are definable in L_{If}, so we will leave them out to avoid clutter.

Aside: Shortcut operations

- Many languages (e.g. C, Java) offer *shortcut* versions of "and" and "or":

$$e ::= \cdots \mid e_1 \;\texttt{\&\&}\; e_2 \mid e_1 \;\texttt{||}\; e_2$$

- e_1 && e_2 stops early if e_1 is false (since e_2's value then doesn't matter).
- e_1 || e_2 stops early if e_1 is true (since e_2's value then doesn't matter).
- We can model their semantics using rules like this:

$$\frac{e_1 \Downarrow \texttt{false}}{e_1 \;\texttt{\&\&}\; e_2 \Downarrow \texttt{false}} \qquad \frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow v_2}{e_1 \;\texttt{\&\&}\; e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \texttt{true}}{e_1 \;\texttt{||}\; e_2 \Downarrow \texttt{true}} \qquad \frac{e_1 \Downarrow \texttt{false} \quad e_2 \Downarrow v_2}{e_1 \;\texttt{||}\; e_2 \Downarrow v_2}$$

What else can we do?

- We can also do strange things like this:

$$e_1 = 1 + (2 == 3)$$

- Or this:

$$e_2 = \texttt{if}\ 1\ \texttt{then}\ 2\ \texttt{else}\ 3$$

  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 $\mathsf{L}_{\mathsf{Arith}}$ 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 `true` to 1 and `false` to 0"
  - Likewise, "if a boolean is needed and an integer is available, convert 0 to `false` and other values to `true`"
  - LISP family languages have a similar convention: if we need a Boolean value, `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.

Another answer: Types

- Should programs like:

$$1 + (2 == 3) \qquad \texttt{if } 1 \texttt{ then } 2 \texttt{ else } 3$$

  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 $e =$ `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 $e =$ `if` e_0 `then` e_1 `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:

$$\mathit{Type} \ni \tau ::= \mathtt{int} \mid \mathtt{bool}$$

  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 τ (or "e has type τ").

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.

$\boxed{\vdash e : \tau}$ for $\mathrm{L}_{\mathrm{Arith}}$

$$\frac{n \in \mathbb{N}}{\vdash n : \mathtt{int}} \qquad \frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 + e_2 : \mathtt{int}}$$

$$\frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 \times e_2 : \mathtt{int}}$$

Typing rules, more formally: equality and conditionals

$\vdash e : \tau$ for $\mathrm{L_{If}}$

$$\frac{b \in \mathbb{B}}{\vdash b : \texttt{bool}} \qquad \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{\vdash e : \texttt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

- We indicate that the types of subexpressions of $==$ must be equal by using the same τ
- Similarly, we indicate that the result of a conditional has the same type as the two branches using the same τ for all three

Typing judgments: examples

$$\dfrac{\dfrac{\overline{\vdash 1 : \texttt{int}} \quad \overline{\vdash 2 : \texttt{int}}}{\vdash 1 + 2 : \texttt{int}} \qquad \overline{\vdash 4 : \texttt{int}}}{\vdash 1 + 2 == 4 : \texttt{bool}}$$

$$\dfrac{\begin{array}{c}\vdots \\ \vdash 1 + 2 == 4 : \texttt{bool}\end{array} \quad \overline{\vdash 42 : \texttt{int}} \quad \overline{\vdash 17 : \texttt{int}}}{\vdash \texttt{if } 1 + 2 == 4 \texttt{ then } 42 \texttt{ else } 17 : \texttt{int}}$$

$$\dfrac{\begin{array}{c}\vdots \\ \vdash \texttt{if } 1 + 2 == 4 \texttt{ then } 42 \texttt{ else } 17 : \texttt{int}\end{array} \quad \overline{\vdash 100 : \texttt{int}}}{\vdash (\texttt{if } 1 + 2 == 4 \texttt{ then } 42 \texttt{ else } 17) + 100 : \texttt{int}}$$

Typing judgments: non-examples

But we also want some things **not** to typecheck:

$$\vdash 1 == \texttt{true} : \tau$$

$$\vdash \texttt{if } 42 \texttt{ then } e_1 \texttt{ else } e_2 : \tau$$

These judgments do not hold for any e_1, e_2, τ.

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})

If $\vdash e : \tau$ *then* $e \Downarrow v$ *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