CSC2125H
Types and Programming Languages
From Lambda Calculus to Programming Languages

Ningning Xie
Assistant Professor
Department of Computer Science
University of Toronto

Lambda calculus

- The λ-calculus is exceedingly elegant and minimal, a study of functions in the purest possible form.

- We find versions of it in most, if not all modern programming languages because the abstractions provided by functions are a central structuring mechanism for software.

Lambda calculus

On the other hand, there are some problem with the data-as-functions representation technique of which we have seen Booleans, natural numbers, and trees. Here are a few notes:

1. Generality of typing
2. Expressiveness
3. Observability of functions

(1) Generality of typing

- The untyped λ-calculus can express fixed points but the same is not true for Church's simply-typed λ-calculus.

- Types, however, are needed to understand and classify data representations and the functions defined over them.

- Fortunately, this can be fixed by introducing recursive types, so this is not a deeper obstacle to representing data as functions.

(2) Expressiveness

- While all computable functions on the natural numbers can be represented in the sense of correctly modeling their input/output behavior, some natural algorithms are difficult or impossible to express.
- For example, our predecessor function took O(n) steps.
- Other representations are possible, but they either complicate typing or inflate the size of the representations.

(3) Observability of functions

- Since reduction results in normal forms, to interpret the outcome of a computation we need to be able to inspect the structure of functions.

- But generally we like to compile functions and think of them only as something opaque: we can probe it by applying it to arguments, but its structure should be hidden from us.

- This is a serious and major concern about the pure λ-calculus where all data are expressed as functions.

In this part

- Rather than representing all data as functions, we add data to the language directly, with new types and new primitives.

- At the same time we make the structure of functions unobservable so that implementation can compile them to machine code, optimize them, and manipulate them in other ways.

- Functions become more extensional in nature, characterized via their input/output behavior rather than distinguishing functions that have different internal structure.

Part I Revisit functions

Make functions unobservable

- We have to change our notion of reduction. We call the result of computation *values* and define them with the judgment.

$$e \ \mathit{value}$$

- Also, we write

$$e \mapsto e'$$

for a single step of computation. For now, we want this step relation to be deterministic.

Values and computations

$$\frac{}{\lambda x.\, e \ \mathit{value}} \ \textsf{val/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \ \textsf{step/app}_1 \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1} \ \textsf{beta}$$

Values

$$\frac{}{\lambda x.\, e \; value} \; \mathsf{val/lam}$$

Redcution

- Call by name

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1 \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{beta}$$

Redcution

- Call by value

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \textsf{step/app}_1 \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \textsf{step/app}_2$$

$$\frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \textsf{step/app/lam}$$

Type safety

- Devising a set of rules is usually the key activity in programming language design.

- Proving the required theorems is just a way of checking one's work rather than a primary activity

Preservation. If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Part II Booleans as a primitive type

Types and expressions

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\,\tau \mid \mathsf{bool} \\
\text{Expressions} & e & ::= & x \mid \lambda x.\,e \mid e_1\,e_2 \mid \Lambda\alpha.\,e \mid e\,[\tau] \\
 & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\; e_1\;e_2\;e_3
\end{array}
$$

Typing rules

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \mathsf{tp/true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \mathsf{tp/false}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}\ \mathsf{tp/if}$$

Values

$$\frac{}{\mathsf{true}\ \mathit{value}} \qquad \frac{}{\mathsf{false}\ \mathit{value}}$$

Reduction

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{step/if}$$

$$\frac{}{\mathsf{if\ true}\ e_2\ e_3 \mapsto e_2}\ \mathsf{step/if/true} \qquad \frac{}{\mathsf{if\ false}\ e_2\ e_3 \mapsto e_3}\ \mathsf{step/if/false}$$