Types

Kenneth Lai

Brandeis University

September 12, 2022

Announcements

- Please submit your personal learning goals if you haven't done so already!
- For Wednesday
  - Read van Eijck and Unger Chapter 4.4, 5.2, and 5.3
- For 9/21
  - HW1 due

Today's Plan

- Lambda Calculus Exercises and Resources
- Types
- Functional Programming with Haskell

Today's Plan

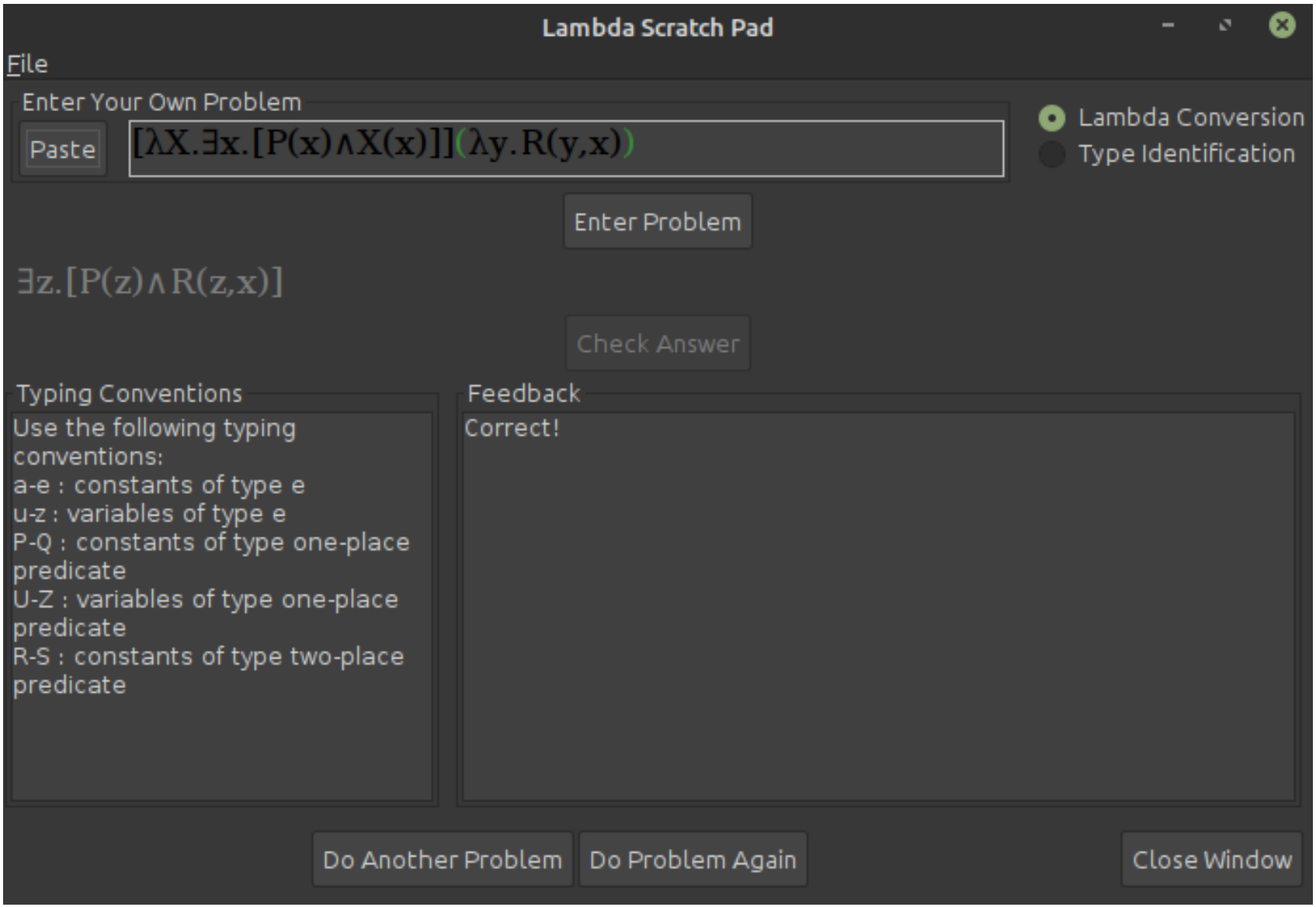
- Lambda Calculus Exercises and Resources
- Types
- Functional Programming with Haskell
- (we'll see how far we get...)

Exercises

- Exercises from Coppock and Champollion (2022) Chapter 5, Exercise 3
- For each of the following lambda expressions, apply beta reduction to give a completely reduced expression (i.e., in beta normal form):
  7. $[[\lambda x \lambda y . R(x, y)](b)](a)$
  8. $[\lambda x . [\lambda y . R(x, y)](b)](a)$
  9. $[\lambda X . \exists x . [P(x) \wedge X(x)]](\lambda y . R(a, y))$
  11. $[\lambda X . \exists x . [P(x) \wedge X(x)]](\lambda y . R(y, x))$

Lambda Calculator

- `http://lambdacalculator.com/`



Computational Semantics
Day 3: Lambda calculus and the composition of meanings

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Observation

We can build expressions that do not make much sense.

- $(+\ x\ \lambda y.(1\ 2))$

Typed lambda calculus

Types

Types are sets of expressions, classifying expressions according to their combinatorial behavior.

Types

$$\tau ::= e \mid t \mid (\tau \to \tau)$$

Where e (for entities) and t (for truth values) are basic types and $\tau \to \tau$ are functional types.

Typed lambda calculus

Each lambda expression is assigned a type, specified as follows:

- **Variables**:
  For each type τ we have variables for that type.
- **Abstraction**:
  If $v :: \delta$ and $E :: \tau$, then $\lambda v.E :: \delta \to \tau$.
- **Application**:
  If $E_1 :: \delta \to \tau$ and $E_2 :: \delta$, then $(E_1\ E_2) :: \tau$.

Examples

Of which types are the following expressions? (Assuming that numbers are of type $\texttt{Int}$, $+$ and * are of type $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$.)

- $\lambda x.(+\ 1\ x)$
- $(\lambda x.(x\ 2)\ \ \lambda y.(*\ y\ y))$
- $(\lambda x.(y\ x)\ \ z)$
- $\lambda z.(z\ z)$

Computational Semantics
Day 1: Getting Started with Haskell + Inference Engine for NL

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

A short history of Haskell

A short history of Haskell

In the 80s, efforts of researchers working on functional programming were scattered across many languages (Lisp, SASL, Miranda, ML,. . .).

In 1987 a dozen functional programmers decided to meet in order to reduce unnecessary diversity in functional programming languages by **designing a common language** that is

- based on ideas that enjoy a wide consensus
- suitable for further language research as well as applications, including building large systems
- freely available

A short history of Haskell

In 1990, they published the first **Haskell** specification, named after the logician and mathematician Haskell B. Curry (1900-1982).

Haskell is functional

A program consists entirely of functions.

- The main program itself is a function with the program's input as argument and the program's output as result.
- Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

A shift in thinking

Imperative thinking:

- Variables are pointers to storage locations whose value can be updated all the time.
- You give a sequence of commands telling the computer what to do step by step.

Examples:

- initialize a variable `examplelist` of type integer list, then add 1, then add 2, then add 3
- in order to compute the factorial of n, initialize an integer variable `f` as 1, then for all `i` from 1 to n, set `f` to `f`$\times$`i`

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

```
factorial :: Int -> Int
factorial n = product [1..n]
```

A shift in thinking

Stop thinking in variable assignments, sequences and loops.

Start thinking in functions, immutable values and recursion.