3. Untyped Arithmetic Expressions (Types and Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Week 3

A Plan

To talk rigorously about type systems, we need to start by dealing formally with basic aspects of programming languages.

In this chapter, a very small programming language of numbers and booleans for the introduction of several fundamental concepts:

- Abstract syntax,
- Inductive definitions and proofs,
- Evaluation (i.e. semantics), and
- Modeling of run-time errors.

After this chapter, more powerful programming languages will be introduced but with the same fundamental concepts.

3.1 Introduction

In the arithmetic programming language,

- boolean constants : `true`, `false`
- conditional expressions : `if` - `then` - `else` -
- numeric constants : 0, `succ` -, `pred` -
- Testing operation : `iszero` -

These forms are called *terms* and they can be summarized by the following grammar in *BNF (Backus-Naur Form)*:

$$\begin{array}{rcl} \texttt{t} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{if t then t else } t \\ & \mid & \texttt{0} \mid \texttt{succ t} \mid \texttt{pred t} \mid \texttt{iszero t} \end{array}$$

cf. Expressions : terms, types, and kinds

3.1 Introduction

A program in the untyped arithmetic programming language is just a term built by the grammar.

- `if false then 0 else succ 0`
- `iszero (pred (succ 0))`   (cf. `pred 1` is 0)

For simple notation, `succ (succ (succ 0))` is written as 3.

The *evaluation* of terms (i.e., the execution of a program)

- "`if false then 0 else succ 0`" *evaluates to* "`succ 0`".
- "`iszero (pred (succ 0))`" *evaluates to* "`true`".

The results of evaluation are terms of a particularly simple form such as boolean constants or numbers. Such terms are called *values*.

3.1 Introduction

Note that the grammar permits to write some nonsensical terms such as "`succ true`" and "`if 0 then then 0 else 0`".

$$\begin{array}{rcl} \mathtt{t} & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{if\ t\ then\ t\ else}\ t \\ & \mid & \mathtt{0} \mid \mathtt{succ\ t} \mid \mathtt{pred\ t} \mid \mathtt{iszero\ t} \end{array}$$

Later, these nonsensical terms will be excluded by a *type system*.

3.2 Syntax

In the arithmetic programming language, a set of terms S that are generated by the grammar can be computed as:

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

Q. $S_3 =$

Q. Show that the sets S_i are cumulative-that is, that for each i we have $S_i \subseteq S_{i+1}$.

3.3 Induction on Terms

Three inductive definitions of functions over the set of terms (P.29$\sim$30)

- *Consts*(`t`) : the set of constants appearing in a term `t`
- *size*(`t`) : the size of a term `t`
- *depth*(`t`) : the depth of a term `t`

cf. Inductive definitions (also called recursive definitions)

- A way to define the elements in a set in terms of other elements in the set

3.3 Induction on Terms: $\mathit{Consts}(\mathtt{t})$

$\mathit{Consts}(\mathtt{t})$: the set of constants appearing in a term $\mathtt{t}$

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

3.3 Induction on Terms: $\mathit{Consts}(\mathtt{t})$

Consts(if false then 0 else succ 0)

$$
\begin{aligned}
&= \mathit{Consts}(\mathit{false}) \cup \mathit{Consts}(0) \cup \mathit{Consts}(\mathit{succ}\ 0) \\
&= \{\mathit{false}\} \cup \{0\} \cup \mathit{Consts}(0) \\
&= \{\mathit{false}\} \cup \{0\} \cup \{0\} \\
&= \{\mathit{false}, 0\}
\end{aligned}
$$

Q. *Consts(iszero (pred (succ 0))) =*

3.3 Induction on Terms: *size*($\texttt{t}$)

size($\texttt{t}$) : the size of a term $\texttt{t}$

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

3.3 Induction on Terms: *depth*(t)

depth(t) : the depth of a term t

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

3.3 Induction on Terms: Three induction principles

Recall that the principle of induction is useful for proving (countably) infinitely many cases.

Three variants of the principle of induction useful for proving over terms

- Induction on the depth of terms
- Induction on the size of terms
- Structural induction

The choice of one induction principle over another may lead to a simpler structure for the proof, but formally they are inter-derivable.

3.3 Induction on Terms: Induction on depth

Suppose P is a predicate on terms.

Informally, assume P is true over terms with depth $\leq d$, and then prove P over terms with depth $d + 1$ using the assumption.

Formally, the principle of induction on the depth of terms:

- If, for each term $\mathtt{s}$,
  given $P(\mathtt{r})$ for all $\mathtt{r}$ such that $\mathit{depth}(\mathtt{r}) < \mathit{depth}(\mathtt{s})$
  we can show $P(\mathtt{s})$,
  then $P(\mathtt{s})$ holds for all $\mathtt{s}$.

3.3 Induction on Terms: Induction on size

Suppose P is a predicate on terms.

Informally, assume P is true over terms with size $\leq sz$, and then prove P over terms with size $sz + 1$ using the assumption.

Formally, the principle of induction on the size of terms:

- If, for each term $\mathtt{s}$,
  given $P(\mathtt{r})$ for all $\mathtt{r}$ such that $\mathit{size}(\mathtt{r}) < \mathit{size}(\mathtt{s})$
  we can show $P(\mathtt{s})$,
  then $P(\mathtt{s})$ holds for all $\mathtt{s}$.

3.3 Induction on Terms: Structural induction

Suppose P is a predicate on terms.

Informally, assume P is true over subterms, and then prove P over terms constructed with the subterm using the assumption.

Formally, the principle of structural induction over terms:

- If, for each term $\mathtt{s}$,
  given $P(\mathtt{r})$ for all (immediate) subterms $\mathtt{r}$ of $\mathtt{s}$
  we can show $P(\mathtt{s})$,
  then $P(\mathtt{s})$ holds for all $\mathtt{s}$.

3.3 Induction on Terms: Exercise

Lemma 3.3.3: The number of distinct constants in a term t is no greater than the size of t (i.e., $|Consts(\mathtt{t})| \leq size(\mathtt{t})$).

- Proof by induction on the depth of t.

Q. Which principle of induction is necessary for this proof? Why?

3.4 Semantics Styles

In operational semantics, the meaning of a program is defined by a sequence of evaluation steps.

- Structural operational semantics (small-step semantics) by Gordon Plotkin
- Natural semantics (big-step semantics) by Gilles Kahn
- Communicating concurrent systems (CCS) by Robin Milner

In denotational semantics, the meaning of a program is defined by mathematical objects such as sets and relations. (cf. Dana Scott)

In axiomatic semantics, the meaning of a program is defined by logical rules. (cf. Tony Hoare and Robert W. Floyd)

3.5 Evaluation: The boolean part of the arithmetic PL

(Partial) Syntax of terms and values

$$\begin{array}{lll} \text{Terms } \texttt{t} & ::= & \texttt{v} \mid \texttt{if t then t else t} \\ \text{Values } \texttt{v} & ::= & \texttt{true} \mid \texttt{false} \end{array}$$

Evaluation

$$\texttt{if true then t2 else t3} \quad \rightarrow \quad \texttt{t2} \qquad \text{(E-IFTrue)}$$

$$\texttt{if false then t2 else t3} \quad \rightarrow \quad \texttt{t3} \qquad \text{(E-IFFalse)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{if t1 then t2 else t3} \rightarrow \texttt{if t1' then t2 else t3}} \qquad \text{(E-IF)}$$

Note evaluation is defined by an evaluation relation →

- $t \rightarrow t'$ saying "a term t evaluates to t' in one step".
  cf. Small-step operational semantics
  cf. $t \rightarrow t' \quad \equiv \quad (t, t') \in \rightarrow$

3.5 Evaluation: how to read evaluation rules

(1) Axiom $\texttt{if true then t2 else t3} \to \texttt{t2}$ (E-IFTrue)

- The left term evaluates to the right term *unconditionally*.

(2) Evaluation rule

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{if t1 then t2 else t3} \to \texttt{if t1' then t2 else t3}} \;\text{(E-IF)}$$

- One thing above the horizontal line is called a condition or a premise.
- Another thing below it is called a conclusion.
- Sometimes, a side condition in (...) is present optionally.

3.5 Evaluation: The boolean part of the arithmetic PL

Q. What does the following term evaluate to?

- `if true then (if false then false else false) else true`

Q. Run the following term until there is no more step by the evaluation rules.

- `if (if false then false else false) then false else true`.

3.5 Evaluation: The boolean part of the arithmetic PL

(E-IFTrue) and (E-IFFalse) are called *computation rules* and (E-IF) is called as a *congruence rule*.

When a pair (t,t') $\in \rightarrow$, we say that "the evaluation statement (or judgment) `t` $\rightarrow$ `t'` is *derivable*.

This can be justified by exhibiting a *derivation tree* whose leaves are (E-IFTrue) and (E-IFFalse) and whose internal nodes are (E-IF).

Q. Show that if `t` $\rightarrow$ `t1` and `t` $\rightarrow$ `t2` then `t1 = t2`.
Proof by structural induction on the derivation of `t` $\rightarrow$ `t1`.

3.5 Evaluation: The boolean part of the arithmetic PL

The one-step evaluation relation shows how a term evaluates from one state to the next state.

A term `t` is in *normal form* if no evaluation rule applies to it.

Every value is in normal form. (But every normal form is not always a value.)

- In the boolean part of the arithmetic PL, if `t` is in normal form then `t` is a value. (Provable by structural induction on `t`)
- But in the arithmetic PL, there is a normal form which is not a value. E.g., `succ true`, `iszero true`, and so on.

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

3.5 Evaluation: The boolean part of the arithmetic PL

The semantics of (the boolean part) of the airthmetic PL is defined by constructing derivation trees

- One-step: A term `t1` evalutes to another `t2`
  if you derive $\texttt{t1} \rightarrow \texttt{t2}$ by (E-IFTrue), (E-IFFalse), and (E-IF).
- Normal execution: A term `t1` normally evaluates to a value
  if $\texttt{t1} \rightarrow^* \texttt{v}$
- Runtime error: A term `t1` will gets stuck at `t2`
  if $\texttt{t1} \rightarrow^* \texttt{t2}$.

Thus the existence of a derivation tree constructed with (E-IFTrue), (E-IFFalse), and (E-IF) defines the semantics.

3.5 Evaluation: The arithmetic PL

An extension of the definition of evaluation to arithmetic expressions.

Syntax

$$
\begin{array}{llcl}
\text{Terms } \texttt{t} & & ::= & \texttt{v} \mid \texttt{if t then t else t} \\
 & & \mid & \texttt{0} \mid \texttt{succ t} \mid \texttt{pred t} \mid \texttt{iszero t} \\
\text{Values} & \texttt{v} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{nv} \\
\text{Num. values} & \texttt{nv} & ::= & \texttt{0} \mid \texttt{succ nv}
\end{array}
$$

Q. Describe a set of terms S_{terms} whose elements are defined by Terms. Also describe a set of values S_{values} whose elements are defined by Values. (Refer to P.6)

3.5 Evaluation: The arithmetic PL

Evaluation

$\cdots$ (the same evaluation rules) $\cdots$

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{succ t1} \to \texttt{succ t1'}} \qquad \text{(E-Succ)}$$

$$\texttt{pred 0} \to \texttt{0} \qquad \text{(E-PredZero)}$$

$$\texttt{pred (succ nv)} \to \texttt{nv} \qquad \text{(E-PredSucc)}$$

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{pred t1} \to \texttt{pred t1'}} \qquad \text{(E-Pred)}$$

$$\texttt{iszero 0} \to \texttt{false} \qquad \text{(E-IsZeroZero)}$$

$$\texttt{iszero (succ nv)} \to \texttt{true} \qquad \text{(E-IsZeroSucc)}$$

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{iszero t1} \to \texttt{iszero t1'}} \qquad \text{(E-IsZero)}$$

3.5 Evaluation: The arithmetic PL

Q. Evaluate "`pred (succ (pred 0))`".

3.5 Evaluation: The arithmetic PL

A term is *stuck* if it is in normal form but not a value. “Stuckness” gives us a simple notion of *run-time error* for our simple conceptual computer.

Q. Show a term that will get stuck on the evaluation.

Summary: The arithmetic programming language

The syntax

$$
\begin{array}{llcl}
\text{Terms } \mathtt{t} & & ::= & \mathtt{v} \mid \mathtt{if\ t\ then\ t\ else\ t} \\
 & & \mid & \mathtt{0} \mid \mathtt{succ\ t} \mid \mathtt{pred\ t} \mid \mathtt{iszero\ t} \\
\text{Values} & \mathtt{v} & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{nv} \\
\text{Num. values} & \mathtt{nv} & ::= & \mathtt{0} \mid \mathtt{succ\ nv}
\end{array}
$$

Summary: The arithmetic PL (cont.)

The evaluation rules

$$\texttt{if true then t2 else t3} \;\rightarrow\; \texttt{t2} \qquad \text{(E-IFTrue)}$$

$$\texttt{if false then t2 else t3} \;\rightarrow\; \texttt{t3} \qquad \text{(E-IFFalse)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{if t1 then t2 else t3} \rightarrow \texttt{if t1' then t2 else t3}} \qquad \text{(E-IF)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{succ t1} \rightarrow \texttt{succ t1'}} \qquad \text{(E-Succ)}$$

$$\texttt{pred 0} \rightarrow \texttt{0} \qquad \text{(E-PredZero)}$$

$$\texttt{pred (succ nv)} \rightarrow \texttt{nv} \qquad \text{(E-PredSucc)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{pred t1} \rightarrow \texttt{pred t1'}} \qquad \text{(E-Pred)}$$

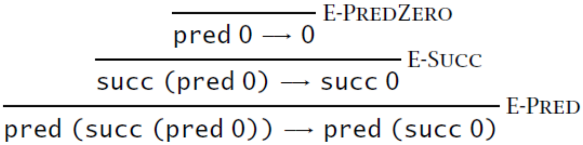
$$\texttt{iszero 0} \rightarrow \texttt{false} \qquad \text{(E-IsZeroZero)}$$

$$\texttt{iszero (succ nv)} \rightarrow \texttt{true} \qquad \text{(E-IsZeroSucc)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{iszero t1} \rightarrow \texttt{iszero t1'}} \qquad \text{(E-IsZero)}$$

Summary: The arithmetic PL - Derivations

A derivation tree of height 3 defines the semantics for a single step "`pred (succ (pred 0))` $\rightarrow$ `pred (succ 0)`" using the three rules: (E-PredZero), (E-Succ), and (E-Pred).



$$\dfrac{\dfrac{\dfrac{}{\text{pred } 0 \longrightarrow 0}\ \textsc{E-PredZero}}{\text{succ (pred } 0) \longrightarrow \text{succ } 0}\ \textsc{E-Succ}}{\text{pred (succ (pred } 0)) \longrightarrow \text{pred (succ } 0)}\ \textsc{E-Pred}$$

Summary: The arithmetic PL - Semantics

The semantics of the airthmetic PL by constructing derivation trees

- One-step: A term `t1` evalutes to another `t2`
  if you can derive `t1` $\rightarrow$ `t2` by the ten evaluation rules.
- Normal execution: A term `t1` normally evaluates to a value
  if `t1` $\rightarrow^*$ `v`
- Runtime error: A term `t1` will get stuck at `t2`
  if `t1` $\rightarrow^*$ `t2`.

Thus the existence of a derivation tree constructed by the evaluation rules defines the semantics.

Summary: The arithmetic PL - Induction

Q. Show that if `t → t1` and `t → t2` then `t1 = t2`.

By the induction, we can prove the uniqueness property for all infinite number of derivation trees for `t → t1`.

Proof by structural induction on the derivation of `t → t1`.

- Base cases: The derivation trees of height 1.
- Inductive cases:
  (1) Assume the property to show is true for all the derivation trees of height k.
  (2) Then prove that the property is also true for all the derivation trees of height $k+1$.