Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Edit links

Hindley–Milner type system

From Wikipedia, the free encyclopedia
(Redirected from Hindley-Milner type inference)

In type theory and functional programming, **Hindley–Milner** (**HM**) (also known as **Damas–Milner** or **Damas–Hindley–Milner**) is a classical type system for the lambda calculus with parametric polymorphism, first described by J. Roger Hindley[1] and later rediscovered by Robin Milner.[2] Luis Damas contributed a close formal analysis and proof of the method in his PhD thesis.[3][4]

Among HM's more notable properties is completeness and its ability to deduce the most general type of a given program without the need of any type annotations or other hints supplied by the programmer. **Algorithm W** is a fast algorithm, performing type inference in almost linear time with respect to the size of the source, making it practically usable to type large programs.[note 1] HM is preferably used for functional languages. It was first implemented as part of the type system of the programming language ML. Since then, HM has been extended in various ways, most notably by constrained types as used in Haskell.

Contents [hide]

Introduction [edit]

Organizing their original paper, Damas and Milner[4] clearly separated two very different tasks. One is to describe what types an expression can have and another to present an algorithm actually computing a type. Keeping both aspects apart from each other allows one to focus separately on the logic (i.e. meaning) behind the algorithm, as well as to establish a benchmark for the algorithm's properties.

How expressions and types fit to each other is described by means of a deductive system. Like any proof system, it allows different ways to come to a conclusion and since one and the same expression arguably might have different types, dissimilar conclusions about an expression are possible. Contrary to this, the type inference method itself (Algorithm W) is defined as a deterministic step-by-step procedure, leaving no choice what to do next. Thus clearly, decisions not present in the logic might have been made constructing the algorithm, which demand a closer look and justifications but would perhaps remain non-obvious without the above differentiation.

Syntax [edit]

Logic and algorithm share the notions of "expression" and "type", whose form is made precise by the syntax.

Expressions

The expressions to be typed are exactly those of the lambda calculus, enhanced by a let-expression. These are shown in the table to the right. For readers unfamiliar with the lambda calculus, here is a brief explanation: The application $e_1 e_2$ represents applying the function e_1 to the argument e_2, often written $e_1(e_2)$. The abstraction $\lambda\ x\ .\ e$ represents an anonymous function that maps the input x to the output e. This is also called function literal, common in most contemporary programming languages, and sometimes written as $\texttt{function}\ (x)\ \texttt{return}\ e\ \texttt{end}$. The let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ represents the result of substituting every occurrence of x in e_2 with e_1.

e	$=$	x	variable
	$\mid$	$e_1\ e_2$	application
	$\mid$	$\lambda\ x\ .\ e$	abstraction
	$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	

Types

mono	τ	$=$	α	variable
		$\mid$	$D\ \tau \ldots \tau$	application
poly	σ	$=$	τ	
		$\mid$	$\forall\ \alpha\ .\ \sigma$	quantifier

Types as a whole are split into two groups, called mono- and polytypes.[note 2]

Monotypes [edit]

Monotypes τ are syntactically represented as terms. A monotype always designates a particular type, in the sense that it is equal only to itself and different from all others.

Examples of monotypes include type constants like $\texttt{int}$ or $\texttt{string}$, and parametric types like $\texttt{Map}\ (\texttt{Set}\ \texttt{string})\ \texttt{int}$. These types are examples of *applications* of type functions, for example, from the set $\{\texttt{Map}^2,\ \texttt{Set}^1,\ \texttt{string}^0,\ \texttt{int}^0\}$, where the superscript indicates the number of type parameters. The complete set of type functions D is arbitrary in HM, except that it *must* contain at least $\to^2$, the type of functions. It is often written in infix notation for convenience. For example, a function mapping integers to strings has type $\texttt{int} \to \texttt{string}$. [note 3]

Type variables are monotypes. Standing alone, a type variable α is meant to be as concrete as $\texttt{int}$ or β, and clearly different from both. Type variables occurring as monotypes behave as if they were type constants whose identity is unknown. Correspondingly, a function typed $\alpha \to \alpha$ only maps values of the particular type α on itself. Such a function can only be applied to values having type α and to no others.

Polytype [edit]

Polytypes (or *type schemes*) are types containing variables bound by one or more for-all quantifiers, e.g. $\forall\alpha.\alpha \to \alpha$.

A function with polytype $\forall\alpha.\alpha \to \alpha$ can map *any* value of the same type to itself, and the identity function is a value for this type. As another example $\forall\alpha.(\texttt{Set}\ \alpha) \to \texttt{int}$ is the type of a function mapping all finite sets to integers. The count of members is a value for this type. Note that quantifiers can only appear top level, i.e. a type $\forall\alpha.\alpha \to \forall\alpha.\alpha$ for instance, is excluded by the syntax of types and that monotypes are included in the polytypes, thus a type has the general form $\forall\alpha_1 \ldots \forall\alpha_n.\tau$.

Free type variables [edit]

In a type $\forall\alpha_1 \ldots \forall\alpha_n.\tau$, the symbol $\forall$ is the quantifier binding the type variables α_i in the monotype τ. The variables α_i are called *quantified* and any occurrence of a quantified type variable in τ is called *bound* and all unbound type variables in τ are called *free*. Like in the lambda calculus, the notion of free and bound variables is essential for the understanding of the meaning of types.

Free Type Variables

$$\begin{aligned} \text{free}(\ \alpha\ ) &= \{\alpha\} \\ \text{free}(\ D\ \tau_1 \ldots \tau_n\ ) &= \bigcup_{i=1}^{n} \text{free}(\ \tau_i\ ) \\ \text{free}(\ \forall\ \alpha\ .\ \sigma\ ) &= \text{free}(\ \sigma\ ) - \{\alpha\} \end{aligned}$$

This is certainly the hardest part of HM, perhaps because polytypes containing free variables are not represented in programming languages like Haskell. Likewise, one does not have clauses with free variables in Prolog. In particular developers experienced with both languages and actually knowing all the prerequisites of HM, are likely to slip this point. In Haskell for example, all type variables implicitly occur quantified, i.e. a Haskell type `a -> a` means $\forall\alpha.\alpha \to \alpha$ here. Because a type like $\alpha \to \alpha$, though it may practically occur in a Haskell program, cannot be expressed there, it can easily be confused with its quantified version.

So what function can have a type like e.g. $\forall\beta.\beta \to \alpha$, i.e. a mixture of both bound and free type variables and what could the free type variable α therein mean?

Consider foo in Example 1, with type annotations in brackets. Its parameter y is not used in the body, but the

Example 1

variable x bound in the outer context of foo surely is. As a consequence, foo accepts every value as argument, while returning a value bound outside and with it its type. bar to the contrary has type $\forall\alpha.\forall\beta.\alpha \to (\beta \to \alpha)$, in which all occurring type variables are bound. Evaluating, for instance $bar\ 1$, results in a function of type $\forall\beta.\beta \to int$, perfectly reflecting that foo's monotype α in $\forall\beta.\beta \to \alpha$ has been refined by this call.

$$\begin{array}{l}\textbf{let } bar\ [\forall\alpha.\forall\beta.\alpha \to (\beta \to \alpha)] = \lambda\ x. \\ \quad \textbf{let } foo\ [\forall\beta.\beta \to \alpha] = \lambda\ y.x \\ \quad \textbf{in } foo \\ \textbf{in } bar\end{array}$$

In this example, the free monotype variable α in foo's type becomes meaningful by being quantified in the outer scope, namely in bar's type. I.e. in context of the example, the same type variable α appears both bound and free in different types. As a consequence, a free type variable cannot be interpreted better than stating it is a monotype without knowing the context. Turning the statement around, in general, a typing is not meaningful without a context.

Context and typing [edit]

Consequently, to get the yet disjoint parts of the syntax, expressions and types together meaningfully, a third part, the context is needed. Syntactically, it is a list of pairs $x : \sigma$, called assignments or assumptions, stating for each value variable x_i therein a type σ_i. All three parts combined gives a *typing judgment* of the form $\Gamma \vdash e : \sigma$, stating, that under assumptions Γ, the expression e has type σ.

Syntax

$$\begin{array}{lcl}\text{Context} & \Gamma & = \epsilon\ (\textbf{empty}) \\ & & \mid\ \Gamma,\ x : \sigma \\ \text{Typing} & & = \Gamma \vdash e : \sigma\end{array}$$

Free Type Variables

$$\text{free}(\ \Gamma\ ) = \bigcup_{x:\sigma\in\Gamma} \text{free}(\ \sigma\ )$$

Now having the complete syntax at hand, one can finally make a meaningful statement about the type of foo in example 1, above, namely $x : \alpha \vdash \lambda\ y.x : \forall\beta.\beta \to \alpha$. Contrary to the above formulations, the monotype variable α no longer appears unbound, i.e. meaningless, but bound in the context as the type of the value variable x. The circumstance whether a type variable is bound or free in the context apparently plays a significant role for a type as part of a typing, so $\text{free}(\ \Gamma\ )$ it is made precise in the side box.

Polymorphic type order [edit]

While the equality of monotypes is purely syntactical, polytypes offer a richer structure by being related to other types through a specialization relation $\sigma \sqsubseteq \sigma'$ expressing that σ' is more special than σ.

When being applied to a value a polymorphic function has to change its shape specializing to deal with this particular type of values. During this process, it also changes its type to match that of the parameter. If for instance the identity function having type $\forall\alpha.\alpha \to \alpha$ is to be applied on a number having type int, both simply cannot work together, because all the types are different and nothing fits. What is needed is a function of type $int \to int$. Thus, during application, the polymorphic identity is specialized to a monomorphic version of itself. In terms of the specialization relation, one writes $\forall\alpha.\alpha \to \alpha \sqsubseteq\ int \to int$

Now the shape shifting of polymorphic values is not fully arbitrary but rather limited by their pristine polytype. Following what has happened in the example one could paraphrase the rule of specialization, saying, a polymorphic type $\forall\alpha.\tau$ is specialized by consistently replacing each occurrence of α in τ and dropping the quantifier. While this rule works well for any monotype used as replacement, it fails when a polytype, say $\forall\beta.\beta$ is tried as a replacement, resulting in the non-syntactical type $\forall\beta.\beta \to \forall\beta.\beta$. But not only that. Even if a type with nested quantified types would be allowed in the syntax, the result of the substitution would not longer preserve the property of the pristine type, in which both the parameter and the result of the function have the same type, which are now only seemingly equal because both subtypes became independent from each other allowing to specialize the parameter and the result with different types resulting in, e.g. $string \to Set\ int$, hardly the right task for an identity function.

The syntactic restriction to allow quantification only top-level is imposed to prevent generalization while specializing. Instead of $\forall\beta.\beta \to \forall\beta.\beta$, the more special type $\forall\beta.\beta \to \beta$ must be produced in this case.

One could undo the former specialization by specializing on some value of type $\forall\alpha.\alpha$ again. In terms of the relation one gains $\forall\alpha.\alpha \to \alpha \sqsubseteq \forall\beta.\beta \to \beta \sqsubseteq \forall\alpha.\alpha \to \alpha$ as a summary, meaning that syntactically different polytypes are equal with respect to renaming their quantified variables.

Now focusing only on the question whether a type is more special than another and no longer what the specialized type is used for, one could summarize the specialization as in the box above. Paraphrasing it clockwise, a type $\forall\alpha_1 \ldots \forall\alpha_n.\tau$ is specialized by consistently replacing any of the quantified variables α_i

Specialization Rule

$$\frac{\tau' = [\alpha_i := \tau_i]\,\tau \quad \beta_i \notin \text{free}(\forall\alpha_1 ... \forall\alpha_n.\tau)}{\forall\alpha_1 ... \forall\alpha_n.\tau \sqsubseteq \forall\beta_1 ... \forall\beta_m.\tau'}$$

by arbitrary monotypes τ_i gaining a monotype τ'.
Finally, type variables in τ' not occurring free in the pristine type can optionally be quantified.

Thus the specialization rules makes sure that no free variable, i.e. monotype in the pristine type becomes unintentionally bound by a quantifier, but originally quantified variable can be replaced with whatever, even with types introducing new quantified or unquantified type variables.

Starting with a polytype $\forall\alpha.\alpha$, the specialization could either replace the body by another quantified variable, actually a rename or by some type constant (including the function type) which may or may not have parameters filled either with monotypes or quantified type variables. Once a quantified variable is replaced by a type application, this specialization cannot be undone through another substitution as it was possible for quantified variables. Thus the type application is there to stay. Only if it contains another quantified type variable, the specialization could continue further replacing for it.

So the specialization introduces no further equivalence on polytype beside the already known renaming. Polytypes are syntactically equal up to renaming their quantified variables. The equality of types is a reflexive, antisymmetric and transitive relation and the remaining specializations of polytypes are transitive and with this the relation $\sqsubseteq$ is an order.

Deductive system [edit]

The syntax of HM is carried forward to the syntax of the inference rules that form the body of the formal system, by using the typings as judgments. Each of the rules define what conclusion could be drawn from what premises. Additionally to the judgments, some extra conditions introduced above might be used as premises, too.

A proof using the rules is a sequence of judgments such that all premises are listed before a conclusion. Please see the Examples 2 and 3 below for a possible format of proofs. From left to right, each line shows the conclusion, the [Name] of the rule applied and the premises, either by referring to an earlier line (number) if the premise is a judgment or by making the predicate explicit.

The Syntax of Rules

$$\begin{array}{lcl} \text{Predicate} & = & \sigma \sqsubseteq \sigma' \\ & \mid & \alpha \notin free(\Gamma) \\ & \mid & x : \alpha \in \Gamma \\ \\ \text{Judgment} & = & \text{Typing} \\ \text{Premise} & = & \text{Judgment} \mid \text{Predicate} \\ \text{Conclusion} & = & \text{Judgment} \\ \\ \text{Rule} & = & \dfrac{\text{Premise} \ \ldots}{\text{Conclusion}} \quad [\texttt{Name}] \end{array}$$

Typing rules [edit]

See also: Type rules

The side box shows the deduction rules of the HM type system. One can roughly divide them into two groups:

The first four rules [Var] (variable or function access), [App] (*application*, i.e. function call with one parameter), [Abs] (*abstraction*, i.e. function declaration) and [Let] (variable declaration) are centered around the syntax, presenting one rule for each of the expression forms. Their meaning is pretty obvious at the first glance, as they decompose each expression, prove their sub-expressions and finally combine the individual types found in the premises to the type in the conclusion.

The second group is formed by the remaining two rules [Inst] and [Gen]. They handle specialization and generalization of types. While the rule [Inst] should be clear from the section on specialization above, [Gen] complements the former, working in the opposite direction. It allows generalization, i.e. to quantify monotype variables that are not bound in the context. The necessity of this restriction $\alpha \notin free(\ \Gamma\ )$ is introduced in the section on free type variables.

Declarative Rule System

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

The following two examples exercise the rule system in action

Example 2: A proof for $\Gamma \vdash_D id(n) : int$ where $\Gamma = id : \forall\alpha.\alpha \to \alpha,\ n : int$, could be written

$$\begin{array}{lll} 1: & \Gamma \vdash_D id : \forall\alpha.\alpha \to \alpha & [\mathtt{Var}] \quad (id : \forall\alpha.\alpha \to \alpha \in \Gamma) \\ 2: & \Gamma \vdash_D id : int \to int & [\mathtt{Inst}] \quad (1),\ (\forall\alpha.\alpha \to \alpha \sqsubseteq int \to int) \\ 3: & \Gamma \vdash_D n : int & [\mathtt{Var}] \quad (n : int \in \Gamma) \\ 4: & \Gamma \vdash_D id(n) : int & [\mathtt{App}] \quad (2),\ (3) \end{array}$$

Example 3: To demonstrate generalization, $\vdash_D \ \mathbf{let}\, id = \lambda x.x\ \mathbf{in}\ id\ :\ \forall\alpha.\alpha \to \alpha$ is shown below:

$$\begin{array}{lll} 1: & x : \alpha \vdash_D x : \alpha & [\mathtt{Var}] \quad (x : \alpha \in \{x : \alpha\}) \\ 2: & \vdash_D \lambda x.x : \alpha \to \alpha & [\mathtt{Abs}] \quad (1) \\ 3: & \vdash_D \lambda x.x : \forall\alpha.\alpha \to \alpha & [\mathtt{Gen}] \quad (2),\ (\alpha \notin free(\epsilon)) \\ 4: & id : \forall\alpha.\alpha \to \alpha \vdash_D id : \forall\alpha.\alpha \to \alpha & [\mathtt{Var}] \quad (id : \forall\alpha.\alpha \to \alpha \in \{id : \forall\alpha.\alpha \to \alpha\}) \\ 5: & \vdash_D \mathbf{let}\, id = \lambda x.x\ \mathbf{in}\ id\ :\ \forall\alpha.\alpha \to \alpha & [\mathtt{Let}] \quad (3),\ (4) \end{array}$$

Principal type [edit]

As mentioned in the introduction, the rules allow one to deduce different types for one and the same expression. See for instance, Example 2, steps 1,2 and Example 3, steps 2,3 for three different typings of the same expression. Clearly, the different results are not fully unrelated, but connected by the type order. It is an important property of the rule system and this order that whenever more than one type can be deduced for an expression, among them is (modulo alpha-renaming of the type variables) a unique most general type in the sense, that all others are specialization of it. Though the rule system must allow to derive specialized types, a type inference algorithm should deliver this most general or principal type as its result.

Let-polymorphism [edit]

Not visible immediately, the rule set encodes a regulation under which circumstances a type might be generalized or not by a slightly varying use of mono- and polytypes in the rules $[\mathtt{Abs}]$ and $[\mathtt{Let}]$.

In rule $[\mathtt{Abs}]$, the value variable of the parameter of the function $\lambda x.e$ is added to the context with a monomorphic type through the premise $\Gamma,\ x : \tau \vdash_D e : \tau'$, while in the rule $[\mathtt{Let}]$, the variable enters the environment in polymorphic form $\Gamma,\ x : \sigma \vdash_D e_1 : \tau$. Though in both cases the presence of x in the context prevents the use of the generalisation rule for any monotype variable in the assignment, this regulation forces the parameter x in a λ-expression to remain monomorphic, while in a let-expression, the variable could already be introduced polymorphic, making specializations possible.

As a consequence of this regulation, *no* type can be inferred for $\lambda f.(f\ \mathrm{true}, f\ 0)$ since the parameter f is in a monomorphic position, while $\mathbf{let}\ f = \lambda x.x\ \mathbf{in}\ (f\ \mathrm{true}, f\ 0)$ yields a type $(bool, int)$, because f has been introduced in a let-expression and is treated polymorphic therefore. Note that this behaviour is in strong contrast to the usual definition $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ ::= (\lambda\, x.e_2)\ e_1$ and the reason why the let-expression appears in the syntax at all. This distinction is called **let-polymorphism** or **let generalization** and is a conception owed to HM.

Towards an algorithm [edit]

Now that the deduction system of HM is at hand, one could present an algorithm and validate it with respect to the rules. Alternatively, it might be possible to derive it by taking a closer look on how the rules interact and proof are formed. This is done in the remainder of this article focusing on the possible decisions one can make while proving a typing.

Degrees of freedom choosing the rules [edit]

Isolating the points in a proof, where no decision is possible at all, the first group of rules centered around the syntax leaves no choice since to each syntactical rule corresponds a unique typing rule, which determines a part of the proof, while between the conclusion and the premises of these fixed parts chains of $[\mathtt{Inst}]$ and $[\mathtt{Gen}]$ could occur. Such a chain could also exist between the conclusion of the proof and the rule for topmost expression. All proofs must have the so sketched shape.

Because the only choice in a proof with respect of rule selection are the $[\mathtt{Inst}]$ and $[\mathtt{Gen}]$ chains, the form of the proof suggests the question whether it can be made more precise, where these chains might be needed. This is in fact possible and leads to a variant of the rules system with no such rules.

Syntax-directed rule system [edit]

A contemporary treatment of HM uses a purely

syntax-directed rule system due to Clement[5] as an intermediate step. In this system, the specialization is located directly after the original $[\mathtt{Var}]$ rule and merged into it, while the generalization becomes part of the $[\mathtt{Let}]$ rule. There the generalization is also determined to always produce the most general type by introducing the function $\bar{\Gamma}(\tau)$, which quantifies all monotype variables not bound in Γ.

Formally, to validate, that this new rule system $\vdash_S$ is equivalent to the original $\vdash_D$, one has to show that $\Gamma \vdash_D \ e : \sigma \Leftrightarrow \Gamma \vdash_S \ e : \sigma$, which falls apart into two sub-proofs:

- $\Gamma \vdash_D \ e : \sigma \Leftarrow \Gamma \vdash_S \ e : \sigma$ (Consistency)
- $\Gamma \vdash_D \ e : \sigma \Rightarrow \Gamma \vdash_S \ e : \sigma$ (Completeness)

Syntactical Rule System

$$\frac{x : \sigma \in \Gamma \quad \tau \sqsubseteq \sigma}{\Gamma \vdash_S x : \tau} \ [\mathtt{Var}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \to \tau' \qquad \Gamma \vdash_S e_1 : \tau}{\Gamma \vdash_S e_0 \ e_1 : \tau'} \ [\mathtt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_S e : \tau'}{\Gamma \vdash_S \lambda\ x\ .\ e : \tau \to \tau'} \ [\mathtt{Abs}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \qquad \Gamma,\ x : \bar{\Gamma}(\tau) \vdash_S e_1 : \tau'}{\Gamma \vdash_S \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau'} \ [\mathtt{Let}]$$

Generalization

$$\bar{\Gamma}(\tau) = \forall\ \hat{\alpha}\ .\ \tau \qquad \hat{\alpha} = \mathrm{free}(\tau) - \mathrm{free}(\Gamma)$$

While consistency can be seen by decomposing the rules $[\mathtt{Let}]$ and $[\mathtt{Var}]$ of $\vdash_S$ into proofs in $\vdash_D$, it is likely visible that $\vdash_S$ is incomplete, as one cannot show $\lambda\ x.x : \forall\alpha.\alpha \to \alpha$ in $\vdash_S$, for instance, but only $\lambda\ x.x : \alpha \to \alpha$. An only slightly weaker version of completeness is provable [6] though, namely

- $\Gamma \vdash_D \ e : \sigma \Rightarrow \Gamma \vdash_S \ e : \tau \wedge \bar{\Gamma}(\tau) \sqsubseteq \sigma$

implying, one can derive the principal type for an expression in $\vdash_S$ allowing to generalize the proof in the end.

Comparing $\vdash_D$ and $\vdash_S$ note that only monotypes appear in the judgments of all rules, now.

Degrees of freedom instantiating the rules [edit]

Within the rules themselves, assuming a given expression, one is free to pick the instances for (rule) variables not occurring in this expression. These are the instances for the type variable in the rules. Working towards finding the most general type, this choice can be limited to picking suitable types for τ in $[\mathtt{Var}]$ and $[\mathtt{Abs}]$. The decision of a suitable choice cannot be made locally, but its quality becomes apparent in the premises of $[\mathtt{App}]$, the only rule, in which two different types, namely the function's formal and actual parameter type have to come together as one.

Therefore, the general strategy for finding a proof would be to make the most general assumption ($\alpha \notin free(\Gamma)$) for τ in $[\mathtt{Abs}]$ and to refine this and the choice to be made in $[\mathtt{Var}]$ until all side conditions imposed by the $[\mathtt{App}]$ rules are finally met. Fortunately, no trial and error is needed, since an effective method is known to compute all the choices, Robinson's Unification in combination with the so-called Union-Find algorithm.

To briefly summarize the union-find algorithm, given the set of all types in a proof, it allows one to group them together into equivalence classes by means of a $\mathtt{union}$ procedure and to pick a representative for each such class using a $\mathtt{find}$ procedure. Emphasizing on the word procedure in the sense of side effect, we're clearly leaving the realm of logic to prepare an effective algorithm. The representative of a $\mathtt{union}(a, b)$ is determined such, that if both a and b are type variables the representative is arbitrarily one of them, while uniting a variable and a term, the term becomes the representative. Assuming an implementation of union-find at hand, one can formulate the unification of two monotypes as follows:

```
unify(ta,tb):
  ta = find(ta)
  tb = find(tb)
  if both ta,tb are terms of the form D p1..pn with identical D,n then
    unify(ta[i],tb[i]) for each corresponding ith parameter
  else
  if at least one of ta,tb is a type variable then
    union(ta,tb)
  else
    error 'types do not match'
```

Algorithm W [edit]

The presentation of

Algorithm W

Algorithm W as shown in the side box does not only deviate significantly from the original[4] but is also a gross abuse of the notation of logical rules, since it includes side effects. It is legitimized here, for allowing a direct comparison with $\vdash_S$ while expressing an efficient implementation at the same time. The rules now specify a procedure with parameters Γ, e yielding τ in the conclusion where the execution of the premises proceeds from left to right. Alternatively to a procedure, it could be viewed as an attributation of the expression.

$$\frac{x : \sigma \in \Gamma \quad \tau = inst(\sigma)}{\Gamma \vdash_W x : \tau} \quad [\mathtt{Var}]$$

$$\frac{\Gamma \vdash_W e_0 : \tau_0 \quad \Gamma \vdash_W e_1 : \tau_1 \quad \tau' = newvar \quad unify(\tau_0,\ \tau_1 \to \tau')}{\Gamma \vdash_W e_0\ e_1 : \tau'} \quad [\mathtt{App}]$$

$$\frac{\tau = newvar \quad \Gamma,\ x : \tau \vdash_W e : \tau'}{\Gamma \vdash_W \lambda\ x\ .\ e : \tau \to \tau'} \quad [\mathtt{Abs}]$$

$$\frac{\Gamma \vdash_W e_0 : \tau \qquad \Gamma,\ x : \bar{\Gamma}(\tau) \vdash_W e_1 : \tau'}{\Gamma \vdash_W \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau'} \quad [\mathtt{Let}]$$

The procedure $inst(\sigma)$ specializes the polytype σ by copying the term and replacing the bound type variables consistently by new monotype variables. '$newvar$' produces a new monotype variable. Likely, $\bar{\Gamma}(\tau)$ has to copy the type introducing new variables for the quantification to avoid unwanted captures. Overall, the algorithm now proceeds by always making the most general choice leaving the specialization to the unification, which by itself produces the most general result. As noted above, the final result τ has to be generalized to $\bar{\Gamma}(\tau)$ in the end, to gain the most general type for a given expression.

Because the procedures used in the algorithm have nearly O(1) cost, the overall cost of the algorithm is close to linear in the size of the expression for which a type is to be inferred. This is in strong contrast to many other attempts to derive type inference algorithms, which often came out to be NP-hard, if not undecidable with respect to termination. Thus the HM performs as well as the best fully informed type-checking algorithms can. Type-checking here means that an algorithm does not have to find a proof, but only to validate a given one.

Efficiency is slightly reduced because the binding of type variables in the context has to be maintained to allow computation of $\bar{\Gamma}(\tau)$ and enable an occurs check to prevent the building of recursive types during $union(\alpha, \tau)$. An example of such a case is $\lambda\ x.(x\ x)$, for which no type can be derived using HM. Practically, types are only small terms and do not build up expanding structures. Thus, in complexity analysis, one can treat comparing them as a constant, retaining O(1) costs.

Original presentation of Algorithm W [edit]

In the original paper,[4] the algorithm is presented more formally using a substitution style instead of side effects in the method above. In the latter form, the side effect invisibly takes care of all places where a type variable is used. Explicitly using substitutions not only makes the algorithm hard to read[*dubious – discuss*], because the side effect occurs virtually everywhere, but also gives the false impression that the method might be costly. When implemented using purely functional means or for the purpose of proving the algorithm to be basically equivalent to the deduction system, full explicitness is of course needed and the original formulation a necessary refinement.

Further topics [edit]

Recursive definitions [edit]

A central property of the lambda calculus is, that recursive definitions are non-elemental, but can instead be expressed by a fixed point combinator. The original paper[4] notes that recursion can realized by this combinator's type $fix : \forall\alpha.(\alpha \to \alpha) \to \alpha$. A possible recursive definitions could thus be formulated as

$$\mathtt{rec}\ v = e_1\ \mathtt{in}\ e_2\ ::= \mathtt{let}\ v = fix(\lambda v.e_1)\ \mathtt{in}\ e_2$$

Alternatively an extension of the expression syntax and an extra typing rule is possible as:

$$\frac{\Gamma, \Gamma' \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, \Gamma' \vdash e_n : \tau_n \quad \Gamma, \Gamma'' \vdash e : \tau}{\Gamma \ \vdash\ \mathtt{rec}\ v_1 = e_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ v_n = e_n\ \mathtt{in}\ e : \tau} \quad [\mathtt{Rec}]$$

where

- $\Gamma' = v_1 : \tau_1,\ \ldots,\ v_n : \tau_n$
- $\Gamma'' = v_1 : \bar{\Gamma}(\ \tau_1\ ),\ \ldots,\ v_n : \bar{\Gamma}(\ \tau_n\ )$

basically merging $\boxed{\texttt{Abs}}$ and $\boxed{\texttt{Let}}$ while including the recursively defined variables in monotype positions where they occur left to the $\texttt{in}$ but as polytypes right to it. This formulation perhaps best summarizes the essence of let-polymorphism.

Notes [edit]

1. ^ Hindley–Milner is DEXPTIME-complete. However, non-linear behaviour only manifests itself on pathological inputs, as such the complexity theoretic proofs by (Mairson 1990) and (Kfoury, Tiuryn & Urzyczyn 1990) came as a surprise to the research community. When the depth of nested let-bindings is bounded—as is the case in realistic programs—Hindley–Milner type inference becomes polynomial.
2. ^ Polytypes are called "type schemes" in the original article.
3. ^ The parametric types $D\ \tau\ \ldots\ \tau$ were not present in the original paper on HM and are not needed to present the method. None of the inference rules below will take care or even note them. The same holds for the non-parametric "primitive types" in said paper. All the machinery for polymorphic type inference can be defined without them. They have been included here for sake of examples but also because the nature of HM is all about parametric types. This comes from the function type $\tau \rightarrow \tau$, hard-wired in the inference rules, below, which already has two parameters and has been presented here as only a special case.

References [edit]

1. ^ Hindley, J. Roger (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". *Transactions of the American Mathematical Society* **146**: 29–60. JSTOR 1995158.
2. ^ Milner, Robin (1978). "A Theory of Type Polymorphism in Programming". *Journal of Computer and System Science (JCSS)* **17**: 348–374. CiteSeerX: 10.1.1.67.5276.
3. ^ Damas, Luis (1985). *Type Assignment in Programming Languages* (PhD thesis). University of Edinburgh (CST-33-85).
4. ^ [a] [b] [c] [d] [e] Damas, Luis; Milner, Robin (1982). *Principal type-schemes for functional programs* (PDF). 9th Symposium on Principles of programming languages (POPL'82). ACM. pp. 207–212.
5. ^ Clement (1986). *A Simple Applicative Language: Mini-ML*. LFP'86. ACM.
6. ^ Vaughan, Jeff (July 23, 2008) [May 5, 2005]. "A proof of correctness for the Hindley–Milner type inference algorithm" (PDF). Archived from the original (PDF) on 2012-03-24.

- Mairson, Harry G. (1990). "Deciding ML typability is complete for deterministic exponential time". *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '90 (ACM): 382–401. doi:10.1145/96709.96748. ISBN 0-89791-343-4.
- Kfoury, A. J.; Tiuryn, J.; Urzyczyn, P. (1990). "ML typability is dexptime-complete". *Lecture Notes in Computer Science*. CAAP '90 **431**: 206–220. doi:10.1007/3-540-52590-4_50.

External links [edit]

- A literate Haskell implementation of Algorithm W along with its source code on GitHub.

Categories: Type systems | Type theory | Type inference | Lambda calculus | Theoretical computer science | Formal methods | 1969 in computer science | 1978 in computer science | 1985 in computer science | Algorithms