Types and the Simply-Typed λ-Calculus

1 Background on Type Systems

A *type* describes a set of related values that all admit the same legal operations, e.g., the type **num** describes the set of all natural numbers (which admit the operations $+, -, \times, \div$). When we say that an expression has some type τ, we are saying that when the expression is evaluated the resulting value will belong to the set described by τ. Here are some examples of common types:

Type	Description	Example
num	natural numbers	42
bool	boolean values	**true**
num $\rightarrow$ **bool**	functions from numbers to booleans	$\lambda x : \textbf{num} \, . \, x \leq 42$

Types prevent specific kinds of bad behavior in programs. They do this by restricting expressions involving certain operators to have certain types. For example, if in our language the + operator only works on numbers, then in the expression $e_1 + e_2$ the type of expressions e_1 and e_2 should both be **num**. If we can't prove that e_1 and e_2 will always evaluate to numbers, then it's possible when the program is run that it will try to add two things that aren't numbers, which doesn't make sense in our language.

1.1 Terminology and Definitions

There is a lot of confusion surrounding typing and its terminology. People often use the terms incorrectly or inconsistently, which can make the debates about typing very frustrating. Here we'll lay out a number of terms and definitions as given in the paper "Type Systems" by Luca Cardelli, an authority on type systems:

A **trapped error** makes the program halt immediately and signal the error. For example, division by zero is a trapped error, as is an array out of bounds access in Java. An **untrapped error** allows the program to proceed execution without signaling that anything bad happened. For example, an array out of bounds access in C is an untrapped error. A program is called **safe** if it cannot cause any untrapped errors. If we can guarantee that all programs in a certain language are safe, it's called a **safe language**. Sometimes we want to guarantee that programs do not cause certain kinds of trapped errors in addition to untrapped errors; in this case we define a set of **forbidden errors** which include all untrapped errors and also some trapped errors (which ones are included is entirely up to the language designer). A program that cannot cause a forbidden error is called **well-behaved**. If we can guarantee that all programs in a certain language are well-behaved, then it's called a **strongly typed** language; otherwise it's a **weakly typed** language. Java, Scala, Haskell, and OCaml are examples of strongly typed languages; C and C++ are examples of weakly typed languages.

Languages can, independently from being strongly or weakly typed, be either **statically typed** or **dynamically typed**. In a statically typed language, we use a type system *before* a program is actually executed to filter out potentially ill-behaved programs (e.g., in a compiler or interpreter right after the program is read in and parsed but before it is compiled or executed). In a dynamically typed language, we use runtime checks *during* program execution to detect and reject ill-behaved programs. Essentially, a dynamically typed language is designed so that all errors are trapped errors. The tradeoffs between statically and dynamically typed languages are hotly debated in the programming community, with strong (almost fanatic) adherents on both sides.

The Case for Static Typing. A statically typed language can guarantee the absence of entire classes of errors in a program; potential errors are detected at compile-time rather than lurking undetected only to be found after the code is deployed. Type annotations serve as code documentation that, unlike programmer comments which are too often missing, inaccurate, or out of date, are guaranteed to be correct. Finally, type information allows the compiler/interpreter to do a much better job optimizing the code for better performance; statically typed programs are often many times faster than equivalent dynamically typed programs. Examples of statically typed languages include C, C++, Java, Scala, Haskell, and OCaml.

The Case for Dynamic Typing. A dynamically typed language can be more flexible than a statically typed language because the programmer does not need to satisfy the type system in order to get running code. This tends to allow much more rapid development, which can be important in some domains where time-to-market is an important consideration. Dynamically typed language enthusiasts argue that unit testing is sufficient to detect and prevent the kinds of errors that type systems guard against. There has also been a lot of research into making dynamically typed languages faster, and though they are still on average much slower than statically typed languages, the gap is narrowing. Examples of dynamically typed languages include JavaScript, Python, Ruby, Scheme, and Lisp.

Statically typed languages can also be divided into two camps: **explicitly typed** and **implicitly typed**. In an explicitly typed language the programmer annotates variables in the program with type information, explicitly telling the compiler/interpreter what the types are. This annotation can be burdensome, especially if we're using a powerful and expressive type system. In an implicitly typed language the programmer leaves out all of the type annotations, making the program syntactically *look* like a dynamically typed program—but rather than using runtime checks as in a dynamically typed language, the compiler/interpreter automatically *infers* the correct type information and checks the program ahead of time. Essentially, the compiler/interpreter inserts an extra step after parsing but before type checking that automatically adds the type annotations into the program, without the programmer needing to do anything; this step is called *type inference*. The more powerful and expressive the type system, the more burdensome the type annotations and the more convenient it is to have type inference—but the more difficult and expensive type inference can be. C, C++, and Java are examples of explicitly typed languages; OCaml and Haskell are examples of implicitly typed languages. Scala is a hybrid of the two: its type system is too powerful to allow for full type inference, but it does perform a local form of type inference to remove a lot of the type annotation burden.

1.2 A Brief History of Type Systems

The story of type systems starts long before the era of electronic computers and writing programs. Type systems were invented by Bertrand Russell and Alfred Whitehead in their book *Principia Mathematica* ("Principles of Mathematics"), written between 1910 to 1913. Russell and Whitehead wrote the Principia Mathematica in an attempt to lay out a formal, logical foundation for all of mathematics. They were inspired to make this effort by a severe flaw in the underlying theory of mathematics exposed by, for example, Russell's Paradox:

Russell's Paradox: Define a set `U` as the set containing all sets that do not contain themselves. Is `U` a member of itself?

- **Yes.** Then `U` is a member of itself, and hence by definition cannot be a member of `U`.
- **No.** Then `U` is not a member of itself, and hence by definition must be a member of `U`.

The question demands a yes or no answer, but we cannot answer either yes or no—thus it is a paradox. There are many other formulations that all end up with similar difficulties. The fundamental problem exposed by this paradox is that naïve set theory, previously used to justify mathematical theory, contains contradictions that render that theory useless.

Russell and Whitehead used type systems to disallow the formulation of definitions that lead to paradox. The definition of `U` in Russell's Paradox would, then, be ill-typed and thus not considered a proper definition. Computer scientists in the 1950s and 60s borrowed the theory of type systems from mathematics and applied it to the theory of programming languages that they were developing at the time. Type systems are still a large and rapidly moving research area in the field of Programming Languages.

1.3 Limits of Type Systems

One way to think of a type system is as a *decision procedure*, and a type checker is a lightweight theorem prover used to prove the absence of certain kinds of program behaviors. A natural question to ask is: can a type system reject all ill-behaved programs while accepting all well-behaved programs? Unfortunately the answer is *no*; it falls in the same class of undecidable problems that we discussed in the context of Hilbert's Entsheidungsproblem. This fact means that any type system which rejects all ill-behaved programs *must necessarily* also reject some well-behaved programs. It is impossible to create a type system that exactly distinguishes between ill-behaved and well-behaved programs.

This result establishes a theoretical limit, but how well do type systems actually do in practice? It turns out that there are a wide spectrum of type systems that range from very simple (thus efficient to compute, but very limiting) to very complex (thus difficult to compute, but very expressive). A large part of ongoing programming languages research centers around trying to increase the power and expressiveness of type systems while still keeping them practical to use.

2 Simply-Typed λ-Calculus

We will examine a explicit, strongly typed version of the extended λ-calculus we previously defined.

2.1 Syntax of the Simply-Typed λ-Calculus

$$x \in \mathit{Variable} \qquad n \in \mathbb{N} \qquad b \in \mathit{Bool}$$

$$\begin{aligned} e \in \mathit{Exp} ::= {} & n \mid b \mid x \mid \lambda x{:}\tau\,.\,e \mid e_1\ e_2 \\ & \mid (e_1, e_2) \mid \textsf{fst}(e) \mid \textsf{snd}(e) \\ & \mid \textsf{inl}_\tau(e) \mid \textsf{inr}_\tau(e) \mid \textsf{case}\ e_1\ \textsf{of inl}\ x \Rightarrow e_2 \mid \textsf{inr}\ x \Rightarrow e_3 \end{aligned}$$

We are using the lambda calculus extended with natural numbers, booleans, pairs, and unions. Note that $\textsf{inl}_\tau$ and $\textsf{inr}_\tau$ are subscripted with the type of the *other* side of the union, i.e., the one that isn't being used. Notice that there are no arithmetic, relational, or logical operators given in the syntax; for simplicity we'll assume that they are all given as builtin functions with the appropriate types.

2.2 Types Used in the Simply-Typed λ-Calculus

$$\tau \in \mathit{Type} = \textsf{num} \mid \textsf{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \rightarrow \tau_2$$

We have the *primitive* (or *base*) types **num** and **bool**, the type of pairs $\tau_1 \times \tau_2$ (i.e., a pair whose first element is of type τ_1 and whose second element is of type τ_2), the type of unions $\tau_1 + \tau_2$ (i.e., a union whose left value is of type τ_1 and whose right value is of type τ_2), and the type of functions $\tau_1 \rightarrow \tau_2$ (i.e., a function whose parameter is type τ_1 and whose return value is type τ_2).

2.3 Type Judgements

Given a program in the simply-typed λ-calculus we need some way to determine whether that program is well-typed, i.e., whether the program is guaranteed to be well-behaved. Being well-typed is a *judgement*, exactly like we had for first-order logic. Here our judgements are of the form $\Gamma \vdash e : \tau$, meaning "under the set of hypotheses Γ, expression e has type τ". The hypotheses used in our judgements are lists of assumptions about the types of variables, e.g., $\Gamma = x_1{:}\tau_1,\ x_2{:}\tau_2$, etc. We define a set of inference rules that allow us to make judgements about expressions in the simply-typed λ-calculus:

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\ \textsc{var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma \vdash \lambda x{:}\tau_1\,.\,e : \tau_1 \rightarrow \tau_2} \rightarrow\mathrm{I} \qquad \frac{\Gamma \vdash e_1{:}\tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2{:}\tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \rightarrow\mathrm{E}$$

$$\frac{}{\Gamma \vdash n : \textsf{num}}\ n\mathrm{I} \qquad \frac{}{\Gamma \vdash b : \textsf{bool}}\ b\mathrm{I}$$

$$\frac{\Gamma \vdash e_1{:}\tau_1 \quad \Gamma \vdash e_2{:}\tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \times\mathrm{I} \qquad \frac{\Gamma \vdash e{:}\tau_1 \times \tau_2}{\Gamma \vdash \textsf{fst}(e) : \tau_1} \times\mathrm{E1} \qquad \frac{\Gamma \vdash e{:}\tau_1 \times \tau_2}{\Gamma \vdash \textsf{snd}(e) : \tau_2} \times\mathrm{E2}$$

$$\frac{\Gamma \vdash e{:}\tau_1}{\Gamma \vdash \textsf{inl}_{\tau_2}(e) : \tau_1 + \tau_2} +\mathrm{I1} \qquad \frac{\Gamma \vdash e{:}\tau_2}{\Gamma \vdash \textsf{inr}_{\tau_1}(e) : \tau_1 + \tau_2} +\mathrm{I2} \qquad \frac{\Gamma \vdash e{:}\tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1{:}\tau_3 \quad \Gamma, x{:}\tau_2 \vdash e_2{:}\tau_3}{\Gamma \vdash \textsf{case}\ e\ \textsf{of inl}\ x \Rightarrow e_1 \mid \textsf{inr}\ x \Rightarrow e_2 : \tau_3} +\mathrm{E}$$

3 Curry-Howard Correspondence

Natural deduction is a formal system for expressing proofs, while λ-calculus is a formal system for expressing functions. There is a powerful connection between these two ideas, called the *Curry-Howard Correspondence*. The central insight is that an expression in the λ-calculus is actually a logical proof derivation using natural deduction, and vice-versa. Let **Prop** be the set of propositional variables in first-order logic (i.e., the 0-arity predicates) and **Prim** be the set of primitive types (e.g., **num** and **bool**). If we make these sets equal, then the set of propositional logic formulas is identical to the set of types that can be given to an expression in λ-calculus. We can extend this correspondance to include quantifiers and all other aspects of first-order logic; in fact, we can even extend the correspondence to include higher-order logic. This correspondence has been a rich source of ideas for programming language research.