**ADVANCED PROGRAMMING LANGUAGES**
Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

# The Simply Typed Lambda Calculus

In our previous lecture, we have introduced a simple static type system for arithmetic expressions. Now we construct a similar type system for the Lambda Calculus: in the interest of brevity, we ignore most of the operators on integers and booleans and focus on the operators from the language of expressions from our last class.

In addition, we introduce a new primitive operator for recursion:
**Syntax.** We start with the following extended syntax, which includes integers and booleans with their operators, conditionals and a primitive version of the $Y$ combinator.

$$\begin{array}{rcll} M, N, P & ::= & x, y, z & \text{variable} \\ & \mid & \lambda x@T.M & \text{abstraction} \\ & \mid & M\,N & \text{application} \\ & \mid & n, m, k \mid M + N & \text{integers} \\ & \mid & \mathsf{true} \mid \mathsf{false} \mid M = N & \text{booleans} \\ & \mid & \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P & \text{conditional} \\ & \mid & \mathsf{fix}@T.M & \text{recursion} \end{array}$$

The evaluation of the fix operator provides a direct recursion mechanism:

$$\frac{}{\mathsf{fix}@T.M \to M(\mathsf{fix}@T.M)}$$

As we will see, a primitive recursive operator is needed because the $Y$ combinator from the lambda calculus cannot be typed with the *simple* types we are going to discuss.

**Types** We start our analysis by introducing the definition of types:

$$\begin{array}{rcll} S, T & ::= & \mathsf{Nat} & \text{natural numbers} \\ & \mid & \mathsf{Bool} & \text{booleans} \\ & \mid & S \to T & \text{functions} \end{array}$$

**Typing judgements** The typing judgements, of the form $\Gamma \vdash M : T$, associate the type $T$ to the term $M$. The *typing context*, also known as *type environment* $\Gamma$ contains a set of type bindings to qualify the types of the free variables of $M$. While we are generally interested in assigning types to closed expressions, for which the simpler judgement form $M : T$ would suffice, the presence of lambda-abstractions requires us to work with type environments as well.

In order to assign a type to an abstraction $\lambda x.M$, we need to know what type of arguments to expect for the bound variable $x$. There are two possible responses: either we simply annotate the lambda abstraction with the intended type of its arguments, or we analyze the body of the abstraction to see how the argument is used and try to deduce, from this, what type it should have. We choose the first alternative and instead of just $\lambda x.M$, we write $\lambda x@T.M$ where the annotation on the bound variable tells us to assume that the argument will be of type $T$. In general, languages in which type annotations in terms are used to help guide the type-checker are called explicitly typed. Languages in which we ask the type-checker to infer or reconstruct this information are called implicitly typed.

Once we know the type of the argument for the abstraction, it is clear that the type of the function's result will be just the type of the body $M$, where the occurrences of $x$ in $M$are assumed to denote terms of the type assumed for $x$. Accordingly, to establish the relation $\lambda x@T_1.M : T_1 \to T_2$ we must be able to show that $M : T_2$ under the assumption that $x : T_1$, written $x : T_1 \vdash M : T_2$.

Since terms may contain nested $\lambda$-abstractions, we will need, in general, to talk about several such assumptions. This explains the format of the typing judgement $\Gamma \vdash M : T$.

Formally, a type environment $\Gamma$ is a sequence of variables and their types, with " , " as an operator to extend $\Gamma$ by adding a new binding on the right. The empty context is sometimes written $\oslash$, but usually we just omit it and write $\vdash M : T$, to state that *the (closed) term $M$ has type $T$ under the empty set of assumptions*.

Type environments may also be thought of as finite maps from variables to their types. Following this intuition, we write $\mathrm{dom}(\Gamma)$ for the set of variables bound by $\Gamma$.

## Typing Rules

We are ready to introduce the typing rules. In most type systems, each syntactic category of the language is associated with a corresponding form of type judgement. In particular, a common scenario for standard type systems includes at least two forms of judgement, for expressions and for statements. Given that the Lambda Calculus has just expressions, we only have the judgement form we have discussed, for expressions.

**Variables.** The typing rule for variables also follows immediately from this discussion: a variable has whatever type we are currently assuming it to have.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

**Abstraction**. The rule for typing abstractions has the general form:

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x@T_1.M : T_1 \to T_2} \quad \text{(T-Abs)}$$

where the premise adds one more assumption to those in the conclusion. To avoid confusion between the new binding for $x$ any bindings that may already appear in $\Gamma$, we require that the name $x$ be chosen so that it is distinct from the variables bound by $\Gamma$. Since our convention is that variables bound by $\lambda$-abstractions may be renamed whenever convenient, this condition can always be satisfied by renaming the bound variable if necessary.

**Application.** The typing rule for applications is dual to the rule for abstraction:

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\,N : T} \quad \text{(T-App)}$$

If $M$ evaluates to a function $S \to T$, (under the assumption that the values represented by its free variables have the types assumed for them in $\Gamma$), and if $N$valuates to a result in $S$, then the result of applying $M$ to $N$ will be a value of type $T$.

**Recursion**. The typing of recursive terms has the following general form:

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash \mathsf{fix}@T.M : T} \quad \text{(T-fix)}$$

Notice that $M$ is required to have a function type in which the domain and codomain types coincide: that is consistent with the intended behavior of the fixpoint operator and the type preservation property we expect of the type system, as $\mathsf{fix}.M$ reduces to $M(\mathsf{fix}.M)$.

**Constants and operators.** The system of inference rules is completed with the direct generalization of the corresponding typing rules for the expression language, adapted to accommodate the context $\Gamma$ in the judgements so as to account for the presence of variables within the expressions.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash n : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M + N : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : T \quad \Gamma \vdash N : T}{\Gamma \vdash M = N : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool} \quad \Gamma \vdash N : T \quad \Gamma \vdash P : T}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : T}$$

## Properties of Typing

We proceed as for the language of arithmetic expressions, with a basic lemmas needed in the proof of type safety. Most of these lemmas are similar to what we saw before - we just need to add contexts to the typing relation and add clauses to each proof for abstractions, applications, and variables. The only significant new result is a substitution lemma for the typing relation

**Lemma [Inversion of the Typing Relation]**

If $\Gamma \vdash x : T$, then $x : T \in \Gamma$
If $\Gamma \vdash \lambda x@S.M : T$ then $T = S \to U$ for some $U$ such that $\Gamma, x : S \vdash M : U$
If $\Gamma \vdash M\ N : T$ then there is some type $S$ such that $\Gamma \vdash M : S \to T$ and $\Gamma \vdash N : S$
If $\Gamma \vdash \mathsf{true} : T$, then $T = \mathsf{Bool}$.

If $\Gamma \vdash \mathsf{false} : T$ then $T = \mathsf{Bool}$.
If $\Gamma \vdash \textbf{if } M' \textbf{ then } N \textbf{ else } P : T$ then $\Gamma \vdash M : \mathsf{Bool},\ N : T$ and $\Gamma \vdash P : T$.
If $\Gamma \vdash n : T$ then $T = \mathsf{Nat}$.
If $\Gamma \vdash M + N : T$ then $T = \mathsf{Nat}$ and $\Gamma \vdash M : \mathsf{Nat}$ and $\Gamma \vdash N : \mathsf{Nat}$.
If $\Gamma \vdash M = N : T$ then $T = \mathsf{Bool}$ and $\Gamma \vdash M : S$ and $\Gamma \vdash N : S$ for some type $S$.
*Proof*: Immediate from the definition of the typing relation.

**Theorem [Uniqueness of Types]**
For each term $M$ and type environment $\Gamma$ there exists at most one type $T$ such that $\Gamma \vdash M : T$.
Moreover, there is just one derivation of the typing judgement from the inference rules.

**Theorem [Canonical Forms]**
If $v$ is a value and $\vdash v : \mathsf{Bool}$ then $v$ is either $\mathsf{true}$ or $\mathsf{false}$;
If $v$ is a value and $\vdash v$: $\mathsf{Nat}$ then $v$ is a numeral $n$;
If $v$ is a value and $\vdash v : S \to T$ then, then $v$ is a closed lambda abstraction $\lambda x @ S.M$.

**Theorem [Progress]**
Suppose $M$ is a closed, well-typed term, that is $\vdash M : T$ for some $T$. Then either $M$ is a value or else there exists $N$ such that $M \to N$.

*Proof.* By rule induction on the typing judgement $\vdash M : T$. We proceed by cases on all the rules that may have derived the judgement in question, and assume (by rule induction) that the theorem holds for the judgements in the premises of the rule.

The cases of the axioms for the boolean and integer constants are immediate. since in all these cases $M$ is a value.

For the other cases, we take the rule for conditionals as a representative.
In this case, $M$ is of the form $\textbf{if } M' \textbf{ then } N \textbf{ else } P : T$, derived from $\vdash M' : \mathsf{Bool}$. By the induction hypothesis we know that either $M'$ is a value or else there is some term $M''$ such that $M' \to M''$. In the first case, by the canonical forms theorem $M'$ is either $\mathsf{true}$ or $\mathsf{false}$: hence either $M \to N$ or $M \to P$. If instead $M' \to M''$ then also $M \to \textbf{if } M'' \textbf{ then } N \textbf{ else } P : T$, as desired.

---

To prove the preservation theorem, we first state a couple of "structural lemmas" that will allow us us to perform some useful manipulations of typing derivations. The first lemma tells us that we may permute the elements of a context, as convenient, without changing the set of typing statements that can be derived under it. The second says that we can add bindings to the context, again without affecting the validity of the judgements being proved.

**Lemma [Permutation]**
If $\Gamma \vdash M : T$ and $\Gamma'$ is a permutation of $\Gamma$, then also $\Gamma' \vdash M : T$.

**Lemma [Weakening]**
If $\Gamma \vdash M : T$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, x : S \vdash M : T$.

Now we can prove a crucial property of the typing relation: that the typing relations is preserved when variables are substituted with terms of appropriate types.

**Lemma [Preservation of Types Under Substitution]**
If $\Gamma, x : S \vdash M : T$ and $\Gamma \vdash N : S$, then $\Gamma \vdash [N/x]M : T$.

*Proof.* By rule induction on the judgement $\Gamma, x : S \vdash M : T$, and a case analysis on the final rule used in the derivation. The most interesting cases are the ones for variables and abstractions.

*Case T-Var*. In this case $M = z$ with $z : T \in (\Gamma, x : S)$. There are two sub-cases to consider, depending on whether $z$ is $x$ or another variable. If $z = x$, then $[N/x]z = N$. Then the desired result $\Gamma \vdash N : S$ is among the assumptions of the lemma. Otherwise, [N /x]z = z, and the dsired result is immediate.

*Case T-Abs*. The judgement in question is $\Gamma, x : S \vdash \lambda y@T_1.M' : T_1 \to T_2$, derived from $\Gamma, x : S, y : T_1 \vdash M' : T_2$. By $\alpha$-conversion, we may assume that $x \neq y$ and $y \notin \mathrm{fv}(N)$. Now:

- from $\Gamma, x : S, y : T_1 \vdash M' : T_2$, using permutation we obtain $\Gamma, y : T_1, x : S \vdash M' : T_2$.
- from $\Gamma \vdash N : S$using weakening we have $\Gamma, y : T_1 \vdash N : S$.

By the induction hypothesis, we then have $\Gamma, y : T_1 \vdash [N/x]M' : T_2$, and then by an application of (T-Abs) we may derive $\Gamma \vdash \lambda y@T_1[N/x]M' : T_1 \to T_2$. But this is precisely the result we are looking for, since, by the definition of substitution, $[N/x]M = \lambda y@T_1[N/x]M'$ given that $y \notin \mathrm{fv}(N)$

---

**Theorem [Preservation]** If $\Gamma \vdash M : T$ and $M \to N$ then $\Gamma \vdash N : T$.
*Proof*. By rule induction on the judgement $\Gamma \vdash M : T$. We show the case of application as a representative for the other cases.

*Case T-App*. $M$ is the application term $M'\,N'$ and the judgement in question is of the form $\Gamma \vdash M'\,N' : T$, derived from the two premises $\Gamma \vdash M' : T' \to T$ and $\Gamma \vdash N' : T'$. From the hypothesis $M \to N$ we isolate three possible subcases.

- $M \to N$ because $M' \to M''$ and $N \equiv M''\,N'$.
  From $\Gamma \vdash M' : T' \to T$ and $M' \to M''$, by the induction hypothesis we know that $\Gamma \vdash M'' : T' \to T$. From this, and from $\Gamma \vdash N' : T'$ by an applicaton of (T-App) we derive $\Gamma \vdash M''\,N' : T$ as desired.
- $M \to N$ because $N' \to N''$ and $N \equiv M'\,N''$.
  From $\Gamma \vdash N' : T'$ and $N' \to N''$, by the induction hypothesis we derive $\Gamma \vdash N'' : T'$. From this and from $\Gamma \vdash M' : T' \to T$ and $\Gamma \vdash N' : T'$. we derive the desired judgement $\Gamma \vdash M'\,N'' : T$ by an application of (T-app) .
- $M \to N$ because $M' = \lambda x@T'.M''$ and $N \equiv [N'/x]M''$.
  Then $\Gamma \vdash M' : T' \to T$ is the conclusion of the (T-Abs) rule with premise $\Gamma, x : T' \vdash M'' : T$. From this judgement, and from $\Gamma \vdash N' : T'$, by the substitution lemma we have $\Gamma \vdash [N'/x]M'' : T$, which is just what we wanted to prove.

## From theory to practice: strong and static typing

---

Just as in the Typed Lambda Calculus, type systems generally accomplish their task by imposing a static type structure on programs which assigns types to all constants, operators, variables, and function symbols defined by the program. Then, the typing rules verify the consistency between definition and use, again following the same rationale we have discussed for the Typed Lambda Calculus.

Static typing is a useful property, but the requirement that all expressions are assigned a type at compile time turns out to be too restrictive in certain situations. Indeed, traditional statically typed languages have long suffered of a significant loss of flexibility, as they excluded programming idioms which, although sound, where incompatible with the early binding of program expressions to a specific type. In modern systems, such as in Scala's type systems, most of these limitations have been overcome thanks to the powerful techniques, developed by the research community, adopted in such systems.
For the remaining cases for which static typing is still infeasible, the static checks may be replaced by the weaker requirement that all expressions are guaranteed to be type-consistent by introducing some run-time type checking.

Languages in which all expressions are type-consistent are called strongly typed languages. If a language is strongly typed its compiler can guarantee that the programs it accepts will execute without type errors.

In general, we should strive for strong typing, and adopt static typing whenever possible, for at least two good reasons:

1. *Robustness*: strong, static typing typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent, hence that no type-related error may arise at run time. At the same type, strong static typing enforces a programming discipline on the programmer that makes programs more structured and easier to read.
2. *Efficiency*: dynamic checks require run-time type information on values and therefore take more memory; in addition to that, dynamic checks take CPU time and make the compiled code slower;

Having said that, we must remark that types do not always provide all the necessary information to ensure the absence of *all* errors at run time, most often because some of the properties to check are undecidable. Examples of runtime errors escaping strong static typing:

1. a division between two integers (or doubles) may cause a run-time error, because the denominator may be zero;
2. a method invocation `o.m` with `o : T` may fail because `o` is the null reference, even thought the type `T` provides an `m` method.
3. an array access `as[e]` with `as : Array[T]` and `e : Int` may fail at runtime, because $e$may evaluate to an index that does not correspond to any element of `as` (the typical “array index out of bounds” exception).

Of course, in principle, nothing prevents us from refining the notion of type to achieve more accurate static checks, and thus obtain more robust notions of type safety. For example, we could define types as follows:

```
Int+ = positive integers
Int- = negative integers
Int* = nonzero integers
Int = integer
```

and establish the following type for the integer division operator:

```
/ : Int x Int* => Int
```

The problem, then, is to determine whether a given variable may be assigned one of the refined types. For example, given `x: Int+` and `y:Int-` , what can we say about the type of the expression `x + y` ? The only correct type to give to `x + y` would be `Int` , so that we could not type-check a division of the form `z / (x + y)` .

Similarly, we could say that field/method access operations on an object should check not only that the object type is a class defining that field/method, but also that the object is not null. Again, the problem is to judge (statically!) whether a given reference is null, which is not feasible in general. Finally, we could make array types more precise by including the array size in the type and then check the array-out-of-bounds error statically, but again that would not be effective in practice.

Indeed, modern programming languages take a more pragmatic approach to typing and carry out these checks dynamically, by automatically generating code that implements them when a given operation is performed.

There are, however, interesting research proposals to enhance new typing techniques to provide additional power to the static checks.

## Refinement Types

A rather interesting example is based on *Refinement Types* a technique that enables the specification of complex invariants by extending the base type system with refinement predicates drawn from decidable logics. For example,

$$\begin{aligned}&\mathsf{type\ Nat} = \{\mathsf{v : Int} \mid 0 \leq \mathsf{v}\}\\&\mathsf{type\ Pos} = \{\mathsf{v : Int} \mid 0 < \mathsf{v}\}\end{aligned}$$

are refinements of the basic type $\mathsf{Int}$ that capture the types `Int*` and `Int+` introduced above by means of a logical predicate stating that the values $\mathsf{v}$ being described must be non-negative and positive respectively. Based on such definitions, we can specify contracts of functions by refining function types. For example, the following contract

$$\mathsf{div} :: (\mathsf{n : Nat}) \to (\mathsf{d : Pos}) \to \{\mathsf{v : Nat} | \mathsf{v} \leq \mathsf{n}\}$$

states that $\mathsf{div}$ requires a non-negative dividend $\mathsf{n}$ and a positive divisor $\mathsf{d}$, and ensures that the result is less than the dividend. If a program (refinement) type checks, we can be sure that div will never throw a divide-by-zero exception.

We will not discuss refinement types further. For more information, please check the following papers:

- An Introduction to Liquid Haskell by Ricardo Peña
- LiquidHaskell: Experience with Refinement Types in the Real World by Niki Vazou, Eric L. Seidel, Ranjit Jhala (you'll have to access the link with your Ca' Foscari account)

and / or the following online tutorial:

- An Introduction to LiquidHaskell by Ranjit Jhala, Eric Seidel, and Niki Vazou.

## CREDITS

The material on the type system properties is adapted from Chapter 9 of Prof. Benjamin's Pierce's book on Types and Programming Languages. MIT Press. 2001.