A brief theoretical introduction

Danya Rogozin

Lomonosov Moscow State University

February, 2019

Lecture plan

1. Type theory and λ-calculus: a bit of history.
2. λ-calculus: formal definition, related concepts and basic results.
3. Typed λ-calculus: a bit of history again and basic definitions.
4. Basic functional programming concepts and its connection with type theory and λ-calculus.

A bit of history

- At the beginning of the 20th century, there has been much controversy on the foundation of mathematics after the discovering of paradoxes Cantor's set theory, such as Russel's paradox;
- Mainstream programs: logicism (Frege, Russell, and Whitehead), formalism (Hilbert and Goettingen School of Mathematics) and intuitionism/constructivism (Brouwer and Heyting/ Markov, Kolmogorov).
- Initially, type theory was proposed by Bertrand Russell as the alternative to Cantor's set theory. In that sense, type theory is one of the oldest branches in mathematical logic and foundations of mathematics:

Bertrand Russell, 1903, "The Principles of Mathematics", Cambridge: Cambridge University Press.

Bertrand Russell, 1908, "Mathematical logic as based on the theory of types," American Journal of Mathematics, 30: 222–262.

Alfred North Whitehead and Bertrand Russell, 1910, 1912, 1913, Principia Mathematica, 3 vols, Cambridge: Cambridge University Press.

A bit of history

- At the end of 1920-s, Alonzo Church provided the alternative approach to the foundations of mathematics. You have already heard about this approach because Church developed well-known λ-calculus for this purpose;
- λ-calculus is a foundation of functional programming like von Neumann princilples for imperative programming.

Alonzo Church in person

A bit of history

Moreover, Church used λ-calculus to show that Peano arithmetic is undecidable. In other words, there is no algorithm that defines whether an arithmetical formula ϕ is provable.

See:

1. Martin Davis. 2004. The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Dover Publications, Inc., New York, NY, USA.
2. Alonzo Church. An unsolvable problem of elementary number theory. American journal of mathematics, vol. 58 (1936), pp. 345–363.

A bit of history: type theory meet λ-calculus

- The initial version of λ-calculus was inconsistent, that was shown by Kleene and Rosser.
- Idea of typing in λ-calculus was orinigally as the instrument that allows one to avoid paradoxes.
- The first system of typed λ-calculus is a hybrid from λ-calculus and type theory developed by Russell and Whitehead in Principia Mathematica.

See:

- Alonzo Church. A Formulation of the Simple Theory of Types. J. Symbolic Logic 5 (1940), no. 2, 56–68.

A bit of history again

- After Church's works, type theory as the branch of λ-calculus and combinatory logic was developed by Haskell Curry and William Howard within a context of intuitionistic proof theory (1950–1960-s);
- Polymorphic λ-calculus (John Reynolds and Jean-Yves Girard (1970-s));
- Polymorphic type inference (Roger Hindley, and Luis Damas (1970–1980-s));
- ML: the first language with polymorphic inferred type system (Robin Milner, 1973);
- Haskell appeared at the beginning of 1990-s. Haskell desinged by Simon Peyton Jones, Philip Wadler, and others.

General conceptual aspects: the notion of function

Let's focus on the notion of function in itself. We consider the ordinary set-theoretical notion of function:

Definition

Set-theoretical definition of a function.

Let A, B be sets. A functional relation is a relation $f \subseteq A \times B$, such that, if $\langle x, y_1 \rangle \in f$ and $\langle x, y_1 \rangle \in f$, then $y_1 = y_2$.

Notation: $f(x) = y$ denotes $\langle x, y \rangle \in f$

Formally, a function is an ordered triple $\langle A, B, f \rangle$. Brief notation: $f : A \to B$.

General conceptual aspects

- In such set-theoretical approach, we identify a function and its graph (it's a quite old idea proposed by R. Dedekind at the end of the 19th century);
- In lambda-calculus, a notion of function is introduced as primitive. A function (informally) is a method of object transformation, which is not exactly the same as some subset of the cartesian product.
- As λ-calculus is a Turing complete model of computation: any Turing machine is λ-definable and vice-versa. It's not so painful to prove this equivalence between λ-calculus and recursive functions via Church encoding.

General conceptual aspects

We may provide the following functional programming principles:

1. We have no assignment as in imperative languages, we introduce variables as nullary constant functions (so far as any "value" might be considered as a function).
2. More generally, we have no states.
3. We use recursion instead of loops.
4. For instance, we apply fix-point combinators instead of iterations in λ-calculus, so far as fix-point combinator is the same function as each other.
5. A program evaluation is a simplification of some expression via reduction rules.

Reminder: formal definition

Definition

Let $Var = \{x, y, z, \dots\}$ *be the countably infinite set of atomic variables. The set of preterms is generated by the following grammar:*

$\Lambda_{pre} ::= Var \mid (\lambda V.\Lambda_{pre}) \mid (\Lambda_{pre}\Lambda_{pre})$

Bounded and free variables are defined similarly to first-order logic.

Definition

- α*-conversion is an operation on* Λ_{pre} *that renames bounded variables;*
- $\equiv_\alpha$ *is a reflexive-symmetric-transitive closure of* α*-conversion.*

Definition

The set of λ*-terms* (Λ)*:*

$\Lambda = \Lambda_{pre}/\equiv_\alpha = \{[M]_N \mid M \equiv_\alpha N, \text{where } M, N \in \Lambda_{pre}\}$

It is easy to see that λ-term is a preterm modulo $\equiv_\alpha$, so far as the set Λ is defined as the quotient $\Lambda_{pre}/\equiv_\alpha$.

Reminder: substitution

A substitution is defined as follows:

1. $x[z := M] = M$, if $x = z$
2. $x[z := M] = x$, if $\neg(x = z)$
3. $(\lambda x.N)[z := M] = \lambda x.(N[z := M])$, if x is free in M
4. $(NP)[z := M] = (N[z := M])(P[z := M])$

Reminder: β-reduction and normal form

Definition

β-reduction is the following rewriting rule on terms: $(\lambda x.M)N \to_\beta M[x := N]$

where $(\lambda x.M)N$ is so-called *β-redex*.

Multistep reduction ($\twoheadrightarrow_\beta$) is a reflexive-transitive closure of $\to_\beta$. In other words, $\twoheadrightarrow_\beta$ is preorder on Λ.

β-equality ($=_\beta$) is a symmetric closure of $\twoheadrightarrow_\beta$, i.e. $=_\beta$ is an equivalence relation on Λ.

Definition

A term M is said to be in normal norm, if M contains no β-redexes.

Examples

1. Any variable is in normal form;
2. $\lambda fg.f(gx)$;
3. $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

Confluence

Theorem

Confluence (Church-Rosser property)
Let $M, N, P \in \Lambda$ *and* $M \twoheadrightarrow_\beta N \wedge M \twoheadrightarrow P$. *Then there exists* $Q \in \Lambda$, *s.t.* $N \twoheadrightarrow_\beta Q \wedge P \twoheadrightarrow_\beta Q$

Corollary

Let M *be term, then, if there exists a normal form, then it's unique.*

From a practical point of view, confluece is a property of so-called pure functions, which we will be discussing a bit later.

Normal form

Definition

A term M is weakly normalizable (WN), if there exists some halting reduction path that starts from M.

Definition

A term M is strongly normalizable (SN), if any reduction path that starts from M halts.

Note that, SN implies WN, not vice versa. In other words, there exists a term, that has an infinite reduction path, but might be reduced to some normal form via some another reduction path.

Example

Let us consider the following term:
$(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx))$

On the one hand:

$$(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx)) \to_\beta$$
$$(\lambda y.x[x := \lambda x.x])((\lambda x.xx)(\lambda x.xx)) \to_\beta$$
$$(\lambda y.(\lambda x.x))((\lambda x.xx)(\lambda x.xx)) \to_\beta$$
$$(\lambda x.x)[y := (\lambda x.xx)(\lambda x.xx)] \to_\beta \lambda x.x$$

On the other hand:

$$(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx)) \to_\beta$$
$$(\lambda xy.x)(\lambda x.x)(xx[x := \lambda x.xx]) \to_\beta$$
$$(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx)) \to_\beta$$
$$(\lambda xy.x)(\lambda x.x)(xx[x := \lambda x.xx]) \to_\beta$$
$$(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx)) \to_\beta \ldots \text{pamagiti } ((((((9((9$$

Hence, this term is WN, but not SN.

Applicative and normal strategies: motivation

Let us consider the example from the previous slide in some other aspect.

- In the first case, we have got a sensible result by four reduction steps. On the other hand, we have a loop in the second case.
- Also, in the first case, we started our reduction from the leftmost innermost, but when we tried to start our reduction from the right redex $(\lambda x.xx)(\lambda x.xx)$, we have found ourselves in a spot of trouble, because something went wrong.
- Can we distinguish all possible ways of term reduction?

In a matter of fact, we need to distinguish all possible ways of application reduction, so for as we have no other options in the remaining cases:

1. If $M \equiv x$ is a variable, then reduction is finished;
2. If $M \equiv (\lambda x.N)$, then we reduce N.

Applicative and normal order: motivation

We have two chairs:

1. $(\dots((MN_1)N_2)\dots)N_m$: we firslty reduce N_i for all $i \in \{1, \dots, n\}$ (from left to right or vice versa, it doesn't really matter);
2. $(\dots(((\lambda x.M)N_1)N_2)\dots)N_m$: we $(\lambda x.M)N_1$ and move from left to right.

Definition

1. *An order is called applicative, if we prefer the first way of application reduction.*
2. *An order is called normal, if we prefer the second one way.*

Moreover, there is the following theorem:

Theorem

If M has a normal form, then M reduces to its normal form via normal order of reduction.

That is, normal order allows one to reduce any "reducible" term to its normal form.

Normal reduction

Informally, applicative order corresponds to strict evaluation (the most part of mainstream programming languages), normal order corresponds to lazy evaluation (Haskell).
Generally, we may the following benefits and shortcomings:

- Normal benefit:
  - Most of the time, we don't produce any redundant reductions, E.g., $((\lambda xy.x)N)M \twoheadrightarrow_\beta N$, but we haven't reducted M at once.
  - if M has normal form, it doesn't mean that M may be reduced to its normal form via applicative order. See example above.
- Normal shortcoming:
  - Possible troubles with efficiency: e.g., $(((\lambda fgx.f(gx))N)N)P \twoheadrightarrow_\beta N(NP)$, but we have to reduce N two times, but N is reduced once via applicative order.

In a matter of fact, normal and applicative orders are equivalent in a case strong normalizability, just because any reduction path always halts by the definition.

Normal order and Haskell

- There is no need to perform full β-reduction each time.
- Evaluation in Haskell is the closest to reduction via normal order up to weak head normal form (or, call-by-name up to WHNF).
- In basic untyped λ-calculus, a term M is in weak head normal form, if M is either λ-abstraction or $M = y\,\hat{N}$, where $\hat{N}$ is a finite sequence of arbitrary λ-terms.

Examples of WHNFs in λ-calculus:

- $(\lambda xy.M)$ is in WHNF (M is a metavariable on terms);
- $(\lambda x.f(gx))(\lambda x.h(hx))$ is not in WHNF;
- Any variable is in WHNF;
- Any closed term is in WHNF.

In Haskell, a term M is WHNF, if M is either λ-abstraction or constructor, as you'll see furthermore.

Currying and partial application

- In mathematics, a binary function is some map $f : A \times B \to C$.
- On the other hand, we may transform this binary function to some unary function $f^* : A \to C^B$, where $C^B = \{g \mid g : B \to C\}$, such that $f(x, y) = f^*(x)(y)$.
- This property is also known as the universal exponentiation property in category theory: $Hom(A \times B, C) \cong Hom(A, C^B)$, that is, we have a bijection between these sets for all A, B, C.
- In the first case, a function f should be applied to some ordered pair $\langle x, y \rangle \in A \times B$. In the second one case, we apply arguments sequentially. That is, $f^*(x)$ is a function $B \to C$, hence $f^*(x)(y) \in C$.
- In other words, sequential applying allows one to perform a partial application.
- This move is called *currying*.
- Initially, currying was proposed by Frege and Schoenfinkel, but named after Haskell Curry.
- In untyped λ-calculus, all functions are curried by default that follows from the definition of abstraction.

Pure functions and so-called side-effects

- Pure function is a function that returns the same value for the same argument.
- Remember the Church-Rosser property. A pure function is a function for which confluence holds. In other words, an uniqueness of the normal form is guaranteed for each output.
- This principle is also knows as referential transparency.
- Thus, side-effect function is a function with no confluece property, i.e. outputs may differ for the same argument. From a mathematical point of view, a side-effect function is not function at all (remember the set-theoretic definition).
- The principal reason of confluece failure for side-effect function: dependency from external state and outside world.
- Unfortunately, we have no reason to claim that Haskell is confluent.

Types: general words

- Type is a syntax construction that should be assigned to terms according to the special list of rules;
- Types defines a kind of partial specification;
- Type checking allows one to catch a huge class of errors;
- Standard definition by Benjamin Peirce (from TAPL):

  > *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

Types: motivation

Quite old and silly joke from calculus:
$(x^2)' = 2x$, let $x := 3$. On the one hand, $(3^2)' = 9' = 0$, on the other hand, $2 \cdot 3 = 6$.

- Why $0 \neq 6$?
- Well, we use x^2 in two ways: the first one, x^2 is a number, the second one: we have a deal with function $f : x \mapsto x^2$. Thus, $f'(x) = (x^2)' = 2 \cdot x$.
- In the first case, $x \in \mathbb{R}$, in the second one, $f : \mathbb{R} \to \mathbb{R}$.
- So, we need to distinguish objects considered above with respect to their types.

Type theory

Type theory is often considered in the following two aspects:

- The branch of proof theory. For example, a term assignment allows one to simplify the proof of cut elimination.
- The branch of computer science.

As you have already heard before a lot of times, these branches are closely connected and, moreover, equivalent (in a certain degree). Logically, lambda-term denotes a program that proves some intuitionistic formula (inhabits some type).

Kinds of typing in real world languages

We may classify possible ways of typing as follows:

- Strong or weak typing:
  - Strong typing: Java, Haskell, Ocaml, F ♯, Rust.
  - Weak typing: JavaScript, PHP, C, C++, etc.
- Static or dynamic typing:
  - Static typing: C, C++, Java, Haskell, etc.
  - Dynamic typing: JavaScript, Ruby, PHP, etc.
- Implicit or explicit typing:
  - Implicit typing: JavaScript, Ruby, PHP, etc.
  - Explicit typing: C++, Java, etc.
- Inferred typing:
  - Haskell, Standard ML, Ocaml, Idris, etc.

In such classification, Haskell is a strong, static and inferred, similarly to Ocaml or F ♯.

Reminder on simple type theory: the simplest type system

$$\frac{}{\Gamma, x : A \to x : A}\ \mathrm{Ax}$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x.u : A \to B}\ \to_I$$

$$\frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B}\ \to_E$$

We will consider the typing style in a manner of Curry by default. Curry-style typing is closer to Haskell or Ocaml: we introduce type annotations only on the whole expression, but we don't annotate its bounded variables.

Higher-order functions

- Higher-order functions are widely used in ordinary mathematics, such as differential operator, that has type (informally) $\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}}$.
- In pure λ-calculus, all functions are higher-order by default.
- In typed λ-calculus, a function of type $A \to B$ is called higher-order, if $A = C \to D$, for some types C and D, for example:
  $\lambda fgx.f(gx) : (B \to C) \to ((A \to B) \to (A \to C))$
- Idea is quite simple: a function might be an arguments like any other value. For instance, we may compose arbitrary unary functions via combinator describe above.

Type-safety and type uniqueness

- Type safety is a provable metatheoretical property of type systems.
- Type-safety is so-called subject reduction. Subject reduction claims
- that a type is an invariant with respect to an evaluation:

Theorem

If $\Gamma \vdash M : A$ *and* $M \twoheadrightarrow_\beta N$, *then* $\Gamma \vdash N : A$

- From a logical (more precisely, proof-theoretical) point of view, a type safety property is exactly proof normalization (or cut elimination, in terms of sequent calculi).
- Subject reduction expresses the well-known principle expressed by Robin Milner that "well typed programs cannot go wrong".

Example

$\pi_2 \langle x, f\ x \rangle \to_\beta f\ x$

$$\dfrac{\dfrac{x : A \vdash x : A \qquad \dfrac{f : A \to B \vdash f : A \to B \qquad x : A \vdash x : B}{f : A \to B, x : A \vdash f\ x : B}}{f : A \to B, x : A \vdash \langle x, f\ x \rangle : A \times B}}{f : A \to B, x : A \vdash \pi_2 \langle x, f\ x \rangle : B}$$

We may transform this derivation as follows:

$$\dfrac{f : A \to B \vdash f : A \to B \qquad x : A \vdash x : B}{f : A \to B, x : A \vdash f\ x : B}$$

Type uniqueness

Theorem

Let $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$*, then* $A \sim B$.

- Logically, type uniquness denotes that single prove proves the same formula, in other words, if M is a proof of A and M is a proof of B, then A and B are the same modulo equivalence on formulas.
- From a computational point of view, this property denotes that all type assigments for a given term in the same environment are equivalent.
- What $\sim$ is in type systems?
- That depends on type systems, in richer type systems, where type reduction is enabled, $\sim$ is $\beta\eta$-equivalence.

λ-cube and Haskell

As you know, we have the following ways of dependency between terms and types:

- A term depends on type (polymorphism in system F);
- A type depends on type (so-called type operators in λ_ω);
- A type depends on terms (dependent types in basic DT system called P).

All possible combinations of these dependencies may be illustrated via Barendregt's λ-cube.

- Simply typed λ-calculus is the weakest system in λ-cube;
- The strongest system is C, so-called calculus of constructions, (foundation of Agda and Coq proof assistants).

To read more about λ-cube, see:

1. Rob Nederpelt and Herman Geuvers. Type Theory and Formal Proof – An Introduction. Cambridge University Press, 2014.
2. Morten Heine Soerensen and Pawel Urzyczyin. Lectures on the Curry-Howard Isomorphism. Elsivier, Studies in Logic and the Foundations of Mathematics, 2006.

λ-cube and Haskell

- As you've already heard, we cannot have type inference in polymorphic Curry-style typing;
- Type inference in Haskell based on Hindley-Milner inference proposed for special system F restriction (so-called Hindley-Milner type systems);
- Roughly speaking, "basic" Haskell is a little bit weaker than system F;
- As you will see furthermore, we may enable system F typing via `RankNTypes` extension;
- Similarly, λ_ω typing might be enabled using `TypeOperators` extension;
- Roughly speaking again, system F_ω (polymorphism + type operators) is already implemented in GHC-core.

Actual Haskell type system

So-called system FC is a type system that might be considered as Haskell Core. This system is quite sophisticated to be defined formally on slides. We just speak about basic features:

- $F_\omega \subset FC$;
- Coercions and equality constraints;
- Algebraic datatypes;
- Coercions and ATD allows one to encode GADT;
- etc.

To read about system *FC* in more detail, see:

1. Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones. System F with Type Equality Coercions, 2007.
2. Richard A. Eisenberg. System FC, as implemented in GHC, 2013.
3. Stephanie Weirich, Justin Hsu, Richard A. Eisenberg. System FC with Explicit Kind Equality, 2015.

Actual Haskell type system

- It seems that *FC* as implemented in GHC Code will be extended soon to so-called system *D* to develop dependent Haskell, that was planned to finish in 2020.
- Thus, *FC* will be Haskell Core for a couple of years, no longer.

See:

- Vladislav Zavialov. Why Dependent Haskell is the Future of Software Development, 2018.
- Richard A. Eisenberg. Dependent Types in Haskell: Theory and Practice (PhD Thesis), 2016.

Thank you!

Any questions?