Chapter 3: Untyped Arithmetic Expressions

A small language of numbers and booleans
Basic aspects of programming languages
Introduction

Grammar
Programs
Evaluation
Grammar (Syntax)

t ::=  
  true  
  false  
  if t then t else t  
  0  
  succ t  
  pred t  
  iszero t

t: meta-variable (non-terminal symbol)

terms:
  constant true
  constant false
  conditional constant
  zero
  successor
  predecessor
  zero test
Programs and Evaluations

- A program in the language is just a term built from the forms given by the grammar.

\[
\text{if false then 0 else 1} \quad (1 = \text{succ } 0) \\
\rightarrow 1
\]

\[
\text{iszero (pred (succ 0))} \\
\rightarrow \text{true}
\]
Syntax

Many ways of defining syntax (besides grammar)
Terms, Inductively

The set of terms is the **smallest set** $T$ such that

1. $\{\text{true}, \text{false}, 0\} \subseteq T$;
2. if $t_1 \in T$, then $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq T$;
3. if $t_1 \in T$, $t_2 \in T$, and $t_3 \in T$,
   then if $t_1$ then $t_2$ else $t_3 \in T$. 
Terms, by Inference Rules

The set of terms is defined by the following rules:

\[
\begin{align*}
\text{true} & \in \mathcal{T} \\
\text{false} & \in \mathcal{T} \\
0 & \in \mathcal{T} \\
\frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \\
\frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \\
\frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\
\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}}
\end{align*}
\]

Inference rules = Axioms + Proper rules
Terms, Concretely

For each natural number $i$, define a set $S_i$ as follows:

\[
S_0 = \emptyset \\
S_{i+1} = \{\text{true, false, 0}\} \\
\cup \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\
\cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}.
\]

Finally, let

\[
S = \bigcup_i S_i.
\]

Exercise [**]: How many elements does $S_3$ have?

Proposition: $T = S$
Induction on Terms

Inductive definitions
Inductive proofs
Inductive Definitions

The set of constants appearing in a term \( t \), written \( \text{Consts}(t) \), is defined as follows:

\[
\begin{align*}
\text{Consts}(\text{true}) & = \{\text{true}\} \\
\text{Consts}(\text{false}) & = \{\text{false}\} \\
\text{Consts}(0) & = \{0\} \\
\text{Consts}(\text{succ } t_1) & = \text{Consts}(t_1) \\
\text{Consts}(\text{pred } t_1) & = \text{Consts}(t_1) \\
\text{Consts}(\text{iszero } t_1) & = \text{Consts}(t_1) \\
\text{Consts}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) & = \text{Consts}(t_1) \cup \text{Consts}(t_2) \cup \text{Consts}(t_3)
\end{align*}
\]
Inductive Definitions

The size of a term $t$, written $\text{size}(t)$, is defined as follows:

\[
\begin{align*}
\text{size}(\text{true}) &= 1 \\
\text{size}(\text{false}) &= 1 \\
\text{size}(0) &= 1 \\
\text{size}(\text{succ } t_1) &= \text{size}(t_1) + 1 \\
\text{size}(\text{pred } t_1) &= \text{size}(t_1) + 1 \\
\text{size}(\text{iszero } t_1) &= \text{size}(t_1) + 1 \\
\text{size}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + 1
\end{align*}
\]
Inductive Definitions

The depth of a term \( t \), written \( \text{depth}(t) \), is defined as follows:

\[
\begin{align*}
\text{depth}(\text{true}) & = 1 \\
\text{depth}(\text{false}) & = 1 \\
\text{depth}(0) & = 1 \\
\text{depth}(\text{succ } t_1) & = \text{depth}(t_1) + 1 \\
\text{depth}(\text{pred } t_1) & = \text{depth}(t_1) + 1 \\
\text{depth}(\text{iszero } t_1) & = \text{depth}(t_1) + 1 \\
\text{depth}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) & = \max(\text{depth}(t_1), \text{depth}(t_2), \text{depth}(t_3)) + 1
\end{align*}
\]
Lemma. The number of distinct constants in a term $t$ is no greater than the size of $t$:
\[ |\text{Consts}(t) | \leq \text{size}(t) \]

Proof. By induction over the depth of $t$.
- Case $t$ is a constant
- Case $t$ is $\text{pred } t1$, $\text{succ } t1$, or $\text{iszero } t1$
- Case $t$ is if $t1$ then $t2$ else $t3$
Inductive Proof

**Theorem [Structural Induction]**

If, for each term $s$, given $P(r)$ for all immediate subterms $r$ of $s$ we can show $P(s)$, then $P(s)$ holds for all $s$. 
Semantic Styles

Three basic approaches
Operational Semantics

• Operational semantics specifies the behavior of a programming language by defining a simple abstract machine for it.

• An example (often used in this course):
  – terms as states
  – transition from one state to another as simplification
  – meaning of t is the final state starting from the state corresponding to t
Denotational Semantics

• Giving denotational semantics for a language consists of
  – finding a collection of semantic domains, and then
  – defining an interpretation function mapping terms into elements of these domains.

• Main advantage: It abstracts from the gritty details of evaluation and highlights the essential concepts of the language.
Axiomatic Semantics

• Axiomatic methods take the laws (properties) themselves as the definition of the language. The meaning of a term is just what can be proved about it.

  – They focus attention on the process of reasoning about programs.

  – Hoare logic: define the meaning of imperative languages
Evaluation

Evaluation relation (small-step/big-step)

Normal form

Confluence and termination
Evaluation on Booleans

**Syntax**

\[ t ::= \]

- true
- false
- if \( t \) then \( t \) else \( t \)

**Evaluation**

- \( t \rightarrow t' \) (E-IFTRUE)
- \( t \rightarrow t' \) (E-IFFALSE)
- \( \frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \) (E-IF)
One-step Evaluation Relation

• The one-step evaluation relation $\rightarrow$ is the smallest binary relation on terms satisfying the three rules in the previous slide.

• When the pair $(t,t')$ is in the evaluation relation, we say that “$t \rightarrow t'$ is derivable.”
“if $t$ then false else false $\rightarrow$ if $u$ then false else false” is witnessed by the following derivation tree:

where

\[
\begin{align*}
  s & \overset{\text{def}}{=} \text{if true then false else false} \\
  t & \overset{\text{def}}{=} \text{if } s \text{ then true else true} \\
  u & \overset{\text{def}}{=} \text{if false then true else true}
\end{align*}
\]
Induction on Derivation

**Theorem** [Determinacy of one-step evaluation]:
If $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$.

**Proof.** By *induction on derivation* of $t \rightarrow t'$.

If the last rule used in the derivation of $t \rightarrow t'$ is E-IfTrue, then $t$ has the form if true then $t_2$ else $t_3$.
It can be shown that there is only one way to reduce such $t$.

...
Normal Form

• **Definition**: A term $t$ is in **normal form** if no evaluation rule applies to it.

• **Theorem**: Every value is in normal form.

• **Theorem**: If $t$ is in normal form, then $t$ is a value.
  – Prove by contradiction (then by structural induction).
Multi-step Evaluation Relation

- **Definition**: The multi-step evaluation relation $\rightarrow^*$ is the reflexive, transitive closure of one-step evaluation.

- **Theorem** [Uniqueness of normal forms]: If $t \rightarrow^* u$ and $t \rightarrow^* u'$, where $u$ and $u'$ are both normal forms, then $u = u'$.

- **Theorem** [Termination of Evaluation]: For every term $t$ there is some normal form $t'$ such that $t \rightarrow^* t'$. 
Extending Evaluation to Numbers

New syntactic forms
\[ t ::= \ldots \]
- 0
- \text{succ } t
- \text{pred } t
- \text{iszero } t

\[ v ::= \ldots \]
- \text{nv}

\[ \text{nv} ::= \ldots \]
- 0
- \text{succ } \text{nv}

New evaluation rules
- terms:
  \[ t_1 \rightarrow t'_1 \]
- \text{succ } t_1 \rightarrow \text{succ } t'_1 \]
- \text{pred } 0 \rightarrow 0 \]
- \text{pred } (\text{succ } \text{nv}_1) \rightarrow \text{nv}_1 \]
- \text{pred } t_1 \rightarrow \text{pred } t'_1 \]
- \text{iszero } 0 \rightarrow \text{true} \]
- \text{iszero } (\text{succ } \text{nv}_1) \rightarrow \text{false} \]
- \text{iszero } t_1 \rightarrow \text{iszero } t'_1 \]

values:
- numeric value
- numeric values:
  - zero value
  - successor value
Stuckness

• **Definition**: A closed term is **stuck** if it is in normal form but not a value.

• **Examples**:
  – succ true
  – succ false
  – If zero then true else false
Big-step Evaluation

\[ v \downarrow v \]

\[ t_1 \downarrow \text{true} \quad t_2 \downarrow v_2 \]
\[ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \downarrow v_2 \]

\[ t_1 \downarrow \text{false} \quad t_3 \downarrow v_3 \]
\[ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \downarrow v_3 \]

\[ t_1 \downarrow \text{nv}_1 \]
\[ \text{succ } t_1 \downarrow \text{succ } \text{nv}_1 \]

\[ t_1 \downarrow 0 \]
\[ \text{pred } t_1 \downarrow 0 \]

\[ t_1 \downarrow \text{succ } \text{nv}_1 \]
\[ \text{pred } t_1 \downarrow \text{nv}_1 \]

\[ t_1 \downarrow 0 \]
\[ \text{iszero } t_1 \downarrow \text{true} \]

\[ t_1 \downarrow \text{succ } \text{nv}_1 \]
\[ \text{iszero } t_1 \downarrow \text{false} \]

(B-VALUE)

(B-IFTRUE)

(B-IFFALSE)

(B-SUCC)

(B-PREDZERO)

(B-PREDSUCC)

(B-ISZEROZERO)

(B-ISZEROSUCC)
Big-step vs small-step

• Big-step is usually easier to understand
  – called “natural semantics” in some articles
• Big-step often leads to simpler proof
• Big-step cannot describe computations that do not produce a value
  – Non-terminating computation
  – “Stuck” computation
Summary

- How to define syntax?
  - Grammar, Inductively, Inference Rules, Generative

- How to define semantics?
  - Operational, Denotational, Axomatic

- How to define evaluation relation (operational semantics)?
  - Small-step/Big-step evaluation relation
  - Normal form
  - Confluence/termination
Homework

- Do Exercise 3.5.16 in Chapter 3.

3.5.16 EXERCISE [RECOMMENDED, ★★★]: A different way of formalizing meaningless states of the abstract machine is to introduce a new term called wrong and augment the operational semantics with rules that explicitly generate wrong in all the situations where the present semantics gets stuck. To do this in detail, we introduce two new syntactic categories

\[
\begin{align*}
\text{badnat} & : = \quad \text{non-numeric normal forms:} \\
\text{wrong} & \quad \text{run-time error} \\
\text{true} & \quad \text{constant true} \\
\text{false} & \quad \text{constant false} \\
\text{badbool} & : = \quad \text{non-boolean normal forms:} \\
\text{wrong} & \quad \text{run-time error} \\
\text{nv} & \quad \text{numeric value}
\end{align*}
\]

and we augment the evaluation relation with the following rules:

\[
\begin{align*}
\text{if badbool then } t_1 \text{ else } t_2 & \rightarrow \text{ wrong} & \text{(E-IF-WRONG)} \\
\text{succ badnat} & \rightarrow \text{ wrong} & \text{(E-SUCC-WRONG)} \\
\text{pred badnat} & \rightarrow \text{ wrong} & \text{(E-PRED-WRONG)} \\
\text{iszero badnat} & \rightarrow \text{ wrong} & \text{(E-ISZERO-WRONG)}
\end{align*}
\]

Show that these two treatments of run-time errors agree by (1) finding a precise way of stating the intuition that “the two treatments agree,” and (2) proving it. As is often the case when proving things about programming languages, the tricky part here is formulating a precise statement to be proved—the proof itself should be straightforward.