2

SIMPLY TYPED λ-CALCULUS

Types were introduced in mathematics as a response to the Russell's paradox, found in the first naive axiomatizations of set theory. An attempt to use untyped λ-calculus as a foundational logical system by Church suffered from the ***Rosser-Kleene paradox***, as detailed in [KR35] and [Cur46]; and types were a way to avoid it. Once types are added, a deep connection between λ-calculus and logic arises. This connection will be discussed in the next chapter.

In programming languages, types indicate how the programmer intends to use the data, prevent errors and enforce certain invariants and levels of abstraction in programs. The role of types in λ-calculus when interpreted as a programming language closely matches what we would expect of types in any common programming language, and typed λ-calculus has been the basis of many modern type systems for programming languages.

Simply typed λ-calculus is a refinment of the untyped λ-calculus. In it, each term has a type, which limits how it can be combined with other terms. Only a set of basic types and function types between any to types are considered in this system. Whereas functions in untyped λ-calculus could be applied over any term, now a function of type $A \to B$ can only be applied over a term of type A, to produce a new term of type B, where A and B could be, themselves, function types.

We will give now a presentation of simply typed λ-calculus based on [HS08]. Our presentation will rely only on the *arrow type constructor* $\to$. While other presentations of simply typed λ-calculus extend this definition with type constructors providing pairs or union types, as it is done in [Sel13], it seems clearer to present a first minimal version of the λ-calculus. Such extensions will be explained later, and its exposition will profit from the logical interpretation that we will explain in "propositions as types".

2.1 SIMPLE TYPES

We start assuming a set of **basic types**. Those basic types would correspond, in a programming language interpretation, with the fundamental types of the language. Examples would be the type of strings or the type of integers. Minimal presentations of λ-calculus tend to use only one basic type.

Definition 17 (Simple types)**.** The set of **simple types** is given by the following Backus-Naur form

$$\texttt{Type} ::= \iota \mid \texttt{Type} \to \texttt{Type},$$

where ι would be any *basic type*.

That is to say that, for every two types A, B, there exists a **function type** $A \to B$ between them.

2.2 TYPING RULES FOR SIMPLY TYPED λ-CALCULUS

We will now define the terms of simply typed λ-calculus using the same constructors we used on the untyped version. Those are the ***raw typed λ-terms***.

Definition 18 (Raw typed lambda terms)**.** The set of **typed lambda terms** is given by the following Backus-Naus form

$$\texttt{Term} ::= x \mid \texttt{Term}\ \texttt{Term} \mid \lambda x^{\texttt{Type}}.\texttt{Term}.$$

The main difference here with Definition 1 is that every bound variable has a type, and therefore, every λ-abstraction of the form $(\lambda x^A.M)$ can be applied only over terms type A; if M is of type B, this term will be of type $A \to B$.

However, the set of raw typed λ-terms contains some meaningless terms under this type interpretation, such as $(\lambda x^A.M)(\lambda x^A.M)$.[1] **Typing rules** will give them the desired expressive power; only a subset of these raw lambda terms will be typeable, and we will choose to work only with that subset. When a particular term M has type A, we write this relation as $M : A$, where the : symbol should be read as "is of type".

Definition 19 (Typing context)**.** A **typing context** is a sequence of type assumptions $x_1 : A_1, \ldots, x_n : A_n$, where no variable x_i appears more than once. We will implicitely assume that the order in which these assumptions appear does not matter.

Every typing rule assumes a typing context, usually denoted by Γ. Concatenation of typing contexts is written as Γ, Γ'; and the fact that ψ follows from Γ is written as $\Gamma \vdash \psi$. Typing rules are written as rules of inference; the premises are listed above and the conclusion is written below the line.

1 : In particular, we can not apply a function of type $A \to B$ to a term of type $A \to B$; it is expecting a term of type A.

1. The (var) rule simply makes explicit the type of a variable from the context. That is, a context that assumes that $x : A$ can be written as $\Gamma, x : A$; and we can trivially deduce from it that $x : A$.

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

2. The (abs) rule declares that the type of a λ-abstraction is the type of functions from the variable type to the result type. If a term $M : B$ can be built from the assumption that $x : A$, then $\lambda x^A.M : A \to B$. It acts as a *constructor* of function terms.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\,(abs)$$

3. The (app) rule declares the type of a well-typed application. A term $f : A \to B$ applied to a term $a : A$ is a term $f\ a : B$. It acts as a *destructor* of function terms.

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B}\,(app)$$

Definition 20. A term M is **typeable** in a giving context Γ if a typing judgement of the form $\Gamma \vdash M : T$ can be derived using only the previous typing rules.

From now on, we only consider typeable terms as the only terms of simply typed λ-calculus. As a consequence, the set of λ-terms of simply typed λ-calculus is only a subset of the terms of untyped λ-calculus.

Example 1 (Typeable and non-typeable terms). The term $\lambda f.\lambda x.f(fx)$ is typeable. If we abbreviate $\Gamma = f : A \to A,\ x : A$, the detailed typing derivation can be written as

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : A \to A}\,(var) \quad \dfrac{\dfrac{}{\Gamma \vdash x : A}\,(var) \quad \dfrac{}{\Gamma \vdash f : A \to A}\,(var)}{\Gamma \vdash f\ x : A}\,(app)}{f : A \to A, x : A \vdash f(fx) : A}\,(app)}{f : A \to A \vdash \lambda x.f(fx) : A \to A}\,(abs)}{\vdash \lambda f.\lambda x.f(fx) : (A \to A) \to A \to A}\,(abs)$$

The term $(\lambda x.x\ x)$, however, is not typeable. If x were of type ψ, it also should be of type $\psi \to \sigma$ for some σ in order for $x\ x$ to be well-typed; but $\psi \equiv \psi \to \sigma$ is not solvable, as it can be shown by structural induction on the term ψ.

It can be seen that the typing derivation of a term somehow encodes the complete λ-term. If we were to derive the term bottom-up, there would be only one possible choice at each step on which rule to use. In the following sections we will discuss a type inference algorithm that determines if a type is typeable and what its type should be, and we will make precise these intuitions.

2.3 CURRY-STYLE TYPES

Two different approaches to typing in λ-calculus are commonly used.

- **Church-style** typing, also known as *explicit typing*, originated from the work of Alonzo Church in [Chu40], where he described a simply-typed lambda calculus with two basic types. The term's type is defined as an intrinsic property of the term; and the same term has to be always interpreted with the same type.
- **Curry-style** typing, also known as *implicit typing*; which creates a formalism where every single term can be given an infinite number of types. This technique is called ***polymorphism*** when it is a formal part of the language; but here, it is only used to allow us to build intermediate terms without having to directly specify their type.

As an example, we can consider the identity term $I = \lambda x.x$. It would have to be defined for each possible type. That is, we should consider a family of different identity terms $I_A = \lambda x.x : A \to A$. Curry-style typing allows us to consider parametric types with type variables, and to type the identity as $I = \lambda x.x : \sigma \to \sigma$ where σ would a free type variable.

Definition 21 (Parametric types)**.** Given a infinite numerable set of *type variables*, we define **parametric types** or **type templates** inductively as

$$\texttt{PType} ::= \iota \mid \texttt{Tvar} \mid \texttt{PType} \to \texttt{PType},$$

where ι is a basic type, `Tvar` is a type variable and `PType` is a parametric type. That is, all basic types and type variables are atomic parametric types; and we also consider the arrow type between two parametric types.

The difference between the two typing styles is then not a mere notational convention, but a difference on the expressive power that we assign to each term. The interesting property of type variables is that they can act as placeholders for other type templates. This is formalized with the notion of type substitution.

Definition 22 (Type substitution)**.** A **substitution** ψ is any function from type variables to type templates. Any substitution ψ can be extended to a function between type templates called $\overline{\psi}$ and defined inductively by

- $\overline{\psi}\iota = \iota$, for any basic type ι;
- $\overline{\psi}\sigma = \psi\sigma$, for any type variable σ;
- $\overline{\psi}(A \to B) = \overline{\psi}A \to \overline{\psi}B$.

That is, the parametric type $\overline{\psi}A$ is the same as A but with every type variable replaced according to the substitution ψ.

We consider a type to be *more general* than other if the latter can be obtained by applying a substitution to the former. In this case, the latter is called an *instance* of the former. For example, $A \to B$ is more general than its instance $(C \to D) \to B$, where

A has been substituted by $C \to D$. An crucial property of simply typed λ-calculus is that every type has a most general type, called its *principal type*; this will be proved in Theorem 5.

Definition 23 (Principal type)**.** A closed λ-term M has a **principal type** π if $M : \pi$ and given any $M : \tau$, we can obtain τ as an instance of π, that is, $\overline{\sigma}\pi = \tau$.

2.4 UNIFICATION AND TYPE INFERENCE

The unification of two type templates is the construction of two substitutions making them equal as type templates; that is, the construction of a type that is a particular instance of both at the same time. We will not only aim for an unifier but for the most general one between them.

Definition 24 (Most general unifier)**.** A substitution ψ is called an **unifier** of two sequences of type templates $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ if $\overline{\psi}A_i = \overline{\psi}B_i$ for any i. We say that it is the **most general unifier** if given any other unifier ϕ exists a substitution φ such that $\phi = \overline{\varphi} \circ \psi$.

Lemma 4 (Unification)**.** *If an unifier of $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ exists, the most general unifier can be found using the following recursive definition of $\mathtt{unify}(A_1, \dots, A_n; B_1, \dots, B_n)$.*

1. $\mathtt{unify}(x; x) = \mathrm{id}$ *and* $\mathtt{unify}(\iota, \iota) = \mathrm{id}$*;*
2. $\mathtt{unify}(x; B) = (x \mapsto B)$*, the substitution that only changes x by B; if x does not occur in B. The algorithm* ***fails*** *if x occurs in B;*
3. $\mathtt{unify}(A; x)$ *is defined symmetrically;*
4. $\mathtt{unify}(A \to A'; B \to B') = \mathtt{unify}(A, A'; B, B')$*;*
5. $\mathtt{unify}(A, A_1, \dots; B, B_1, \dots) = \overline{\psi} \circ \rho$ *where* $\rho = \mathtt{unify}(A_1, \dots; B_1, \dots)$ *and* $\psi = \mathtt{unify}(\overline{\rho}A; \overline{\rho}B)$*;*
6. $\mathtt{unify}$ *fails in any other case;*

where x is any type variable. The two sequences $\{A_i\}, \{B_i\}$ of types have no unifier if and only if $\mathtt{unify}(\{A_i\}; \{B_i\})$ fails.

Proof. It is easy to notice by structural induction that, if $\mathtt{unify}(A; B)$ exists, it is in fact an unifier.

If the unifier fails in clause 2, there is obviously no possible unifier: the number of constructors on the first type template will be always smaller than the second one. If the unifier fails in clause 6, the type templates are fundamentally different, they have different head constructors and this is invariant to substitutions. This proves that the failure of the algorithm implies the non existence of an unifier.

We now prove that, if A and B can be unified, $\mathtt{unify}(A, B)$ is the most general unifier. For instance, in the clause 2, if we call $\psi = (x \mapsto B)$ and, if η were another unifier,

then $\eta x = \overline{\eta} x = \overline{\eta} B = \overline{\eta}(\psi(x))$; hence $\overline{\eta} \circ \psi = \eta$ by definition of ψ. A similar argument can be applied to clauses 3 and 4. In the clause 5, we suppose the existence of some unifier ψ'. The recursive call gives us the most general unifier ρ of $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$; and since it is more general than ψ', there exists an α such that $\overline{\alpha} \circ \rho = \psi'$. Now, $\overline{\alpha}(\overline{\rho} A) = \psi'(A) = \psi'(B) = \overline{\alpha}(\overline{\rho} B)$, hence α is a unifier of $\overline{\rho} A$ and $\overline{\rho} B$; we can take the most general unifier to be ψ, so $\overline{\beta} \circ \psi = \overline{\alpha}$; and finally, $\overline{\beta} \circ (\overline{\psi} \circ \rho) = \overline{\alpha} \circ \rho = \psi'$.

We also need to prove that the unification algorithm terminates. Firstly, we note that every substitution generated by the algorithm is either the identity or it removes at least one type variable. We can perform induction on the size of the argument on all clauses except for clause 5, where a substitution is applied and the number of type variables is reduced. Therefore, we need to apply induction on the number of type variables and only then apply induction on the size of the arguments. □

Using unification, we can define type inference.

Theorem 5 (Type inference)**.** *The algorithm* $\texttt{typeinfer}(M, B)$*, defined as follows, finds the most general substitution* σ *such that* $x_1 : \sigma A_1, \ldots, x_n : \sigma A_n \vdash M : \overline{\sigma} B$ *is a valid typing judgment if it exists; and fails otherwise.*

1. $\texttt{typeinfer}(x_i : A_i, \Gamma \vdash x_i : B) = \texttt{unify}(A_i, B)$;
2. $\texttt{typeinfer}(\Gamma \vdash MN : B) = \overline{\varphi} \circ \psi$*, where* $\psi = \texttt{typeinfer}(\Gamma \vdash M : x \to B)$ *and* $\varphi = \texttt{typeinfer}(\overline{\psi}\Gamma \vdash N : \overline{\psi} x)$ *for a fresh type variable* x*;*
3. $\texttt{typeinfer}(\Gamma \vdash \lambda x.M : B) = \overline{\varphi} \circ \psi$ *where* $\psi = \texttt{unify}(B; z \to z')$ *and* $\varphi = \texttt{typeinfer}(\overline{\psi}\Gamma, x : \overline{\psi} z \vdash M : \overline{\psi} z')$ *for fresh type variables* z, z'*.*

Note that the existence of fresh type variables is always asserted by the set of type variables being infinite. The output of this algorithm is defined up to a permutation of type variables.

Proof. The algorithm terminates by induction on the size of M. It is easy to check by structural induction that the inferred type judgments are in fact valid. If the algorithm fails, by Lemma 4, it is also clear that the type inference is not possible.

On the first case, the type is obviously the most general substitution by virtue of the previous Lemma 4. On the second case, if α were another possible substitution, in particular, it should be less general than ψ, so $\alpha = \beta \circ \psi$. As β would be then a possible substitution making $\overline{\psi}\Gamma \vdash N : \overline{\psi} x$ valid, it should be less general than φ, so $\alpha = \overline{\beta} \circ \psi = \overline{\gamma} \circ \overline{\varphi} \circ \beta$. On the third case, if α were another possible substitution, it should unify B to a function type, so $\alpha = \overline{\beta} \circ \psi$. Then β should make the type inference $\overline{\psi}\Gamma, x : \overline{\psi} z \vdash M : \overline{\psi} z'$ possible, so $\beta = \overline{\gamma} \circ \varphi$. We have proved that the inferred type is in general the most general one. □

Corollary 2 (Principal type property)**.** *Every typeable pure* λ*-term has a principal type.*

Proof. Given a typeable term M, we can compute $\texttt{typeinfer}(x_1 : A_1, \ldots, x_n : A_n \vdash M : B)$, where $x_1, \ldots, x_n$ are the free variables on M and $A_1, \ldots, A_n, B$ are fresh type

variables. By virtue of Theorem 5, the result is the most general type of M if we assume the variables to have the given types. □

2.5 SUBJECT REDUCTION AND NORMALIZATION

A crucial property is that type inference and β-reductions do not interfere with each other. A term can be β-reduced without changing its type.

Theorem 6 (Subject reduction)**.** *The type is preserved on β-reductions; that is, if $\Gamma \vdash M : A$ and and $M \twoheadrightarrow_\beta M'$, then $\Gamma \vdash M' : A$.*

Proof. If M' has been derived by β-reduction, $M = (\lambda x.P)$ and $M' = P[Q/x]$. $\Gamma \vdash M : A$ implies $\Gamma, x : B \vdash P : A$ and $\Gamma \vdash Q : B$. Again by structural induction on P (where the only crucial case uses that x and Q have the same type) we can prove that substitutions do not alter the type and thus, $\Gamma, Q : B \vdash P[Q/x] : A$. □

We have seen previously that the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not weakly normalizing; but it is also non-typeable. In this section we will prove that, in fact, every typeable term is strongly normalizing. We start proving some lemmas about the notion of *reducibility*, which will lead us to the Strong Normalization Theorem. This proof will follow [GTL89].

The notion of ***reducibility*** is an abstract concept originally defined by Tait in [Tai67] which we will use to ease this proof. It should not be confused with the notion of β-reduction.

Definition 25 (Reducibility)**.** We inductively define the set of **reducible** terms of type T for basic and arrow types.

- If $t : T$ where T a basic type, $t \in \mathtt{RED}_T$ if t is strongly normalizable.
- If $t : U \to V$, an arrow type, $t \in \mathtt{RED}_{U \to V}$ if $t\ u \in \mathtt{RED}_V$ for all $u \in \mathtt{RED}_U$.

Properties of reducibility will be used directly in the Strong Normalization Theorem. We prove three of them at the same time in order to use mutual induction.

Proposition 1 (Properties of reducibility)**.** *The following three properties hold;*

1. *if $t \in \mathtt{RED}_T$, then t is strongly normalizable;*
2. *if $t \in \mathtt{RED}_T$ and $t \to_\beta t'$, $t' \in \mathtt{RED}_T$; and*
3. *if t is not a λ-abstraction and $t' \in \mathtt{RED}_T$ for every $t \to_\beta t'$, then $t \in \mathtt{RED}_T$.*

Proof. For basic types,

1. holds trivially;
2. holds by the definition of strong normalization;

3. if any one-step β-reduction leads to a strongly normalizing term, the term itself must be strongly normalizing.

For arrow types,

1. if $x : U$ is a variable, we can inductively apply (3) to get $x \in \mathtt{RED}_U$; then, $t\ x \in \mathtt{RED}_V$ is strongly normalizing and t in particular must be strongly normalizing;
2. if $t \to_\beta t'$ then for every $u \in \mathtt{RED}_U$, $t\ u \in \mathtt{RED}_V$ and $t\ u \to_\beta t'\ u$. By induction, $t'\ u \in \mathtt{RED}_V$;
3. if $u \in \mathtt{RED}_U$, it is strongly normalizable. As t is not a λ-abstraction, he term $t\ u$ can only be reduced to $t'\ u$ or $t\ u'$. If $t \to_\beta t'$; by induction, $t'\ u \in \mathtt{RED}_V$. If $u \to_\beta u'$, we could proceed by induction over the length of the longest chain of β-reductions starting from u and assume that $t\ u'$ is irreducible. In every case, we have proved that $t\ u$ only reduces to already reducible terms; thus, $t\ u \in \mathtt{RED}_U$.

□

Lemma 5 (Abstraction lemma)**.** *If $v[u/x] \in \mathtt{RED}_V$ for all $u \in \mathtt{RED}_U$, then $\lambda x.v \in \mathtt{RED}_{U\to V}$.*

Proof. We apply induction over the sum of the lengths of the longest β-reduction sequences from $v[x/x]$ and u. The term $(\lambda x.v)u$ can be β-reduced to

- $v[u/x] \in \mathtt{RED}_U$; in the base case of induction, this is the only choice;
- $(\lambda x.v')u$ where $v \to_\beta v'$, and, by induction, $(\lambda x.v')u \in \mathtt{RED}_V$;
- $(\lambda x.v)u'$ where $u \to_\beta u'$, and, again by induction, $(\lambda x.v)u' \in \mathtt{RED}_V$.

Thus, by Proposition 1, $(\lambda x.v) \in \mathtt{RED}_{U\to V}$. □

A final lemma is needed before the proof of the Strong Normalization Theorem. It is a generalization of the main theorem, useful because of the stronger induction hypothesis it provides.

Lemma 6 (Strong Normalization lemma)**.** *Given an arbitrary $t : T$ with free variables $x_1 : U_1, \ldots, x_n : U_n$, and reducible terms $u_1 \in \mathtt{RED}_{U_1}, \ldots, u_n \in \mathtt{RED}_{U_2}$, we know that*

$$t[u_1/x_1][u_2/x_2]\ldots[u_n/x_n] \in \mathtt{RED}_T.$$

Proof. We call $\tilde{t} = t[u_1/x_1][u_2/x_2]\ldots[u_n/x_n]$ and apply structural induction over t,

- if $t = x_i$, then we simply use that $u_i \in \mathtt{RED}_{U_i}$,
- if $t = v\ w$, then we apply induction hypothesis to get $\tilde{v} \in \mathtt{RED}_{R\to T}, \tilde{w} \in \mathtt{RED}_R$ for some type R. Then, by definition, $\tilde{t} = \tilde{v}\ \tilde{w} \in \mathtt{RED}_T$,
- if $t = \lambda y.v : R \to S$, then by induction $\tilde{v}[r/y] \in \mathtt{RED}_S$ for every $r : R$. We can then apply Lemma 5 to get that $\tilde{t} = \lambda y.\tilde{v} \in \mathtt{RED}_{R\to S}$. □

Theorem 7 (Strong Normalization Theorem)**.** *In simply typed λ-calculus, all terms are strongly normalizing.*

Proof. It is the particular case of Lemma 6 where we take $u_i = x_i$. □

Every term normalizes in simply typed λ-calculus and every computation ends. We know, however, that the Halting Problem is unsolvable, so simply typed λ-calculus must be not Turing complete.