Type Theory & Functional Programming

- **Publisher**: Addison-Wesley
- **Author**: Simon Thompson
- **Presenter**: Wen-Bin Luo
- **Link**: https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/ (https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/)

Contents

Introduction

- **Constructive type theory**: A system which is simultaneously a logic and a programming language, and in which propositions and types are *identical*.
- **Functional programming language**: A program is simply a value of a particular explicit type, rather than a state transformer.
- If the language allows general recursion, then every type contains at least one value, defined by the equation $x = x$.
- **Curry Howard isomorphism**: the propositions-as-types notion.
- $p : P$: p is of type P, or p is a proof of proposition P.
- Functions defined by recursion have their properties proved by induction.
- $(a, b) : (\exists x : A). B(x)$: a of type A meets the specification $B(x)$, as proved by $b : B(a)$.
- The logic is an extension of many-sorted, first-order predicate logic.

- The system here integrates the process of program development and proof: to show that a program meets a specification we provide the program/proof pair.

Introduction to Logic

- Propositional Logic
- Predicate Logic

Propositional Logic

- Propositional **formula** (φ) are made up of propositional **atoms** (P) and **connectives** ($\land | \lor | \implies$).
- Backus-Naur form: $\varphi ::= P | (\neg\varphi) | (\varphi \land \varphi) | (\varphi \lor \varphi) | (\varphi \implies \varphi)$.
- Natural deduction rules: ($\land | \lor | \implies | \neg | \bot$) (introduction|elimination)
- Propositional logic is a subset of the predicate logic.

Predicate Logic

- Predicate **formula** (φ) are made up of **terms**, **predicates** (P), **quantifiers** ($\forall | \exists$), and **connectives** ($\land | \lor | \implies$).
  - Terms (t): **variables** (x), **constants** (c), **functions** (f).
- Backus-Naur form: $\varphi ::= P(t\ldots) | \forall x.\ \varphi | \exists x.\ \varphi | (\neg\varphi) | (\varphi \land \varphi) | (\varphi \lor \varphi) | (\varphi \implies \varphi)$.
- Natural deduction rules: ($\forall | \exists | \land | \lor | \implies | \neg | \bot$) (introduction|elimination)
- In a sense, $\forall$ is a combination of infinite $\land$, while $\exists$ is a combination of infinite $\lor$.

Functional Programming and λ-Calculi

- Functional Programming
- The Untyped λ-Calculus
- Evaluation
- Convertibility
- Expressiveness
- Typed λ-Calculus
- Strong Normalization
- Further Type Constructors: The Product
- Base Types: Natural Numbers

- General Recursion
- Evaluation Revisited

Functional Programming

- FP is characterized by first-class functions, strong type systems, polymorphic types, algebraic types, and modularity.
  - **First-class functions**: Functions may be passed as arguments to and returned as results of other functions.
  - **Algebraic types**: Algebraic types generalizes enumerated types, (variant) records, certain sorts of pointer type definitions, and also permits type definitions to be parametrized over types.

The Untyped λ-Calculus

- **λ-expression** (e) is made up of *variables*, *applications*, and *abstractions*.
  - **Variables** (x)
  - **Applications** ($e_1 e_2$): The application of expression e_1 to e_2.
  - **Abstractions** ($\lambda x.\ e$): The function which returns the value e when given formal parameter x.
- Backus-Naur form: $e ::= x | ee | \lambda x.\ e$.
- An expression is **closed** if it contains no free variables, otherwise it is **open**.
- The **substitution** of x' for the free occurrences of x in e is written $e[x'/x]$.
- **β-reduction**: For all x, e and e', we can reduce the application $(\lambda x.\ e)e' \to_\beta e[e'/x]$.
- **β-redex**: A sub-expression of a lambda expression of the form $(\lambda x.\ e)e'$.

Evaluation

- Normal form variants:
  - **Normal form**: An expression is in normal form if it contains no redexes.
  - **Head normal form**: All expressions of the form $\lambda x_1 \ldots \lambda x_n.\ y e_1 \ldots e_m$ where x and y are variables and e are expressions.
  - **Weak head normal form**: All expressions which are either abstractions or of the form $y e_1 \ldots e_m$.
- A normal form can be thought of as the result of a computation.
- Evaluation of an expression fails to terminate if no sequence of reductions ends in a weak head normal form.

- **Church-Rosser Theorem**: For all expressions e, e_1, and e_2, if $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there exists an expression e' such that $e_1 \twoheadrightarrow e'$ and $e_2 \twoheadrightarrow e'$.
- The method of **structural induction**: To prove the result $P(x)$ for all λ-expressions e, it is sufficient to prove
  - $\forall x.\ P(x)$ holds.
  - If $P(e_1)$ and $P(e_2)$ hold, then $P(e_1 e_2)$ holds.
  - If $P(e)$ holds, then $P(\lambda x.\ e)$ holds.
- **Theorem**: If a term has a normal form, then it is unique.
- If an expression contains more than one redex, then we say that the **leftmost outermost** redex is that found by searching the parse tree top-down, going down the left hand subtree of a non-redex application before the right.
- **Normalization Theorem**: The reduction sequence formed by choosing for reduction at each stage the leftmost-outermost redex will result in a normal form, head normal form or weak head normal form if any exists.
- **Lazy** evaluation mechanism:
  - Corresponds to the strategy of choosing the leftmost-outermost redex at each stage.
  - Avoids duplication of evaluation caused by duplication of redexes.
- The strict or applicative order discipline will not always lead to termination, even when it is possible.
- **η-reduction**: For all x and e, if x is not free in e, then we can perform the reduction $\lambda x.\ (ex) \rightarrow_\eta e$.
- It is not clear that η-reduction is strictly a rule of computation.
- The η-reduction rule identifies certain (terms for) functions which have the same behavior, yet which are represented in different ways.

Convertibility

- **Convertibility** relations: equivalence relations which are also substitutive.
- **Definition**: $e \leftrightarrow f$ if and only if there is a sequence $e_0, \ldots, e_n$ such that $e \equiv e_0, e_n \equiv f$ and for each $i < n$, $e_i \twoheadrightarrow e_{i+1}$ or $e_{i+1} \twoheadrightarrow e_i$.
- $\leftrightarrow$ is the smallest equivalence relation extending $\twoheadrightarrow$.
- As a consequence of the Church-Rosser theorems, two expressions e_1 and e_2 will be (βη-)convertible if and only if there exists a common (βη-)reduct of e_1 and e_2.
- Two functions with normal forms are convertible if and only if they have the *same* normal form.
- The convertibility relations are not necessary to explain the computational behavior of λ-expressions.

Expressiveness

- The untyped λ-calculus is Turing-complete.
- Objects such as the natural numbers, booleans and so forth can be represented as λ-terms.
- To derive recursive functions, we need to be able to solve equations of the form $f := Rf$ where R is a λ-terms.
- **Fixed-point combinators** (F) solve the equation $f := Rf$. Thus, $FR \twoheadrightarrow R(FR)$.

Typed λ-Calculus

- The untyped λ-calculus is characterized by
  - Powerful representatives of all the common base types and their combinations under standard type-forming operations.
  - The presence of non-termination since not every term has even a weak head normal form.
- Given a set B of base types, we form the set S of **simple types** closing under the rule of function type formation - If σ and τ are types, then so is $(\sigma \implies \tau)$.
- The expressions (e) of the typed λ-calculus have three forms:
  - **Variables** (x)
  - **Applications**: If $e_1 : (\sigma \implies \tau)$ and $e_2 : \sigma$, then $(e_1 e_2) : \tau$.
  - **Abstractions**: If $x : \sigma$ and $e : \tau$, then $(\lambda x.\ e) : (\sigma \implies \tau)$.
- **Strong Normalization Theorem**: Every reduction sequence terminates.
  - The system is less expressive than the untyped calculus.
- **Type assumption (declaration)**: When a variable is used, it is associated with a type.
- **Type context** (Γ): Types are assigned to expressions in the type context of a number of type assumption.
- All contexts Γ are *consistent* in containing at most one occurrence of each variable.

Strong Normalization

- **Reducibility** method involves an induction over the complexity of the types, rather than over syntactic complexity.
- **Strong Normalization Theorem**: For all expressions e of the simply typed λ-calculus, all reduction sequences beginning with e are finite.
- The method of **induction over types** states that to prove the result $P(\tau)$ for all types τ it is sufficient to prove
  - *Base case*: For all base types $\sigma \in B$, $P(\sigma)$ holds.
  - *Induction step*: If $P(\sigma)$ and $P(\tau)$ hold, then $P(\sigma \implies \tau)$ holds.
- An expression e of type τ is **stable** (denoted by $e \in \|\tau\|$) if either
  - e is of base type and e is strongly normalizing.
  - e is of type $\sigma \implies \tau$ and for all $e \in \|\sigma\|$, $(ee) \in \|\tau\|$.

- Stability for a function type is defined in terms of stability for its domain and range types.
- **Lemma**:
  - $x \in \mathrm{SN}$.
  - If $e_1, \ldots, e_n \in \mathrm{SN}$, then $xe_1 \ldots e_n \in \mathrm{SN}$.
  - If $ex \in \mathrm{SN}$, then $e \in \mathrm{SN}$.
  - If $e \in \mathrm{SN}$, then $(\lambda x.\ e) \in \mathrm{SN}$.
- **Lemma**:
  - If $e \in \|\tau\|$, then $e \in \mathrm{SN}$.
  - If $xe_1 \ldots e_n : \tau$ and $e_1, \ldots, e_n \in \mathrm{SN}$, then $xe_1 \ldots e_n \in \|\tau\|$.
  - If $x : \tau$, then $x \in \|\tau\|$.
- **s-instance**: A **s-instance** e' of an expression e is a substitution instance $e' \equiv e[e_1/x_1, \ldots, e_n/x_n]$ where $e_1, \ldots, e_n$ are stable expressions.
- **Lemma**:
  - If e_1 and e_2 are stable, then so is $(e_1 e_2)$.
  - For all $n \geq 0$, if $e[e'/x]e_1 \ldots e_n \in \|\tau\|$ and $e' \in \mathrm{SN}$, then $(\lambda x.\ e)e' e_1 \ldots e_n \in \|\tau\|$.
  - All s-instances e' of expressions e are stable.

Further Type Constructors: The Product

- **Product**:
  - $\sigma \times \tau$ is a type if σ and τ are.
  - **Pairs**: If $x : \sigma$ and $y : \tau$, then $(x, y) : \sigma \times \tau$.
- **Projections**: If $p : \sigma \times \tau$, then
  - $\mathrm{first}\ p : \sigma$ where first returns the first element of p.
  - $\mathrm{second}\ p : \tau$ where second returns the second element of p.
- The rules of reduction:
  - *Computation (β-reduction) rules*: $\mathrm{first}(x, y) \to x$ and $\mathrm{second}(x, y) \to y$.
  - *Equivalence (η-reduction) rules*: $(\mathrm{first}\ p, \mathrm{second}\ p) \to p$.
- **Extensionality**: An element of a product type is characterized by its components.
- The operations first and second as primitives:
  - $\mathrm{first} : (\sigma \times \tau) \implies \sigma$.
  - $\mathrm{second} : (\sigma \times \tau) \implies \tau$.

Base Types: Natural Numbers

- **Numbers**:
  - $\mathbb{N}$ is in the set of base types B.
  - $0 : \mathbb{N}$, and if $n : \mathbb{N}$, then $\mathrm{successor}\ n : \mathbb{N}$.

- **Primitive recursion**: For all types τ, if $e_0 : \tau$ and $f : (\mathbb{N} \implies \tau \implies \tau)$, then $Re_0f : \mathbb{N} \implies \tau$ where R is the **primitive recursor**.
- The rules of reduction:
  - $Re_0f0 \to e_0$.
  - $Re_0f(n+1) \to fn(Re_0fn)$.
- R that represents a natural number $n : \mathbb{N}$ is a function that maps any function f to its n-fold composition.

General Recursion

- **General recursion**: A **general recursor** R also has the property that $Rf \to f(Rf)$.
- Rf is a *fixed point* of the functional f.

Evaluation Revisited

- Final results of programs are non-functional.
- **Order** (∂) of a type τ (denoted as $\partial(\tau)$) is defined as:
  - $\partial(\tau) = 0$ if $\tau \in B$.
  - $\partial(\sigma \times \tau) = \max(\partial(\sigma), \partial(\tau))$.
  - $\partial(\sigma \implies \tau) = \max(\partial(\sigma) + 1, \partial(\tau))$.
- The terms we evaluate are not only zeroth-order (**ground types**), they also have the second property of being closed containing as they do no free variables. The results will thus be closed (β-)normal forms of zeroth-order type. It is these that we call the *printable* values.

Constructive Mathematics

- Existence and Logic
- Mathematical Objects
- Formalizing Constructive Mathematics
- Conclusion

Existence and Logic

- Systems of constructive logic do not include the *law of the excluded middle* and *double negation elimination*.
- Sanction for proof by contradiction is given by the *law of the excluded middle*.

- An idealistic view of truth: every statement is seen as true or false, independently of any evidence either way.
- Bishops states that the classical theorem that every bounded non-empty set of reals has a least upper bound not only seems to depend for its proof upon non-constructive reasoning, it implies certain cases of the law of the excluded middle which are *not* constructively valid.
- Not only will a constructive mathematics depend upon a different logic, but also it will not consist of the same results.
- The negation of a formula $\neg A$ can be defined to be an implication $A \implies \bot$.
- A proof of a negated formula has no computational content.
- To give a proof of an existential statement $\exists x.\, P(x)$, we have to give a **witness** a and the proof of $P(a)$.
- A constructive proof of $\exists x.\, P(x) \lor \neg\exists x.\, P(x)$ constitutes a demonstration of the *limited principle of omniscience*.

Mathematical Objects

- The nature of objects in classical mathematics is simple: everything is a set.
- Every object in constructive mathematics is either finite or has a finitary description.
- Constructive mathematics is naturally typed.
- Two algorithms are deemed equal if they give the same results on every input (the *extensional* equality on the function space).
- **Principle of Complete Presentation**: If an object is supposed to have a certain type, then that object should contain sufficient witnessing information so that the assertion can be verified.
- Negative assertions should be replaced by positive assertions whenever possible.

Conclusion

- Objects are given by rules, and the validity of an assertion is guaranteed by a proof from which we can extract relevant computational information, rather than on idealist semantic principles.

Introduction to Type Theory

- Propositional Logic: An Informal View
- Judgements, Proofs and Derivations
- The Rules for Propositional Calculus
- The Curry Howard Isomorphism
- Some Examples

- Quantifiers
- Base Types
- The Natural Numbers
- Well-founded Types: Trees
- Equality
- Convertibility

- Central to type theory is the duality between propositions and types, proofs and elements: a proof of a proposition T can be seen as a member of the type T, and conversely.
- Infinite data types are characterized by principles of definition by recursion and proof by induction.
- A proof by induction is nothing other than a proof object defined using recursion.
- Our system gives an *integrated* treatment of programming and verification.

Propositional Logic: An Informal View

- $A \wedge B$: A proof of $A \wedge B$ will be a pair of proofs p and q, $p : A$ and $q : B$.
- $A \vee B$: A proof of $A \vee B$ will either be a proof of A or be a proof of B, together with an indication of which formula the proof is of.
- $A \implies B$: A proof of $A \implies B$ consists of a method or function which transforms any proof of A into a proof of B.

The Rules for Propositional Calculus

- Each connective has its formation, introduction, elimination, and computation rule.
- Rules for $\wedge$:
  - Formation: If A is a formula and B is a formula, then $(A \wedge B)$ is a formula.
  - Introduction: If $p : A$ and $q : B$, then $(p, q) : (A \wedge B)$.
  - Elimination:
    - If $r : (A \wedge B)$, then $\text{first } r : A$.
    - If $r : (A \wedge B)$, then $\text{second } r : B$.
  - Computation:
    - $\text{first}(p, q) \to p$.
    - $\text{second}(p, q) \to q$.
- Rules for $\vee$:
  - Formation: If A is a formula and B is a formula, then $(A \vee B)$ is a formula.
  - Introduction:

        - If $q : A$, then $\text{inl}\ q : (A \vee B)$.
        - If $r : B$, then $\text{inr}\ r : (A \vee B)$.
    - Elimination: If $p : (A \vee B), f : (A \implies C)$, and $g : (B \implies C)$, then $\text{cases}\ pfg : C$.
    - Computation:
        - $\text{cases}(\text{inl}\ q)fg \to fq$.
        - $\text{cases}(\text{inr}\ r)fg \to gr$.
- Rules for $\implies$:
    - Formation: If A is a formula and B is a formula, then $(A \implies B)$ is a formula.
    - Introduction: If from the assumption $x : A$ the conclusion $e : B$ is derived, then $(\lambda x : A)e : (A \implies B)$.
    - Elimination: If $q : (A \implies B)$ and $a : A$, then $(qa) : B$.
    - Computation: $((\lambda x : A)e)a \to e[a/x]$.
- Rules for $\bot$:
    - Formation: $\bot$ is a formula.
    - Elimination: If $p : A$, then $\text{abort}\ p : A$.
- Rule of Assumption: If A is a formula, then $x : A$.

The Curry Howard Isomorphism

- Under the isomorphism, types correspond to propositions and members of those types to proofs.
- The rules are seen to explain:
    - Formation rule: What the types of the system are.
    - Introduction and Elimination rules: Which expressions are members of which types.
    - Computation rule: How these objects can be reduced to simpler forms, i.e. how we can evaluate expressions.
- Rules for $\wedge$:
    - Formation: If A is a type and B is a type, then $(A \wedge B)$ is a type.
    - Introduction: If $p : A$ and $q : B$, then $(p, q) : (A \wedge B)$.
    - Elimination:
        - If $r : (A \wedge B)$, then $\text{first}\ r : A$.
        - If $r : (A \wedge B)$, then $\text{second}\ r : B$.
    - Computation:
        - $\text{first}(p, q) \to p$.
        - $\text{second}(p, q) \to q$.
- Rules for $\vee$:
    - Formation: If A is a type and B is a type, then $(A \vee B)$ is a type.
    - Introduction:
        - If $q : A$, then $\text{inl}\ q : (A \vee B)$.

    - If $r : B$, then $\mathrm{inr}\ r : (A \vee B)$.
  - Elimination: If $p : (A \vee B), f : (A \implies C)$, and $g : (B \implies C)$, then $\mathrm{cases}\ pfg : C$.
  - Computation:
    - $\mathrm{cases}(\mathrm{inl}\ q) f g \to fq$.
    - $\mathrm{cases}(\mathrm{inr}\ r) f g \to gr$.
- Rules for $\implies$:
  - Formation: If A is a type and B is a type, then $(A \implies B)$ is a type.
  - Introduction: If from the assumption $x : A$ the conclusion $e : B$ is derived, then $(\lambda x : A)e : (A \implies B)$.
  - Elimination: If $q : (A \implies B)$ and $a : A$, then $(qa) : B$.
  - Computation: $((\lambda x : A)e)a \to e[a/x]$.
- Rules for $\bot$:
  - Formation: $\bot$ is a type.
  - Elimination: If $p : A$, then $\mathrm{abort}\ p : A$.
- Rule of Assumption: If A is a type, then $x : A$.

Quantifiers

- Rules for $\forall$:
  - Formation: If A is a formula and from the assumption $x : A$ the conclusion P is a formula, then $(\forall x : A). P$ is a formula.
  - Introduction: If from the assumption $x : A$ the conclusion $p : P$ is derived, then $(\lambda x : A)e : (\forall x : A). P$.
  - Elimination: If $a : A$ and $f : (\forall x : A). P$, then $fa : P[a/x]$.
  - Computation: $((\lambda x : A)p)a \to p[a/x]$.
- Rules for $\exists$:
  - Formation: If A is a formula and from the assumption $x : A$ the conclusion P is a formula, then $(\exists x : A). P$ is a formula.
  - Introduction: If $a : A$ and $p : P[a/x]$, then $(a, p) : (\exists x : A). P$.
  - Elimination:
    - If $p : (\exists x : A). P$, then $\mathrm{first}\ p : A$.
    - If $p : (\exists x : A). P$, then $\mathrm{second}\ p : P[\mathrm{first}\ p/x]$.
  - Computation:
    - $\mathrm{first}(p, q) \to p$.
    - $\mathrm{second}(p, q) \to q$.