Handout 5: Simply Typed Lambda Calculus

Most programming languages used for serious software development are typed languages. The reason for their popularity is that types allow us to check mechanically whether we are plugging together compatible pieces of programs. When we plug together incompatible pieces either due to lack of understanding, confusion, or misunderstandings, we end up constructing programs that do not function correctly. Type checking allows us to catch such problems early and leads to more reliable software.

In Mathematics, typed languages have been developed because their untyped variants have led to paradoxes. The most famous of them is the Russell’s paradox. If we don’t distinguish between the type of elements and the type of sets of such elements, then the language allows us to express “the set of all sets” which cannot be shown to exist or not exist. However, the typed languages are constraining. So, there is ever a temptation to devise untyped languages.

Church first developed a typed version of the lambda calculus which he called the “simple theory of types”. He later generalized it to the untyped lambda calculus in order to create a more expressive language. However, the untyped lambda calculus, if used as the foundation for logic, leads one back to paradoxes. Paradoxes are not a problem in programming. Our interest in types is pragmatic.

Simple type system

1. Syntax of types

Consider an applied lambda calculus with integers and boolean values. (We omit cons-pairs for now.) The types for this language are given by the following inductive definition:

- int and bool are types.
- If $T_1$ and $T_2$ are types, then $T_1 \to T_2$ is a type.

We can also write this definition as the grammar

$$T ::= \text{int} \mid \text{bool} \mid T_1 \to T_2$$

We use the convention that the “$\to$” symbol associates to the right, i.e., $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$.

The obvious interpretation is that int is the type of integers, bool is the type of boolean values, and $T_1 \to T_2$ is the type of functions that take arguments of type $T_1$ and produce results of type $T_2$.

The types int and bool are called base types; the others are called higher types.

Note that we use $T$ as a mathematical variable to stand for types. But “$T$” is itself not a type. Rather, “int”, “int $\to$ int”, . . . are types. (This is similar to the use of a variable like $n$ to stand for integers. Clearly, “$n$” is not an integer. Rather, 0, 1, . . . are integers.)

2. Types of constants

We list the types of various constants used in our applied lambda calculus:

- The integer constants . . . , −2, −1, 0, 1, 2, . . . are all of type int.
- The boolean constants true and false are of type bool.
- The arithmetic operations $+$, $-$, $\ast$, $/$ and mod are of type int $\to$ int $\to$ int.
- The relational operations $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ are of type int $\to$ int $\to$ bool.
• We need two conditional branching functions: if\textsubscript{int} of type bool → int → int → int, and if\textsubscript{bool} of type bool → bool → bool → bool.

Note that, in the untyped calculus, we only had one constant for if for all types of values. Unfortunately, in the typed calculus, we need two separate constants for use with the two base types. And, we don’t have any conditionals for higher types! This is just one part of the price to pay for moving to a typed language. There is more.

3. Terms
The syntax of terms is similar to that of the untyped calculus. The main difference is that we need to declare types for the parameters of \( \lambda \) functions, as we do in Java. (However, unlike Java, we do not declare result types for \( \lambda \) functions. They can be inferred.) So, the grammar is:

\[
M ::= c \mid x \mid \lambda x : T. M' \mid M_1 M_2
\]

Terms satisfying this grammar are not necessarily valid terms of the typed lambda calculus. The grammar specifies only the “context-free syntax” of the terms. To be valid, the terms should also satisfy the typing rules of the language, which we describe below.

4. Type checking
A simple way to type check a term is to annotate the term and all its subterms with their types. This is cumbersome but quite effective. We show below the annotated term for the “and” function. (We abbreviate int to i and bool to b for readability.)

\[
(\lambda p : b. (\lambda q : b. (((((\text{if}_b \rightarrow b \rightarrow b \rightarrow b \rightarrow b) p \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b) (\text{false}_b) \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b)
\]

The typing rules of the simply typed lambda calculus are the following:

**Constants.** All the constants should be annotated with their types in the language definition.

**Variables.** For every abstraction term \( \lambda x : T. M \) all the free occurrences of \( x \) in \( M \) should be annotated with \( T \) which is the declared type of \( x \).

**Abstraction terms.** An abstraction term \( (\lambda x : T. M^U) \) is annotated with the type \( T \rightarrow U \).

**Application terms.** An application term \( (M^T N^S) \) is type correct only if \( T \) is a function type of the form \( T_1 \rightarrow T_2 \), and the type of the argument \( S \) is equal to the argument type of the function, i.e., \( T_1 = S \). In that case, the application term is annotated with the type \( T_2 \). If the rules are not satisfied, then the term is not type correct.

Let us see how these rules apply to the above example. The first two rules (the constant and variable rules) give the following annotation:

\[
(\lambda p : b. (\lambda q : b. (((((\text{if}_b \rightarrow b \rightarrow b \rightarrow b \rightarrow b) p \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b)
\]

Next, we can use the application rule to type check and annotate all the application terms. We do this inside out.

\[
(\lambda p : b. (\lambda q : b. (((((\text{if}_b \rightarrow b \rightarrow b \rightarrow b \rightarrow b) p \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b)))
\]

Finally, we use the abstraction rule to annotate the two abstraction terms, again inside out. Note that, if we had used a wrong type of argument, e.g., the integer 0 instead of false, then the type checking rules would not be satisfied.

\[
(\lambda p : b. (\lambda q : b. (((((\text{if}_b \rightarrow b \rightarrow b \rightarrow b \rightarrow b) p \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \rightarrow b)(0^i))
\]

The function has the type \( b \rightarrow b \) and the argument has the type \( i \), and this violates the constraint \( b = i \) required for the application to type-check.
5. Limitations of type systems

The use of type systems, while promoting program reliability, also places some severe restrictions on the kind of programs that can be written. Consider the following points:

- In the untyped lambda calculus, we can write functions that can take arguments of a variety of types. The simplest example is the identity function $\lambda x. x$. It can be applied to integers, reals, booleans, functions on such values, functions on functions and so on. But in the typed calculus, there is no such universal identity function. Instead, we have to write an identity function for integers, $\lambda x : \text{int}. x$, another for booleans $\lambda x : \text{bool}. x$, yet another for functions on integers $\lambda x : \text{int} \rightarrow \text{int}. x$ etc. Functions like the identity function, which can be applied to arguments of many (in fact, infinitely many) types, are called polymorphic functions. Most of the examples we have seen in Handout 2 are such polymorphic functions, e.g., $\text{fst} = \lambda x. \lambda y. x$ or $\text{twice} = \lambda f. \lambda x. f(f(x))$. The simply typed lambda calculus does not allow polymorphic functions. However, we will see that we can devise more sophisticated type systems for this purpose.

- In the simply typed lambda calculus, it is not possible to apply a function to itself. Why not? Suppose a term $M$ is of type $T_1 \rightarrow T_2$. Here, $T_1$ is a subterm of the type term $T_1 \rightarrow T_2$. So, it is strictly “smaller” than $T_1 \rightarrow T_2$. However, we want self-application $M(M)$ then $T_1$ should be equal to $T_1 \rightarrow T_2$. This is not possible. So, even though in the untyped lambda calculus, we can write $\text{id}(\text{id})$, $\text{twice(twice)}$ etc., we can’t write them in the typed calculus. This limitation can also be relaxed by devising more powerful type systems for polymorphic functions.

- Since the definition of the Y combinator fundamentally depends on self-application, we cannot write a term for it in the simply typed lambda calculus. However, the effect of the Y combinator can be given a type. To say that $x = Y f$ is to say that $x$ is recursively defined by the definition $x = f x$. So, if $x$ is supposed to be of type $T$, then $f$ must be of type $T \rightarrow T$. In other words, the Y combinator, when applied to functions of type $T \rightarrow T$, gives values of type $T$. So, even though Y combinator cannot be defined within the simply typed lambda calculus, it is possible to add it as an extension.

6. Overloading

In the simple type system, we need two separate constants for the conditional if_{int} and if_{bool}. However, in practice, it would be simpler to just write if for both of these constants. This is called overloading. We say that a symbol is overloaded if it is used to stand for different functions which differ in their types. We can resolve the overloading, i.e., figure out which function is meant by any particular use of the symbol, by looking at the types of the arguments. For example,

\[
\text{(if } p \text{ false) } \equiv (\text{if}_{\text{bool}} p \text{ false}) \\
\text{(if } p \text{ 0 (f x)) } \equiv (\text{if}_{\text{int}} p \text{ 0 (f x)})
\]

In the first example, the third argument of if is false which is of type bool. Hence, it is the bool version of if that is meant. In the second example, the second argument of if is 0, which is of type int. Hence, it is the int version of if.

Overloading is common in most programming languages. For example, if the language has a type of integers as well as a type of reals, then it is likely to overload all the arithmetic operator symbols $+, -, \times, \div$, as well as the comparison operators $=, \neq, \ldots$, for both the integer operations and real operations. That means, there is a $+_{\text{int}}$ function and a separate $+_{\text{real}}$ function and the same symbol $+$ is used for both of them. In all such cases, the ambiguity of the notation is resolved by type checking. If both the arguments are of type int, then it is $+_{\text{int}}$. If both the arguments are of type real, then it is of $+_{\text{real}}$.

What if one argument is of type int and another is of type real? In this case, a separate phenomenon called type conversion is used. Integers can always be treated as real numbers. So, the integer argument is converted to type real and then the real version of the operator to the two real-typed arguments.

7. Formal typing rules. The type checking rules described in paragraph 4 suffer from two problems. Firstly, they are informal and subject to vagueness and ambiguity that can arise in informal English.
prose. Secondly, they only describe how to type check closed terms (terms without free variables). We now describe formal typing rules which solve both of these problems.

8. Typing contexts
Types of free variables cannot be deduced by type checking. Instead, we must be given the types of the free variables in order to type check terms. The types of free variables are given as a list of typings such as:

\[ x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n \]

Here \( x_1, \ldots, x_n \) are distinct variables (or identifiers) and \( T_1, \ldots, T_n \) are types. A list of this form is called a typing context. The idea is that such a list forms the “context” for type checking a term. We use capitalized Greek letters \( \Gamma, \Delta \) to denote typing contexts.

A statement of the typing rules takes the form “in the context of \( \Gamma \), the term \( M \) has type \( T \)”. This is denoted symbolically as \( \Gamma \vdash M : T \). Here are some examples of valid typing statements:

\[ x : \text{int} \rightarrow \text{int}, y : \text{int}, z : \text{bool} \vdash x y : \text{int} \]
\[ \lambda x : \text{int}. x + 1 : \text{int} \rightarrow \text{int} \]
\[ f : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \vdash f (\lambda x : \text{int}. x + 1) \]

In the first example, we have been given the types \( x : \text{int} \rightarrow \text{int} \) and \( y : \text{int} \). So, \( x y \) is of type \( \text{int} \) by the rule for function application. The fact that there is another variable \( z \) in the typing context makes no difference. In the second example, the typing context is empty. This is fine too because \( \lambda x : \text{int}. x + 1 \) is a closed term. So, we don’t need any types of variables to type check it.

9. Typing rules for the simply typed lambda calculus
The typing rules specify how to give a type to a term, provided its subterms have been given types. So, the rules are typically of the form

if \( \Gamma_1 \vdash M_1 : T_1 \) and \( \Gamma_2 \vdash M_2 : T_2 \) and \ldots, then \( \Gamma \vdash M : T \).

We graphically depict such rules using the horizontal bar notation:

\[
\Gamma_1 \vdash M_1 : T_1 \quad \Gamma_2 \vdash M_2 : T_2 \quad \ldots \\
\Gamma \vdash M : T
\]

The statements above the horizontal bar are called the “premises” of the rule and the statement below it is called the “conclusion”. There is always a single conclusion in a rule. However, there can be any number of premises. There can also be zero premises, in which case we understand that the conclusion is always true.

Here are the typing rules of the simply typed lambda calculus. In addition to the rules for constants, variables, abstraction and application terms, we have also added a rule for the \( Y \) combinator extension:

\[
\text{(Const)} \quad \Gamma \vdash c : T \quad \text{if } c \text{ has the type } T \text{ in the language definition}
\]
\[
\text{(Variable)} \quad \Gamma \vdash x : T \quad \text{if } x : T \text{ is in } \Gamma
\]
\[
\text{(--Intro)} \quad \Gamma, x : T \vdash M : T' \quad \Gamma \vdash \lambda x : T. M : T \rightarrow T'
\]
\[
\text{(--Elim)} \quad \Gamma \vdash M : T_1 \rightarrow T_2 \quad \Gamma \vdash N : T_1 \quad \Gamma \vdash M N : T_2
\]
\[
\text{(Rec)} \quad \Gamma \vdash M : T \rightarrow T \quad \Gamma \vdash Y M : T
\]

Several observations should be made regarding the rules:
The rules \textit{Const} and \textit{Variable} have conditions stated on the side of the rule. These are often called “side conditions”. These conditions must be satisfied for the rules to be applicable.

The rules \(\rightarrow\text{Intro}\) and \(\rightarrow\text{Elim}\) are so called because they introduce and eliminate the \(\rightarrow\) symbol in the types of terms. The rule \(\rightarrow\text{Intro}\) has the type \(T \rightarrow T'\) in the conclusion whereas only \(T\) and \(T'\) in the premise. Thus an \(\rightarrow\) symbol has been introduced. The rule \(\rightarrow\text{Elim}\) has \(T_1 \rightarrow T_2\) in one of the premises, but this \(\rightarrow\) symbol doesn’t occur in the conclusion. This structure of \(\rightarrow\text{Intro}\) and \(\rightarrow\text{Elim}\) rules is a common pattern in well-designed type systems.

Some of the rules have the same mathematical variable appearing multiple times in the premises. For example, the rule \(\rightarrow\text{Elim}\), has \(\Gamma\) occurring twice and \(T_1\) occurring twice. The rule \(\text{Rec}\) has \(T\) appearing twice. In all such cases, we understand that the types or typing contexts in the different positions must be exactly the same.

The rule \(\rightarrow\text{Intro}\) has a subtlety. To give a type to the term \(\lambda x : T. M\) in the context \(\Gamma\), the rule requires that the term \(M\) should be given a type in the context \(\Gamma, x : T\). If \(\Gamma\) is the list of typings \(x_1 : T_1, \ldots, x_n : T_n\), then \(\Gamma, x : T\) means the list \(x_1 : T_1, \ldots, x_n : T_n, x : T\). However, recall that all the variables in the list should be distinct. That means that \(x\) should be distinct from each \(x_i\) in the remaining context. So, the rule does not allow us to type check a term like \(\lambda x : \text{int}. \lambda x' : \text{bool}. x\) because it would lead to a typing context of the form \(x : \text{int}, x : \text{bool}\) which is not well-formed. The solution is to rename one of the bound variables (using \(\alpha\) equivalence), e.g., \(\lambda x : \text{int}. \lambda x' : \text{bool}. x'\). This term can be typed using the rules.

It is possible to reformulate the \(\rightarrow\text{Intro}\) rule to eliminate this awkwardness with bound variables. However, the rule given above has the advantage of simplicity.

### 10. Typing derivations

A typing derivation is a tree structure of inferences, each of which is an instance of a typing rule, which serves to show that a term has a particular type. For example, here is a typing derivation for the \texttt{and} function mentioned in the paragraph 4. We abbreviate the typing context \(p : b, q : b\) as \(\Gamma\). (We again shorten \texttt{bool} to \(b\) for readability).

\[
\begin{array}{c}
\Gamma \vdash \text{if}_\text{bool} : b \rightarrow b \rightarrow b & \text{\texttt{Const}} \\
\Gamma \vdash p : b & \text{\texttt{Variable}} \\
\hline
\Gamma \vdash \text{if}_\text{bool} p : b \rightarrow b \rightarrow b & \rightarrow\text{Elim} \\
\Gamma \vdash q : b & \text{\texttt{Variable}} \\
\hline
\Gamma \vdash \text{if}_\text{bool} p q : b \rightarrow b & \rightarrow\text{Elim} \\
\Gamma \vdash \text{false} : b & \text{\texttt{Const}}
\end{array}
\]

The derivation can also be written in a linear form, by labeling typing statements and referring to them as needed from other statements which depend on them for their derivation. Here is the linear form of the above derivation:

1. \(\Gamma \vdash \text{if}_\text{bool} : b \rightarrow b \rightarrow b\) \hspace{1cm} using \texttt{Const}
2. \(\Gamma \vdash p : b\) \hspace{1cm} using \texttt{Variable}
3. \(\Gamma \vdash \text{if}_\text{bool} p : b \rightarrow b \rightarrow b\) \hspace{1cm} from 1 and 2, using \(\rightarrow\text{Elim}\)
4. \(\Gamma \vdash q : b\) \hspace{1cm} using \texttt{Variable}
5. \(\Gamma \vdash \text{if}_\text{bool} p q : b \rightarrow b\) \hspace{1cm} from 3 and 4, using \(\rightarrow\text{Elim}\)
6. \(\Gamma \vdash \text{false} : b\) \hspace{1cm} using \texttt{Const}
7. \(\Gamma \vdash \text{if}_\text{bool} p q \text{false} : b\) \hspace{1cm} from 5 and 6, using \(\rightarrow\text{Elim}\)
8. \(p : b \vdash \lambda q : b. \text{if}_\text{bool} p q \text{false} : b \rightarrow b\) \hspace{1cm} from 7, using \(\rightarrow\text{Intro}\)
9. \(\vdash \lambda p : b. \lambda q : b. \text{if}_\text{bool} p q \text{false} : b \rightarrow b\) \hspace{1cm} from 8, using \(\rightarrow\text{Intro}\)
In practice, it is rarely necessary to write out typing derivations in full. It is sufficient to annotate terms and subterms as shown in paragraph 4, but we must ensure that the annotation is done in accordance with the type rules.

Product types

11. Product types. Other kinds of type constructions can be added to the type system with no real interference. We show this for product types. A product type is of the form \( T_1 \times \ldots \times T_n \) where \( T_1, \ldots, T_n \) are arbitrary types. (In ML, the \( \times \) symbol is changed to \( \ast \) so that it can be entered from ASCII keyboards.) The nullary product type (the case of \( n = 0 \)) is denoted \textit{unit}.

12. Typing rules. Here are the typing rules for the term forms dealing with product types:

\[
\begin{align*}
(\times\text{Intro}) & \quad \Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n \\
& \quad \Gamma \vdash (M_1, \ldots, M_n) : T_1 \times \ldots \times T_n \\
\end{align*}
\]

where \( n > 0 \)

\[
\begin{align*}
(\times\text{Elim}) & \quad \Gamma \vdash M : T_1 \times \ldots \times T_n \\
& \quad \Gamma \vdash \text{sel}[i] M : T_i \\
\end{align*}
\]

for \( i = 1, \ldots, n \)

\[
\begin{align*}
(\text{unit Intro}) & \quad \Gamma \vdash () : \text{unit} \\
\end{align*}
\]

Record types

14. Record types. In Handout 8, we have introduced records (or structs) as a stylized form of tuples which allow access via mnemonic field names. We can formulate a type system for them in a similar fashion to tuples. A record type is written as:

\[
\{ x_1 : T_1 ; \ldots ; x_n : T_n \}
\]

where \( x_1, \ldots, x_n \) are distinct field names (identifiers) and \( T_1, \ldots, T_n \) are types. Such a record type is essentially a notational variant of the product type \( T_1 \times \ldots \times T_n \).

15. Type rules. Here are the typing rules for the term forms dealing with record types:

\[
\begin{align*}
(\{}\text{Intro}\) & \quad \Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n \\
& \quad \Gamma \vdash \text{struct} \{ x_1 = M_1 ; \ldots ; x_n = M_n \} : \{ x_1 : T_1 ; \ldots ; x_n : T_n \} \\
\end{align*}
\]

where \( n > 0 \)

\[
\begin{align*}
(\{}\text{Elim}\) & \quad \Gamma \vdash M : \{ x_1 : T_1 ; \ldots ; x_n : T_n \} \\
& \quad \Gamma \vdash M \cdot x_i : T_i \\
\end{align*}
\]

for \( i = 1, \ldots, n \)

The \textit{struct} form evaluates to a record which has a record type. The field selection operation selects a component of a record.
Connections to programming languages

16. Java

The Java type system is based on that of the simply typed lambda calculus. An obvious notational difference is that type declarations are written in the form $T x$, where $T$ is a type and $x$ is an identifier, instead of the notation $x : T$. (For example, we write `int x` to declare an integer parameter.) This notation for declarations is inherited from Algol 60. It works well for simple types but it can become cumbersome and confusing when the type $T$ is complex.

Somewhat less obviously, the Java type system is a first-order type system. It does not support functions to be passed as parameters or returned as results of functions. (However, it is possible to pass and return objects which have functions as components.) Java also does not support tuples to be returned as results of functions.

Record types are similar to interfaces in Java. Higher-order record types, i.e., records whose fields are functions, are supported. However, the records are always constructed as instances of “classes”. They cannot be defined directly. See Handout 8, paragraph 9, for an example of how this is done.

In addition to these type-theoretic types, Java also uses class names as types. The meaning of this is subtle. Every class provides certain fields and methods as members. The types of these fields and methods constitute an interface, i.e., a record type. When a class name is used as a type, what is meant is this record type.

17. ML

The type system we have described here is very similar to that of ML. In addition to product types and record types, ML also provides sum types (also called “data types”). Here is an example of a binary tree data structure:

```
datatype tree = Empty of unit | Internal of (int * tree * tree)
```

This says that the type tree is a “sum” or “union” of two variants. One variant is that the tree is empty, which is indicated by the tag “Empty”. There is no further data associated with an empty tree. The other variant is that the tree has an internal node. In this case, the associated data consists of an integer that is stored in the internal node and two subtrees. Note that this type is also recursive, tree being defined in terms of itself. Sum types of this form are very convenient for defining recursive data structures and to define functions on recursive data structures.

Java does not have sum types. However, an extension of Java called Pizza adds sum types.