Foundations of Software
Fall 2020

Week 2

Readings

You should try to at least look at the reading for a particular lecture *before* that lecture.

We're starting with Chapter 3 of the textbook. Chapter 2 contains some mathematical preliminaries which we assume you are familiar with.

Where we're going

Going Meta...

The functional programming style used in OCaml and Scala is based on treating *programs as data* — i.e., on writing functions that manipulate other functions as their inputs and outputs.

Everything in this course is based on treating *programs as mathematical objects* — i.e., we will be building mathematical theories whose basic objects of study are programs (and whole programming languages).

Jargon: We will be studying the *metatheory* of programming languages.

Warning!

The material in the next couple of lectures is more slippery than it may first appear.

"I believe it when I hear it" is not a sufficient test of understanding.

A much better test is "I can explain it so that someone else believes it."

"You never really misunderstand something
until you try to teach it..."
— Anon.

Basics of Induction (Review)

Induction

Principle of *ordinary induction* on natural numbers:

Suppose that P is a predicate on the natural numbers. Then:
If $P(0)$
and, for all i, $P(i)$ implies $P(i+1)$,
then $P(n)$ holds for all n.

Example

Theorem: $2^0 + 2^1 + ... + 2^n = 2^{n+1} - 1$, for every n.
Proof: Let $P(i)$ be "$2^0 + 2^1 + ... + 2^i = 2^{i+1} - 1$."

- Show $P(0)$:
$$2^0 = 1 = 2^1 - 1$$
- Show that $P(i)$ implies $P(i+1)$:
$$\begin{aligned} 2^0 + 2^1 + ... + 2^{i+1} &= (2^0 + 2^1 + ... + 2^i) + 2^{i+1} \\ &= (2^{i+1} - 1) + 2^{i+1} && \text{by IH} \\ &= 2 \cdot (2^{i+1}) - 1 \\ &= 2^{i+2} - 1 \end{aligned}$$
- The result ($P(n)$ for all n) follows by the principle of (ordinary) induction.

Shorthand form

Theorem: $2^0 + 2^1 + ... + 2^n = 2^{n+1} - 1$, for every n.
Proof: By induction on n.

- Base case ($n = 0$):

$$2^0 = 1 = 2^1 - 1$$

- Inductive case ($n = i + 1$):

$$\begin{array}{rcll} 2^0 + 2^1 + ... + 2^{i+1} & = & (2^0 + 2^1 + ... + 2^i) + 2^{i+1} & \\ & = & (2^{i+1} - 1) + 2^{i+1} & \text{IH} \\ & = & 2 \cdot (2^{i+1}) - 1 & \\ & = & 2^{i+2} - 1 & \end{array}$$

Complete Induction

Principle of *complete induction* on natural numbers:

Suppose that P is a predicate on the natural numbers. Then:
If, for each natural number n,
given $P(i)$ for all $i < n$
we can show $P(n)$,
then $P(n)$ holds for all n.

Complete versus ordinary induction

Ordinary and complete induction are *interderivable* — assuming one, we can prove the other.

Thus, the choice of which to use for a particular proof is purely a question of style.

We'll see some other (equivalent) styles as we go along.

Syntax

Simple Arithmetic Expressions

Here is a BNF grammar for a very simple language of arithmetic expressions:

`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*

Terminology:

- `t` here is a *metavariable*

Abstract vs. concrete syntax

Q: Does this grammar define a set of *character strings*, a set of *token lists*, or a set of *abstract syntax trees*?

Abstract vs. concrete syntax

Q: Does this grammar define a set of *character strings*, a set of *token lists*, or a set of *abstract syntax trees*?

A: In a sense, all three. But we are primarily interested, here, in abstract syntax trees.

For this reason, grammars like the one on the previous slide are sometimes called *abstract grammars*. An abstract grammar *defines* a set of abstract syntax trees and *suggests* a mapping from character strings to trees.

We then *write* terms as linear character strings rather than trees simply for convenience. If there is any potential confusion about what tree is intended, we use parentheses to disambiguate.

Q: So, are

```
succ 0
succ (0)
(((succ (((((0))))))))
```

"the same term"?

What about

```
succ 0
pred (succ (succ 0))
```

?

A more explicit form of the definition

The set $\mathcal{T}$ of *terms* is the smallest set such that

1. $\{\texttt{true}, \texttt{false}, 0\} \subseteq \mathcal{T}$;
2. if $\texttt{t}_1 \in \mathcal{T}$, then $\{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1\} \subseteq \mathcal{T}$;
3. if $\texttt{t}_1 \in \mathcal{T}$, $\texttt{t}_2 \in \mathcal{T}$, and $\texttt{t}_3 \in \mathcal{T}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}$.

Inference rules

An alternate notation for the same definition:

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

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}}$$

Note that "the smallest set closed under..." is implied (but often not stated explicitly).

Terminology:

- axiom vs. rule
- concrete rule vs. rule scheme

Terms, concretely

Define an infinite sequence of sets, $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2$, ..., as follows:

$$\begin{aligned} \mathcal{S}_0 &= \emptyset \\ \mathcal{S}_{i+1} &= \quad \{\texttt{true}, \texttt{false}, 0\} \\ &\quad \cup \ \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \mid \texttt{t}_1 \in \mathcal{S}_i\} \\ &\quad \cup \ \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in \mathcal{S}_i\} \end{aligned}$$

Now let

$$\mathcal{S} = \bigcup_i \mathcal{S}_i$$

Comparing the definitions

We have seen two different presentations of terms:

1. as the *smallest* set that is *closed* under certain rules ($\mathcal{T}$)
   - explicit inductive definition
   - BNF shorthand
   - inference rule shorthand
2. as the *limit* ($\mathcal{S}$) of a series of sets (of larger and larger terms)

Comparing the definitions

We have seen two different presentations of terms:

1. as the *smallest* set that is *closed* under certain rules ($\mathcal{T}$)
   - explicit inductive definition
   - BNF shorthand
   - inference rule shorthand
2. as the *limit* ($\mathcal{S}$) of a series of sets (of larger and larger terms)

What does it mean to assert that "these presentations are equivalent"?

Induction on Syntax

Why two definitions?

The two ways of defining the set of terms are both useful:

1. the definition of terms as the smallest set with a certain closure property is compact and easy to read
2. the definition of the set of terms as the limit of a sequence gives us an *induction principle* for proving things about terms...

Induction on Terms

Definition: The *depth* of a term $\mathtt{t}$ is the smallest i such that $\mathtt{t} \in \mathcal{S}_i$.

From the definition of $\mathcal{S}$, it is clear that, if a term $\mathtt{t}$ is in $\mathcal{S}_i$, then all of its immediate subterms must be in $\mathcal{S}_{i-1}$, i.e., they must have strictly smaller depths.

This observation justifies the *principle of induction on terms.* Let P be a predicate on terms.

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

Inductive Function Definitions

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

$$
\begin{array}{lcl}
\mathit{Consts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{Consts}(\mathtt{0}) & = & \{\mathtt{0}\} \\
\mathit{Consts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \\
 & & \cup \mathit{Consts}(\mathtt{t}_3)
\end{array}
$$

Simple, right?

First question:

Normally, a "definition" just assigns a convenient name to a previously-known thing. But here, the "thing" on the right-hand side involves the very name that we are "defining"!

So in what sense is this a definition??

Second question:

Suppose we had written this instead...

The set of constants appearing in a term $\mathtt{t}$, written $\mathit{BadConsts}(\mathtt{t})$, is defined as follows:

$$
\begin{array}{lcl}
\mathit{BadConsts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{BadConsts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{BadConsts}(\mathtt{0}) & = & \{\mathtt{0}\} \\
\mathit{BadConsts}(\mathtt{0}) & = & \{\} \\
\mathit{BadConsts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{BadConsts}(\mathtt{t}_1) \\
\mathit{BadConsts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{BadConsts}(\mathtt{t}_1) \\
\mathit{BadConsts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{BadConsts}(\mathtt{iszero}\ (\mathtt{iszero}\ \mathtt{t}_1))
\end{array}
$$

What is the essential difference between these two definitions?
How do we tell the difference between well-formed inductive definitions and ill-formed ones?
What, exactly, does a well-formed inductive definition mean?

What is a function?

Recall that a *function* f from A (its domain) to B (its co-domain) can be viewed as a two-place *relation* (called the "graph" of the function) with certain properties:

- It is *total*: Every element of its domain occurs at least once in its graph. More precisely:
  For every $a \in A$, *there exists some* $b \in B$ *such that* $(a, b) \in f$.
- It is *deterministic*: every element of its domain occurs at most once in its graph. More precisely:
  If $(a, b_1) \in f$ *and* $(a, b_2) \in f$, *then* $b_1 = b_2$.

We have seen how to define relations inductively. E.g....
Let *Consts* be the smallest two-place relation closed under the following rules:

$$(\texttt{true},\ \{\texttt{true}\}) \in \mathit{Consts}$$

$$(\texttt{false},\ \{\texttt{false}\}) \in \mathit{Consts}$$

$$(\texttt{0},\ \{\texttt{0}\}) \in \mathit{Consts}$$

$$\frac{(\texttt{t}_1,\ C) \in \mathit{Consts}}{(\texttt{succ t}_1,\ C) \in \mathit{Consts}}$$

$$\frac{(\texttt{t}_1,\ C) \in \mathit{Consts}}{(\texttt{pred t}_1,\ C) \in \mathit{Consts}}$$

$$\frac{(\texttt{t}_1,\ C) \in \mathit{Consts}}{(\texttt{iszero t}_1,\ C) \in \mathit{Consts}}$$

$$\frac{(\texttt{t}_1,\ C_1) \in \mathit{Consts} \quad (\texttt{t}_2,\ C_2) \in \mathit{Consts} \quad (\texttt{t}_3,\ C_3) \in \mathit{Consts}}{(\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3,\ C_1 \cup C_2 \cup C_3) \in \mathit{Consts}}$$

This definition certainly defines a *relation* (i.e., the smallest one with a certain closure property).

Q: How can we be sure that this relation is a *function*?

This definition certainly defines a *relation* (i.e., the smallest one with a certain closure property).

Q: How can we be sure that this relation is a *function*?

A: *Prove it!*

Theorem:

The relation *Consts* defined by the inference rules a couple of slides ago is total and deterministic.

I.e., for each term $\texttt{t}$ there is exactly one set of terms C such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proof:

Theorem:

The relation *Consts* defined by the inference rules a couple of slides ago is total and deterministic.

I.e., for each term `t` there is exactly one set of terms C such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proof: By induction on `t`.

Theorem:

The relation *Consts* defined by the inference rules a couple of slides ago is total and deterministic.

I.e., for each term `t` there is exactly one set of terms C such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proof: By induction on `t`.

To apply the induction principle for terms, we must show, for an arbitrary term `t`, that if

> for each immediate subterm `s` of `t`, there is exactly one set of terms C_s such that $(\texttt{s}, C_s) \in \mathit{Consts}$

then

> there is exactly one set of terms C such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proceed by cases on the form of `t`.

- If `t` is `0`, `true`, or `false`, then we can immediately see from the definition of *Consts* that there is exactly one set of terms C (namely $\{\texttt{t}\}$) such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proceed by cases on the form of `t`.

- If `t` is `0`, `true`, or `false`, then we can immediately see from the definition of *Consts* that there is exactly one set of terms C (namely $\{\texttt{t}\}$) such that $(\texttt{t}, C) \in \mathit{Consts}$.

- If `t` is $\texttt{succ t}_1$, then the induction hypothesis tells us that there is exactly one set of terms C_1 such that $(\texttt{t}_1, C_1) \in \mathit{Consts}$. But then it is clear from the definition of *Consts* that there is exactly one set C (namely C_1) such that $(\texttt{t}, C) \in \mathit{Consts}$.

Proceed by cases on the form of $\mathtt{t}$.

- If $\mathtt{t}$ is $\mathtt{0}$, $\mathtt{true}$, or $\mathtt{false}$, then we can immediately see from the definition of *Consts* that there is exactly one set of terms C (namely $\{\mathtt{t}\}$) such that $(\mathtt{t}, C) \in \mathit{Consts}$.
- If $\mathtt{t}$ is $\mathtt{succ}\ \mathtt{t}_1$, then the induction hypothesis tells us that there is exactly one set of terms C_1 such that $(\mathtt{t}_1, C_1) \in \mathit{Consts}$. But then it is clear from the definition of *Consts* that there is exactly one set C (namely C_1) such that $(\mathtt{t}, C) \in \mathit{Consts}$.

  Similarly when $\mathtt{t}$ is $\mathtt{pred}\ \mathtt{t}_1$ or $\mathtt{iszero}\ \mathtt{t}_1$.

- If $\mathtt{t}$ is $\mathtt{if}\ \mathtt{s}_1\ \mathtt{then}\ \mathtt{s}_2\ \mathtt{else}\ \mathtt{s}_3$, then the induction hypothesis tells us
  - there is exactly one set of terms C_1 such that $(\mathtt{t}_1, C_1) \in \mathit{Consts}$
  - there is exactly one set of terms C_2 such that $(\mathtt{t}_2, C_2) \in \mathit{Consts}$
  - there is exactly one set of terms C_3 such that $(\mathtt{t}_3, C_3) \in \mathit{Consts}$

  But then it is clear from the definition of *Consts* that there is exactly one set C (namely $C_1 \cup C_2 \cup C_3$) such that $(\mathtt{t}, C) \in \mathit{Consts}$.

How about the bad definition?

$$(\mathtt{true}, \{\mathtt{true}\}) \in \mathit{BadConsts}$$

$$(\mathtt{false}, \{\mathtt{false}\}) \in \mathit{BadConsts}$$

$$(\mathtt{0}, \{\mathtt{0}\}) \in \mathit{BadConsts}$$

$$(\mathtt{0}, \{\}) \in \mathit{BadConsts}$$

$$\frac{(\mathtt{t}_1, C) \in \mathit{BadConsts}}{(\mathtt{succ}\ \mathtt{t}_1, C) \in \mathit{BadConsts}}$$

$$\frac{(\mathtt{t}_1, C) \in \mathit{BadConsts}}{(\mathtt{pred}\ \mathtt{t}_1, C) \in \mathit{BadConsts}}$$

$$\frac{(\mathtt{iszero}\ (\mathtt{iszero}\ \mathtt{t}_1), C) \in \mathit{BadConsts}}{(\mathtt{iszero}\ \mathtt{t}_1, C) \in \mathit{BadConsts}}$$

This set of rules defines a perfectly good *relation* — it's just that this relation does not happen to be a function!

Just for fun, let's calculate some cases of this relation...

- For what values of C do we have $(\mathtt{false}, C) \in \mathit{BadConsts}$?

This set of rules defines a perfectly good *relation* — it's just that this relation does not happen to be a function!

Just for fun, let's calculate some cases of this relation...

- For what values of C do we have $(\texttt{false}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{succ 0}, C) \in \mathit{BadConsts}$?

This set of rules defines a perfectly good *relation* — it's just that this relation does not happen to be a function!

Just for fun, let's calculate some cases of this relation...

- For what values of C do we have $(\texttt{false}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{succ 0}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{if false then 0 else 0}, C) \in \mathit{BadConsts}$?

This set of rules defines a perfectly good *relation* — it's just that this relation does not happen to be a function!

Just for fun, let's calculate some cases of this relation...

- For what values of C do we have $(\texttt{false}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{succ 0}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{if false then 0 else 0}, C) \in \mathit{BadConsts}$?
- For what values of C do we have $(\texttt{iszero 0}, C) \in \mathit{BadConsts}$?

Another Inductive Definition

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

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$.

Proof:

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$.

Proof: By induction on $\mathtt{t}$.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$.

Proof: By induction on $\mathtt{t}$.
Assuming the desired property for immediate subterms of $\mathtt{t}$, we must prove it for $\mathtt{t}$ itself.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$.

Proof: By induction on $\mathtt{t}$.
Assuming the desired property for immediate subterms of $\mathtt{t}$, we must prove it for $\mathtt{t}$ itself.

There are "three" cases to consider:

Case: $\mathtt{t}$ is a constant

Immediate: $|\mathit{Consts}(\mathtt{t})| = |\{\mathtt{t}\}| = 1 = \mathit{size}(\mathtt{t})$.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|Consts(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on $\mathtt{t}$.
Assuming the desired property for immediate subterms of $\mathtt{t}$, we must prove it for $\mathtt{t}$ itself.

There are "three" cases to consider:

Case: $\mathtt{t}$ is a constant

Immediate: $|Consts(\mathtt{t})| = |\{\mathtt{t}\}| = 1 = size(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1$, $\mathtt{pred}\ \mathtt{t}_1$, or $\mathtt{iszero}\ \mathtt{t}_1$

By the induction hypothesis, $|Consts(\mathtt{t}_1)| \leq size(\mathtt{t}_1)$. We now calculate as follows:
$|Consts(\mathtt{t})| = |Consts(\mathtt{t}_1)| \leq size(\mathtt{t}_1) < size(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

By the induction hypothesis, $|Consts(\mathtt{t}_1)| \leq size(\mathtt{t}_1)$, $|Consts(\mathtt{t}_2)| \leq size(\mathtt{t}_2)$, and $|Consts(\mathtt{t}_3)| \leq size(\mathtt{t}_3)$. We now calculate as follows:

$$\begin{array}{rcl} |Consts(\mathtt{t})| & = & |Consts(\mathtt{t}_1) \cup Consts(\mathtt{t}_2) \cup Consts(\mathtt{t}_3)| \\ & \leq & |Consts(\mathtt{t}_1)| + |Consts(\mathtt{t}_2)| + |Consts(\mathtt{t}_3)| \\ & \leq & size(\mathtt{t}_1) + size(\mathtt{t}_2) + size(\mathtt{t}_3) \\ & < & size(\mathtt{t}). \end{array}$$

Operational Semantics

Abstract Machines

An *abstract machine* consists of:

- a set of *states*
- a *transition relation* on states, written $\longrightarrow$

We read "$\mathtt{t} \longrightarrow \mathtt{t}'$" as "$\mathtt{t}$ evaluates to $\mathtt{t}'$ in one step".

A state records *all* the information in the machine at a given moment. For example, an abstract-machine-style description of a conventional microprocessor would include the program counter, the contents of the registers, the contents of main memory, and the machine code program being executed.

Abstract Machines

For the very simple languages we are considering at the moment, however, the term being evaluated is the whole state of the abstract machine.

Nb. Often, the transition relation is actually a partial function: i.e., from a given state, there is at most one possible next state. But in general there may be many.

Operational semantics for Booleans

Syntax of terms and values

`t ::=`		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v ::=`		*values*
	`true`	*true value*
	`false`	*false value*

Evaluation relation for Booleans

The evaluation relation $\mathtt{t} \longrightarrow \mathtt{t}'$ is the smallest relation closed under the following rules:

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_2} \qquad \text{(E-IfTrue)}$$

$$\mathtt{if\ false\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3} \qquad \text{(E-IfFalse)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}} \qquad \text{(E-If)}$$

Terminology

Computation rules:

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_2} \qquad \text{(E-IfTrue)}$$

$$\mathtt{if\ false\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3} \qquad \text{(E-IfFalse)}$$

Congruence rule:

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}} \qquad \text{(E-If)}$$

Computation rules perform "real" computation steps.
Congruence rules determine *where* computation rules can be applied next.

Evaluation, more explicitly

$\longrightarrow$ is the smallest two-place relation closed under the following rules:

$$((\texttt{if true then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3), \mathtt{t}_2) \quad \in \quad \longrightarrow$$

$$((\texttt{if false then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3), \mathtt{t}_3) \quad \in \quad \longrightarrow$$

$$\frac{(\mathtt{t}_1, \mathtt{t}_1') \quad \in \quad \longrightarrow}{((\texttt{if } \mathtt{t}_1 \texttt{ then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3), (\texttt{if } \mathtt{t}_1' \texttt{ then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3)) \quad \in \quad \longrightarrow}$$

The notation $\mathtt{t} \longrightarrow \mathtt{t}'$ is short-hand for $(\mathtt{t}, \mathtt{t}') \in \longrightarrow$.