The University of Birmingham
School of Computer Science

Handout 4: 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.

Alonzo Church first developed a typed version of the lambda calculus which he called the "simple theory of types". He later generalised 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. Despite paradoxes, the untyped calculus was useful for some foundational purposes. In his Master's thesis, Turing proved that the untyped lambda calculus and (what we now call) "Turing machines" were equally expressive.

The term "simply typed lambda calculus" duplicates Alonzo Church's terminology of "simple types". However, the prefix "simply" has more significance. It represents the fact that this calculus does not have polymorphism. For practical programming, we need polymorphic functions as well. We will study **polymorphism** later in the course.

Our interest in types is pragmatic rather than foundational, i.e., we use it to understand how type systems work. But, the typed lambda calculus is not as expressive as Turing machines. It can only express *terminating'/* functional programs. However, by adding **general recursion** to the typed lambda calculus, we can again obtain a Turing-powerful programming language.

Context-free syntax

1. Syntax of types

We will define a typed lambda calculus with integers and boolean values. 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

$$T ::= \textbf{Int} \mid \textbf{Bool} \mid T_1 \to T_2$$

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, $0, 1, \ldots$ are integers.)

The obvious interpretation of these type expressions 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.

2. Bracketing convention We use the convention that the "$\rightarrow$" symbol "associates to the right," i.e.,

$$T_1 \rightarrow T_2 \rightarrow T_3 \quad \text{means} \quad T_1 \rightarrow (T_2 \rightarrow T_3)$$

Recall that, in set-theoretic interpretation of types, $T_1 \rightarrow T_2 \rightarrow T_3$ is isomorphic to $T_1 \times T_2 \rightarrow T_3$. So, $T_1 \rightarrow T_2 \rightarrow T_3$ is essentially a function with two arguments, but the arguments come one at a time rather than both together.

3. Types of constants We will use a whole range of "constants", i.e., primitive terms, in our calculus. Here are their types:

- The integer constants $\ldots, -2, -1, 0, 1, 2, \ldots$ are all of type **Int**.
- The boolean constants true and false are of type **Bool**.
- The arithmetic operations $+, -, *, /$ and *mod* are of type **Int** $\rightarrow$ **Int** $\rightarrow$ **Int**.
- The relational operations $=, \neq, <, \leq, >, \geq$ are of type **Int** $\rightarrow$ **Int** $\rightarrow$ **Bool**.
- We need two conditional branching functions: $\text{if}_{\textbf{Int}}$ of type **Bool** $\rightarrow$ **Int** $\rightarrow$ **Int** $\rightarrow$ **Int**, and $\text{if}_{\textbf{Bool}}$ of type **Bool** $\rightarrow$ **Bool** $\rightarrow$ **Bool** $\rightarrow$ **Bool**.

  (We need these subscripts **Int** and **Bool** for the "if" operator because we clearly have two separate functions of different types. In practice, we omit the subscripts in writing program terms.)

Note that the primitive operations of the programming language are regarded as "constants" in the simply typed lambda calculus. This is possible because it is a *higher-order* programming language.

4. Terms
The terms of the typed lambda calculus are given by the following inductive definition: *A term M can be one of:*

- *a constant c,*
- *a variable x,*
- *a lambda abstraction $\lambda x\colon T.\, M'$, where x is a variable, T a type and M' is another term, and*
- *a function application term $M_1\ M_2$ where M_1 and M_2 are terms.*

The defiition is more compactly expressed by the grammar:

$$M ::= c \mid x \mid \lambda x\colon T.\, M' \mid M_1\ M_2$$

The term $\lambda x\colon T.\, M'$ represents a function that takes an argument x of type T and returns the corresponding value of M'. (Recall the λ notation from Handout 1.) In this construction, x is a *local variable* of the lambda term. It is called the *formal parameter*. The term $M_1\ M_2$ represents function application, with M_1 representing the function and M_2 the argument it is applied to.

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.

5. Bracketing conventions To minimise the number of brackets we need in writing expressions, we use the following conventions:

1. Function application *associates to the left*, i.e., a term with multiple applications such as $M_1\ M_2\ M_3$ means $(M_1\ M_2)\ M_3$.

2. The scope of a lambda abstraction extends *as far to the right as possible*, e.g., $\lambda x\colon T.\, M_1\ M_2$ means $\lambda x\colon T.\, (M_1\ M_2)$, not $(\lambda x\colon T.\, M_1)\ M_2$.

6. Example As an example function in the typed lamdba calulus, consider the "and" function that takes two boolean values p and q and returns (p and q):

$$\lambda p\colon b.\, \lambda q\colon b.\, \text{if}_b\ p\ q\ \text{false}$$

(We abbreviate **Int** to i and **Bool** to b for readability.)

The fully bracketed form of the term is:

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b\ p)\ q)\ \text{false}))))$$

Note the two rules at play: function application associates to the left, and the scope of λ extends as far to the right as possible. Evidently, the bracketing conventions we have adopted reduce a lot of the clutter!

Type syntax

7. 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 mentioned above:

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b\ p)\ q)\ \text{false}))))$$

The type-annotated version of the term is:

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b^{b\to b\to b\to b}\ p^b)^{b\to b\to b}\ q^b)^{b\to b})\ \text{false}^b)^b)^{b\to b})^{b\to b\to b}$$

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 $\lambda x\colon T.\, M$, 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 $(\lambda x\colon T.\, M^U)$ is annotated with the type $T \to U$.

Application terms. An application term $(M^T\ N^S)$ is type correct only if T is a function type of the form $T_1 \to T_2$, 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:

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b^{b\to b\to b\to b}\ p^b)\ q^b)\ \text{false}^b))))$$

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

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b^{b\to b\to b\to b}\ p^b)^{b\to b\to b}\ q^b)^{b\to b})\ \text{false}^b)^b))$$

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.

$$(\lambda p\colon b.\, (\lambda q\colon b.\, ((((\text{if}_b^{b\to b\to b\to b}\ p^b)^{b\to b\to b}\ q^b)^{b\to b})\ 0^i)^?))$$

The function has the type $b \to b$ and the argument has the type i, and this violates the constraint $b = i$ required for the application to type-check.

8. Overloading

In the simple type system, we need two separate constants for the conditional $\text{if}_{\textbf{Int}}$ and $\text{if}_{\textbf{Bool}}$. 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,

$$\begin{array}{rcl}(\text{if } p\ q\ \text{false}) & \equiv & (\text{if}_{\textbf{Bool}}\ p\ q\ \text{false}) \\ (\text{if } p\ 0\ (f\ x)) & \equiv & (\text{if}_{\textbf{Int}}\ p\ 0\ (f\ x))\end{array}$$

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 $+$, $-$, $*$, $/$, as well as the comparison operators $=$, $\neq$, ..., for both the integer operations and real operations. That means, there is a $+_{\textbf{Int}}$ function and a separate $+_{\textbf{Real}}$ 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 $+_{\textbf{Int}}$. If both the arguments are of type **Real**, then it is of $+_{\textbf{Real}}$.

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.

9. Formal typing rules. The type checking rules described in paragraph 7 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.

10. 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 capitalised Greek letters Γ, Δ to denote typing contexts.

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

$$\begin{array}{rcl}x : \textbf{Int} \to \textbf{Int},\ y : \textbf{Int},\ z : \textbf{Bool} & \vdash & x\ y : \textbf{Int} \\ & \vdash & \lambda x\colon \textbf{Int}.\ x + 1 : \textbf{Int} \to \textbf{Int} \\ f : (\textbf{Int} \to \textbf{Int}) \to \textbf{Int} & \vdash & f\ (\lambda x\colon \textbf{Int}.\ x + 1)\end{array}$$

In the first example, we have been given the types $x : \textbf{Int} \to \textbf{Int}$ and $y : \textbf{Int}$. So, $x\ y$ is of type **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 $\lambda x\colon \textbf{Int}.\ x + 1$ is a closed term. So, we don't need any types of variables to type check it.

11. 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 ..., then $\Gamma \vdash M : T$.

We graphically depict such rules using the horizontal bar notation:

$$\frac{\Gamma_1 \vdash M_1 : T_1 \quad \Gamma_2 \vdash M_2 : T_2 \quad \ldots}{\Gamma \vdash M : T}$$

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. We have rules for constants, variables, abstraction and application terms.

(Const) $\dfrac{}{\Gamma \vdash c : T}$ if c has the type T in the language definition

(Variable) $\dfrac{}{\Gamma \vdash x : T}$ if $x : T$ is in Γ

(→Intro) $\dfrac{\Gamma,\, x : T \vdash M : T'}{\Gamma \vdash \lambda x\colon T.\, M : T \to T'}$

(→Elim) $\dfrac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1}{\Gamma \vdash M\ N : T_2}$

Several observations should be made regarding the rules:

- The rules *Const* and *Variable* have conditions stated on the side of the rule. They are often called "side conditions". These conditions must be satisfied for the rules to be applicable.
- The rules *→Intro* and *→Elim* are so called because they introduce and eliminate the $\to$ symbol in the types of terms. The rule *→Intro* has the type $T \to T'$ in the conclusion whereas only T and T' in the premise. Thus an $\to$ symbol has been introduced. The rule *→Elim* has $T_1 \to T_2$ in one of the premises, but this $\to$ symbol doesn't occur in the conclusion. This structure of Intro and 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 *→ Elim*, has Γ occurring twice and T_1 occurring twice. The rule *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 *→Intro* has a subtlety. To give a type to the term $\lambda x\colon T.\, M$ in the context of Γ, the rule requires that the term M should be given a type in the context $\Gamma,\, x : T$. If Γ 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 $\lambda x\colon \mathbf{Int}.\, \lambda x\colon \mathbf{Bool}.\, x$ because it would lead to a typing context of the form $x\colon \mathbf{Int},\, x\colon \mathbf{Bool}$ which is not well-formed. The solution is to rename one of the bound variables (using α equivalence), e.g., $\lambda x\colon \mathbf{Int}.\, \lambda x'\colon \mathbf{Bool}.\, x'$. This term can be typed using the rules.

  It is possible to reformulate the *→Intro* rule to eliminate this awkwardness with bound variables. However, the rule given above has the advantage of simplicity.
- The rule *Rec* specifies how to write a type-correct recursive definition. The expression $\mathbf{rec}\ z.\, M$ means the quantity z defined recursively using the defining term M. The variable z can occur inside M. That is what it means for it to be a "recursive" definition. For example, here is a term that defines the factorial function recursively:

$$\mathbf{rec}\ f\colon \mathbf{Int} \to \mathbf{Int}.\ \lambda n\colon \mathbf{Int}.\ \mathrm{if}_{\mathbf{Int}}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

12. 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 *and* function mentioned in the paragraph 7. We abbreviate the typing context $p : b,\ q : b$ as Γ. (We again shorten **Bool** to b for readability).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \text{if}_{\textbf{Bool}} : b \to b \to b \to b}\,\textit{Const} \quad \dfrac{}{\Gamma \vdash p : b}\,\textit{Variable}}{\Gamma \vdash \text{if}_{\textbf{Bool}}\ p : b \to b \to b}\,{\to}\textit{Elim} \quad \dfrac{}{\Gamma \vdash q : b}\,\textit{Variable}}{\Gamma \vdash \text{if}_{\textbf{Bool}}\ p\ q : b \to b}\,{\to}\textit{Elim} \quad \dfrac{}{\Gamma \vdash \text{false} : b}\,\textit{Const}}{\Gamma \vdash \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b}\,{\to}\textit{Elim}}{p : b \vdash \lambda q{:}\ b.\ \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b \to b}\,{\to}\textit{Intro}}{\vdash \lambda p{:}\ b.\ \lambda q{:}\ b.\ \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b \to b \to b}\,{\to}\textit{Intro}$$

The derivation can also be written in a linear form, by labelling 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.	$\Gamma \vdash \text{if}_{\textbf{Bool}} : b \to b \to b \to b$	using *Const*
2.	$\Gamma \vdash p : b$	using *Variable*
3.	$\Gamma \vdash \text{if}_{\textbf{Bool}}\ p : b \to b \to b$	from 1 and 2, using ${\to}\textit{Elim}$
4.	$\Gamma \vdash q : b$	using *Variable*
5.	$\Gamma \vdash \text{if}_{\textbf{Bool}}\ p\ q : b \to b$	from 3 and 4, using ${\to}\textit{Elim}$
6.	$\Gamma \vdash \text{false} : b$	using *Const*
7.	$\Gamma \vdash \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b$	from 5 and 6, using ${\to}\textit{Elim}$
8.	$p : b \vdash \lambda q{:}\ b.\ \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b \to b$	from 7, using ${\to}\textit{Intro}$
9.	$\vdash \lambda p{:}\ b.\ \lambda q{:}\ b.\ \text{if}_{\textbf{Bool}}\ p\ q\ \text{false} : b \to b \to b$	from 8, using ${\to}\textit{Intro}$

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 7, but we must ensure that the annotation is done in accordance with the type rules.

13. Typed lambda calculus is a framework The typed lambda calculus is not a programming language, but rather a *framework* for defining programming languages. By choosing appropriate basic types and constants, one can generate different programming languages from the same framework. We can also extend the framework in various ways, while sticking to the same formalism for the syntax and type rules for terms. Thus we can talk about "typed lambda calculus with recursion," "typed lambda calculus with product types," and so on. For simplicity of terminology, we will call of them just "typed lambda calculus." In the following, we will consider various extensions of the framework which are found in real-life languages like Haskell and others.

Recursion

14. Fixed point operator When we evaluate typed lambda calculus terms by simplification, all evaluations *terminate* in finite amount of time. That is because there is no mechanism for iteration.[1]

To allow recursive programs, we add a (generic) operation:

$$\textbf{fix}_T : (T \to T) \to T$$

so that $\textbf{fix}_T\ F$ represents the infinite expansion

$$F(F(F(\ldots)))$$

[1]The untyped lambda calculus has something called the **Y** combinator that can be used to get the effect of recursion. However the **Y** combinator cannot be expressed in the typed lambda calculus.

Such an infinite expansion is a "fixed point" of F, i.e., if you apply F to it, you get back the same value.

$$\mathbf{fix}_T\ F = F(\mathbf{fix}_T\ F)$$

Note: $F(F(F(\ldots))) = F\big(F(F(F(\ldots)))\big)$. You can't even see a difference!

You might wonder if there is always a unique fixed point of F in a type T. The answer is yes. There is only one possible polymorphic fixed point operator which works uniformly for all types T.

Any recursive program that we can write would be of the form:

$$f = F(f)$$

The value defined by such a recursive definition is simply $\mathbf{fix}_T F$. Here are some examples:

infinite list of 1s	
$ones = 1 : ones$	$ones = \mathbf{fix}_{[\mathbf{Int}]}\ (\lambda f\colon [\mathbf{Int}].\ 1 : f)$
list of all natural numbers	
$nats = 0 : (map\ (+1)\ nats)$	$nats = \mathbf{fix}_{[\mathbf{Int}]}\ (\lambda f\colon [\mathbf{Int}].\ 0 : (map\ (+1)\ f))$
the factorial function	
$fact = \lambda n\colon \mathbf{Int}.\ \mathrm{if}_{\mathbf{Int}}(=\ n\ 0)\ 1$ $(*\ n\ (fact\ (-\ n\ 1)))$	$fact = \mathbf{fix}_{\mathbf{Int}\to\mathbf{Int}}\ (\lambda f\colon \mathbf{Int}\to\mathbf{Int}.$ $\lambda n\colon \mathbf{Int}.\ \mathrm{if}_{\mathbf{Int}}(=\ n\ 0)\ 1$ $(*\ n\ (f\ (-\ n\ 1))))$

15. A recursion construct

Sometimes, we use a recursion "construct," i.e., a *term form*, instead of a primitive fixed point operator. We write it as **rec** $f\colon T.\ M$, and read it as "recursively defined f as M."

It has the type rule:

$$\textit{(Rec)}\quad \frac{\Gamma, f : T \vdash M : T}{\Gamma \vdash \mathbf{rec}\ f\colon T.\ M : T}$$

Note that this says f is a local variable in **rec** $f\colon T.\ M : T$. It can be used inside the body term M, but not outside.

The above examples can be rewritten using the **rec** construct:

infinite list of 1s	
$ones = 1 : ones$	$ones = \mathbf{rec}\ f\colon [\mathbf{Int}].\ 1 : f$
list of all natural numbers	
$nats = 0 : (map\ (+1)\ nats)$	$nats = \mathbf{rec}\ f\colon [\mathbf{Int}].\ 0 : (map\ (+1)\ f)$
the factorial function	
$fact = \lambda n\colon \mathbf{Int}.\ \mathrm{if}_{\mathbf{Int}}(=\ n\ 0)\ 1$ $(*\ n\ (fact\ (-\ n\ 1)))$	$fact = \mathbf{rec}\ f\colon \mathbf{Int}\to\mathbf{Int}.$ $\lambda n\colon \mathbf{Int}.\ \mathrm{if}_{\mathbf{Int}}(=\ n\ 0)\ 1$ $(*\ n\ (f\ (-\ n\ 1))))$

Note that **rec** and **fix** are equivalent: **rec** $f\colon T.\ M \equiv \mathbf{fix}_T\ (\lambda f\colon T.\ M)$. The reason for introducing the **rec** construct is that it is a litte easier to use; it doesn't involve an additional λ abstraction.

Product types

16. 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. The "nullary" product type (the case of $n = 0$) is denoted "**Unit**."

In ML, the $\times$ symbol is changed to $*$ so that it can be entered from ASCII keyboards.

In Haskell, the product type is written as if it were a tuple $(T_1, \ldots, T_n)$. The nullary product type (the case of $n = 0$) is denoted $()$.

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

$$(\times\textit{Intro)} \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash (M_1, \ldots, M_n) : T_1 \times \ldots \times T_n} \quad \text{where } n \geq 2$$

$$(\times\textit{Elim)} \qquad \frac{\Gamma \vdash M : T_1 \times \ldots \times T_n}{\Gamma \vdash \texttt{sel}[i]\, M : T_i} \qquad \text{for } i = 1, \ldots, n$$

$$(\textbf{Unit}\ \text{Intro}) \qquad \frac{}{\Gamma \vdash () : \textbf{Unit}}$$

The term $(M_1, \ldots, M_n)$ evaluates to n-tuples of values whose type is a product type. The term $\texttt{sel}[i]\, M$ selects the i'th component of a product-typed term, where i is an integer constant in the range $1, \ldots, n$. The case of the empty tuple has the type **Unit**. Note that there is no elimination rule for **Unit**, because an empty tuple has no data to be extracted.

18. Multiple-argument functions. In a language with product types, it is common to use a λ binder that binds a tuple of variables as part of a lambda abstraction. We refer to them as "multiple-argument functions." Here is its typing rule:

$$(\times\!\rightarrow\text{Intro}) \qquad \frac{\Gamma,\, x_1 : T_1,\, \ldots, x_n : T_n \vdash M : T}{\Gamma \vdash \lambda(x_1 \colon T_1, \ldots, x_n \colon T_n).\, M : T_1 \times \ldots \times T_n \rightarrow T} \quad \text{where } n > 0$$

$$(\textbf{Unit}\!\rightarrow\text{Intro}) \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda().\, M : \textbf{Unit} \rightarrow T}$$

Record types

19. Record types. We can introduce 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 $T_1 \times \ldots \times T_n$.

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

$$(\{\}\textit{Intro)} \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash \textbf{struct}\ \{x_1 = M_1; \ldots; x_n = M_n\} : \{x_1 \colon T_1; \ldots; x_n \colon T_n\}} \quad \text{where } n > 0$$

$$(\{\}\textit{Elim)} \qquad \frac{\Gamma \vdash M : \{x_1 \colon T_1; \ldots; x_n \colon T_n\}}{\Gamma \vdash M.x_i : T_i} \qquad \text{for } i = 1, \ldots, n$$

The **struct** form evaluates to a record which has a record type. The field selection operation selects a component of a record.

Sum types

21. Sum types Languages like ML and Haskell provide, in addition to product types and record types, a notion of *sum types* (also called "data types"). Here is an example of a binary tree data structure:

```
data Tree = Empty () | Node (Tree,  Int,  Tree)
```

This says that the type tree is a "sum" or "disjoint 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.

22. Typing rules. Consider a sum type of the form $c_1T_1 \mid c_2T_2$ where c_1 and c_2 are constructor symbols. Then to build a term of the sum type, we just apply the constructor to an argument of the appropriate type. To use a value of the sum type, we use a **case** expression.

$$(\mid \text{Intro}) \quad \frac{\Gamma \vdash M : T_1}{\Gamma \vdash c_1 M : (c_1T_1 \mid c_2T_2)} \qquad (\mid \text{Intro}) \quad \frac{\Gamma \vdash N : T_2}{\Gamma \vdash c_2 N : (c_1T_1 \mid c_2T_2)}$$

$$(\mid \text{Elim}) \quad \frac{\Gamma \vdash V : (c_1T_1 \mid c_2T_2) \quad \Gamma,\, x\colon T_1 \vdash M : T' \quad \Gamma,\, y\colon T_2 \vdash N : T'}{\Gamma \vdash (\textbf{case}\ V\ \textbf{of}\ c_1x \Rightarrow M \mid c_2y \Rightarrow N) : T'}$$

The **case** term as well as its type rule are more complex than all other term forms we have seen. This complexity is inherent in the notion of sum types. The **case** term first performs a case analysis on a value V of a sum type to see which variant it belongs to. If it is a c_1 variant, then it extracts a value x of type T_1; if it is a c_2 variant, it extracts a value y of type T_2. In both cases, it uses a term (M or N) to construct a value of an arbitrary type T'.

Connections to the Java type system

23. 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.

Our record types are similar to *interfaces* in Java. Higher-order record types, i.e., records whose fields are functions, are supported in Java. However, the records are always constructed as instances of "classes". They cannot be defined directly. The Handout on Java, to come later, gives examples 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.