Administrivia

- HW3 is available now; due next Monday.
- Acceptable submission formats:
  - PDF generated by LaTeX (preferred!)
  - PDF generated in some other way, e.g. Word (if you must)
  - ascii (as a last resort)
  - not raw Word files or handwritten
- If you foresee a lot of technical writing in your future, you should consider learning how to use LaTeX now.
- Recitations started last week.
  - Review recitations: Ask questions, see examples worked out.

Readings

As mentioned before, 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 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: \( 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \), for every \( n \).
Proof: Let \( P(i) \) be “\( 2^0 + 2^1 + \ldots + 2^i = 2^{i+1} - 1 \).”

▶ Show \( P(0) \):
\[
2^0 = 1 = 2^1 - 1
\]

▶ Show that \( P(i) \) implies \( P(i + 1) \):
\[
\begin{align*}
2^0 + 2^1 + \ldots + 2^{i+1} &= (2^0 + 2^1 + \ldots + 2^i) + 2^{i+1} \\
&= (2^{i+1} - 1) + 2^{i+1} \quad \text{by IH} \\
&= 2 \cdot (2^{i+1}) - 1 \\
&= 2^{i+2} - 1
\end{align*}
\]

▶ The result (\( P(n) \) for all \( n \)) follows by the principle of (ordinary) induction.

Shorthand form
Theorem: \( 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \), for every \( n \).
Proof: By induction on \( n \).

▶ Base case (\( n = 0 \)):
\[
2^0 = 1 = 2^1 - 1
\]

▶ Inductive case (\( n = i + 1 \)):
\[
\begin{align*}
2^0 + 2^1 + \ldots + 2^{i+1} &= (2^0 + 2^1 + \ldots + 2^i) + 2^{i+1} \\
&= (2^{i+1} - 1) + 2^{i+1} \quad \text{IH} \\
&= 2 \cdot (2^{i+1}) - 1 \\
&= 2^{i+2} - 1
\end{align*}
\]

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.
Simple Arithmetic Expressions

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

\[
t ::= \text{true} \quad \text{false} \quad \text{if } t \text{ then } t \text{ else } t \quad 0 \quad \text{succ } t \quad \text{pred } t \quad \text{iszero } t
\]

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?

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 \( T \) of terms is the smallest set such that

1. \( \{\text{true, false, 0}\} \subseteq T \);
2. if \( t_1 \in T \), then \( \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq T \);
3. if \( t_1 \in T, t_2 \in T, \text{ and } t_3 \in T \), then
   - if \( t_1 \) then \( t_2 \) else \( t_3 \) \( \in T \).
Inference rules

An alternate notation for the same definition:

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

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, \( S_0, S_1, S_2, \ldots \), as follows:

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

Now let \( S = \bigcup S_i \)

Comparing the definitions

We have seen two different presentations of terms:

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

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

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 Syntax
Induction on Terms

Definition: The depth of a term \( t \) is the smallest \( i \) such that \( t \in S_i \).

From the definition of \( S \), it is clear that, if a term \( t \) is in \( S_i \), then all of its immediate subterms must be in \( S_{i-1} \), 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 \( t \).

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 \( t \), written \( \text{Consts}(t) \), is defined as follows:

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

Simple, right?

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 \( \text{Consts} \) be the smallest two-place relation closed under the following rules:

\[
\begin{align*}
(\text{true}, \{\text{true}\}) & \in \text{Consts} \\
(\text{false}, \{\text{false}\}) & \in \text{Consts} \\
(0, \{0\}) & \in \text{Consts} \\
(\text{succ } t_1, C) & \in \text{Consts} \\
(t_1, C) & \in \text{Consts} \\
(\text{pred } t_1, C) & \in \text{Consts} \\
(t_1, C) & \in \text{Consts} \\
(\text{iszero } t_1, C) & \in \text{Consts} \\
(t_1, C_1) & \in \text{Consts} \\
(t_2, C_2) & \in \text{Consts} \\
(t_3, C_3) & \in \text{Consts} \\
(\text{if } t_1 \text{ then } t_2 \text{ else } t_3, (\text{Consts}(t_1) \cup \text{Consts}(t_2) \cup \text{Consts}(t_3))) & \in \text{Consts}
\end{align*}
\]

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?
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 \( t \) there is exactly one set of terms \( C \) such that \( (t, C) \in \text{Consts} \).

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 \( (s, C_s) \in \text{Consts} \)

then

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

Proceed by cases on the form of \( t \).

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

- If $t$ is $\text{succ } t_1$, then the induction hypothesis tells us that there is exactly one set of terms $C_1$ such that $(t_1, C_1) \in Consts$. But then it is clear from the definition of $Consts$ that there is exactly one set $C$ (namely $C_1$) such that $(t, C) \in Consts$.

- If $t$ is $\text{if } s_1 \text{ then } s_2 \text{ else } s_3$, then the induction hypothesis tells us that
  - there is exactly one set of terms $C_1$ such that $(t_1, C_1) \in Consts$
  - there is exactly one set of terms $C_2$ such that $(t_2, C_2) \in Consts$
  - there is exactly one set of terms $C_3$ such that $(t_3, C_3) \in Consts$

  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 $(t, C) \in Consts$.

How about the bad definition?

\[
\begin{align*}
\text{true, } \{\text{true}\} & \in \text{BadConsts} \\
\text{false, } \{\text{false}\} & \in \text{BadConsts} \\
(0, \{0\}) & \in \text{BadConsts} \\
(0, \{\}\}) & \in \text{BadConsts} \\
(t_1, C) & \in \text{BadConsts} \\
(\text{succ } t_1, C) & \in \text{BadConsts} \\
(t_1, C) & \in \text{BadConsts} \\
(\text{pred } t_1, C) & \in \text{BadConsts} \\
(\text{iszero } (\text{iszero } t_1), C) & \in \text{BadConsts} \\
(\text{iszero } t_1, C) & \in \text{BadConsts}
\end{align*}
\]

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

Another proof by induction

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

**Proof:** By induction on $t$.

Assuming the desired property for immediate subterms of $t$, we must prove it for $t$ itself.
Another proof by induction

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

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

There are "three" cases to consider:

**Case:** $t$ is a constant
Immediate: $|\text{Consts}(t)| = |\{t\}| = 1 = \text{size}(t)$.

**Case:** $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
By the induction hypothesis, $|\text{Consts}(t_1)| \leq \text{size}(t_1)$, $|\text{Consts}(t_2)| \leq \text{size}(t_2)$, and $|\text{Consts}(t_3)| \leq \text{size}(t_3)$. We now calculate as follows:

\[
|\text{Consts}(t)| = |\text{Consts}(t_1) \cup \text{Consts}(t_2) \cup \text{Consts}(t_3)| \\
\leq |\text{Consts}(t_1)| + |\text{Consts}(t_2)| + |\text{Consts}(t_3)| \\
\leq \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) \\
< \text{size}(t).
\]

**Case:** $t = \text{succ } t_1, \text{pred } t_1$, or $\text{iszero } t_1$
By the induction hypothesis, $|\text{Consts}(t_1)| \leq \text{size}(t_1)$. We now calculate as follows:

\[
|\text{Consts}(t)| = |\text{Consts}(t_1)| \leq \text{size}(t_1) < \text{size}(t).
\]

**Operational Semantics**

**Abstract Machines**

An abstract machine consists of:
- a set of states
- a transition relation on states, written $\rightarrow$

We read "$t \rightarrow t'$" as "$t$ evaluates to $t'$ 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
\[ t ::= \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{if } t \text{ then } t \text{ else } t \]

\[ v ::= \]
\[ \text{true} \]
\[ \text{false} \]

Evaluation relation for Booleans

The evaluation relation \( t \xrightarrow{} t' \) is the smallest relation closed under the following rules:

\[ \text{if true then } t_2 \text{ else } t_3 \xrightarrow{} t_2 \quad \text{(E-IfTrue)} \]
\[ \text{if false then } t_2 \text{ else } t_3 \xrightarrow{} t_3 \quad \text{(E-IfFalse)} \]

\[ t_1 \xrightarrow{} t'_1 \quad \text{(E-If)} \]
\[ (\text{if } t_1 \text{ then } t_2 \text{ else } t_3), (\text{if } t'_1 \text{ then } t_2 \text{ else } t_3) \xrightarrow{} \]

Terminology

Computation rules:

if true then t_2 else t_3 \xrightarrow{} t_2 \quad \text{(E-IfTrue)}

if false then t_2 else t_3 \xrightarrow{} t_3 \quad \text{(E-IfFalse)}

Congruence rule:

\[ t_1 \xrightarrow{} t'_1 \quad \text{(E-If)} \]
\[ (\text{if } t_1 \text{ then } t_2 \text{ else } t_3), (\text{if } t'_1 \text{ then } t_2 \text{ else } t_3) \xrightarrow{} \]

Evaluation, more explicitly

\( \xrightarrow{} \) is the smallest two-place relation closed under the following rules:

\[ ((\text{if true then } t_2 \text{ else } t_3), t_2) \xrightarrow{} \]
\[ ((\text{if false then } t_2 \text{ else } t_3), t_3) \xrightarrow{} \]
\[ (t_1, t'_1) \xrightarrow{} \]
\[ ((\text{if } t_1 \text{ then } t_2 \text{ else } t_3), (\text{if } t'_1 \text{ then } t_2 \text{ else } t_3)) \xrightarrow{} \]

The notation \( t \xrightarrow{} t' \) is short-hand for \((t, t') \xrightarrow{}\).