Elements of Programming Languages

Lecture 2: Evaluation

James Cheney

University of Edinburgh

September 29, 2022
Overview

- Last time:
  - Concrete vs. abstract syntax
  - Programming with abstract syntax trees
  - A taste of induction over expressions

- Today:
  - Evaluation
  - A simple interpreter
  - Modeling evaluation using rules
Values

- Recall $L_{\text{Arith}}$ expressions:

  $\text{Expr} \ni e ::= e_1 + e_2 \mid e_1 \times e_2 \mid n \in \mathbb{N}$

- Some expressions, like 1,2,3, are special
- They have no remaining “computation” to do
- We call such expressions \textit{values}.
- We can define a BNF grammar rule for values:

  $\text{Value} \ni v ::= n \in \mathbb{N}$
Evaluation, informally

- Given an expression $e$, what is its value?
  - If $e = n$, a number, then it is already a value.
  - If $e = e_1 + e_2$, evaluate $e_1$ to $v_1$ and $e_2$ to $v_2$. Then add $v_1$ and $v_2$, the result is the value of $e$.
  - If $e = e_1 \times e_2$, evaluate $e_1$ to $v_1$ and $e_2$ to $v_2$. Then multiply $v_1$ and $v_2$, the result is the value of $e$.  

Evaluation, in Scala

- If \( e = n \), a number, then it is already a value.
- If \( e = e_1 + e_2 \), evaluate \( e_1 \) to \( v_1 \) and \( e_2 \) to \( v_2 \). Then add \( v_1 \) and \( v_2 \), the result is the value of \( e \).
- If \( e = e_1 \times e_2 \), evaluate \( e_1 \) to \( v_1 \) and \( e_2 \) to \( v_2 \). Then multiply \( v_1 \) and \( v_2 \), the result is the value of \( e \).

```scala
def eval(e: Expr): Int = e match {
  case Num(n) => n
  case Plus(e1,e2) => eval(e1) + eval(e2)
  case Times(e1,e2) => eval(e1) * eval(e2)
}
```
Example

\[ \text{eval} \left( \frac{1}{+} \frac{2}{\times} \frac{3}{=} \right) = \text{eval}(1) + \text{eval} \]

\[ \left( \frac{2}{\times} \frac{3}{=} \right) \]
Example

\[
eval(1) + eval \left( \begin{array}{c} 
\times \\
2 \\
\end{array} 
\begin{array}{c} 
3 
\end{array} \right) = eval(1) + (eval(2) \times eval(3))
\]

\[
eval(1) + (eval(2) \times eval(3)) = 1 + (2 \times 3) = 1 + 6 = 7
\]
Expression evaluation, more formally

- To specify and reason about evaluation, we use an evaluation judgment.

**Definition (Evaluation judgment)**

Given expression $e$ and value $v$, we say $v$ is the value of $e$ if evaluating $e$ results in $v$, and we write $e \Downarrow v$ to indicate this.

(A judgment is a relation between abstract syntax trees.)

- Examples:
  
  $$1 + 2 \Downarrow 3 \quad 1 + 2 \times 3 \Downarrow 7 \quad (1 + 2) \times 3 \Downarrow 9$$
Evaluation of Values

- A value is already evaluated. So, for any \( v \), we have \( v \Downarrow v \).
- We can express the fact that \( v \Downarrow v \) always holds (for any \( v \)) as follows:

\[
\frac{\quad}{v \Downarrow v}
\]

- This is a rule that says that \( v \) evaluates to \( v \) always (no preconditions).
- So, for example, we can derive:

\[
0 \Downarrow 0 \quad 1 \Downarrow 1 \quad \ldots
\]
Evaluation of Addition

- How to evaluate expression $e_1 + e_2$?
- Suppose we know that $e_1 \downarrow v_1$ and $e_2 \downarrow v_2$.
- Then the value of $e_1 + e_2$ is the number we get by adding numbers $v_1$ and $v_2$.
- We can express this as follows:

$$\begin{align*}
  e_1 \downarrow v_1 & \quad e_2 \downarrow v_2 \\
  e_1 + e_2 \downarrow v_1 +_N v_2
\end{align*}$$

- This is a rule that says that $e_1 + e_2$ evaluates to $v_1 +_N v_2$ provided $e_1$ evaluates to $v_1$ and $e_2$ evaluates to $v_2$
- Note that we write $+_N$ for the mathematical function that adds two numbers, to avoid confusion with the abstract syntax tree $v_1 + v_2$. 
Expression evaluation: Summary

- Multiplication can be handled exactly like addition.
- We will define the meaning of $L_{\text{Arith}}$ expressions using the following rules:

- This evaluation judgment is an example of *big-step semantics* (or *natural semantics*)
  - so-called because we evaluate the whole expression “in one step”
Examples

- We can use these rules to *derive* evaluation judgments for complex expressions:

\[
\begin{align*}
1 & \Downarrow 1 & 1 & \Downarrow 1 & 1 & \Downarrow 1 &2 & \Downarrow 2 & 3 & \Downarrow 3 & 2 & \Downarrow 2 & 3 & \Downarrow 3 & 1 & \Downarrow 1 & 2 & \Downarrow 2 & 1 + 2 & \Downarrow 3 & 1 + (2 \times 3) & \Downarrow 7 & (1 + 2) \times 3 & \Downarrow 9 \\
\end{align*}
\]

- These figures are *derivation trees* showing how we can derive a conclusion from axioms.

- The rules govern how we can construct derivation trees.
  - A leaf node must match a rule with no preconditions.
  - Other nodes must match rules with preconditions. (Order matters.)

- Note that derivation trees “grow up” (root is at the bottom)
Totality and Structural induction

- Question: Given any expression $e$, does it evaluate to a value?
- To answer this question, we can use structural induction:

**Induction on structure of expressions**

Given a property $P$ of expressions, if:

- $P(n)$ holds for every number $n \in \mathbb{N}$
- for any expressions $e_1, e_2$, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 + e_2)$ also holds
- for any expressions $e_1, e_2$, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 \times e_2)$ also holds

Then $P(e)$ holds for all expressions $e$. 
Proof by structural induction

Let's illustrate with an example

**Theorem**

If $e$ is an expression, then there exists $v \in \mathbb{N}$ such that $e \Downarrow v$ holds.

**Proof: Base case.**

If $e = n$ then $e$ is already a value. Take $v = n$, then we can derive $e \Downarrow n$.
Proof by structural induction

Proof: Inductive case 1.

If \( e = e_1 + e_2 \) then suppose \( e_1 \downarrow v_1 \) and \( e_2 \downarrow v_2 \) for some \( v_1, v_2 \). Then we can use the rule:

\[
\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{e_1 + e_2 \downarrow v_1 +_N v_2}
\]

to conclude that there exists \( v = v_1 +_N v_2 \) such that \( e \downarrow v \) holds.

Note that again it’s important to distinguish \( v_1 +_N v_2 \) (the number) from \( v_1 + v_2 \) the expression.
Proof by structural induction

Proof: Inductive case 2.

If $e = e_1 \times e_2$ then suppose $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$ for some $v_1, v_2$. Then we can use the rule:

$$
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \times e_2 \Downarrow v_1 \times_\mathbb{N} v_2}
$$

to conclude that there exists $v = v_1 \times_\mathbb{N} v_2$ such that $e \Downarrow v$ holds.

- This case is basically identical to case 1 (modulo $+$ vs. $\times$).
- From now on we will typically skip over such “essentially identical” cases (but it is important to really check them).
We can also prove the uniqueness of the value of $v$ by induction:

**Theorem (Uniqueness of evaluation)**

If $e \Downarrow v$ and $e \Downarrow v'$, then $v = v'$.

**Base case.**

If $e = n$ then since $n \Downarrow v$ and $n \Downarrow v'$ hold, the only way we could derive these judgments is for $v, v'$ to both equal $n$. 

\[\square\]
Uniqueness

Inductive case.

If \( e = e_1 + e_2 \) then the derivations must be of the form

\[
\begin{align*}
  e_1 & \Downarrow v_1 & e_2 & \Downarrow v_2 \\
  e_1 + e_2 & \Downarrow v_1 +_N v_2 \\
\end{align*}
\]

By induction, \( e_1 \Downarrow v_1 \) and \( e_1 \Downarrow v_1' \) implies \( v_1 = v_1' \), and similarly for \( e_2 \) so \( v_2 = v_2' \). Therefore \( v_1 +_N v_2 = v_1' +_N v_2' \).

The proof for \( e_1 \times e_2 \) is similar.
Totality, uniqueness, and correctness

- The Scala interpreter code defined earlier says how to interpret a \text{L}_{\text{Arith}} expression as a function.
- The big-step rules, in contrast, specify the meaning of expressions as a relation.
- Nevertheless, \textit{totality} and \textit{uniqueness} guarantee that for each \( e \) there is a unique \( v \) such that \( e \downarrow v \).
- In fact, \( v = \text{eval}(e) \), that is:

**Theorem (Interpreter Correctness)**

\textit{For any \text{L}_{\text{Arith}} expression} \( e \), \textit{we have} \( e \downarrow v \) \textit{if and only if} \( v = \text{eval}(e) \).

- Proof: induction on \( e \).
Summary

- In this lecture, we’ve covered:
  - A simple interpreter
  - Evaluation via rules
  - Totality and uniqueness (via structural induction)
- all for the simple language $L_{\text{Arith}}$
- Next time:
  - Booleans, equality, conditionals
  - Types