

Midterm Review (v1.3)

March 5, 2018

1 Syntax

1.1 Inductive Definitions

Inference Rules:

$$\frac{}{\mathbf{T} \in \text{exp}} \quad \frac{}{\mathbf{F} \in \text{exp}} \quad \frac{e_1 \in \text{exp} \quad e_2 \in \text{exp} \quad e_3 \in \text{exp}}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \in \text{exp}}$$

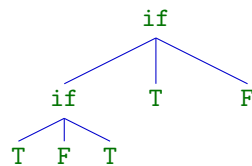
Grammar Schema:

$$e ::= \mathbf{T} \mid \mathbf{F} \mid \text{if } e \text{ then } e \text{ else } e$$

An example expression:

$$\text{if } (\text{if } \mathbf{T} \text{ else } \mathbf{F} \text{ else } \mathbf{T}) \text{ else } \mathbf{T} \text{ else } \mathbf{F}$$

is the tree:



1.2 Metafunctions

$$\begin{aligned} \text{size} &\in \text{exp} \rightarrow \mathbb{N} \\ \text{size}(\mathbf{T}) &:= 1 \\ \text{size}(\mathbf{F}) &:= 1 \\ \text{size}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &:= \text{size}(e_1) + \text{size}(e_2) + \text{size}(e_3) + 1 \end{aligned}$$

2 Semantics

One purpose of semantics is to distinguish the syntax of expressions with the results of their evaluation. E.g.:

$$\begin{aligned} e_1 &:= \text{if } T \text{ else } F \text{ else } T \\ e_2 &:= \text{if } F \text{ else } T \text{ else } F \end{aligned}$$

These expressions are different:

$$e_1 \neq e_2$$

But they evaluate to the same value:

$$\begin{aligned} e_1 &\longrightarrow^* F \\ e_2 &\longrightarrow^* F \end{aligned}$$

2.1 Small-step Operational Semantics

Three steps to defining a small-step operational semantics:

1. Define a one-step relation $e \longrightarrow e$.
2. Identify a subset of terms which are called *values*.
3. Induce a many-step relation $e \longrightarrow^* e$ as the reflexive, transitive closure of $e \longrightarrow e$.

$$\boxed{e \longrightarrow^* e}$$

$$\begin{array}{c} \frac{}{\text{if } T \text{ then } e_2 \text{ else } e_3 \longrightarrow e_2} \text{If-T} \qquad \frac{}{\text{if } F \text{ then } e_2 \text{ else } e_3 \longrightarrow e_3} \text{If-F} \\[10pt] \frac{e_1 \longrightarrow e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3} \text{If-CONG} \end{array}$$

Values:

$$v ::= T \mid F$$

Many-step Relation:

$$e \longrightarrow^* e'$$

means:

$$e \longrightarrow e_1 \longrightarrow \cdots \longrightarrow e_{n-1} \longrightarrow e'$$

for some n .

E.g.:

$$\text{if } (\text{if } T \text{ then } F \text{ else } T) \text{ then } F \text{ else } T \longrightarrow^* T$$

in two steps:

$$\begin{array}{c}
 \text{if } (\text{if } T \text{ then } F \text{ else } T) \text{ then } F \text{ else } T \\
 \longrightarrow \\
 \text{if } F \text{ then } F \text{ else } T \\
 \longrightarrow \\
 T
 \end{array}$$

Each step is justified via a derivation tree. The first step:

$$\frac{\frac{}{\text{if } T \text{ then } F \text{ else } T \longrightarrow F} \text{IF-T}}{\text{if } (\text{if } T \text{ then } F \text{ else } T) \text{ then } F \text{ else } T \longrightarrow \text{if } F \text{ then } F \text{ else } T} \text{IF-CONG}$$

and the second step:

$$\frac{}{\text{if } F \text{ then } F \text{ else } T \longrightarrow T} \text{IF-F}$$

3 Arithmetic Expressions

Syntax:

$$\begin{array}{l}
 e ::= \dots \mid 0 \mid \text{succ } e \mid \text{pred } e \mid \text{iszero } e \\
 v ::= \dots \mid nv \\
 nv ::= 0 \mid \text{succ } nv
 \end{array}$$

Semantics:

$e \longrightarrow e$

$$\begin{array}{c}
 \dots \quad \frac{e \longrightarrow e'}{\text{succ } e \longrightarrow \text{succ } e'} \quad \frac{}{\text{pred } 0 \longrightarrow 0} \quad \frac{}{\text{pred } (\text{succ } nv) \longrightarrow nv} \quad \frac{e \longrightarrow e'}{\text{pred } e \longrightarrow \text{pred } e'} \\
 \frac{}{\text{iszero } 0 \longrightarrow T} \quad \frac{}{\text{iszero } (\text{succ } nv) \longrightarrow F} \quad \frac{e \longrightarrow e'}{\text{iszero } e \longrightarrow \text{iszero } e'}
 \end{array}$$

4 Safety

A term e is *stuck* iff e is not a value and there is no e' such that $e \longrightarrow e'$.

A term e is *unsafe* iff there exists some e' such that $e \longrightarrow e'$ and e' is stuck.

A term e is *safe* iff for every e' such that $e \longrightarrow e'$, e' is not stuck.

If a term is not safe, then it is unsafe. If a term is not unsafe, then it is safe.

E.g.:

$$\text{if } T \text{ then pred } 0 \text{ else } 0$$

is safe.

However:

$$\text{if } T \text{ then pred } F \text{ else } 0$$

is unsafe.

5 Types

Typing is a syntactic relation $e : \tau$.

If there exists a type τ such that $e : \tau$ then e is *well-typed*.

If there does not exist a type τ such that $e : \tau$ then e is *untypeable*.

Type safety is a theorem that says if e is well-typed then e is safe.

Types for the Boolean Arithmetic Language:

$e : \tau$

$$\begin{array}{c}
 \frac{}{T : \text{bool}} \qquad \frac{}{F : \text{bool}} \qquad \frac{e_1 : \text{bool} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \\
 \\
 \frac{}{0 : \text{nat}} \qquad \frac{e : \text{nat}}{\text{succ } e : \text{nat}} \qquad \frac{e : \text{nat}}{\text{pred } e : \text{nat}} \qquad \frac{e : \text{nat}}{\text{iszero } e : \text{bool}}
 \end{array}$$

5.1 Progress + Preservation = Safety

The proof mechanism for type safety is to use two lemmas: progress and preservation.

Progress: if $e : \tau$ then e is not stuck.

Preservation: if $e : \tau$ and $e \longrightarrow e'$ then $e' : \tau$.

Safety is a direct corollary from Progress + Preservation.

5.2 Inversion Lemmas

Type judgments can be “inverted”, which give rise to a number of inversion lemmas. Here are two examples of inversion lemmas:

- If $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ and $e : \tau$, then:
 1. $e_1 : \text{bool}$
 2. $e_2 : \tau$
 3. $e_3 : \tau$
- If $e : \text{bool}$, then *either* of the following are true:
 1. $e = T$
 2. $e = F$
 3. $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$, $e_1 : \text{bool}$, $e_2 : \text{bool}$ and $e_3 : \text{bool}$
 4. $e = \text{iszero } e'$ and $e' : \text{nat}$

6 Lambda Calculus

Syntax:

$$e ::= x \mid \lambda x. e \mid e e$$

When we write multiple terms in sequence:

$$e_1 \ e_2 \ e_3 \ e_4$$

the implicit parenthesis placement is left-associative:

$$((e_1 \ e_2) \ e_3) \ e_4$$

and when we write multiple lambda terms in sequence:

$$\lambda x. \lambda y. \lambda z. e$$

the implicit parenthesis placement is right-associative:

$$\lambda x. (\lambda y. (\lambda z. e))$$

Values:

$$v ::= \lambda x. e$$

Metafunctions for free variables and substitution:

$$\begin{aligned} FV &\in \text{exp} \rightarrow \mathcal{P}(\text{var}) \\ FV(x) &:= \{x\} \\ FV(\lambda x. e) &:= FV(e) \setminus \{x\} \\ FV(e_1 \ e_2) &:= FV(e_1) \cup FV(e_2) \\ [x \mapsto e_2] &\in \text{exp} \rightarrow \text{exp} \\ [x \mapsto e_2]y &:= e_2 && \text{if } x = y \\ [x \mapsto e_2]y &:= y && \text{if } x \neq y \\ [x \mapsto e_2](\lambda y. e) &:= \lambda y. [x \mapsto e_2](e) && \text{if } x \neq y \text{ and } y \notin FV(e_2) \\ [x \mapsto e_2](e_{11} \ e_{12}) &:= ([x \mapsto e_2]e_{11}) \ ([x \mapsto e_2]e_{12}) \end{aligned}$$

E.g.:

$$\begin{aligned} FV(\lambda x. x \ y) &= \{y\} \\ [x \mapsto z](\lambda x. x \ y) &= \lambda x. x \ y \\ [y \mapsto z](\lambda x. x \ y) &= \lambda x. x \ z \\ [y \mapsto x](\lambda x. x \ y) &\neq \end{aligned}$$

It is always okay to convert lambda terms to alpha-equivalent terms.

E.g.:

$$\lambda x. \lambda y. x \ y \approx \lambda x. \lambda z. x \ z \approx^a \lambda y. \lambda z. y \ z \approx^a \lambda y. \lambda x. y \ x$$

This allows for the last substitution example to be given meaning:

$$[y \mapsto x](\lambda x. x \ y) \approx^a [y \mapsto x](\lambda z. z \ y) = \lambda z. z \ x$$

There is one rule which describes how lambda terms take a step, the β rule:

$$\overline{(\lambda x. e_1)e_2 \longrightarrow [x \mapsto e_2]e_1}^\beta$$

The full semantics for call-by-value lambda calculus:

$$\boxed{e \longrightarrow e}$$

$$\frac{}{(\lambda x. e) v \longrightarrow [x \mapsto v]e}^{\beta} \qquad \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \qquad \frac{e \longrightarrow e'}{v e \longrightarrow v e'}$$

These three rules fully describe universal computation (i.e., equivalent in computational power to Turing machines.)

Boolean encodings:

$$\begin{aligned} \text{true} &:= \lambda x. \lambda y. x \\ \text{false} &:= \lambda x. \lambda y. y \\ \text{cond} &:= \lambda b. \lambda x. \lambda y. b x y \\ \text{cond-alt} &:= \lambda b. b \end{aligned}$$

An infinite loop:

$$\Omega := (\lambda x. x x) (\lambda x. x x)$$

Recursion can be defined using the Y-combinator:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

which has the following property:

$$Y f \longrightarrow^* f (Y f)$$

7 Typed Lambda Calculus

$$\begin{aligned} \tau &::= \dots \mid \tau \Rightarrow \tau \\ \Gamma &::= [] \mid \Gamma, x : \tau \end{aligned}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x. e) : \tau_1 \Rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : (\tau_1 \Rightarrow \tau_2) \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

The same type safety theorem holds: if $e : \tau$ then e is safe.

8 Typed Lambda Calculus Extensions

Empty type:

$$\begin{array}{l} \tau ::= \dots \mid \text{empty} \\ e ::= \dots \mid \text{absurd } e \\ v ::= \dots \quad (\text{no new values}) \end{array}$$

$$\boxed{e \longrightarrow e}$$

(this rule is safe to have, but not useful or necessary...)

$$\frac{e \longrightarrow e'}{\text{absurd } e \longrightarrow \text{absurd } e'}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash e : \text{empty}}{\Gamma \vdash \text{absurd } e : \tau} \text{EMPTY-ELIM}$$

Unit type:

$$\begin{array}{l} \tau ::= \dots \mid \text{unit} \\ e ::= \dots \mid \bullet \\ v ::= \dots \mid \bullet \end{array}$$

$$\boxed{e \longrightarrow e}$$

(no new rules)

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash \bullet : \text{unit}} \text{UNIT-INTRO}$$

Sum type:

$$\begin{aligned}\tau &::= \dots \mid \tau + \tau \\ e &::= \dots \mid \text{inl } e \mid \text{inr } e \mid \text{case}(e)\{x. e\}\{x. e\} \\ v &::= \dots \mid \text{inl } v \mid \text{inr } v\end{aligned}$$

$$\boxed{e \longrightarrow e}$$

$$\begin{array}{c} \frac{e \longrightarrow e'}{\text{inl } e \longrightarrow \text{inl } e'} \quad \frac{e \longrightarrow e'}{\text{inr } e \longrightarrow \text{inr } e'} \quad \frac{}{\text{case}(\text{inl}(v))\{x_2. e_2\}\{x_3. e_3\} \longrightarrow [x_2 \mapsto v]e_2} \\[10pt] \frac{}{\text{case}(\text{inr}(v))\{x_2. e_2\}\{x_3. e_3\} \longrightarrow [x_3 \mapsto v]e_3} \quad \frac{e_1 \longrightarrow e'_1}{\text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} \longrightarrow \text{case}(e'_1)\{x_2. e_2\}\{x_3. e_3\}} \end{array}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\begin{array}{c} \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl } e : \tau_1 \times \tau_2} \text{SUM-INTRO-1} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr } e : \tau_1 \times \tau_2} \text{SUM-INTRO-2} \\[10pt] \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x_2 : \tau_1 \vdash e_2 : \tau \quad \Gamma, x_3 : \tau_2 \vdash e_3 : \tau}{\Gamma \vdash \text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} : \tau} \text{SUM-ELIM} \end{array}$$

Product type:

$$\begin{aligned}\tau &::= \dots \mid \tau \times \tau \\ e &::= \dots \mid \langle e_1, e_2 \rangle \mid \text{projl } e \mid \text{projr } e \\ v &::= \dots \mid \langle v_1, v_2 \rangle\end{aligned}$$

$$\boxed{e \longrightarrow e}$$

$$\begin{array}{c} \frac{e_1 \longrightarrow e'_1}{\langle e_1, e_2 \rangle \longrightarrow \langle e'_1, e_2 \rangle} \quad \frac{e \longrightarrow e'}{\langle v, e \rangle \longrightarrow \langle v, e' \rangle} \quad \frac{}{\text{projl } \langle v_1, v_2 \rangle \longrightarrow v_1} \quad \frac{e \longrightarrow e'}{\text{projl } e \longrightarrow \text{projl } e'} \\[10pt] \frac{}{\text{projr } \langle v_1, v_2 \rangle \longrightarrow v_2} \quad \frac{e \longrightarrow e'}{\text{projr } e \longrightarrow \text{projr } e'} \end{array}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\begin{array}{c} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{PROD-INTRO} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{projl } e : \tau_1} \text{PROD-ELIM-1} \quad \frac{\Gamma \vdash e : \tau_2 \times \tau_2}{\Gamma \vdash \text{projr } e : \tau_2} \text{PROD-ELIM-2} \end{array}$$

Errata

v1.1 Fixed typo in the small-step rule for `case`:

$$\frac{e_1 \longrightarrow \boxed{e'_2}}{\text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} \longrightarrow \text{case}(e'_1)\{x_2. e_2\}\{x_3. e_3\}}$$

which has been corrected to be:

$$\frac{e_1 \longrightarrow e'_1}{\text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} \longrightarrow \text{case}(e'_1)\{x_2. e_2\}\{x_3. e_3\}}$$

v1.2 Fixed typo in the small-step rule for **inl**:

$$\frac{e \longrightarrow e'}{\boxed{\text{inl } e} \longrightarrow \boxed{\text{inl } e'}}$$

which has been corrected to be:

$$\frac{e \longrightarrow e'}{\text{inr } e \longrightarrow \text{inr } e'}$$

v1.3 Fixed typo in the typing rule for **case**:

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x_2 : \tau_1 \vdash e_2 : \tau \quad \Gamma, x_3 : \tau_2 \vdash \boxed{e_2} : \tau}{\Gamma \vdash \text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} : \tau} \text{SUM-ELIM}$$

which has been corrected to be:

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x_2 : \tau_1 \vdash e_2 : \tau \quad \Gamma, x_3 : \tau_2 \vdash e_3 : \tau}{\Gamma \vdash \text{case}(e_1)\{x_2. e_2\}\{x_3. e_3\} : \tau} \text{SUM-ELIM}$$