Foundations of Software
Spring 2025

Week 1

Logistic, Proof Principles, Abstract Syntax and Abstract Machine

Week 1

Logistic

The Foundations of Software course consists of

- Lectures (Mon 8:15-11:00, ELD 020)
- Exercises: solve written exercise sheets (Tue 8:15-10:00, GRB 330)
- Projects: graded programming and mechanization assignments (Tue 10:15-12:00, CE 1105). (20% of the grade)
- One large homework mid-may (20% of the grade)
- A written final exam (60 % of the grade)

Assignments

- Five programming and mechanization assignments throughout the semester (in Lean4) - due every 2-3 weeks
- Each project focuses on theoretical aspects covered in class
- New assignments released on Moodle after previous deadline
- Live explanation to TA required during Tuesday sessions (10:00, CE1105)

Teaching Team

Lecturers:

- Martin Odersky
  martin.odersky@epfl.ch
- Thomas Bourgeat
  thomas.bourgeat@epfl.ch

Teaching Assistants:

- Martina Camaioni
  martina.camaioni@epfl.ch
- Yichen Xu
  yichen.xu@epfl.ch
- Cao Nguyen Pham
  nguyen.pham@epfl.ch

Textbook

The lectures are closely following the textbook:

> Types and Programming Languages,
> Benjamin C. Pierce, MIT Press, 2002.

The electronic version of the book is available for free in the EPFL digital library.

Textbook

The lectures are closely following the textbook:

> Types and Programming Languages,
> Benjamin C. Pierce, MIT Press, 2002.

The electronic version of the book is available for free in the EPFL digital library.

These slides are based on teaching materials originally developed by Benjamin C. Pierce, with contributions from Martin Odersky, Sébastien Doeraene, and Thomas Bourgeat.

Today the material is based on Chapter 2 and Chapter 3.

Part I
Modeling programming languages

Where we're going

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.

Usefulness

In theory, what is this theory useful/good for:

- help design principled programming languages
- shed some light on compilation
- verify the correctness of programs (and of hardware)
- serve as a foundation for logic
- build proof systems

Usefulness

In theory, what is this theory useful/good for:

- help design principled programming languages
- shed some light on compilation
- verify the correctness of programs (and of hardware)
- serve as a foundation for logic
- build proof systems

This theory is not only paper theory. We will *mechanize* most concepts in the Lean4 proof assistant (by the end of tomorrow, it will be clearer what this means).

Lean4

- Lean4 is both a proof system *and* a programming language
- FoS is *not* a Lean4 class, though you will learn Lean4, and it might end up being one of the challenges of the course

Exercise of the week: an intuitive/gamified introduction to Lean4 (ungraded):

adam.math.hhu.de/#/g/leanprover-community/nng4

Lean4

- Lean4 is both a proof system *and* a programming language
- FoS is *not* a Lean4 class, though you will learn Lean4, and it might end up being one of the challenges of the course

Exercise of the week: an intuitive/gamified introduction to Lean4 (ungraded):

adam.math.hhu.de/#/g/leanprover-community/nng4

By tomorrow morning for exercise and project sessions, you should install Lean4 on your machine. If you have trouble you can ask for help to the TAs in the sessions.

Warning!

The material in the next couple of slides 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."

Mathematical Foundations - Memories of BA1 and BA3 (though not so easy!)

Functions

- A function $f : A \to B$ is a mapping that associates each element in set A with exactly one element in set B
- Key properties:
  - Total: every element in A must be mapped
  - Well-defined: each input maps to exactly one output
- Example: $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = n + 1$

Relations

A binary relation R between sets A and B is a subset of $A \times B$

- Example: $R = \{(x, 2x) | x \in N\}$
- Example of a few common properties for binary relations:
  - Reflexive: $\forall x.(x, x) \in R$
  - Symmetric: $(x, y) \in R \implies (y, x) \in R$
  - Transitive: $(x, y) \in R \wedge (y, z) \in R \implies (x, z) \in R$
- Functions are binary relations. Not every binary relation is a function. Why?

Relations

A binary relation R between sets A and B is a subset of $A \times B$

- Example: $R = \{(x, 2x) | x \in N\}$
- Example of a few common properties for binary relations:
  - Reflexive: $\forall x.(x, x) \in R$
  - Symmetric: $(x, y) \in R \implies (y, x) \in R$
  - Transitive: $(x, y) \in R \wedge (y, z) \in R \implies (x, z) \in R$
- Functions are binary relations. Not every binary relation is a function. Why?

One can also have n-ary relations, as subsets of $A_1, \ldots, A_n$.
Example:

$$S = \{(x, y, z) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mid x + y = z\}$$

- $(2, 3, 5) \in S$ since $2 + 3 = 5$
- $(0, 7, 7) \in S$ since $0 + 7 = 7$
- $(2, 2, 5) \notin S$ since $2 + 2 \neq 5$

Induction

Principle of *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 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 ($n = i + 1$):

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

Strong Induction

Principle of *strong 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.

Example of strong induction (shorthand form)

Theorem: Every natural $n > 1$ is the product of (one or more) prime numbers.

Proof: By strong induction on n.

- IH: Every natural $1 < m < n$ is the product of prime numbers.
- If n is a prime number, then it is the product of itself.
- Otherwise,
  - By definition, there exist $1 < m_1, m_2 < n$ such that $n = m_1 m_2$.
  - By the IH, m_1 and m_2 are both the product of prime numbers.
  - Therefore, $n = m_1 m_2$ is also the product of prime numbers.

Strong versus ordinary induction

Ordinary and complete induction are *interderivable* — assuming one, we can prove the other (can you do it?).

Inductively Defined Sets

An inductively defined set S is specified by:

- Base cases: elements that are in S
- Inductive rules: ways to construct new elements in S

Example: Natural numbers $\mathbb{N}$

- Base case: $0 \in \mathbb{N}$
- Inductive rule: If $n \in \mathbb{N}$, then $1 + n \in \mathbb{N}$

Mathematically, $\mathbb{N}$ is defined as the smallest set that contains the base case and that is closed by the inductive rules.

Lists of Integers

Let's define the set L of objects that can represent lists of integers inductively:

- Base case:
  - There is an empty list nil in L
- Inductive rule:
  - If $\ell \in L$ and $n \in \mathbb{Z}$, then we can build a new list by adding n at the front, $\text{cons}(n, \ell)$, which is in L

Lists of Integers

Let's define the set L of objects that can represent lists of integers inductively:

- Base case:
  - There is an empty list nil in L
- Inductive rule:
  - If $\ell \in L$ and $n \in \mathbb{Z}$, then we can build a new list by adding n at the front, $\text{cons}(n, \ell)$, which is in L

Examples:

- $\text{cons}(3, \text{nil})$ is in L (singleton list [3])
- $\text{cons}(1, \text{cons}(2, \text{nil}))$ is in L (list [1,2])
- $\text{cons}(-1, \text{cons}(0, \text{cons}(1, \text{nil})))$ is in L (list [-1,0,1])

Reasoning about inductively defined sets

An inductively defined set comes with *an induction principle* (often called structural induction).
We can prove a property for all the elements of an inductively defined set, using structural induction.

Reasoning about inductively defined sets

An inductively defined set comes with *an induction principle* (often called structural induction).
We can prove a property for all the elements of an inductively defined set, using structural induction.

Example: Natural numbers $\mathbb{N}$

- Base case: $0 \in \mathbb{N}$
- Inductive rule: If $n \in \mathbb{N}$, then $1 + n \in \mathbb{N}$

You already know the induction principle corresponding to this inductively defined set!

Well-Parenthesized Words

Let's define the set W of objects to model well-parenthesized words:

- Base case:
  - There is the empty word ε in W
- Inductive rules:
  - If $w \in W$, then we can build a wrapped word, (w), which is in W
  - If $w_1, w_2 \in W$, then we build the concatenation of the two words, $w_1 w_2$, which is in W

Well-Parenthesized Words

Let's define the set W of objects to model well-parenthesized words:

- Base case:
  - There is the empty word ε in W
- Inductive rules:
  - If $w \in W$, then we can build a wrapped word, (w), which is in W
  - If $w_1, w_2 \in W$, then we build the concatenation of the two words, $w_1 w_2$, which is in W

Examples:

- $()$ is in W (using base case then first rule)
- $(())$ is in W (using previous example and first rule)
- $()()$ is in W (using second rule with $w_1 = w_2 = ()$)

Length Function and Evenness Property

Length function $len : W \to \mathbb{N}$ defined *recursively*:

- $len\ \varepsilon = 0$
- $len\ (w) = len\ w + 2$
- $len\ w_1 w_2 = len\ w_1 + len\ w_2$

Theorem: For all $w \in W$, $len\ w$ is even.

Length Function and Evenness Property

Length function $len : W \to \mathbb{N}$ defined *recursively*:

- $len\ \varepsilon = 0$
- $len\ (w) = len\ w + 2$
- $len\ w_1 w_2 = len\ w_1 + len\ w_2$

Theorem: For all $w \in W$, $len\ w$ is even.

Proof: By induction on the structure of $w \in W$:

- Base case: $len(\varepsilon) = 0$ is even
- Case (w): If $len\ w$ is even, then $len\ (w) = len\ w + 2$ is even
- Case $w_1 w_2$: If $len\ w_1$ and $len\ w_2$ are even, then their sum $len\ w_1 w_2$ is even

Indexed family of sets: Lists of Size n

Instead of defining a *single* set, we can define simultaneously a *family* of sets.
Let's define a family of sets $\{L_n\}_{n\in\mathbb{N}}$ where L_n contains all lists of integers of length exactly n:

- Base case: $L_0 = \{\text{nil}\}$
- Inductive rule: For any $n \geq 0$ and $k \in \mathbb{Z}$:
  If $\ell \in L_n$ then $\text{cons}(k, \ell) \in L_{n+1}$

Indexed family of sets: Lists of Size n

Instead of defining a *single* set, we can define simultaneously a *family* of sets.
Let's define a family of sets $\{L_n\}_{n\in\mathbb{N}}$ where L_n contains all lists of integers of length exactly n:

- Base case: $L_0 = \{\text{nil}\}$
- Inductive rule: For any $n \geq 0$ and $k \in \mathbb{Z}$:
  If $\ell \in L_n$ then $\text{cons}(k, \ell) \in L_{n+1}$

Examples:

- $L_0 = \{\text{nil}\}$
- $L_1 = \{\text{cons}(k, \text{nil}) \mid k \in \mathbb{Z}\}$
  (all singleton lists)
- $L_2 = \{\text{cons}(k_1, \text{cons}(k_2, \text{nil})) \mid k_1, k_2 \in \mathbb{Z}\}$
  (all lists of length 2)

Inductively Defined Predicate (or Proposition)

A predicate $P(x)$, which is a function, can be thought of as a set $\{x|P(x) \text{ is true}\}$. Hence, we can define propositions (logical statements) inductively:

- An inductively defined predicate P is defined by:
  - Base cases: directly true statements
  - Inference rules: ways to derive new true statements

Inductively Defined Predicate (or Proposition)

A predicate $P(x)$, which is a function, can be thought of as a set $\{x|P(x) \text{ is true}\}$. Hence, we can define propositions (logical statements) inductively:

- An inductively defined predicate P is defined by:
  - Base cases: directly true statements
  - Inference rules: ways to derive new true statements

- Example: A predicate $\text{String} \to \text{Bool}$ that returns true for well-parenthesized strings (written $WP\ s$)
  - Base: WP "" is true
  - Rules: Adding matching parentheses and concatenation
    If we have s such that $WP\ s$, then WP "("++s++")"

    If we have s_1 and s_2 such that $WP\ s_1$ and $WP\ s_2$ then $WP\ s_1$++s_2

Inductively Defined Propositions - alternative formulation

Example: Well-parenthesized strings (written $WP\ s$)

- Base: Empty string is well-parenthesized

$$\frac{}{WP\ \varepsilon}$$ [WP-Empty]

- Rules: Adding matching parentheses and concatenation

$$\frac{WP\ s}{WP\ \texttt{"("}\texttt{++}s\texttt{++"}\texttt{)"}}$$ [WP-Wrap]

$$\frac{WP\ s_1 \quad WP\ s_2}{WP\ s_1\texttt{++}s_2}$$ [WP-Concat]

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. $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$;
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:

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

$$\frac{t_1 \in \mathcal{T}}{\texttt{succ } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{pred } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{iszero } t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \in \mathcal{T}}$$

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

Terminology:

- axiom vs. rule
- concrete rule vs. rule schema

Terms, concretely

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

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

Now let

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

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"?

Operational Semantics

Abstract Machines

An *abstract machine* consists of:

- a set of *states*
- a *transition relation* on states, written $\longrightarrow$. Note that mathematically $\longrightarrow$ is a binary relation!

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:

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

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

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

Terminology

Computation rules:

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

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

Congruence rule:

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

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:

$$((\texttt{if true then } t_2 \texttt{ else } t_3), t_2) \quad \in \quad \longrightarrow$$

$$((\texttt{if false then } t_2 \texttt{ else } t_3), t_3) \quad \in \quad \longrightarrow$$

$$\frac{(t_1, t_1') \quad \in \quad \longrightarrow}{((\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3), (\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3)) \quad \in \quad \longrightarrow}$$

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

An example

Let $\mathtt{t}$ be the term

$\mathtt{if\ true\ then\ (if\ false\ then\ false\ else\ true)\ else\ false}$

What is $\mathtt{t}'$ in $\mathtt{t} \longrightarrow \mathtt{t}'$?

1. $\mathtt{t}' = \mathtt{true}$
2. $\mathtt{t}' = \mathtt{if\ true\ then\ true\ else\ false}$
3. $\mathtt{t}' = \mathtt{if\ false\ then\ false\ else\ true}$
4. There is no such $\mathtt{t}'$
5. I don't know

Reading for next week

- Chapter 3 – Untyped Arithmetic Expressions
- Some of it is recap; most important:
  - 3.3 Induction on terms
  - 3.5 Evaluation

Induction on Syntax

Recall: terms, concretely

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

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

Now let

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

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

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.*

Recursive Function Definitions

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

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

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 *BadConsts*($\mathtt{t}$), is defined as follows:

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

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

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

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

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

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

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

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

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

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

Proof:

Theorem:

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

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

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

Theorem:

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

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

Proof: By induction on $\mathtt{t}$.
To apply the induction principle for terms, we must show, for an arbitrary term $\mathtt{t}$, that if

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

then

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

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 $(\mathtt{t}, C) \in \mathit{Consts}$.

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 $(\mathtt{t}, C) \in \mathit{Consts}$.

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

- If $\texttt{t}$ is $\texttt{0}$, $\texttt{true}$, or $\texttt{false}$, 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 $\texttt{t}$ is $\texttt{succ}\ \texttt{t}_1$, 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 $(\texttt{t}, C) \in \mathit{Consts}$.

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

- If $\texttt{t}$ is $\texttt{if } \texttt{s}_1 \texttt{ then } \texttt{s}_2 \texttt{ else } \texttt{s}_3$, then the induction hypothesis tells us

  - there is exactly one set of terms C_1 such that $(\texttt{t}_1, C_1) \in \mathit{Consts}$
  - there is exactly one set of terms C_2 such that $(\texttt{t}_2, C_2) \in \mathit{Consts}$
  - there is exactly one set of terms C_3 such that $(\texttt{t}_3, C_3) \in \mathit{Consts}$

  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 $(\texttt{t}, C) \in \mathit{Consts}$.

How about the bad definition?

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

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

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

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

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

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

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

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 $(\texttt{false}, C) \in \mathit{BadConsts}$?

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 Recursive 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., $|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$.

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

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

By the induction hypothesis, $|\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1)$. We now calculate as follows:
$|\mathit{Consts}(\mathtt{t})| = |\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1) < \mathit{size}(\mathtt{t}_1) + 1 = \mathit{size}(\mathtt{t})$.

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

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

$$\begin{array}{rcl} |\mathit{Consts}(\mathtt{t})| & = & |\mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \cup \mathit{Consts}(\mathtt{t}_3)| \\ & \leq & |\mathit{Consts}(\mathtt{t}_1)| + |\mathit{Consts}(\mathtt{t}_2)| + |\mathit{Consts}(\mathtt{t}_3)| \\ & \leq & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) \\ & < & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) + 1 \\ & = & \mathit{size}(\mathtt{t}). \end{array}$$