Type Theory

Zhaohui Luo
Department of Computer Science
Royal Holloway, University of London

Lecture slides & distribution files:

http://www.cs.rhul.ac.uk/home/zhaohui/TTlectures.html

Type theory:

a foundational and practical language for the working computer scientist.

- Logic + Computation
  - Two fundamental features in single language
- Rich structural mechanisms
  - Abstraction and modularisation
- Nice properties
  - Basis for simple semantics and implementations

Type theories as …

- Logical systems
  - propositional, first-order, higher-order, …
  - intuitionaistic, classical, …
- Programming languages
  - Functional programming (via λ-functions and computation)
  - Modular programming (via rich type structures)
- Mathematical modelling calculus
  - Formalisation of mathematics
  - Natural language semantics

Some applications of type theory

- Proof assistants for interactive theorem proving
  - ALF/Agda (Sweden), Coq (France), Lego/Plastic (UK), Matita (Italy), NuPRL (USA), …
  - Formalisation of mathematics (eg, four-colour theorem)
  - Verification (eg, of security protocols)
- Dependently-typed programming
  - Agda/Cayenne (Sweden), DML (USA), Epigram (UK), …
  - Dependent, richer types in programming languages
- Modelling in type theories
  - Eg, linguistic reasoning with type-theoretical semantics (Leverhulme research project at Royal Holloway)

Historical remarks

- Early development (from early 1900's)
  - Logical paradoxes (eg, $S \in S$ if $S = \{ x \mid x \notin x \}$?)
  - Ramified type theory (Russell)
  - Simple type theory (Ramsay 1926 & Church 1940)
- Modern development (since 1970's)
  - Martin-Löf's predicative type theory (Martin-Löf 1973, 1984)
  - Impredicative type theories (with type Prop of all propositions)
    - Polymorphic λ-calculi (F & F^ω, Girard 1972, Reynolds 1974)
    - Calculus of Constructions (CC, Coquand & Huet 1988)
    - Unifying Theory of dependent Types (ECC/UTT, Luo 1989/1994)
    - Calculus of Inductive Constructions (CIC, implemented in Coq)

This lecture series

- Basics of type theory
  - Introduction
  - Embedded logics in type theories
  - Inductive data types and universes
- Subtyping in type theory
  - Coercive subtyping – theory and implementation
  - Applications (eg, in proof development and linguistic semantics)

We shall start from the simplest TT – the simply typed λ-calculus.

I. Typed λ-calculi

- Type-free v.s. typed λ-calculi
  - Type-free λ-calculus (Barendregt 1980)
    - Type-free terms: x, λx.M, MN
    - eg, $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ and $\Omega \triangleright_\beta \Omega$.
  - Typed λ-calculi
    - Only well-typed terms are "legal" (eg, λx:A.x : A→A).
    - eg, Self-applications such as (λx:A.xx)(λx:A.xx) are not well-typed.
- Typed λ-calculus – basis of type theory
  - Example calculi: function types, dependent types, … …

Simply typed λ-calculus: syntax

- Types ::= σ | A→B
  - Intuitively, an object of A→B is a function from A to B (eg, λ-functions).
- Terms ::= x | λx:A.b | f(a)
- Judgement

$$\Gamma \vdash a : A$$

means "*a is an object of type A under assumptions Γ*", where Γ, called a *context*, is a finite set of entries of the form x:T.
  - Eg,

$$\emptyset \vdash \lambda x{:}\mathrm{Nat}.\,x : \mathrm{Nat}\to\mathrm{Nat}$$
$$x{:}\mathrm{Nat} \vdash x+2 : \mathrm{Nat}$$
$$f : \mathrm{Nat}\to\mathrm{Nat} \vdash f(0) : \mathrm{Nat}$$
$$(?)\quad f : \mathrm{Nat}\to\mathrm{Nat} \vdash f(f) : \mathrm{Nat}$$

Here, the first three are correct, not the fourth – governed by the rules.

Simply typed λ-calculus: inference rules

- Inference rules

$$(Var)\quad \frac{}{\Gamma,\ x : A \vdash x : A}$$

$$(Abs)\quad \frac{\Gamma,\ x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : A \to B}$$

$$(App)\quad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

- Correctness is given by *derivability.*
  - *A judgement J is derivable if there is a sequence of judgements $J_1, \ldots, J_n$ with $J_n \equiv J$ such that, for $1 \le i \le n$, J_i is the conclusion of some instance of a rule whose premises are all in $\{J_j \mid j < i\}$.*
  - *Eg,* $f : \mathrm{Nat}\to\mathrm{Nat} \vdash f(0) : \mathrm{Nat}$ is derivable;
    $f : \mathrm{Nat}\to\mathrm{Nat} \vdash f(f) : \mathrm{Nat}$ is not derivable.

Computation: β-reduction

- Compatible relation ▷:
  - If M ▷ N, then λx:A.M ▷ λx:A.N, M(a) ▷ N(a), and f(M) ▷ f(N).
- β-reduction $\triangleright_\beta$ is the reflexive and transitive closure of the least compatible relation satisfying (β):

$$(\beta)\qquad (\lambda x{:}A.b)(a) \triangleright_\beta [a/x]b$$

  - Eg, $(\lambda x{:}A.\ x+2)(3) \triangleright_\beta 3+2$
- β-conversion $=_\beta$ is the corresponding equivalence.

Meta-theoretic properties

- Properties of typing, computation and their relationship.
- Remarks:
  - Properties held for all "well-behaving" calculi, but only illustrated here for the simply typed λ-calculus.
  - These properties are the basis for
    - Simple operational semantics (see "canonical objects" later)
    - Implementations (of, eg, proof assistants)

We now explain some example properties.

Church-Rosser (CR)

- If $M =_\beta N$, then ∃ P. $M \rhd_\beta P$ and $N \rhd_\beta P$.
  - Diamond property: an alternative formulation
  - Equivalence between the two formulations
- Uniqueness of values (if they exist)!
- Remark:
  - CR as a property of "raw terms"
  - For some calculi, CR only holds for well-typed terms.
    - Eg, for dependent types with type labels and βη-reduction, λx:A.(λy:B.y)(x) is not CR as a raw term, but CR if well-typed.

Subject reduction (SR)

- If a : A and $a \rhd_\beta b$, then b : A.
- Computation preserves typing!
  - When performing computation, there is no need to check well-typedness.
  - Important for implementation

Strong normalisation (SN)

- If a : A, then a is strongly normalisable.
  - Every reduction sequence starting from any well-typed term is finite.
  - Proof in Appendix 2 of (Hindley & Seldin 1986)
- Every computation terminates!
- Implications
  - Usually implying logical consistency, ie, false is not provable. (cf, in FP languages: no consistent logic)
  - Decidability and others

Decidability

- Decidability of
  - Type checking: $\Gamma \vdash a : A$?
  - Type inference: $\Gamma \vdash a :$?
- Basis for implementations (of, eg, proof assistants)
- Remarks
  - Compare this with the undecidability of a∈A in set theory.
  - For dependent TTs, the above two problems are equivalent – type checking requires type inference.

More sophisticated types: higher-order

- Higher-order types
  - System F – 2nd-order polymorphic λ-calculus (Girard 1972, Reynolds 1974)
    - 2nd-order type ∀X.X (or ∀X:Type.X/∀X:Prop.X), where X ranges over *all types/propositions*)
    - Logical constant "false"
  - System F^ω (Girard 1972)
    - Higher-order types: quantifications over connectives as well as propositions (eg, ∀C:Prop→Prop→Prop. …)

Remark

- For (non-dependent) higher-order types, we still have
  - Separation (syntactically) between terms and types (terms cannot occur in types/propositions).
  - Eg, we cannot have ∀P:Nat→Prop.P(m)⊃P(n) (ie, m and n are Leibniz equal.) To do this, we need *dependent* types.

More sophisticated types: dependency

- Families of types – types dependent on terms
  - Per Martin-Löf 1970s-1980s (1973, 1984)
- Examples
  - Vect(n) – type of lists of exactly n elements, a type depending on n : Nat
  - m≤n – proposition that depends on m, n : Nat.

Π-types

- Informally,

  Πx:A.B(x) = { f | for any a : A, f(a) : B(a) }

  (formal rules later)
- Examples
  - λx:Nat.[1,…,x] : Πx:Nat.Vect(x)
  - ∀x:Nat.0≤x
  - Combining dependency & higher-order, we can have: ∀P:Nat→Prop.P(m)⊃P(n).

Σ-types

- Informally,

  Σx:A.B(x) = { (a,b) | a : A & b : B(a) }
- Examples
  - Types of modules such as

    〈 S : type, f : S→S→S 〉 : ΣS:type. S→S→S,

    where "type" is a type of types ("universe" – see later)

Curry-style typed λ-calculus

- So far: Church-style
- Curry-style: an equivalent presentation for simply typed λ-calculus
  - Terms ::= x | λx.b | f(a)
  - (Var)(App) are the same, except $(Abs')\quad \dfrac{\Gamma,\ x:A \vdash b:B}{\Gamma \vdash \lambda x.b : A \to B}$
- Type assignment
  - Eg, λx.x can be assigned α→α for any type α

    (cf, in Church-style, λx:A.x must be of type A→A.)
  - Terms can be assigned many types or "overloaded".
  - Adopted in various programming languages such as ML/Haskell/…
- These are two very different styles.
  - Only equivalent for simpler calculi, not for others …

Church-style v.s. Curry-style

- In appearance, only a syntactic difference of type labels: between λx:A.b and λx.b.
- In fact, much deeper – two different views of types!
- Curry-style – type assignment systems
  - Objects exist first – types are then assigned to objects.
  - Overloading λ-terms, which may reside in different types.
- Church-style – type theories with canonical objects
  - Types and their objects co-exist.
  - This is the kind of TTs we are about to study … …

Revision Questions

- What are
  - Simple types?
  - Higher-order types?
  - Dependent types?
- What are the meta-theoretic properties such CR, SR and SN? What are their theoretical and practical implications?
- What are the differences between
  - type-free λ-calculus and typed λ-calculi?
  - Church-style and Curry-style λ-calculi?

Selected References

- H.P. Barendregt. The Lambda Calculus, Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, Volume 103. 1980.
- H.P. Barendregt. Lambda-calculus with types. In *Handbook of Logic in Computer Science*. OUP, 1992.
- Th. Coquand and G. Huet. The calculus of constructions. Infor and Comp 76(2-3). 1988.
- J.-Y. Girard. PhD thesis,1972.
- J. Hindley and J. Seldin. Introduction to Combinators and λ-Calculus. CUP, 1986. (2nd edition in 2008 with a slightly different title)
- Z. Luo. Computation and Reasoning: A Type Theory for Computer Science. OUP, 1994.
- P. Martin-Löf. Intuitionistic Type Theory. Bibliopolis, 1984.
- J.C. Reynolds. Towards a theory of type structure. LNCS 19, 1974.
- M. Sorensen and P. Urzyczyn. Lectures on the Curry-Howard Isomorphism. Studies in Logic and the Foundations of Mathematics, Vol 149. Elsevier, 2006.