Notes on types

Author: **Amitabha Sanyal**

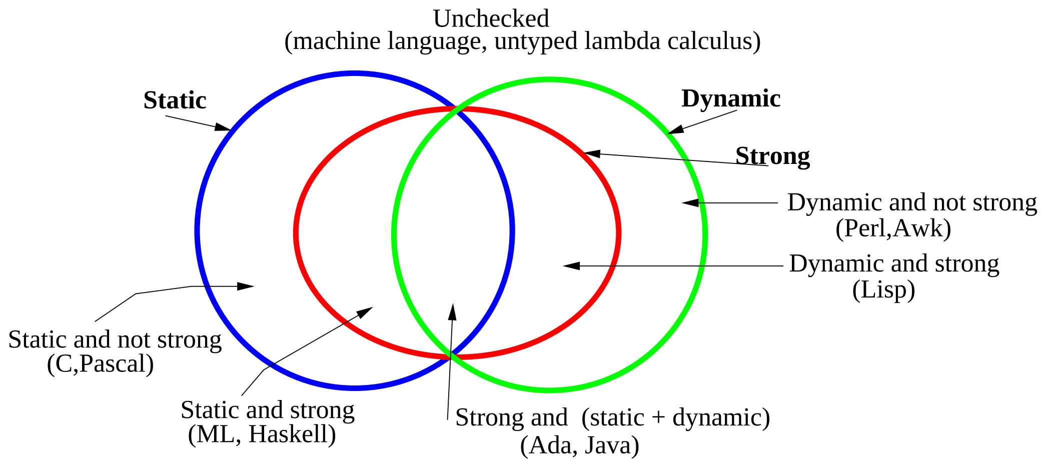
Last updated: 28th September 2001

Abstract

In this report we discuss the basic issues related to type checking and type inferencing. After introducing the framework for presenting type systems, we prsent type systems for two languages $\lambda \rightarrow$ Curry and Milner's languge ($\lambda \rightarrow$ Curry + let). We then present a type inferencing algorithm for the second language. This algorithm forms the core of all type inferencing algorithms based on the Hindley-Milner type system.

1 Introduction

To see the place of type systems in programming languages, look at the diagram given below:



Every programming language definition includes a list of *forbidden errors*. Examples of forbidden errors are adding a pair of booleans and accessing an array element beyond its bound.

A programming lnguage may take one of the following stands with respect to forbidden errors.

1. It may check for and prevent *all* forbidden errors. Then it is called a *strongly typed language*. Further, it may do so statically (at compile time like ML and Haskell), dynamically (at runtime like LISP), or both statically and dynamically (as in Ada and Java).
2. It may check for only a proper subset of forbidden errors. This is the case with languages like C and Pascal. Both these languages consider out-of-bounds array access as a forbidden error. However, both these languages are statically checked and ignore out-of-bounds accesses that occur at run time.
3. Finally there are languages like untyped lambda calculus which have no forbidden errors and consequently do not do any checking.

Type systems are concerned with languages which perform static checking. The algorithm doing so is called a *typechecker*. A typechecker is based on certain rules given by a *type system*.

2 Type Systems

So what does a type system consist of?

1. A set of terms (or programs, we shall use the two words interchangeably) which are to be typed.
2. A set of type expressions to describe the types of the above terms.
3. Assignment or type rules which enable us to make judgements such as: *a program M has the type σ.*

With a type system in place, one can define two problems:

1. Prove that a program M has the type σ under the type system. This is the *type checking* problem.
2. Find out a type σ for the program M under the type system. This is the *type inferencing* problem.

In general, the second problem, is harder than the first. Now let us look at the three components of a type system in turns.

2.1 Terms

This is essentially the programming language whose programs we want to type check. Eventually we want to type a reasonable subset of Haskell. But, to start with, we shall attempt to type λ-terms augmented with constants. Thus our terms are given by the following grammar:

$M \rightarrow var \mid constant \mid \lambda var.M \mid M_1\ M_2$

We shall use x, y and z to range over variables (var) and c to range over constants. M and N shall range over terms. This language is called $\lambda \rightarrow$ Curry.

2.2 Types

The language of types is given by the following grammar:

$T = V \mid C \mid T \to T \mid \chi\ T\ T \mid (T)$ *(monomorphic types)*
$\Sigma = T \mid \forall V.\Sigma$ *(polymorphic types)*

This says that a monomorphic type is either a type variable (V), a constant type C, a function type $T \to T$ or a type constructor χ (like *List*) applied to type expressions. Whereas a polymorphic type is either a monomorphic type or a type with quantified type variables. Note that the quantifiers appear at the outermost level of a type expression.

We shall use α, β to range over type variables, τ to range over monomorphic types and σ to range over polymorphic types. Further, we shall assume that $\to$ associates to the right.

Let us take some examples. Examples of monomorphic types are Int, α, $\alpha \to Int$ and $\alpha \to \beta$. Examples of polymorphic types which are not monomorphic are $\forall\alpha.\alpha$, $\forall\alpha.\alpha \to Int$ and $\forall\alpha\forall\beta.\alpha \to \beta$.

What does a type like $\forall\alpha.\alpha \to Int$ really mean? If f has this type, then it can be used in a context which requires a value of the type $Int \to Int$ or $Bool \to Int$ or $(List\ Int) \to Int$. We can summarize this by saying that f can be used in context which requires the type $\alpha \to Int$.

The difference between a monomorphic type like $\alpha \to Int$ and a polymorphic type like $\forall\alpha.\alpha \to Int$ is often a source of confusion. Read $\alpha \to Int$ as the type of a function from *some specific* type α (whose details are not important) to Int. Whereas $\forall\alpha.\alpha \to Int$ is the type of functions which can take values of *any* concrete type as argument and return a Int as result. $\forall\alpha.\alpha \to Int$ can be "instantiated" to a monomorphic type, say $Int \to Int$, whereas $\alpha \to Int$ cannot. Also, a type like $\alpha \to Int$ is used during the process of reasoning about types and *is not the final type of any term.* As an analogy, you might also like to think of the difference between the terms $\lambda x.x + 2$ and $x + 2$.

The use of type expressions is illustrated by type checking a term such as $foldr\ Cons\ [\,]\ [1, 2, 3]$. First $foldr$ has the polymorphic type $\forall a \forall b.\ (a \to b \to b) \to b \to List\ a \to b$. Instantiating a to Int and b to $List\ Int$, we see that $foldr$ can work in the context $(Int \to (List\ Int) \to (List\ Int)) \to (List\ Int) \to List\ Int \to (List\ Int)$. Similarly, $Cons$ has the polymorhic type $\forall a.a \to (List\ a) \to (List\ a)$, which can work in the context of $Int \to (List\ Int) \to (List\ Int)$. Finally $[\,]$ has the type $\forall a.\ List\ a$, which can work in the context of $List\ Int$. So $foldr\ Cons\ [\,]\ [1, 2, 3]$ is a proper application.

As another example, consider $foldr\ Cons$. In this case, we instantiate $foldr$ to $(a \to (List\ a) \to (List\ a)) \to (List\ a) \to (List\ a) \to (List\ a)$ and $Cons$ to $(a \to (List\ a) \to (List\ a)$. Therefore $(foldr\ Cons)$ has the monomorphic type $(List\ a) \to (List\ a) \to (List\ a)$. We can also say that since no assumption was made regarding the type of a, $(foldr\ Cons)$ also has the polymorphic type $\forall a.(List\ a) \to (List\ a) \to (List\ a)$

Also note that the type variables in a Haskell type expression are implicitly quantified. Thus the type of $foldr$, expressed in Haskell as $(a \to b \to b) \to b \to List\ a \to b$, is in our notation, $\forall a \forall b.\ (a \to b \to b) \to b \to List\ a \to b$.

2.3 Type rules

Type rules allow us to make judgment of the form $\Gamma \vdash M :: \sigma$. Read this as – *from the set of assumptions* Γ *it can be judged that* M *is of the type* σ. An assumption is of the form $x :: \sigma$ or $c :: \sigma$, and is to be read as – *the variable* x *(or constant* c*) is of the type* σ. Thus the judgment

$$\{x :: \alpha\} \vdash \lambda y.x :: \forall\beta.\beta \to \alpha$$

says that from the set containing the sole assumption $x :: \alpha$, one can judge that $\lambda y.x$ has the type $\forall\beta.\beta \to \alpha$.

Why do we need assumptions? Assumptions are akin to symbol tables used by compilers. For example, a compiler extracts information from the declaration part of a program, and puts it in the symbol table. Subsequently it uses the symbol table to process the imperative part. Similarly, here we use assumptions to collect information about the 'lambdas' and use this information to process the 'bodies'. As an example, to show that

$$\{\} \vdash \lambda x \lambda y.x :: \forall\alpha\forall\beta.\alpha \to \beta \to \alpha$$

we would have show as an intermediate step that, assuming that x has the type α, $\lambda y.x$ has the type $\beta \to \alpha$. This is represented as:

$$\{x :: \alpha\} \vdash \lambda y.x :: \beta \to \alpha$$

Another reason for having the assumption set is to collect assumptions regarding constants, built-in functions and library functions. A typical type judgment in such a situation might be:

$$\{+ :: Int \to Int \to Int, 1 :: Int\} \vdash \lambda x.x + 1 :: Int \to Int$$

If J_i are judgments, then a type rule has one of two possible forms:

$$J$$

– This is read as *the judgement* J *can be inferred unconditionally.*

$$\frac{J_1 \qquad J_2 \qquad \dots \qquad J_n}{J}$$

Which is read as: *from the judgments* J_i*, we can infer the judgment* J.

We shall now have a look at the type rules of the language discussed earlier.

3 Type rules for $\lambda \to$ Curry

$$\Gamma \cup \{x :: \sigma\} \vdash x :: \sigma \qquad \text{(Var)}$$

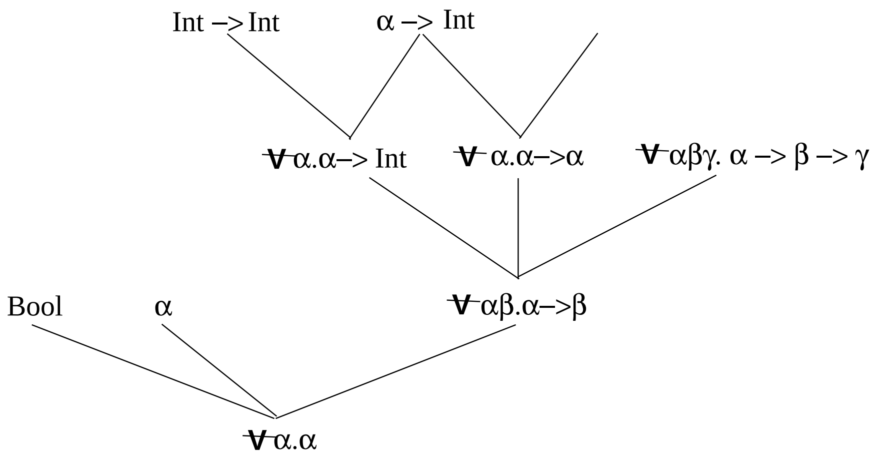
The first rule (named VAR) says that if the assumption $\{x :: \sigma\}$ is already present in the assumption set, then we can have this fact as conclusion. We have similarly,

$$\Gamma \cup \{c :: \sigma\} \vdash c :: \sigma \qquad \text{(Con)}$$

The next rule that we are about to introduce, would allow us to make inferences of the form

$$\frac{\Gamma \vdash x :: \forall\alpha.\alpha \rightarrow \alpha}{\Gamma \vdash x :: Int \rightarrow Int}$$

The type $Int \rightarrow Int$ is called a *generic instance* of $\forall\alpha.\alpha \rightarrow \alpha$. Intuitively, σ' is a generic instance of σ if σ can be used in any context in which σ' can be used. In the lattice diagram shown below, $\sigma \leq \sigma'$ iff σ' is a generic instance of σ.



Now for the formal definition of generic instances. First a substitution is a pair consisting of a type variable and a monomorphic type expression, and is denoted by α/τ. A substitution list S is a set of such substitutions $\{\alpha_1/\tau_1 \ldots \alpha_n/\tau_n\}$. A substitution list S applied on a type expression σ, denoted by $S\ \sigma$ involves simultaneous substitution of the variables $\alpha_1 \ldots \alpha_n$, if they occur free in σ, by the corresponding type expressions $\tau_1 \ldots \tau_n$.

Let $\sigma = \forall\alpha_1 \ldots \alpha_m.\tau$ and $\sigma' = \forall\beta_1 \ldots \beta_n.\tau'$. Then σ' is a generic instance of σ, iff there is a substitution S acting only on $\{\alpha_1 \ldots \alpha_m\}$ such that $\tau' = S(\tau)$ and no β_i is free in σ.

We are now in a position to give the next rule:

$$\frac{\Gamma \vdash M :: \sigma \qquad \sigma' \geq \sigma}{\Gamma \vdash M :: \sigma'} \qquad \text{(Inst)}$$

Clearly, the restriction that no β_i is free in σ is needed, else we would have absurdities like $\alpha \rightarrow Int \leq \forall\alpha.\alpha \rightarrow Int$.

To illustrate the next rule, we shall consider a proof of $\forall a \forall b.(a-b)(a+b) = a^2 - b^2$. To prove this, we consider two *arbitrary* variables a and b, where *arbitrary* means that we do not ascribe any particular properties to these variables. Now we do the usual calculations:

$(a-b)(a+b)$
$a^2+ab-ba-b^2$
$\ldots$
a^2-b^2

We then say that since nothing was assumed about a and b, we have in effect proved that $\forall a \forall b.(a-b)(a+b)=a^2-b^2$. In a similar vein, our next rule allows us to add quantifiers to a judged type.

If a type variable α occurs free within a type expression σ, we denote it as $\alpha \in FV(\sigma)$. Further $\alpha \in FV(\Gamma)$, if $\alpha \in FV(\sigma)$ for some σ occurring in Γ.

Now for the next rule, which is called GEN, standing for generalization:

$$\frac{\Gamma \vdash M :: \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash M :: \forall \alpha.\sigma} \tag{Gen}$$

Our final two rules relate to abstraction and application and are called M-ABS and M-APP. The prefix M stands for monomorphic. Recall that τ ranges over monomorphic types and σ ranges over polymorphic types.

$$\frac{\Gamma \vdash M :: \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N :: \tau_1}{\Gamma \vdash M\ N :: \tau_2} \tag{M-App}$$

$$\frac{\Gamma, x :: \tau_1 \vdash M :: \tau_2}{\Gamma \vdash \lambda x.M :: \tau_1 \rightarrow \tau_2} \tag{M-Abs}$$

We shall now look at a series of examples. In each example, the judgment on a particular line follows from the judgment on the next line using the rule specified between the two lines.

Example 1:
$\{\} \vdash \lambda x.x :: \forall \alpha.\alpha \rightarrow \alpha$
GEN
$\{\} \vdash \lambda x.x :: \alpha \rightarrow \alpha$
M-ABS
$\{x :: \alpha\} \vdash x :: \alpha$
VAR

Example 2:
$\{\} \vdash \lambda xyz.x\ z(y\ z) :: \forall \alpha\beta\gamma.(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$
GEN (3 times)
$\{\} \vdash \lambda xyz.x\ z(y\ z) :: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$
M-ABS (3 times)
$\{x :: \alpha \rightarrow \beta \rightarrow \gamma, y :: \alpha \rightarrow \beta, z :: \alpha\} \vdash x\ z(y\ z) :: \gamma$
M-APP
$\{x :: \alpha \rightarrow \beta \rightarrow \gamma, y :: \alpha \rightarrow \beta, z :: \alpha\} \vdash x\ z :: \beta \rightarrow \gamma$ and
$\{x :: \alpha \rightarrow \beta \rightarrow \gamma, y :: \alpha \rightarrow \beta, z :: \alpha\} \vdash (y\ z) :: \beta$

We shall consider the first conjunct only:
M-APP
$\{x :: \alpha \to \beta \to \gamma, y :: \alpha \to \beta, z :: \alpha\} \vdash x :: \alpha \to \beta \to \gamma$ and
$\{x :: \alpha \to \beta \to \gamma, y :: \alpha \to \beta, z :: \alpha\} \vdash z :: \beta$
VAR (once for each conjunct)

Example 3:
$\lambda x.x\ x$ cannot be typed. The reason is: assume that the type of the second occurrence of x is α. Then the type of the first occurrence of x is $\alpha \to \beta$. Since the types of the two occurrences of x must be the same, the type of the second occurrence now is $\alpha \to \beta$ and that of the first is $\alpha \to \beta \to \gamma$ and so on.

Example 4:
$\{\} \vdash \lambda x.x :: \mathit{Int} \to \mathit{Int}$
M-ABS
$\{x :: \mathit{Int}\} \vdash x :: \mathit{Int}$
VAR
The point of the above example is to show that the type system allows more than one type judgments $\forall\alpha.\alpha \to \alpha$ and $\mathit{Int} \to \mathit{Int}$ for the same term $\lambda x.x$. However, it is desirable that a type-inferencing algorithm should return a unique type (the principal type) for each term.

Example 4:
In this example we shall try to type the term $\lambda fxy.(f\ x, f\ y)$. Remember that $(f\ x, f\ y)$ is actually $\mathit{Pair}\ (f\ x)\ (f\ y)$. We shall use the notation (,) both for the type constructor Tuple as well as the data constructor Pair. We shall try to show the judgment
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)\} \vdash \lambda fxy.(f\ x, f\ y) :: \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta,\beta)$
GEN
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)\} \vdash \lambda fxy.(f\ x, f\ y) :: (\alpha \to \beta) \to \alpha \to \alpha \to (\beta,\beta)$
M-ABS
$\boxed{\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ x, f\ y) :: (\beta,\beta)}$
M-APP
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (\ ,\ )\ (f\ x) :: \beta \to (\beta,\beta)$ and
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ y) :: \beta$
Once again we shall prove the first conjunct, which is more interesting.
M-APP
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (\ ,\ ) :: \beta \to \beta \to (\beta,\beta)$ and
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ x) :: \beta$
the first conjunct is proved by:
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (\ ,\ ) :: \beta \to \beta \to (\beta,\beta)$
INST
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)$
VAR

and the second conjunct by M-APP
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash f :: \alpha \to \beta$ and
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash x :: \alpha$
Both of which are proved by VAR.

It can be figured out that $\forall \alpha\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta,\beta)$ is the most general type of the term $\lambda fxy.(f\ x, f\ y)$. In this, the types of the arguments x and y are forced to be identical. This makes a seemingly sensible term like

$(\lambda fxy.(f\ x, f\ y))\ (\lambda x.x)\ \ 3\ \ True$

ill-typed under this type system!

Notice carefully the source of the problem. This type system forces one to judge the type of a lambda body from *monomorphic type assumptions regarding lambda bound variables.* Thus *all occurrence of the lambda variable in the body are forced to have the same monomorphic type.* This is illustrated by the boxed step. The type of (fx, fy) is being judged from the assumption $f :: \alpha \to \beta$. Thus both x and y are forced to have the same type α.

To fix this problem, we first have to change the language of types

$T = V \mid C \mid T \to T \quad (monomorphic\ types)$
$\Sigma = T \mid \forall V.\Sigma \mid \Sigma \to \Sigma \quad (polymorphic\ types)$

Notice that we now permit quantifiers at inner levels of a type expression. Thus $\forall \beta\gamma.(\forall \alpha.\alpha \to \alpha) \to \beta \to \gamma \to (\beta,\gamma)$ is now a valid type expression. We also introduce the following two rules instead of M-ABS and M-APP.

$$\frac{\Gamma \vdash M :: \sigma_1 \to \sigma_2 \qquad \Gamma \vdash N :: \sigma_1}{\Gamma \vdash M\ N :: \sigma_2} \qquad \text{(P-App)}$$

$$\frac{\Gamma, x :: \sigma_1 \vdash M :: \sigma_2}{\Gamma \vdash \lambda x.M :: \sigma_1 \to \sigma_2} \qquad \text{(P-Abs)}$$

Example 5:
Let us now show the following type judgment:
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta)\} \vdash \lambda fxy.(f\ x, f\ y) :: \forall \beta\gamma.(\forall \alpha.\alpha \to \alpha) \to \beta \to \gamma \to (\beta,\gamma)$
GEN (3 times) followed by P-ABS (3 times) gives
$\boxed{\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall \alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash (f\ x, f\ y) :: (\beta,\gamma)}$
P-APP (2 times)
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall \alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash (\ ,\ ) :: \beta \to \gamma \to (\beta,\gamma)$ and
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall \alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f\ x :: \beta$ and
$\{(\ ,\ ) :: \forall \alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall \alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f\ y :: \gamma$

Notice that if the judgment goes through, the $f\ x$ and $f\ y$ would have different types. We just show that $f\ x :: \beta$. This follows from
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f :: \beta \to \beta$ and
$\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash x :: \beta$

The first conjunct is got by an INST followed by a VAR. The second conjunct directly by a VAR. Observe, once again, in the boxed step of this derivation, that the type of (fx, fy) is being judged from a polymorphic assumption $f :: \forall\alpha\forall\beta.\ \alpha \to \beta$ regarding the lambda variable f.

Exercise:

1. Complete the above judgment and make sure that you understand it.
2. Now derive the following judgment:
   $\{(\ ,\ ) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), 3 :: Int, True :: Bool\} \vdash (\lambda fxy.(f\ x, f\ y))\ (\lambda x.x)\ 3\ True :: (Int, Bool)$

So it would seem that things would be fine if had P-ABS and P-APP instead of M-ABS and M-APP. However, based on the reference (Kfoury, 1989), Barendregt reports that the problem of designing an algorithm based on the above type system is open.

4 Milner's language – $\lambda \to$ Curry + *let*

Though the language discussed so far allows certain functions to be judged polymorphic, the M-APP and M-ABS rules inhibit usage of this polymorphism. To get around this problem, we introduce the *let* and the *letrec* expressions.

4.1 *let* and *letrec*

The *let* expression introduces a local environment. The expression

$$\begin{array}{l} let \\ \quad x_1 = e_1 \\ in\ \ e \end{array}$$

is evaluated as follows: First the expression e_1 is evaluated in the environment surrounding the *let* expression. The expression e is then evaluated in a modified environment in which the variable x_1 is bound to this value. Thus assuming the value of x in the surrounding context to be 5, the the value of

$$\begin{array}{l} let \\ \quad x = x + 1 \\ in\ \ 2 * x \end{array}$$

is 12. The *let* expression can also be generalized to

```
let
  $x_1 = e_1$
  $x_2 = e_2$
  $\ldots$
  $x_n = e_n$
in  $e$.
```

This is equivalent to:

```
let
  $x_1 = e_1$
in let
     $x_2 = e_2$
       $\ldots$
          in let
               $x_n = e_n$
             in $e$.
```

We abbreviate the general let as:

```
let
  $x_i = e_i$
in  $e$
```

On the other hand the expression

```
letrec
  $x = x + 1$
in $2 * x$
```

yields an undefined value because the x on the left and right hand side of the = represents the same x, and there is no (integer) value of x satisfying $x = x + 1$. *letrec* are used for recursive definitions such as

```
letrec
  $fact = \lambda n.if\ (n == 0)\ then\ 1\ else\ fact(n-1)$
in $fact\ 5$
```

Note that there is no *let* in the above sense in Haskell. The Haskell *let* is actually a *letrec*. We shall now present the rule for a *let* expression and illustrate it by an example.

$$\frac{\Gamma_0 = \Gamma \qquad \Gamma_i = \Gamma_{i-1} \cup \{x_i :: \sigma_i\} \qquad \Gamma_{i-1} \vdash e_i :: \sigma_i \qquad \Gamma_n \vdash e :: \sigma}{\Gamma \vdash let\ x_i = e_i\ in\ e :: \sigma} \quad \text{(LET)}$$

Let us illustrate the rule with the following judgment:
$\{3 :: Int, True :: Bool\} \vdash let\ id = \lambda x.x\ in\ (id\ 3, id\ True) :: (Int, Bool)$

LET
$\{3 :: Int, True :: Bool\} \vdash \lambda x.x :: \forall\alpha.\alpha \rightarrow \alpha$ and
$\{3 :: Int, True :: Bool, id :: \forall\alpha.\alpha \rightarrow \alpha\} \vdash \ (id\ 3, id\ True) :: (Int, Bool)$

Complete the rest of the proof. Also notice that the parametric polymorphism of *id* is actually being used. Now the rule for *letrec.*

$$\frac{\Gamma \cup \{x_i :: \tau_i\} \vdash e_i :: \tau_i \qquad \Gamma \cup \{x_i :: \sigma_i\} \vdash e :: \sigma \qquad \sigma_i = \forall\alpha_1 \ldots \alpha_n \tau_i \qquad \alpha_i \notin FV(\Gamma)}{\Gamma \vdash letrec\, x_i = e_i\ in\ e :: \sigma} \quad \text{(LETREC)}$$

Read this rule as follows: Assuming a monomorphic type τ_i for x_i, suppose we can judge that e_i too has the same type τ_i. Further, assuming that x_i has the polymorphic type σ_i, where σ_i is an appropriate generalization of τ_i, suppose we can show that e has the type σ. Then the entire *letrec* expression too has the type σ.

5 An Algorithm for type inferencing

We shall now describe a type inferencing algorithm for Milner's language. Recall that the type inference problem is: Given a program M, find a type σ such that $M :: \sigma$. To explain the type inferencing algorithm, we first need to explain *unification.* We shall therefore make a brief detour to present the unification algorithm.

To see why unification is needed, consider once again the description of the letrec rule – *Assuming a monomorphic type τ_i for x_i, if we can show that e_i too has the type τ_i ....* The work of guessing the right type τ_i for x_i, which would seem to require an oracle, is actually done by unification.

5.1 Unification

A *term*[1] is either a *constant* (denoted by a, b, c ...), a *variable* (denoted by x, y, z ...) or an entity of the form $f(t_1, t_2, \ldots t_n)$ where f is a n-ary function symbol (other function symbols will be denoted by g, h) and each t_i is a term. In the specific context of type checking, terms are made of type variables (α, β) and type constants (Int, $Bool$), and the role of function symbols is played by type constructors like $\rightarrow$ and $List$. So example of a term is $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow List\ a \rightarrow b$.[2].

Once again, a substitution is a pair consisting of a variable x and a term t, and is denoted by x/t. A substitution list θ is a set of such substitutions $\{x_1/t_1 \ldots x_n/t_n\}$. A substitution list θ applied to a term t, denoted by $\theta\ t$, involves simultaneous substitution of the variables $x_1 \ldots x_n$, if they occur in t, by the corresponding type expressions $t_i \ldots t_n$. For example if $\theta = \{x/h(d), y/d\}$ and $t = f(x, g(y), c)$, then $\theta\ t = f(h(d), g(d), c)$.

[1]Here we are talking about terms in general, and not necessarily in the sense used earlier.
[2]Here we are using $\rightarrow$ in a infix fashion in contrast to the general description of terms where function symbols (like f) were shown as being prefix.

If θ is a substitution list and t_1 and t_2 are terms such that $\theta\, t_1 = \theta\, t_2$, then θ is called an *unifier* of t_1 and t_2. For example, $\{x/a, y/c, z/b, w/c\}$ is an unifier of $f(x,b,y)$ and $f(a,z,w)$.

A unifier σ for a pair of terms is also a *most general unifier* (mgu), if for any other unifier σ_1 for the same terms, there exists a substitution list ρ such that $\sigma_1 = \rho\ \sigma$, where $\rho\ \sigma$ is an obvious extension of application of a substitution to another substitution list instead of a term. $\{x/a, y/w, z/b\}$ is a mgu of the terms $f(x,b,y)$ and $f(a,z,w)$.

The Unification Algorithm
Input: Two terms t_1 and t_2.
Output: The mgu, if it exists, else an error message

```
1. $mgu = \{\}$; $ws = \{< t1, t2 >\}$
2. While $ws$ is not empty do
   (a) Remove a pair $\{< t, t' >\}$ from $ws$.
   (b) Consider the following cases:
   case 1. If $t$ and $t'$ are identical variables or constants, do nothing.
   case 2. If t is a variable not occurring in $t'$
           Replace all terms $t''$ in $ws$ and in $mgu$ by $\{t/t'\}\ t''$.
           $mgu = mgu \cup \{< t, t' >\}$
   case 3. If $t'$ is a variable not occurring in $t$
           Replace all terms $t''$ in $ws$ and in $mgu$ by $\{t'/t\}\ t''$.
           $mgu = mgu \cup \{< t', t >\}$
   case 4. if $t = f(t_{11}, \ldots t_{1n})$ and $t' = f(t'_{11}, \ldots t'_{1n})$, then
           $ws = ws \cup \{< t_{11}, t'_{1n} > \ldots < t_{11}, t'_{1n} >\}$.
   case 5. Raise an error.
```

We now give some examples to illustrate the unification algorithm.

Example 1: Unify $f(g(x),\ h(b,\ g(h(c,\ d))),\ y)$ and $f(g(h(w,\ y)),\ x,\ g(z))$.

We show below the values of the variables mgu and ws for each iteration of the while loop.

Iteration 1: $mgu = \{\}$
$ws = \{< f(g(x),\ h(b,\ g(h(c,\ d))),\ y),\ f(g(h(w,\ y)),\ x,\ g(z)) >\}$

Iteration 2: $mgu = \{\}$
$ws = \{< g(x),\ g(h(w,\ y)) >,\ < h(b,\ g(h(c,\ d))),\ x) >,\ < y,\ g(z) >\}$

Iteration 3: $mgu = \{\}$
$ws = \{< x,\ h(w,\ y) >,\ < h(b,\ g(h(c,\ d))),\ x) >,\ < y,\ g(z) >\}$

Iteration 4: $mgu = \{< x,\ h(w,\ y) >\}$
$ws = \{< h(b,\ g(h(c,\ d))),\ h(w,\ y)) >,\ < y,\ g(z) >\}$

Iteration 5: $mgu = \{< x,\ h(w,\ y) >\}$
$ws = \{< b,\ w >, < g(h(c,d)),\ y >,\ < y,\ g(z) >\}$

Iteration 6: $mgu = \{< x,\ h(b,y) >,\ < w,b >\}$
$ws = \{< g(h(c,d)),\ y >,\ < y,\ g(z) >\}$

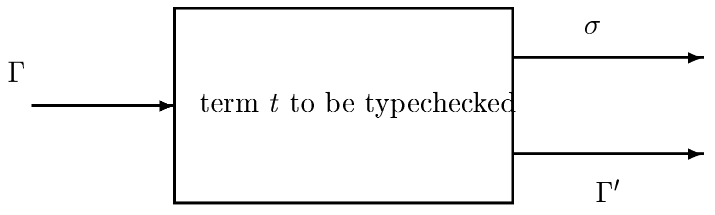
Iteration 7: $mgu = \{< x,\ h(b,\ g(h(c,d))) >,\ < w,b >,\ < y,\ g(h(c,d)) >\}$
$ws = \{< g(h(c,d)),\ g(z) >\}$

Iteration 8: $mgu = \{< x,\ h(b,\ g(h(c,d))) >,\ < w,b >,\ < y,\ g(h(c,d)) >\}$
$ws = \{< h(c,d),\ z >\}$

Iteration 9: $mgu = \{< x,\ h(b,\ g(h(c,d))) >,\ < w,b >,\ < y,\ g(h(c,d)) >,\ < z,\ h(c,d) >\}$
$ws = \{\}$

5.2 The algorithm:

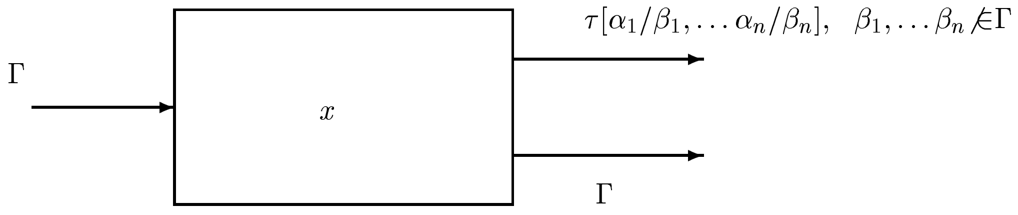
Given a term to be type checked, various inputs that the algorithm requires and the outputs that it produces are given by the following diagram:



Here Γ is called the *type environment* and plays the same role as type assumptions in the type system described earlier. The type checker takes a term t and a type environment Γ and produces as output the type σ of t and a modified environment Γ'. T

We now present the algorithm by a case analysis on the structure of the term. For each case the details of the algorithm will be presented through a diagram and some comments.

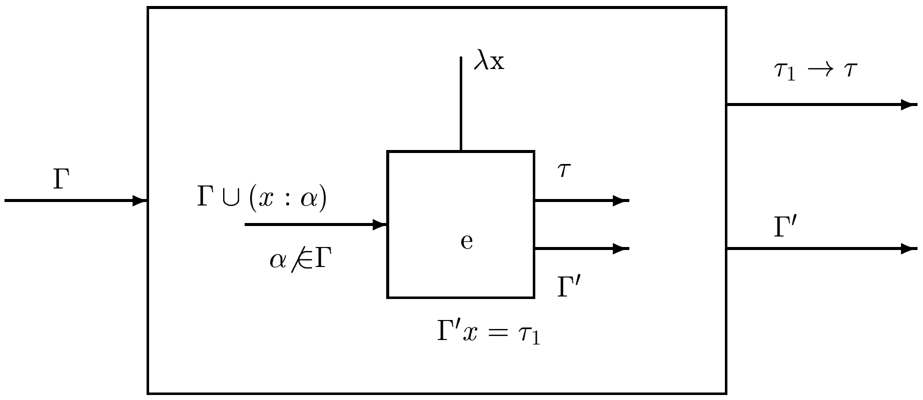
Case 1: t **is a variable** x


$\tau[\alpha_1/\beta_1, \ldots \alpha_n/\beta_n],\quad \beta_1, \ldots \beta_n \not\in \Gamma$
Γ
x
Γ

$\Gamma\ x = \forall \alpha_1, \ldots \alpha_n \cdot \tau$

Here $\beta_1, \ldots, \beta_n$ are fresh variables. The rule says that the type of a variable is found by looking into the environment. The reason for monomorphising the type of x is that we try to find the type of a variable only in the context of an application, and our application is monomorphic.

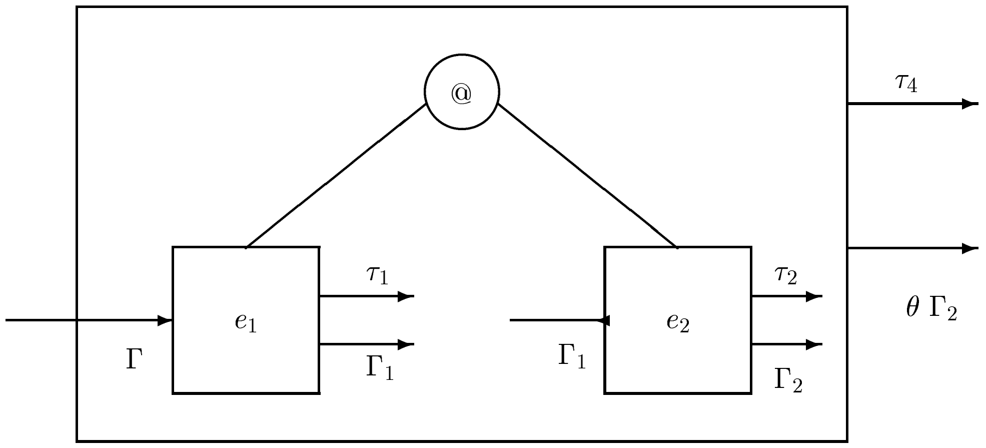
Case 2: t **is a lambda abstraction** $\lambda x.e$



To find the type of $\lambda x.e$ in an environment Γ,

1. typecheck e in an environment Γ augmented with an assumed type α for x. Assume that the result is a type τ and a changed environment Γ'.
2. Let the (refined) type of x in Γ' be τ_1.
3. The type of $\lambda x.e$ is $\tau_1 \to \tau$ and the final environment is Γ'.

Case 3: t **is an application** $(e_1 \ \ e_2)$



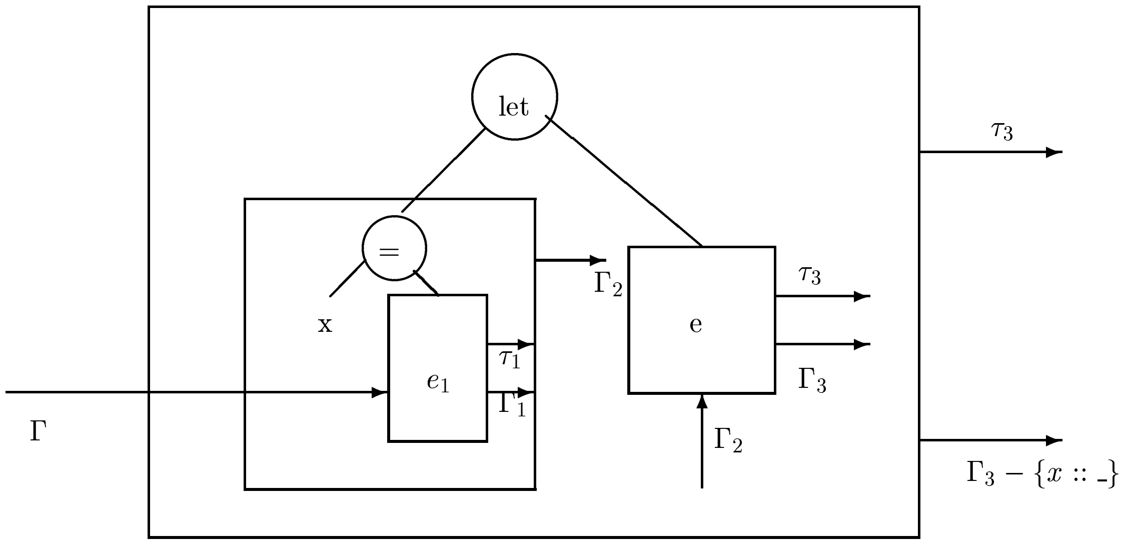
unify $(\tau_1, \tau_2 \to \alpha) = \theta$ and $\theta\ \tau_1 = \tau_3 \to \tau_4$

This rule is to be read as:

1. Typecheck e_1 with the initial environment Γ. Let the result be τ_1 and Γ_1.

2. Typecheck e_2 with the environment Γ_1. Let the result be τ_2 and Γ_2.
3. Unify τ_1 and $\tau_2 \rightarrow \alpha$. Assume that the unifier is θ. And the unified term $(\theta\ \tau_1)$ is $\tau_3 \rightarrow \tau_4$.
4. Then the type of the application is τ_4 and the modified environment is $\theta\ \Gamma_2$.

Case 4: t **is a let expression** *let* $x = e_1$ *in* e



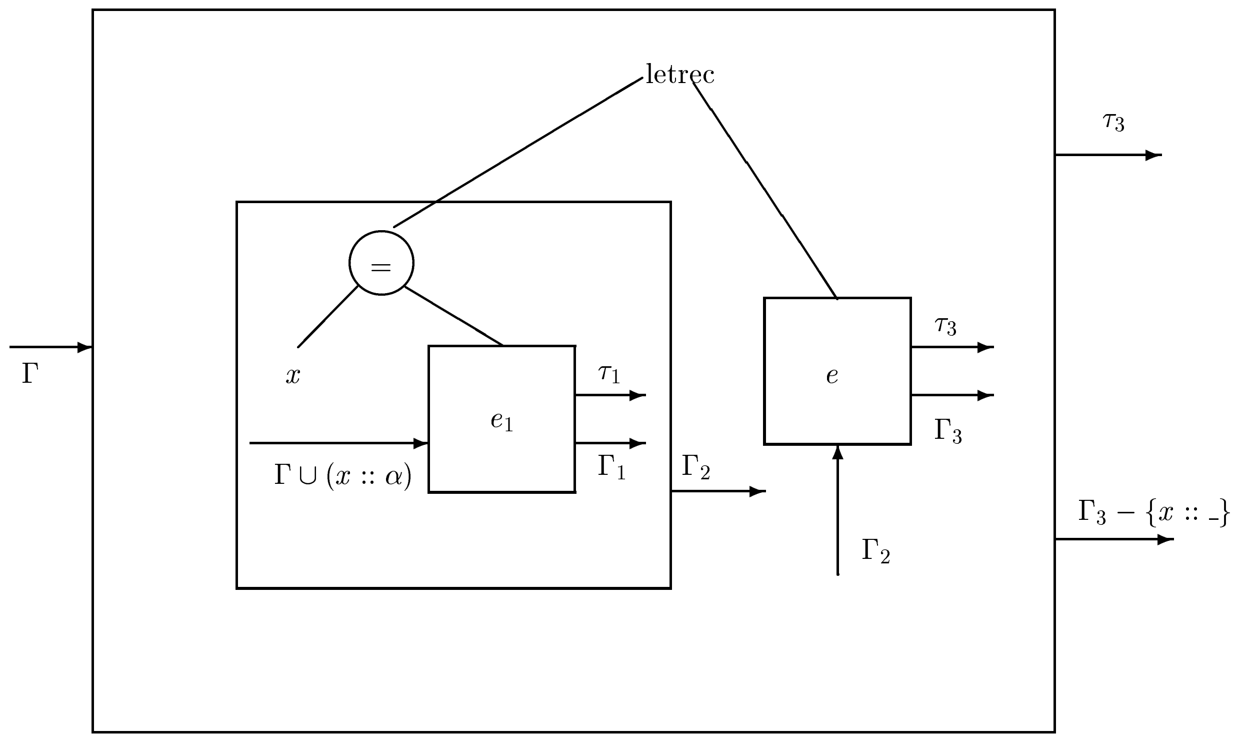
$$\sigma = \forall \alpha_1 \ldots \alpha_n \cdot \tau_1,\ \alpha_1 \ldots \alpha_n \in \tau_1,\ \alpha_1 \ldots \alpha_n \notin \Gamma$$

$$\Gamma_2 = \Gamma_1 \cup (x : \sigma)$$

To typecheck *let* $x = e_1$ *in* e in the environment Γ,

1. Typecheck e_1 in the environment Γ resulting in a type τ_1 and a modified environment Γ_1
2. Let σ be an appropriate polymorphic form of τ_1 and Γ_2 be Γ_1 augmented with the type of x as σ.
3. Typecheck e in the environment Γ_2 resulting in a type τ_3 and a modified environment Γ_3.
4. The type of the let expression is τ_3, and the modified environment is Γ_3 with the type of x deleted.

Case 5: t **is a letrec expression** *letrec* $x = e_1$ *in* e



$$\tau_2 = \Gamma_1\ x, \quad \theta = \text{unify}\ (\tau_1, \tau_2), \quad \tau' = \theta\ \tau_1$$
$$\sigma = \forall \alpha_1 \ldots \alpha_n \cdot \tau', \quad \alpha_1, \ldots, \alpha_n \in \tau', \quad \alpha_1, \ldots, \alpha_n \notin \Gamma$$
$$\Gamma_2 = (\theta\ \Gamma_1) \cup (x : \sigma)$$

The above rule is to be read as follows. To typecheck *letrec* $x = e_1$ *in* e in the environment Γ,

1. Typecheck e_1 in the environment Γ augmented with a type assumption α for the variable x. Assume that this results in a type τ_1 and a modified environment Γ_1
2. Let τ_2 be the refined type of x in Γ_1. Unify this with the type τ_1 of e_1. Let the unifier be θ and the unified type be τ'.
3. Let σ be an appropriate polymorphic form of τ'. Also let Γ_2 be Γ_1 modified taking the unification process into account and further augmented with the type of x as σ.
4. Typecheck e in the environment Γ_2 resulting in a type τ_3 and a modified environment Γ_3.
5. The type of the let expression is τ_3, and the modified environment is Γ_3 with the type of x deleted.