Chapter 3: Untyped Arithmetic Expressions

A small language of numbers and Booleans

Basic aspects of programming languages

Introduction

Grammar

Programs

Evaluation

Grammar (Syntax)

t ::=	terms:
true	constant true
false	constant false
if t then t else t	conditional
0	constant zero
succ t	successor
pred t	predecessor
iszero t	zero test

t: meta-variable (non-terminal symbol)

Programs and Evaluations

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

if false then 0 else 1   (1 = succ 0)

→1

iszero (pred (succ 0))

→ true

Syntax

Many ways of defining syntax (besides grammar)

Terms, Inductively

The set of terms is the smallest set T such that

1. {true, false, 0} $\subseteq$ T;
2. if $t_1 \in T$, then {succ t_1, pred t_1, 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:

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

$$\frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{pred}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{iszero}\ t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in \mathcal{T}}$$

Inference rules = Axioms + Proper rules

Terms, Concretely

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

$$
\begin{aligned}
S_0 &= \emptyset \\
S_{i+1} &= \quad \{\texttt{true}, \texttt{false}, 0\} \\
&\cup \quad \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1 \mid t_1 \in S_i\} \\
&\cup \quad \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid t_1, t_2, t_3 \in S_i\}.
\end{aligned}
$$

Finally, let

$$S = \bigcup_i S_i.$$

3.2.4 EXERCISE [★★]: How many elements does S_3 have? □

3.2.5 EXERCISE [★★]: Show that the sets S_i are *cumulative*—that is, that for each i we have $S_i \subseteq S_{i+1}$. □

Induction on Terms

Inductive definitions

Inductive proofs

Inductive Definitions

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

$$
\begin{aligned}
\mathit{Consts}(\mathtt{true}) &= \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) &= \{\mathtt{false}\} \\
\mathit{Consts}(0) &= \{0\} \\
\mathit{Consts}(\mathtt{succ}\ t_1) &= \mathit{Consts}(t_1) \\
\mathit{Consts}(\mathtt{pred}\ t_1) &= \mathit{Consts}(t_1) \\
\mathit{Consts}(\mathtt{iszero}\ t_1) &= \mathit{Consts}(t_1) \\
\mathit{Consts}(\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3) &= \mathit{Consts}(t_1) \cup \mathit{Consts}(t_2) \cup \mathit{Consts}(t_3)
\end{aligned}
$$

Inductive Definitions

The size of a term t, written size(t), is defined as follows:

$$
\begin{aligned}
size(\mathtt{true}) &= 1 \\
size(\mathtt{false}) &= 1 \\
size(\mathtt{0}) &= 1 \\
size(\mathtt{succ}\ t_1) &= size(t_1) + 1 \\
size(\mathtt{pred}\ t_1) &= size(t_1) + 1 \\
size(\mathtt{iszero}\ t_1) &= size(t_1) + 1 \\
size(\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3) &= size(t_1) + size(t_2) + size(t_3) + 1
\end{aligned}
$$

Inductive Definitions

The depth of a term t, written depth(t), is defined as follows:

$$
\begin{array}{lcl}
\textit{depth}(\texttt{true}) & = & 1 \\
\textit{depth}(\texttt{false}) & = & 1 \\
\textit{depth}(\texttt{0}) & = & 1 \\
\textit{depth}(\texttt{succ } \texttt{t}_1) & = & \textit{depth}(\texttt{t}_1) + 1 \\
\textit{depth}(\texttt{pred } \texttt{t}_1) & = & \textit{depth}(\texttt{t}_1) + 1 \\
\textit{depth}(\texttt{iszero } \texttt{t}_1) & = & \textit{depth}(\texttt{t}_1) + 1 \\
\textit{depth}(\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3) & = & \max(\textit{depth}(\texttt{t}_1), \textit{depth}(\texttt{t}_2), \textit{depth}(\texttt{t}_3)) + 1
\end{array}
$$

Inductive Proof

Lemma. The number of distinct constants in a term t is no greater than the size of t:

$$|\,Consts(t)\,| \le size(t)$$

Proof. By induction over the depth of t.

- Case t is a constant
- Case t is pred t_1, succ t_1, or iszero t_1
- Case t is if t_1 then t_2 else t_3

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

$\texttt{t}$	::=		*terms:*
		$\texttt{true}$	*constant true*
		$\texttt{false}$	*constant false*
		$\texttt{if t then t else t}$	*conditional*
$\texttt{v}$	::=		*values:*
		$\texttt{true}$	*true value*
		$\texttt{false}$	*false value*

Evaluation $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \quad \text{(E-IfTrue)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \quad \text{(E-IfFalse)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \quad \text{(E-If)}$$

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."

Derivation Tree

"if t then false else false → if u then false else false" is witnessed by the following derivation tree:

$$\cfrac{\cfrac{\overline{\texttt{s} \longrightarrow \texttt{false}}\ \textsc{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \textsc{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \textsc{E-If}$$

where

$$\begin{aligned}
\texttt{s} &\stackrel{\text{def}}{=} \texttt{if true then false else false} \\
\texttt{t} &\stackrel{\text{def}}{=} \texttt{if s then true else true} \\
\texttt{u} &\stackrel{\text{def}}{=} \texttt{if false then true else true}
\end{aligned}$$

Induction on Derivation

Theorem [Determinacy of one-step evaluation]:

If $t \longrightarrow t'$ and $t \longrightarrow t''$, then $t' = t''$.

Proof. By induction on derivation of $t \longrightarrow t'$.

If the last rule used in the derivation of $t \longrightarrow t'$ 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 $\longrightarrow*$ is the reflexive, transitive closure of one-step evaluation.

- **Theorem** [Uniqueness of normal forms]: If t $\longrightarrow*$ u and t $\longrightarrow*$ 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 $\longrightarrow*$ t′.

Big-step Evaluation

$$\mathtt{v} \Downarrow \mathtt{v} \qquad \text{(B-VALUE)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{true} \qquad \mathtt{t_2} \Downarrow \mathtt{v_2}}{\mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3} \Downarrow \mathtt{v_2}} \qquad \text{(B-IFTRUE)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{false} \qquad \mathtt{t_3} \Downarrow \mathtt{v_3}}{\mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3} \Downarrow \mathtt{v_3}} \qquad \text{(B-IFFALSE)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{nv_1}}{\mathtt{succ}\ \mathtt{t_1} \Downarrow \mathtt{succ}\ \mathtt{nv_1}} \qquad \text{(B-SUCC)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{0}}{\mathtt{pred}\ \mathtt{t_1} \Downarrow \mathtt{0}} \qquad \text{(B-PREDZERO)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{succ}\ \mathtt{nv_1}}{\mathtt{pred}\ \mathtt{t_1} \Downarrow \mathtt{nv_1}} \qquad \text{(B-PREDSUCC)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{0}}{\mathtt{iszero}\ \mathtt{t_1} \Downarrow \mathtt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{\mathtt{t_1} \Downarrow \mathtt{succ}\ \mathtt{nv_1}}{\mathtt{iszero}\ \mathtt{t_1} \Downarrow \mathtt{false}} \qquad \text{(B-ISZEROSUCC)}$$

Extending Evaluation to Numbers

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\text{succ } t_1 \longrightarrow \text{succ } t_1'} \quad \text{(E-Succ)}$$

$$\text{pred } 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\text{pred (succ } nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{pred } t_1 \longrightarrow \text{pred } t_1'} \quad \text{(E-Pred)}$$

$$\text{iszero } 0 \longrightarrow \text{true} \quad \text{(E-IszeroZero)}$$

$$\text{iszero (succ } nv_1) \longrightarrow \text{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{iszero } t_1 \longrightarrow \text{iszero } t_1'} \quad \text{(E-IsZero)}$$

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

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

`badnat ::=`	*non-numeric normal forms:*
`wrong`	*run-time error*
`true`	*constant true*
`false`	*constant false*
`badbool ::=`	*non-boolean normal forms:*
`wrong`	*run-time error*
`nv`	*numeric value*

and we augment the evaluation relation with the following rules:

$$\texttt{if badbool then } t_1 \texttt{ else } t_2 \longrightarrow \texttt{wrong} \quad \text{(E-IF-WRONG)}$$

$$\texttt{succ badnat} \longrightarrow \texttt{wrong} \quad \text{(E-SUCC-WRONG)}$$

$$\texttt{pred badnat} \longrightarrow \texttt{wrong} \quad \text{(E-PRED-WRONG)}$$

$$\texttt{iszero badnat} \longrightarrow \texttt{wrong} \quad \text{(E-ISZERO-WRONG)}$$

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. □