4. Hindley - Milner Type System

Goal: Describe a type system
for λ-calculus with parametric polymorphism

J. Roger Hindley, Trans. AMS 1969
Robin Milner, JCSS 1978 } POPL '82.
Luis Damas, PhD Edinburgh 1985

Properties:
- Completeness (some form)
- Infers most general type of a program
- Does not need type annotations
- Efficient type inference
  ↳ Often linear time in the size of the source program
  ↳ In general: ML typability is DEXPTIME-complete
  Kfoury, Tiuryn, Urzyczyn CAAP'90
- First implementation part of the type system for ML (1973).
  Extended to type class constraints in Haskell (1990).

Polymorphism:
Some kinds of data are very generic: List of something.
Functions on such generic data are again generic: Counting list items.
Types for such generic data and functions are called polymorphic
in that they can be used for more than one type:
List of numbers, list of words, list of something.

More precisely, this polymorphism is <u>parametric polymorphism</u>,

Something is the parameter in list of something.

Formally, List T with T being a type parameter.

Type of a function adding an item to a list:

$$\forall T.\ T \longrightarrow List\ T \longrightarrow List\ T.$$

Type system has to <u>express</u> parametric polymorphism.

<u>Type Inference</u>:

When using the above typing scheme with a type <u>checker</u>,

the type checker must be continuously informed about the types.

The above needs T as the first parameter → cluttered program text.

↳ List (1 2) would be

```
addItem Number 1 (addItem Number 2 (emptyList Number))
```

HM is strong enough to infer types,

not only for expressions but for whole programs,

including procedures and local definitions

Leads to a <u>type-less</u> style of programming:

```
quickSort [] = []
quickSort (x:xs) = quickSort (filter (<x) xs)
                   ++ [x] ++
                   quickSort (filter (>=x) xs).
```

Parametric types occur also in other programming languages:

↳ C++ templates 1998

↳ Java generics 2004.

Maintaining large untyped programs is a problem

C++ "auto" feature: Automatic return type deduction.

Not as powerful as in the functional setting.

Features of the HM Method:

Type Checking vs. Type Inference

In typing, an expression E is opposed to a type T, written $E : T$.

(Usually needs context, omitted here).

Questions of interest:

Type Checking: Given E, T, does $E : T$ hold?

Type Inference: Given E, derive a type for E, $E : -$?

Proofs: Given T, is there an expression with this type, $- : T$?

Curry-Howard-Isomorphism: Is there a proof for T?

For simply-typed λ-calculus: All three problems decidable.

↳ Makes types of parameters explicit, not needed in HM.

HM is a type inference method but can also be used for type checking.

-3- Third question interesting for recursively defined functions.

Monomorphism vs. Polymorphism

In *simply-typed* λ-calculus, types are
atomic type constants T or
function types $T \to T$.

Such types are *monomorphic*

$3 : \text{Number}$

$\text{add } 3\ 4 : \text{Number}$

$\text{add} : \text{Number} \to \text{Number} \to \text{Number}.$

In contrast, *untyped* λ-calculus is neutral to typing at all.
Many functions can be specified/ meaningfully applied to all kinds of arguments:

$$\lambda x.x$$

Polymorphism in general means
operators accept values of more than one type.

Here, polymorphism is *parametric*, also called *type schemes*.
In addition to type constants, there are *type variables*:

$\text{cons} : \forall a.\ a \to \text{List } a \to \text{List } a.$

$\text{nil} : \forall a.\ \text{List } a$

$\text{id} : \forall a.\ a \to a.$

Polymorphic types can become monomorphic
by *consistent substitutions*

$\text{id}' : \text{String} \to \text{String}$ $\quad$ $\text{nil}' : \text{List Number}$

C++ and Java focus on different kinds of polymorphism: subtyping or overloading.

Subtyping is incompatible with HM.

A variant of systematic overloading is added to an HM-based type system in Haskell.

Let-Polymorphism:

When is deriving a type admissible?

Ideally everywhere:

$$(\lambda\, id. \;\ldots\; (id\ 3) \ldots (id\ \text{"text"}) \ldots)\ (\lambda x. x)$$

↳ Type inference in such a system is undecidable (in the presence of recursion)

HM provides let-polymorphism

$$\underline{\text{let}}\ id = \lambda x. x\ \underline{\text{in}}\ \ldots (id\ 3) \ldots (id\ \text{"text"}) \ldots$$

Only types bound in let-constructs are subject to instantiation / are polymorphic.

Parameters in λ-abstractions are monomorphic.

1.1 Types

- We consider a λ-calculus with let-expressions

$$e ::= \underbrace{x}_{\text{variable}} \mid \underbrace{e_1\ e_2}_{\text{application}} \mid \underbrace{\lambda x. e}_{\text{abstraction}} \mid \underbrace{\text{let } x = e_1 \text{ in } e_2}_{\text{true polymorphism}}$$

Conventions

that allow us to drop brackets:

↳ Application associates to the left,
so

$$x\ y\ z \quad \text{stands for} \quad (x\ y)\ z$$

↳ Application binds stronger than abstraction and let

$$\lambda x\ \lambda y\ x\ y\ z \quad \text{stands for} \quad \lambda x.(\lambda y.((x\ y)\ z))$$

This is also formulated as

"Bodies of lambdas extend as far as possible."

↳ Nested lambdas can be collapsed.

$$\lambda xyz.\ x\ y\ z \quad \text{stands for} \quad \lambda x.\lambda y.\lambda z.\ x\ y\ z$$

- Types are split into two groups, <u>monotypes</u> and <u>polytypes</u>

<u>Monotypes:</u> $$\tau ::= \underbrace{\alpha}_{\text{variable}} \mid \underbrace{C\ \tau \dots \tau}_{\text{application}}$$

Monotypes include int or string,
but also parametric types like

Map (Set string) int.

This is an example of an application of type functions.
The set of type functions C is arbitrary in HM,
but <u>must</u> <u>contain</u> $\rightarrow_2$.

Application binds stronger than $\to$
and $\to$ binds to the right.
Monotypes are <u>equal</u> if they are equal as terms.

<u>Example</u>:

$C = \{ Map/_2, Set/_1, string/_0, int/_0, \to/_2 \}$.

<u>Note</u>:
Type variables are admitted as monotypes.
Therefore, monotypes are <u>not</u> <u>monomorphic</u>
in that they admit only ground terms (see above).

<u>Polytypes</u>:

$\sigma ::= \tau \mid \forall \alpha . \sigma .$

So types contain variables that are bound by $\forall$.

$\forall \alpha .\ \alpha \to \alpha$

$\forall \alpha .\ (Set\ \alpha) \to int.$

Note, however, that quantifiers only appear top-level:

$\forall \alpha . \alpha \to \forall \alpha . \alpha$ is forbidden.

Monotypes are also polytypes.
In general, polytypes have the form

$\forall \alpha_1 \ldots \forall \alpha_n . \tau$, where τ is a monotype.

Polytypes equal up to
- reordering of quantifiers
- α-conversion of quantified variables
- dropping quantified variables not in the monotype.

- **Context and Typing:**

  To bring together expressions and types, we need a context.

  A context is a set of pairs

  $$x : \sigma,$$

  called assignments, assumptions, or bindings, stating that variable x (program) has type σ.

  All three parts combined yield a type judgement

  $$\Gamma \vdash e : \sigma,$$

  stating that under the assumption Γ, e has type σ.

- **Free Type Variables:**
  - In a type $\forall \alpha_1 \dots \forall \alpha_n . \tau$, $\forall$ binds the variables α_i in the monotype τ.
  - All unbound variables in τ are free.
  - Additionally, variables can be bound by occurring in the context. In this case, they behave like type constants in the rhs of $\vdash$.
  - Finally, a type variable may occur unbound. In this case, it is implicitly $\forall$-quantified.

    $$a \to a \quad \leadsto \quad \forall a.\ a \to a.$$

1.2 Type Order

- Types are related by the parametric polymorphism.

  $\lambda x.x$ can have types $\forall\alpha.\ \alpha \to \alpha$

  string $\to$ string

  int $\to$ int

  but not string $\to$ int.

  The most general type would be $\forall\alpha.\ \alpha\to\alpha$.

  More specific types can be obtained by replacing the type parameter by other types.

- (Consistent) replacement is formalized by substitution, mappings of the form

  $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$

  Application of S to type τ yields type $S\tau$ where every free occurrence of α_i is replaced by τ_i.

- Substitution yields a partial order on types, stating that types are more or less special.

Definition (Specialization order):

Type σ is called more general than σ', $\sigma \sqsubseteq \sigma'$ if the following rule applies:

$$\frac{\tau' = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}\tau \qquad \beta_i \notin free(\forall\alpha_1 \ldots \forall\alpha_n.\ \tau)}{\forall\alpha_1 \ldots \forall\alpha_n.\ \tau \sqsubseteq \forall\beta_1 \ldots \forall\beta_m.\ \tau'}$$

<u>Idea:</u> · Imagine polytypes without quantifiers,
but where quantified variables have different symbols.

· In this setting, specialization reduces to
replacement of variables/these symbols.

· Here, this is expressed as follows.
Free variables must not be replaced
but are treated as constants.

· Note that variables contained in the τ_i
can again be quantified (by the new type variables β_i).

<u>Example:</u>

$$\forall a.\ a \to a \sqsubseteq \forall b,c.\ (b \to c) \to (b \to c).$$

<u>Lemma:</u> · $\sqsubseteq$ is a partial order

· There is a least element, $\forall \alpha. \alpha$.

· Downward directed sets (subsets S with
$\forall x,y \in S\ \exists z \in S: x \geq z \wedge y \geq z$)
of types have a greatest lower bound.

<u>Principle Type:</u>

Type inference faces the challenge
of summarizing all types an expression may have.
The above order guarantees that such a summary exists,
in the form of a most general type of an expression.

Substitution in Type Judgements:

The order on types can be lifted to type judgements.

Lemma:

$$\Gamma \vdash e : \sigma \quad \Rightarrow \quad S\Gamma \vdash e : S\sigma.$$

Note that this lemma is not part of the definition of $\sqsubseteq$.
Instead, it follows from the typing rules given in a moment.
Free variables serve as placeholders for refinements.
The binding effect of the environment
is handled by requiring a consistent substitution
in Γ and in σ.

1.3 Deductive System

Typings are derived as proofs in a proof system.

1.3.1 Typing Rules

$$(VAR)\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\, e_1 : \tau'}\ (APP)$$

$$(ABS)\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \mathrm{let}\ x = e_0\ \mathrm{in}\ e_1 : \tau}\ (LET)$$

$$(INST)\ \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \qquad \frac{\Gamma \vdash e : \sigma\ , \quad \alpha \notin free(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}\ (GEN)$$

The rules can be decomposed into two groups.

<u>Centered around the syntax of programs</u>:

(VAR), (APP), (ABS), (LET)

These rules decompose each expression, prove the subexpressions, combine the individual types found in the premises to the type given in the conclusion.

<u>On specialization and generalization of types</u>:

(INST), (GEN).

Note that (GEN) is the implicit $\forall$-quantification mentioned above.

<u>Example</u>:

1.) Let $\Gamma = \{ id : \forall\alpha.\alpha \to \alpha,\ n : int \}$.

Then $\Gamma \vdash id\ n : int$.

<u>Proof</u>:

$$\dfrac{\text{(INST)}\,\dfrac{\text{(VAR)}\,\dfrac{}{\Gamma \vdash id : \forall\alpha.\alpha\to\alpha}}{\Gamma \vdash id : int \to int} \qquad \dfrac{}{\Gamma \vdash n : int}\,\text{(VAR)}}{\Gamma \vdash id\ n : int}\,\text{(APP)}$$

2.) $\vdash let\ id = \lambda x.x\ in\ id : \forall\alpha.\alpha \to \alpha$.

$$\text{(LET)}\,\dfrac{\text{(GEN)}\,\dfrac{\text{(ABS)}\,\dfrac{\text{(VAR)}\,\dfrac{}{x:\alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \forall\alpha.\alpha\to\alpha} \qquad \text{(VAR)}\,\dfrac{}{id : \forall\alpha.\alpha\to\alpha \vdash id : \forall\alpha.\alpha\to\alpha}}{\vdash let\ id = \lambda x.x\ in\ id : \forall\alpha.\alpha\to\alpha}$$

Let-Polymorphism:

The rules encode a mechanism under which circumstances a type might be generalized or not.

- In (ABS), the variable x of $\lambda x. e$ is added to the context through the premise $\Gamma, x : \tau \vdash e : \tau'$.
- In (LET), the variable enters the environment in polymorphic form: $\Gamma, x : \sigma \vdash e_1 : \tau$.
- In both cases, x in the context prevents us from generalization.
- Hence, x in a λ-expression to remain monomorphic.

In a let-expression, the variable can be introduced polymorphically, making specializations possible.

Consequence:

- $\lambda f. (f \text{ true}, f\ 0)$ cannot be typed, as f is in monomorphic position.
- let $f = \lambda x. x$ in $(f \text{ true}, f\ 0)$ has type (bool, int) as f is treated polymorphically.

1.4 The Inference Algorithm

↳ Understand how the rules interact and proofs are formed.

↳ Group 1 rules are syntax-directed and leave no choice.

↳ Group 2 rules need choices.

Understanding where (INST) and (GEN) are needed leads to a variant of the proof system without such rules.

⇒ Specialization is merged into (VAR)

⇒ Generalization is merged into (LET).

Produces the most general type by quantifying all monotype variables that are not bound.

Rules:

$$(\text{VAR}')\ \frac{x:\sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x:\tau} \qquad \frac{\Gamma \vdash e_0:\tau \to \tau' \quad \Gamma \vdash e_1:\tau}{\Gamma \vdash e_0 e_1:\tau'}\ (\text{APP})$$

$$(\text{ABS})\ \frac{\Gamma, x:\tau \vdash e:\tau'}{\Gamma \vdash \lambda x.e:\tau \to \tau'} \qquad \frac{\Gamma \vdash e_0:\tau \quad \Gamma, x:\overline{\Gamma}(\tau) \vdash e_1:\tau'}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1:\tau'}\ (\text{LET}')$$

Here, $\overline{\Gamma}(\tau) := \forall \bar{\alpha}.\tau$ with $\bar{\alpha} = free(\tau) \setminus free(\Gamma)$.

To show the equivalence between $\vdash_{old}$ and $\vdash_{new}$, one has to prove:

$\Gamma \vdash_{old} e:\sigma \quad \Leftarrow \quad \Gamma \vdash_{new} e:\sigma$ (Consistency) and

$\Gamma \vdash_{old} e:\sigma \quad \Rightarrow \quad \Gamma \vdash_{new} e:\sigma$ (Completeness).

Consistency can be shown by decomposing the rules (LET') and (VAR').

Completeness does not hold,

one cannot show $\vdash_{new} \lambda x.x : \forall\alpha.\alpha \to \alpha$

but only $\vdash_{new} \lambda x.x : \alpha \to \alpha$.

The form of completeness that is provable is this:

$\Gamma \vdash_{old} e:\sigma \Rightarrow \Gamma \vdash_{new} e:\tau \wedge \overline{\Gamma}(\tau) \sqsubseteq \sigma$.

Note that the new proof system only uses monotypes.
Moreover, the shape of the proof is the shape of the expression.

Degrees of freedom in instantiating the rules:

- Because all proofs for a given expression have the same shape, consider monotypes in the proofs' judgements undetermined and develop a way to determine them.
- Here, specialization order comes into play.
  Although types cannot be determined locally,
  the hope is to refine them while traversing the tree.
  So one assumes that the type derived at a premise is a principle one.

Here is the idea how to proceed:

(ABS): The critical choice is τ for x.
Use the most general type $\forall \alpha . \alpha$.
Since a polytype is not permitted here, we use a fresh α.

The type α is not yet fixed, it may be refined later.

(VAR'): The choice is how to refine σ.
Any choice for a τ depends on the usage of the variable.
We keep the most general type,
but instantiate all qualified variables in σ with fresh monotype variables.
This leaves open the possibility of further refinements.

(LET'): No choices, done.

(APP): May force a refinement of the fresh monotype variables introduced so far.

The first premise forces the outcome of the inference to be of the form $\tau \to \tau'$.

↳ If it is, fine.

↳ If not, it may be a fresh monotype variable. Then it can be refined to the required form with two fresh variables.

↳ Otherwise, type inference fails because the first premise inferred a type which is not and cannot be made a function type.

The second premise requires that the inferred type is equal to τ of the first premise.

Now two possibly different types, perhaps with fresh monotype variables, have to be compared and made equal — if possible.

↳ If this works, a refinement is found and has to be applied.

↳ If not, again a type error is detected.

<u>Robinson's Unification Algorithm</u>

John Alan Robinson, A machine-oriented logic based on the resolution principle, JACM 1965.

Effective method of making two terms equal. Implemented by a <u>union-find algorithm</u>:

Given a set of terms, the algorithm
↳ groups them into equivalence classes by the <u>union</u> procedure
↳ picks a representative for each class using the <u>find</u> procedure.

<u>Definition of the representative:</u>

We define find(union(a, b)) as follows. classes

↳ If find(a) and find(b) are fresh monotype variables,
one of them is chosen.

↳ If one is a fresh monotype variable and the other a term,
the term is the representative of union(a, b).

Note that this is a recursive definition.

We assume an implementation of union-find at hand with methods

union (a, b) = union the classes

find (a) = return the representative.

With this, unification of two monotypes works as follows.

<u>unify (ta, tb)</u>:

  ta = find(class(ta));

  tb = find(class(tb));

  if ta and tb are terms of the form

    $ta = D\ p_1 \dots p_n$, $tb = D\ q_1 \dots q_n$, same D, same n <u>then</u>

    <u>for all</u> $i = 1$ to n <u>do</u> unify(p_i, q_i);

  <u>else</u> <u>if</u> at least one of ta, tb is a fresh monotype variable <u>then</u>

    union(ta, tb);

  <u>else</u> error, the types do not match.

- Here, we use class(t) to access the class of a term t.
- Note that when joining classes of terms, we do not adjust the classes of composed terms that contain these terms as subterms.
- The point is that the classes as constructed above satisfy the following compositionality:

$$\mathrm{class}(D\ p_1 \ldots p_n) = \{ D\ q_1 \ldots q_n \mid q_1 \in \mathrm{class}(p_1), \ldots, q_n \in \mathrm{class}(p_n) \} \cup Vars,$$

for some set of fresh monotype variables.
- With this compositionality, and with the definition of find, we can compute the representative of a class as

$$\mathrm{find}(\mathrm{class}(D\ p_1 \ldots p_n)) = D\ \mathrm{find}(\mathrm{class}(p_1)) \ldots \mathrm{find}(\mathrm{class}(p_n)).$$

- This means we can <u>reason about</u> classes of composed terms
compute
representatives
without having to store them explicitly.
This is called <u>symbolic</u> <u>reasoning</u>.

- The algorithm given below will use that classes of the form $\{\alpha, t\}$ can be seen as <u>substitutions</u> $\alpha \mapsto t$.
Later substitutions are composed with previous ones to derive a closed-form representation of the overall equivalence.

Example:

Consider $\underbrace{(\alpha \to \beta) \to \alpha}_{ta}$ and $\underbrace{\gamma \to int}_{tb}$.

- In the call unify(ta, tb), the first if-condition applies:

$$ta = p_1 \to p_2 \quad \text{with} \quad p_1 = \alpha \to \beta,\ p_2 = \alpha$$

$$tb = q_1 \to q_2 \quad \text{with} \quad q_1 = \gamma,\ q_2 = int.$$

We call unify (p_1, q_1).
As γ is a monotype variable, we establish the class $\{\alpha \to \beta, \gamma\}$

We call unify (p_2, q_2).
As α is a monotype variable, we establish the class $\{\alpha, int\}$.

As there are no more open calls, unify terminates successfully.

- If we execute
find (class (ta))
we get

$$\begin{aligned}
&find(class((\alpha \to \beta) \to \alpha))\\
&= find(class(\alpha \to \beta)) \to find(class(\alpha))\\
&= (find(class(\alpha)) \to find(class(\beta))) \to int\\
&= (int \to \beta) \to int.
\end{aligned}$$

This is also the return value of find(class(tb)), as required.
As substitutions: $S_0 = \{\gamma \mapsto \alpha \to \beta\}$, $S_1 = \{\alpha \to int\}$, $S_1 S_0 = \{\gamma \mapsto int \to \beta, \alpha \mapsto int\}$.
We have $(S_1 S_0)\,ta = (int \to \beta) \to int = (S_1 S_0)\,tb$.

1.5 Algorithm W

Goal: Formalize the above idea into an algorithm.
due to Milner 1978.

Idea: Type judgements $\Gamma \vdash e : \tau, S$ should be seen as a recursive procedure: Parameters are Γ, e
Return values τ, S.

New: The substitution S computed by the unification.

In the following rules, the premises are invoked from left to right.
Moreover, $\underline{inst(\sigma)}$ copies σ and consistently replaces bound variables by fresh monotype variables.

$\underline{newvar}$ creates a fresh monotype variable.

Also $\underline{\overline{\Gamma}(\tau)}$ creates a copy of the type where fresh monotype variables are introduced for the variables that should be quantified in $\overline{\Gamma}(\tau)$.

(VAR'') $$\frac{x : \sigma \in \Gamma \qquad \tau = inst(\sigma)}{\Gamma \vdash x : \tau, \emptyset}$$

(APP'') $$\frac{\Gamma \vdash e_0 : \tau_0, S_0 \qquad S_0\Gamma \vdash e_1 : \tau_1, S_1 \qquad \tau' = newvar \qquad S_2 = unify(S_1\tau_0, \tau_1 \to \tau')}{\Gamma \vdash e_0\, e_1 : S_2\tau', S_2 S_1 S_0}$$

(ABS'') $$\frac{\tau = newvar \qquad \Gamma, x : \tau \vdash e : \tau; S}{\Gamma \vdash \lambda x.e : S\tau \to \tau; S}$$

$$(\text{LET}'') \ \frac{\Gamma \vdash e_0 : \tau, S_0 \qquad S_0\Gamma, x : \overline{S_0\Gamma}(\tau) \vdash e_1 : \tau', S_1}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau', S_1 S_0}$$

<u>Note</u>:

- The resulting type τ in $\Gamma \vdash e : \tau, S$ has to be generalized to $\overline{\Gamma}(\tau)$.
- Complexity often linear in the size of the term. However, deep nesting of lets leads to exponential runtime.