Type Systems for Lambda Calculus

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Let's speak about type systems. What is a type?

Is **double** a type?

It can be cost, a position, a speed.

It seems there is no real answer.

Type Checking in *C*

In *C*, the compiler uses type checking in order to decide what machine instruction should be selected (and if there is one at all.)

In *C*, type checking is completely bottom up. The compiler starts at the leaves of the expression tree. The leaves contain variables and constants. Variables have a declared type which can be looked up. Constants have a default type.

After that, the compiler travels upwards in the tree, assigns a type to each node, inserts conversions where necessary, and makes choices for overloaded operators.

```
double d = 3.0;
d = d + 1;
   // assign_double( d, sum_double( load_double(d),
                                    int2double(1))).
```

Two Approaches to Type Checking

Church-style: Church style typing is similar to the C^{++} approach. All identifiers are introduced with a type.

λ is extended to include a type, as for example in $\lambda x{:}N\ x + x$.

Once every identifier has a type, the complete term can be type checked bottom up.

Curry-style: Built-in operators have a fixed type.

λ has no types attached them.

Types are attached to a term afterwards, either by hand (according to certain typing rules), or by an algorithm.

Haskell uses Curry-style typing.

Types for λ-Calculus

We assume that every inductive type definition defines a type. This means that **nat**, **bool**, **unit**, **empty** are types.

Some of the inductive types construct a new type from another type, for these we will use type functions:
list(**nat**), **optional**(**nat**), **prod**(**nat**, **bool**), **union**(**bool**, **nat**).

We will not use Currying for type functions.

The precise definition is on the next slide.

Types for λ-Calculus (2)

- We assume an infinite set of type variables, which we will write with Greek letters α, β, γ, etc.
- We assume that every inductive type definition results in a type function $T(T_1, \ldots, T_n)$, where $T_1, \ldots, T_n$ are the subtypes used in the definition of the new type T. It is possible that $n = 0$.
- If T_1 and T_2 are types, then $T_1 \to T_2$ is also a type.
- If T is a type, α is a type variable, then $\Pi\alpha\, T$ is also a type.

The meaning of $T_1 \to T_2$ is 'function from T_1 to T_2.'

The meaning of $\Pi\alpha\, T$ is 'for every type α, the type T.'

Types containing Π are called polymorphic.

Types for λ-Calculus (3)

We assume that $\to$ is right associative. This means that $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$.

If $\to$ would be left associative, it would be harder to write functions of multiple arguments.

For example, the terms $+$ and $-$ have type $\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$.

The term $\lambda m\, \lambda n\, (-\ n\ m)$ also has type $\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$.

The term $\lambda x\ x$ has type $\Pi\alpha\, (\alpha \to \alpha)$.

Types for λ-Calculus (4)

restriction: We will assume that Π only occurs on the top level (on the outside) of a type.

This implies that types of form $(\ \Pi\alpha\,(\alpha \to \alpha)\ ) \to \mathbf{nat}$ are forbidden.

The reason is that the type checking algorithm that we will introduce shortly, will not be able to find such types, unless one declares types in λ-s:

Consider

$$\lambda \mathrm{id}\ (\mathrm{id}\ \mathrm{succ}\ (\mathrm{id}\ 0)\ ).$$

If one declares id, one gets:

$$\lambda \mathrm{id}{:}\,\Pi\alpha(\alpha \to \alpha) \quad (\mathrm{id}\ \mathrm{succ}\ (\mathrm{id}\ 0)\ ).$$

Examples of Inductively Defined Types

0: **nat**

succ: **nat** $\to$ **nat**

$\mathbf{rec}_N$: $\Pi\beta\ \beta \to (\mathbf{nat} \to \beta \to \beta) \to \mathbf{nat} \to \beta$

f: **bool**

t: **bool**

$\mathbf{rec}_\mathbf{B}$: $\Pi\beta\ \beta \to \beta \to \mathbf{bool} \to \beta$

Examples of Inductively Defined Types (2)

List:

$$\mathbf{nil}: \Pi\alpha\ \mathbf{list}(\alpha)$$

$$\mathbf{cons}: \Pi\alpha\ \alpha \to \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$$

$$\mathbf{rec}_L: \Pi\alpha\beta\ \beta \to (\alpha \to \mathbf{list}(\alpha) \to \beta \to \beta) \to \mathbf{list}(\alpha) \to \beta$$

Pair:

$$\mathbf{pair}: \Pi\alpha_1\alpha_2\ \alpha_1 \to \alpha_2 \to \mathbf{prod}(\alpha_1, \alpha_2)$$

$$\mathbf{rec}_P: \Pi\alpha_1\alpha_2\beta\ (\alpha_1 \to \alpha_2 \to \beta) \to \mathbf{prod}(\alpha_1, \alpha_2) \to \beta$$

Examples of Inductively Defined Types (3)

$\mathbf{union1}\colon \Pi\alpha_1\alpha_2 \;\; \alpha_1 \to \mathbf{union}(\alpha_1, \alpha_2)$

$\mathbf{union2}\colon \Pi\alpha_1\alpha_2 \;\; \alpha_2 \to \mathbf{union}(\alpha_1, \alpha_2)$

$\mathbf{rec}_U\colon \Pi\alpha_1\alpha_2\beta \; (\alpha_1 \to \beta) \to (\alpha_2 \to \beta) \to \mathbf{union}(\alpha_1, \alpha_2) \to \beta$

$\mathbf{just}\colon \Pi\alpha \;\; \alpha \to \mathbf{optional}(\alpha)$

$\mathbf{none}\colon \Pi\alpha \; \mathbf{optional}(\alpha)$

$\mathbf{rec}_N\colon \Pi\alpha\beta \; (\alpha \to \beta) \to \beta \to \mathbf{optional}(\alpha) \to \beta$

Examples of Types of Functions

$+$: $\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$

$\mathbf{eq}$: $\mathbf{nat} \to \mathbf{nat} \to \mathbf{bool}$

$\mathbf{append}$: $\Pi\alpha\ \mathbf{list}(\alpha) \to \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$

$\mathbf{reverse}$: $\Pi\alpha\ \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$

$\mathbf{length}$: $\Pi\alpha\ \mathbf{list}(\alpha) \to \mathbf{nat}$

Assigning a Type to an Expression

A context is a sequence of variable declarations of form

$$v_1{:}V_1, \ldots, v_n{:}V_n.$$

Each v_i is a variable, and V_i is the type with which v_i was declared or defined.

Contexts are always used in a stack-like fashion. Declarations are added at the end, and removed from the end.

The context stores all defined/declared identifiers that are currently known.

Unification

If t has some type $\Pi\alpha\ T$, then one can substitute any type t for α.

For example, **nil** has type $\Pi\alpha$ **list**(α).

This means that it also has types **list**(**nat**) or **list**(**list**(β)).

We want to find substitutions on types automatically, because otherwise the user has to insert them. We want

$$(\mathbf{cons}\ (\mathbf{succ}\ 0)\ (\mathbf{cons}\ 0\ \mathbf{nil}))$$

instead of

$$(\mathbf{cons_{nat}}\ (\mathbf{succ}\ 0)\ (\mathbf{cons_{nat}}\ 0\ \mathbf{nil_{nat}}))$$

In order to find type substitutions automatically, we use unification: Decide which substitution should be used by comparing two types.

Unification (2)

When two types (containing type variables) must be equal, they are compared and the necessary substitution is extracted. This process is called unification. I explain later how it works. Here are some examples:

α	$=$	β	$\alpha := \beta$
α	$=$	$\beta_1 \to \beta_2$	$\alpha := (\beta_1 \to \beta_2)$
$\alpha_1 \to \mathbf{list}(\alpha_2)$	$=$	$\beta_1 \to \beta_2$	$\alpha_1 := \beta_1,\ \beta_2 := \mathbf{list}(\alpha_2)$
$\alpha \to \beta$	$=$	$\mathbf{nat}$	not possible
α	$=$	$(\alpha \to \beta)$	not possible

Type Checking Rules

Let Γ be a context, we write $\Gamma \vdash t{:}T$, when term t has type T in context Γ.

Types in Γ can occur type variables, and special type constants $c_1, \ldots,$

Type constants can occur only in temporary assumptions.

Type Checking Rules (2)

VAR: If Γ contains an assumption of form $x{:}X$, then $\Gamma \vdash x{:}X$.

LAMBDA: If $\Gamma, x{:}X \vdash t{:}T$, then let Θ be a 'substitution' that replaces the type constants in X by distinct type variables that do not occur anywhere else. Then

$$\Gamma \vdash (\lambda x{:}X\ t){:}(X\Theta \to T\Theta).$$

APPLY: If

$$\Gamma \vdash f{:}\ \Pi\overline{\alpha}\ (T \to U), \qquad \Gamma \vdash t{:}\ \Pi\overline{\alpha}'\ T',$$

and the types T, T' can be unified, let Θ be the resulting substitution. Then

$$\Gamma \vdash (f\ t){:}\Pi\overline{\beta}\ U\Theta.$$

Here, $\overline{\beta}$ are the free variables of $U\Theta$.

Algorithm for Type Checking

We are given a context Γ and a term t that needs to be type checked. We recursively define the algorithm TYPE(Γ, t), that tries to find a type T for t in context Γ.

- If t is a variable, then look up t in Γ. If Γ contains $t{:}T$, then return T. Otherwise, fail.
- If t has form $\lambda x\ t'$, we need to guess a type for x. Assume it is X. Note that X contain type variables. It can contain type constants.

  Recursively call TYPE$(\Gamma,\ x{:}X,\ \ t')$. If this results in a type T, then apply the LAMBDA rule.
- If t has form $f\ t'$, then recursively call TYPE(Γ, f) and TYPE(Γ, t').

  If both recursive calls find a type, try to apply rule **APPLY**.

Problems with Algorithm

The algorithm on the previous slide has one obvious problem:

In the case of $\lambda x\ t'$, it has to guess a type for x, and add it to the context.

Making a wrong guess may cause the algorithm to fail, where it could have succeeded.

For example in λx (**append** $x\ x$), one can choose

x:**nat** (will fail),

x:**list**(**nat**) (possible), or

x:**list**(c) (best choice, because c will become a type variable).

A less obvious problem is the strange use of type constants in the LAMBDA case. If there is something strange in an algorithm, this is always caused by a thinking error. Strangeness is more dangerous than incorrectness, because it is harder to catch.

Type Clauses

The root cause of the problem is that intermediate assumptions (introduced when checking λ) must be treated different from permanent assumptions.

We introduce type clauses of form

$$\Pi\overline{\alpha}\ \ x_1{:}X_1, \ldots, x_n{:}X_n \Rightarrow t{:}T.$$

The $x_1, \ldots, x_n$ are the free variables of term t that have no permanent declaration.

The order of the $x_1, \ldots, x_n$ does not matter.

There are no repeated x_i in a type clause.

Since the sequence $\overline{\alpha}$ is always the sequence of type variables in the type clause, we omit $\Pi\overline{\alpha}$.

If T contains type variables not occurring in $x_1, \ldots, x_n$ these must be quantified in T, i.e. T has form $\Pi\overline{\beta}\ T'$.

Hindley/Milner Typing Algorithm

We create type clauses for all subterms of the term that we want to check. The rules are as follows:

VAR: If v is a variable declared with type T, then one can create type clause $\Rightarrow v{:}T$.

If v is a variable without declaration, then create type clause $v{:}\alpha \Rightarrow v{:}\alpha$.

LAMBDA: If we have a type clause

$$x_1{:}X_1, \ldots, x_n{:}X_n,\ y{:}Y \Rightarrow t{:}T,$$

then we can construct

$$x_1{:}X_1, \ldots, x_n{:}X_n \Rightarrow (\lambda y\ t){:}(Y \to T).$$

If no $y{:}Y$ is present, we can add $y{:}\alpha$ before applying the rule, using a new type variable α.

Hindley/Milner Typing Algorithm (2)

APPLY: Assume we have type clauses

$$x_1{:}X_1, \ldots, x_n{:}X_n \Rightarrow f{:}F,$$

and

$$y_1{:}Y_1, \ldots, y_m{:}Y_m \Rightarrow t{:}T'.$$

If F is a single type variable α, then substitute $\alpha := (\alpha_1 \to \alpha_2)$, using two new type variables α_1 and α_2.

If F does not have form $T \to U$ now, then fail.

If F has form $T \to U$, then try to unify T with T'. Let Θ be the resulting substitution.

Construct the clause

$$x_1{:}X_1\Theta, \ldots, x_n{:}X_n\Theta,\ y_1{:}Y_1\Theta, \ldots, y_m{:}Y_m\Theta \Rightarrow (f\ t){:}U\Theta.$$

If there are repeated variables left of $\Rightarrow$, they must be merged.

Hindley/Milner Typing Algorithm (3)

MERGE: If type clause

$$x_1{:}X_1, \ldots, x_n{:}X_n \Rightarrow t{:}T,$$

contains a repeated variable, then assume that $x_1 = x_2$. This is fine, because order does not matter.

Try to unify X_1 with X_2. If no unifier exists, then fail, otherwise let Θ be the resulting substitution.

Replace the clause by

$$x_1{:}X_1\Theta,\ x_3{:}X_3\Theta, \ldots, x_n{:}X_n\Theta \Rightarrow t{:}T\Theta.$$

Example of the HM-Algorithm

We want to determine the type of

$$\lambda x\ (\mathbf{succ}\ (\mathbf{succ}\ x)).$$

The subterms are x, $\mathbf{succ}$, $(\mathbf{succ}\ x)$, $(\mathbf{succ}\ (\mathbf{succ}\ x))$, and $\lambda x\ (\mathbf{succ}\ (\mathbf{succ}\ x))$ itself.

- x has no permanent declaration, so we create the type clause

$$x{:}\,\alpha \Rightarrow x{:}\,\alpha.$$

- **succ** is declared with type $\mathbf{nat} \to \mathbf{nat}$, so we create the type clause

$$\Rightarrow \mathbf{succ}{:}\,(\mathbf{nat} \to \mathbf{nat}).$$

- Now we use **APPLY**. We need to unify **nat** with α, which results in $\Theta = (\alpha := \mathbf{nat})$. The resulting type clause is

$$x{:}\,\mathbf{nat} \Rightarrow (\mathbf{succ}\ 0){:}\,\mathbf{nat}.$$

- We use **APPLY** one more time, this time with empty substitution $\Theta = (\ )$. The result is

$$x{:}\,\mathbf{nat} \Rightarrow (\mathbf{succ}\ (\mathbf{succ}\ 0)){:}\,\mathbf{nat}.$$

- Now we can apply **LAMBDA**, and the result is

$$\Rightarrow \lambda x\ (\mathbf{succ}\ (\mathbf{succ}\ 0)){:}\,(\mathbf{nat} \rightarrow \mathbf{nat}).$$

Another Example of the HM-Algorithm

We want to determine the type of

$$\lambda x\ (\mathbf{cons}\ 0\ x).$$

The subterms are **cons**, 0, x, and $\lambda x\ (\mathbf{cons}\ 0\ x)$ itself.

- **cons** is declared with type $\Pi\alpha\ \alpha \to \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$, so we create the type clause

  $$\Rightarrow \mathbf{cons}{:}\,\alpha \to \mathbf{list}(\alpha) \to \mathbf{list}(\alpha).$$

  We don't write the Π in the type clause, because variables that occur right of $\Rightarrow$, but not left, are implicitly quantified.

- 0 has permanent type **nat**, so we create

  $$\Rightarrow 0{:}\,\mathbf{nat}.$$

- Now we use **APPLY** with substitution $\Theta = (\alpha := \mathbf{nat})$. The result is

$$\Rightarrow (\mathbf{cons}\ 0){:}\,\mathbf{list}(\mathbf{nat}) \rightarrow \mathbf{list}(\mathbf{nat}).$$

- x has no permanent declaration, so we create the type clause

$$x{:}\,\alpha \Rightarrow x{:}\,\alpha.$$

- We use **APPLY** with substitution $\Theta = (\alpha := \mathbf{list}(\mathbf{nat})\ )$. The result is

$$x{:}\,\mathbf{list}(\mathbf{nat}) \Rightarrow (\mathbf{cons}\ 0\ x){:}\,\mathbf{list}(\mathbf{nat}).$$

- One final application of **LAMBDA** results in

$$\Rightarrow \lambda x\ (\mathbf{cons}\ 0\ x){:}\,\mathbf{list}(\mathbf{nat}) \rightarrow \mathbf{list}(\mathbf{nat}).$$

A third Example of the HM-Algorithm

We want to determine the type of

$$\lambda x\ (\,(\lambda f\ f)\ x).$$

The subterms are $f, x,\ (\lambda f\ f),\ (\,(\lambda f\ f)\ x)$, and $\lambda x\ (\,(\lambda f\ f)\ x)$ itself.

- f has no declaration, so we create

  $$f{:}\alpha \Rightarrow f{:}\alpha.$$

- In order to obtain the type of $(\lambda f\ f)$, we apply **LAMBDA**, which results in

  $$\Rightarrow (\lambda f\ f){:}(\alpha \to \alpha).$$

  Since α does not occur left of $\Rightarrow$, we assume that $\alpha \to \alpha$ implicitly means $\Pi\alpha\ (\alpha \to \alpha)$.

- x has no declaration, so we create

$$x{:}\beta \Rightarrow x{:}\beta.$$

- In order to obtain the type of $((\lambda f\ f)\ x)$, we use **APPLY**. Type variable α can be unified with β, which results in $\Theta = (\alpha := \beta)$. Using Θ, we obtain

$$x{:}\beta \Rightarrow ((\lambda f\ f)\ x){:}\beta.$$

- One final application of **LAMBDA** results in

$$\Rightarrow \lambda x\ ((\lambda f\ f)\ x){:}(\beta \to \beta),$$

where again β is implicitly quantified as $\Pi\beta\ (\beta \to \beta)$.

An Example involving **MERGE**

Assume that f is declared with type $\mathbf{nat} \to \mathbf{bool} \to \mathbf{nat}$.

We try find a type for $\lambda x\ (f\ x\ x)$. The subterms are $f, x,\ (f\ x\ x)$, and $\lambda x\ (f\ x\ x)$.

- The type clause for f is $\Rightarrow f{:}(\mathbf{nat} \to \mathbf{bool} \to \mathbf{nat})$, because that's how we declared it.
- Variable x has no declaration, so its type clause is $x{:}\,\alpha \Rightarrow x{:}\,\alpha$.
- A first application of **APPL** results in

$$x{:}\,\mathbf{nat} \Rightarrow (f\ x){:}(\mathbf{bool} \to \mathbf{nat}).$$

- The second application of **APPL** results in

$$x{:}\,\mathbf{nat},\ x{:}\,\mathbf{bool} \Rightarrow (f\ x\ x){:}\,\mathbf{nat}.$$

- We need to apply **MERGE** but the types **nat** and **bool** cannot be unified. It follows that $\lambda x\ (f\ x\ x)$ has no type.

Determining the Type of a New Identifier

The Hindley/Milner can also be used for determining the type of a new identifier, given a few terms that contain the new identifier.

For example:

$$\begin{array}{lcl} (\mathbf{rec}_N\ f_0\ f_s\ 0) & \Rightarrow & f_0, \\ (\mathbf{rec}_N\ f_0\ f_s\ (\mathbf{succ}\ n)) & \Rightarrow & (f_s\ n\ (\mathbf{rec}_N\ f_0\ f_s\ n)). \end{array}$$

Just run the algorithm, until all type clauses have form

$$\mathrm{rec}_N{:}T, x_1{:}X_1 \cdots x_n{:}X_n \Rightarrow \cdots$$

The type of $\mathbf{rec}_N$ can be obtained from these final clauses by unifying their types.

Let

The HM algorithm cannot assign a type to

$$(\lambda f\ (f\ \mathbf{succ}\ (f\ 0)))\ \ (\lambda x\ x).$$

This is not a weakness of the HM-algorithm, but of the type system, which does not allow

$$(\,\Pi\alpha\ (\alpha \to \alpha)\,) \to \mathbf{nat}.$$

This problem can be (somewhat) solved by using **let**:

Instead write

$$\mathbf{let}\ f := \lambda x\ x\ \mathbf{in}\ (f\ \mathbf{succ}\ (f\ 0)).$$

The trick is that we used ordered evaluation, instead of parallel. First obtain the type of f, and after that, treat f as declared in $(f\ \mathbf{succ}\ (f\ 0))$.

Unification Again

We have two types T, U and we want to make them equal by finding a suitable substitution. This process is called unification.

In order to do this, keep a set S of pairs of types that need to be unified, and one of the original terms. Set $S = \{(T, U)\}$.

As long as S is not empty, randomly select and remove a pair of types from S.

- If the selected pair has form (V, V), nothing needs to be done for this pair.
- If the selected pair has form (α, X) or (X, α), and α does not occur in X, then substitute $\alpha := X$ everywhere in S and in the original terms.

- If the selected pair has form ($T_1 \to T_2,\ U_1 \to U_2$), then add (T_1, T_2) and (U_1, U_2) to S.
- If the selected pair has form ($C(T_1, \ldots, T_n),\ C(U_1, \ldots, U_n)$), then add $(T_1, U_1), \ldots, (T_n, U_n)$ to S.

  Here C is one of the type constructors defined on slide 5.
- In all other cases, fail.

Examples

- $\lambda x\ x.$
- $\lambda x\ (+\ x\ x).$
- $\lambda f\ \lambda g\ \lambda x\ (f\ (g\ x))$
- Not normalizing term $(\lambda x\ (x\ x))((\lambda x\ (x\ x)).$

  $\Rightarrow$ ¡No pasarán!

Theorem: All typable terms are strongly normalizing.

Succ (Church Style)

succ can be represented by $\lambda n\ \lambda f_0\ \lambda f_s\ (f_s\ (n\ f_0\ f_s))$. Let's type check.

If everything goes well, the result will be:
succ: $(\beta \to (\delta \to \gamma) \to \delta) \to \beta \to (\delta \to \gamma) \to \gamma$.