### Foundations of Software Winter Semester 2007

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.

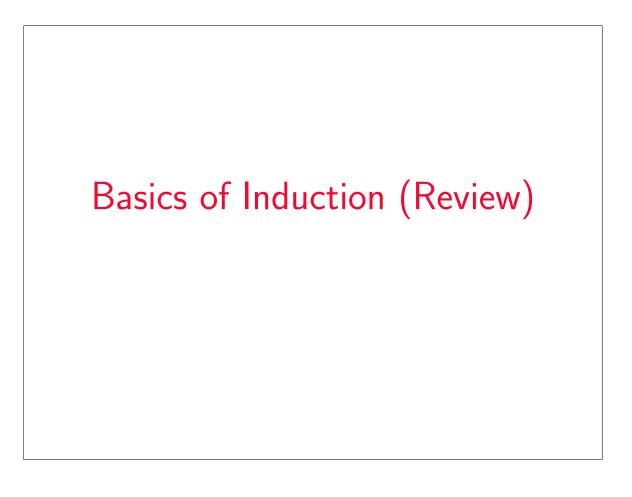
#### Warning!

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.



#### Induction

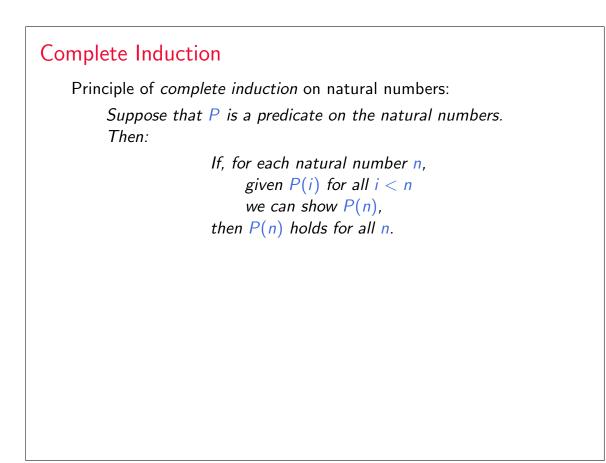
Principle of ordinary induction on natural numbers:

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): $2^{0} + 2^{1} + ... + 2^{i+1} = (2^{0} + 2^{1} + ... + 2^{i}) + 2^{i+1} = (2^{i+1} - 1) + 2^{i+1} = 2 \cdot (2^{i+1} - 1) + 2^{i+1} = 2^{i+2} - 1$ by IH $= 2^{i+2} - 1$ 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):  $2^{0} + 2^{1} + ... + 2^{i+1} = (2^{0} + 2^{1} + ... + 2^{i}) + 2^{i+1} = (2^{i+1} - 1) + 2^{i+1} = 2 \cdot (2^{i+1} - 1) + 2^{i+1}$  IH  $= 2 \cdot (2^{i+1} - 1) = 2^{i+2} - 1$ 



#### 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	::=	
		true
		false
		if t then t else t
		0
		succ t
		pred t
		iszero t

terms constant true constant false conditional constant zero successor predecessor 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

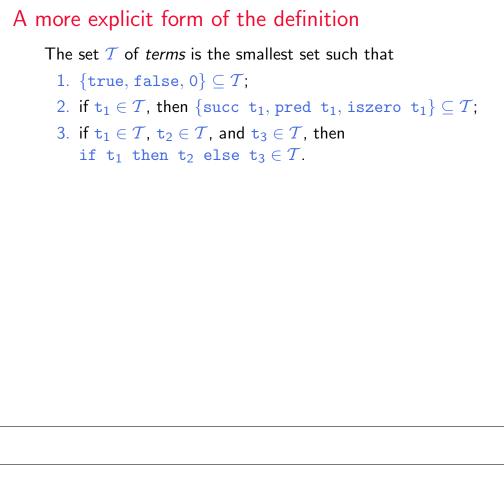
Q: Does this grammar define a set of *character strings*, a set of *token lists*, or a set of *abstract syntax trees*?

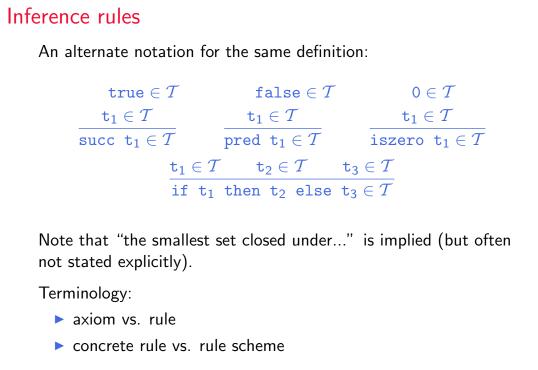
A: In a sense, all three. But we are primarily interested, here, in abstract syntax trees.

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))
?
```





#### Terms, concretely

Define an infinite sequence of sets,  $S_0$ ,  $S_1$ ,  $S_2$ , ..., as follows:

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

Now let

 $S = \bigcup_i S_i$ 

#### 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* (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* (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: the definition of terms as the smallest set with a certain closure property is compact and easy to read 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 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.

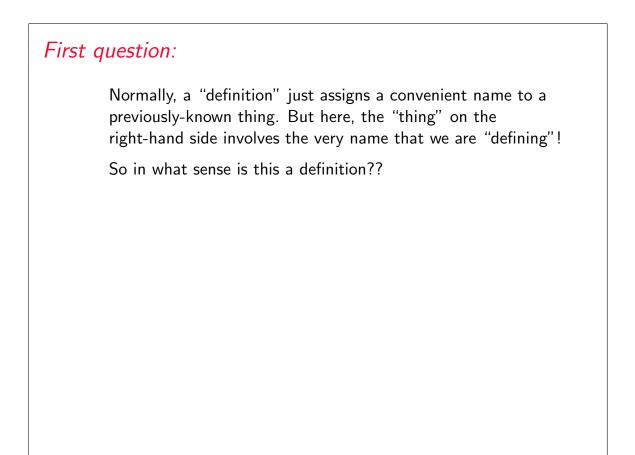
This observation justifies the *principle of induction on terms*. Let P be a predicate on terms.

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.

#### Inductive Function Definitions

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

Simple, right?



#### Second question:

Suppose we had written this instead...

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

BadConsts(true)	=	$\{\texttt{true}\}$		
<pre>BadConsts(false)</pre>	=	{false}		
BadConsts(0)	=	{0}		
BadConsts(0)	=	{}		
$BadConsts(succ t_1)$	=	$BadConsts(t_1)$		
$BadConsts(pred t_1)$	=	$BadConsts(t_1)$		
$BadConsts(iszero t_1)$	=	BadConsts(iszero	(iszero	$t_1))$

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, 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:

 $(true, {true}) \in Consts$   $(false, {false}) \in Consts$   $(0, {0}) \in Consts$   $\frac{(t_1, C) \in Consts}{(succ t_1, C) \in Consts}$   $\frac{(t_1, C) \in Consts}{(pred t_1, C) \in Consts}$   $\frac{(t_1, C) \in Consts}{(iszero t_1, C) \in Consts}$   $\frac{(t_1, C) \in Consts}{(iszero t_1, C) \in Consts}$   $\frac{(t_1, C_1) \in Consts}{(if t_1 then t_2 else t_3, C_1 \cup C_2 \cup C_3) \in Consts}$ 

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

**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  $(t, C) \in Consts$ .

**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  $(t, C) \in Consts$ .

**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  $(s, C_s) \in Consts$ 

then

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

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*) ∈ *Consts*.

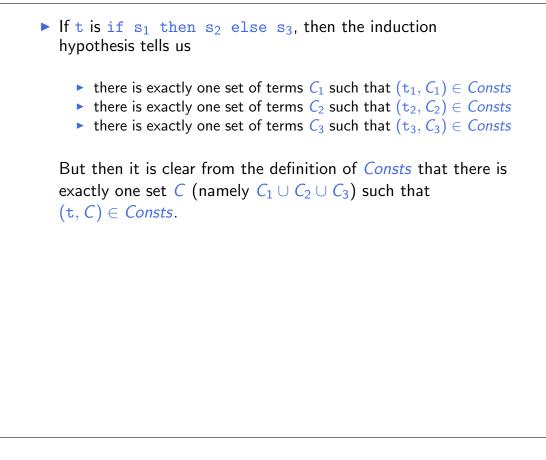
Proceed by cases on the form of t.

- 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*) ∈ *Consts*.
- If t is succ t<sub>1</sub>, then the induction hypothesis tells us that there is exactly one set of terms C<sub>1</sub> such that (t<sub>1</sub>, C<sub>1</sub>) ∈ Consts. But then it is clear from the definition of Consts that there is exactly one set C (namely C<sub>1</sub>) such that (t, C) ∈ 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) ∈ *Consts*.
- If t is succ t<sub>1</sub>, then the induction hypothesis tells us that there is exactly one set of terms C<sub>1</sub> such that
   (t<sub>1</sub>, C<sub>1</sub>) ∈ Consts. But then it is clear from the definition of
   Consts that there is exactly one set C (namely C<sub>1</sub>) such that
   (t, C) ∈ Consts.

Similarly when t is pred  $t_1$  or iszero  $t_1$ .



How about the bad definition?

 $(\texttt{true}, \{\texttt{true}\}) \in BadConsts$   $(\texttt{false}, \{\texttt{false}\}) \in BadConsts$   $(0, \{0\}) \in BadConsts$   $(0, \{\}) \in BadConsts$   $(\texttt{t}_1, C) \in BadConsts$   $(\texttt{iszero } \texttt{t}_1, C) \in BadConsts$   $(\texttt{iszero } \texttt{t}_1, C) \in BadConsts$ 

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 (false, *C*) ∈ *BadConsts*?
- For what values of C do we have (succ  $0, C) \in BadConsts?$

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

#### Another Inductive Definition

#### Another proof by induction

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

**Proof:** 

#### Another proof by induction

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

**Proof:** By induction on t.

#### Another proof by induction

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

**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.,  $|Consts(t)| \leq size(t)$ .

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

#### Another proof by induction

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

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

Case:  $t = succ t_1$ , pred  $t_1$ , or iszero  $t_1$ 

By the induction hypothesis,  $|Consts(t_1)| \le size(t_1)$ . We now calculate as follows:

 $|Consts(t)| = |Consts(t_1)| \le size(t_1) < size(t).$ 

```
 \begin{array}{ll} \textit{Case:} & \texttt{t} = \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \\ \textit{By the induction hypothesis, } |\textit{Consts}(\texttt{t}_1)| \leq \textit{size}(\texttt{t}_1), \\ |\textit{Consts}(\texttt{t}_2)| \leq \textit{size}(\texttt{t}_2), \texttt{ and } |\textit{Consts}(\texttt{t}_3)| \leq \textit{size}(\texttt{t}_3). \textit{ We now calculate as follows:} \\ \\ |\textit{Consts}(\texttt{t})| &= |\textit{Consts}(\texttt{t}_1) \cup \textit{Consts}(\texttt{t}_2) \cup \textit{Consts}(\texttt{t}_3)| \\ \leq |\textit{Consts}(\texttt{t}_1)| + |\textit{Consts}(\texttt{t}_2)| + |\textit{Consts}(\texttt{t}_3)| \\ \leq \textit{size}(\texttt{t}_1) + \textit{size}(\texttt{t}_2) + \textit{size}(\texttt{t}_3) \\ \leq \textit{size}(\texttt{t}). \\ \\ \leq \textit{size}(\texttt{t}). \end{array}
```

## **Operational Semantics**

#### Abstract Machines

An abstract machine consists of:

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

We read "t  $\longrightarrow$  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 ::=

false

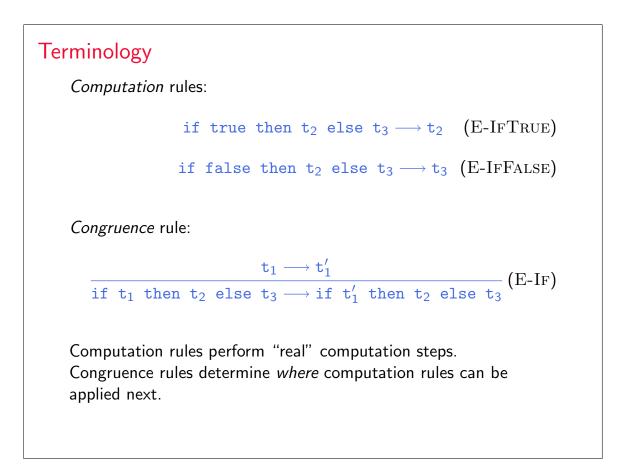
true
false
if t then t else t

v ::= true terms constant true constant false conditional

values true value false value

#### Evaluation relation for Booleans

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



#### Evaluation, more explicitly

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

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