Mathematical Foundations of Programming (G54FOP)

especially Functional Programming
we experiment with Haskell

Lecture 1: Introduction

How does Math help Programming?

- Understanding the meaning of programs
- Specification: stating clearly and precisely what a program must do
- Reasoning: use symbolic logic to prove properties of programs
- Abstract Interpretation: see data structures as mathematical objects to clarify their nature

Two fundamental aspects:

Syntax and Semantics

Syntax: how we write expressions, formulas, programs

- Define precisely the language
  what symbols do we use?
  what are the reserved words and identifiers?

how do we combine symbols to form correct expressions and instructions ?

how do we combine instructions to create programs ?

Semantics: what is the meaning of expressions, formulas, programs

- **Denotational Semantics**
  the meaning of expressions etc. is mathematical objects
- **Operational Semantics**
  the meaning of expressions, programs is the computations they do when we execute them

Example:

A language of arithmetical expressions

A very simple toy language with Booleans and Natural Numbers and simple operations on them

There are two ways to give the syntax

- **Backus-Naur form (BNF)**
  simple and compact
- **Derivation Rules**
  more complex but also more flexible and general

Backus-Naur Form for Arithmetic Expressions:

e ::= true | false | zero | succ e
  | pred e | iszero e | if e then e else e

Explanation:

e stands for any expression

an expression can be constructed by any of the forms on the right-hand side

recursive forms:

replace e by a previously constructed expression

Examples of expressions:

- zero
- true
- succ zero
- succ true — This is a correct expr. even if the meaning is unclear
- iszero (succ zero)
- if (iszero (succ false))
  then zero
  else (pred true)

Definition with Derivation Rules of the set Expr of Arithmetic Expressions

(only some of the rules)

$$\frac{e_1 \in \mathsf{Expr} \quad e_2 \in \mathsf{Expr} \quad e_3 \in \mathsf{Expr}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \in \mathsf{Expr}}$$

Assumptions: if we have already constructed expressions e_1, e_2, e_3

$\Rightarrow$

Conclusion: Then we can construct this new expression

$$\frac{e \in \mathsf{Expr}}{\mathsf{succ}\ e \in \mathsf{Expr}}$$

$$\frac{}{\mathsf{false} \in \mathsf{Expr}}$$

← no assumptions

we can construct this expression without any previous work (base case)

... other rules are similar

Example of a complete derivation of an expression:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{zero} \in \text{Expr}}}{\text{succ zero} \in \text{Expr}}}{\text{iszero (succ zero)} \in \text{Expr}} \qquad \dfrac{\dfrac{}{\text{false} \in \text{Expr}}}{\text{succ false} \in \text{Expr}} \qquad \dfrac{\dfrac{}{\text{zero} \in \text{Expr}}}{\text{pred zero} \in \text{Expr}}}{\text{if (iszero (succ zero)) then (succ false) else (pred zero)} \in \text{Expr}}$$

The derivation has the form of a tree:

nodes = intermediate expressions

leaves = base cases

We use parentheses around sub-expressions to avoid confusion

Defining a language using derivation rules is long and boring

BNF is much simpler and compact

But some complex languages (dependently typed λ-calculus, we'll study it) can't be defined using BNF, derivation rules are essential

Uses of derivation rules:

- Definition of a Language
- Rules for manipulation of expressions

- Operational Semantics: how programs run
- Logical Systems: rules of symbolic logic reasoning about programs

General Shape of a Rule:

$$\frac{\overbrace{A_1 \quad A_2 \quad \cdots \quad A_n}^{\text{premises/assumptions}}}{\underbrace{B}_{\text{conclusion}}}$$

these kinds of formulas are called Judgements

If we have already derived $A_1, \dots, A_n$ then we can derive B