Notes on Types

S. Arun-Kumar

Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi, 110016
email: sak@cse.iitd.ernet.in

April 14, 2002

Chapter 1

The Typed λ Calculus and Type Inferencing in ML

1.1 Typed λ terms

As we have emphasized repeatedly, the λ calculus allows one to define "functions" by λ-abstraction, but in addition also allows the construction of "meaningless functions". In other words, the λ-calculus does not conform to the typical set-theoretic notion of function (with domain and codomain specifications) that is used in mathematics. In particular, this allows the construction of self-referential functions which have no place in mathematics or indeed in any application of mathematics. We rectify that here by setting up systems of the λ-calculus whose terms have expressions for domain and codomain built into their structure.

Definition 1.1 *Given a predefined collection of* **atomic types***, represented by a collection of given symbols (where a typical element is denoted b), we define the language of* **types** *as follows:*

$$\tau ::= b \mid (\tau \to \tau)$$

Each atomic type is intended to denote a particular set (and in the case of programming languages, it also denotes the collection of operations that are available on that set, forming therefore a signature and a structure in ML terminology). These atomic types such as `int` and `bool` actually denote the collections of allowed integers and booleans and their associated operations as given by the structures `Int` and `Bool` respectively.
Each type of the form $\sigma \to \tau$ is a *compound* type and represents the the set of (programmable or computable) functions with domain σ and codomain τ. This language of types is simple enough for us to analyse the concepts of type-checking and type-inferencing for higher order functions in ML-like languages. The "function"-forming operation $\to$ is assume to associate to the right. Hence for typical types α, β and γ, $\alpha \to \beta$ will denote $(\alpha \to \beta)$ and $\alpha \to \beta \to \gamma$ will denote $(\alpha \to (\beta \to \gamma))$ and hence removal of redundant parnetheses should cause no confusion.

Definition 1.2 *Assume X is an infinite collection of typed variables and C is a collection of typed constants*[1]*. Let α and β be typical elements from the language of types given above. The set of* **typed**

[1]Since we assume some basic types are avialable for programming, we also need to assume some constants are available. In the case of the ML-type `int`, there are variety of constants available such as `0`, `+`, `-`, `toString`, `fromString` etc.

λ-terms is defined as follws

$$
\begin{array}{rcll}
L & ::= & c:\alpha & \textit{denotes a typical constant } c \textit{ of type } \alpha \\
 & | & x:\alpha & \textit{denotes a typical variable } x \textit{ of type } \alpha \\
 & | & \lambda x:\alpha[L:\beta]:\alpha\to\beta & \textit{the mathematical notion of a function definition} \\
 & | & (M:\alpha\to\beta \;\; N:\alpha):\beta & \textit{the mathematical notion of function application}
\end{array}
$$

Note

1. We have tried to insert appropriate types so that the language is mathematically consistent with accepted usages of function definitions and function applications.
2. With the imposition of such type constraints in the very definition of the language, it becomes impossible to construct the typed analogues of the untyped terms $\lambda x[(x\ x)]$, $\lambda x[((x\ x)\ x)]$. Hence there is no self-reference in the language.
3. There are no Y or T combinators in the typed calculus. Hence even fairly simple programs and functions such as the factorial function cannot be programmed in this language.
4. Recursion requires the use of names. It is usual in this context to include a **let**-construct to allow for recursive definitions and recursive executions to obtain the full power of computable functions. However, we shall not pursue this much further, knowing full well that languages like ML do have a let-construct.

Example 1.1 *For any type* α, $\mathsf{I}_\alpha \triangleq \lambda x:\alpha[x:\alpha]:\alpha\to\alpha$ *is a typed term. Hence*

- $\mathsf{I}_{\mathtt{int}} \triangleq \lambda x:\mathtt{int}[x:\mathtt{int}]:\mathtt{int}\to\mathtt{int}$ *and*
- $\mathsf{I}_{\mathtt{bool}} \triangleq \lambda x:\mathtt{bool}[x:\mathtt{bool}]:\mathtt{bool}\to\mathtt{bool}$ *and*
- $\mathsf{I}_{\mathtt{int}\to\mathtt{int}} \triangleq \lambda x:\mathtt{int}\to\mathtt{int}[x:\mathtt{int}\to\mathtt{int}]:(\mathtt{int}\to\mathtt{int})\to(\mathtt{int}\to\mathtt{int})$

are all distinct and different terms of the typed λ-calculus.

Definition 1.3 *(***Substitution.***) For typed terms* $M:\alpha$, $N:\beta$ *and a variable* $x:\beta$, $M:\alpha\{N:\beta/x:\beta\}$ *is the term obtained by replacing all free occurrences of* $x:\beta$ *in* $M:\alpha$

Definition 1.4 *(*α**-conversion**$)$

$$\lambda x:\beta[M:\gamma] \equiv_\alpha \lambda y:\beta[M:\gamma\{y:\beta/x:\beta\}] \equiv \lambda y:\beta[M\{y:\beta/x:\beta\}:\gamma]$$

provided $y:\beta \notin FV(M:\gamma)$.

Definition 1.5 Typed β**-reduction** *is the following reduction, closed* [2] *under all typed contexts.*

$$(\lambda x:\alpha[L:\beta]:\alpha\to\beta \;\; M:\alpha) \to_\beta L:\beta\{M:\alpha/x:\alpha\}$$

Note:

1. Typed β-reduction preserves the type of the term.

[2] i.e. we may apply it inside abstraction contexts as well as application contexts as in the case of the untyped calculus, with the only difference that types should match.

2. α-conversion also preserves the type of a term.

3. Hence the type of a term cannot change in the middle of a computation because of either β-reductions or α-conversions.

4. It follows from the structure of the language that it is not possible to do any form of replication by a β-reduction, since all replications in the untyped λ-calculus are due to self-referential terms.

5. More interestingly, all computations are finite and terminating in the typed calculus[3].

6. Since all computations are finite, every typed term has a normal form. Since the type of a term nor the nature of β-reduction changes, the Chirch-Rosser theorem also holds for this language and hence every term has a *unique normal form*.

Clearly, the typed λ-calculus (with recursion combinators) can be a powerful formalism, but it misses out on the essential similarity of the terms of example 1.1. The theory of the typed λ-calculus, therefore essentially tries to split what is intuitively viewed as a single operator into an infinity of different kinds of operators all with the same code but with different types attached to them. It also means that there is going to be costly replication of code in any language which subsribes to the typed calculus.

1.2 Type Assignment to λ terms

In the light of the last paragraph, it would be much more satisfying to have the untyped terms as given and adopt a set of rules and axioms that will assign certain types to certain terms. Most terms that receive types will receive infinitely many of them corresponding to the idea that they represent operations with an infinite number of special cases. This infinite number of special cases will be viewed as a small number of "type-schemes" which will be type expressions containing variables standing for arbitrary types.

Definition 1.6 *Given a predefined collection of* **type constants**, *represented by a collection of given symbols (where a typical element is denoted b), an infinite collection of* **type variables** *(where a typical element is denoted by t), then we define the language of* **type schemes** *as follows:*

$$\tau ::= b \mid t \mid (\tau \to \tau)$$

Definition 1.7 *A* **type assignment formula** *is any expression $L : \alpha$, where L is a term of the* untyped λ*-calculus and α is a type scheme. In such a formula L is the* **subject** *and α is the* **predicate**.

The following natural deduction rules tell us how to infer types for *lambda* abstraction and application. Theformulas of this logical system are all of the form $L : \tau$.

$$(\to -\mathsf{I}) \quad \frac{x : \alpha \vdash L : \beta}{\lambda x[L] : \alpha \to \beta} \qquad x \text{ is not free in any undischarged assumption}$$

$$(\to -\mathsf{E}) \quad \frac{L : \alpha \to \beta, M : \alpha}{(L\ M) : \beta}$$

[3]Note however, that a β-reduction does not necessarily decrease the size of the term, since there may be multiple occurrences of a variable that are substituted by a larger term. In such a case how do you prove that all computations are finite?

Since we are now in the type assignment framework rather than in the typed calculus framework, we need to make sure that terms do not change type due to α-conversion. This rule is called α-invariance.

$$(\equiv_\alpha -\mathsf{inv}) \quad \frac{L : \beta \quad L \equiv_\alpha M}{M : \beta}$$

Example 1.2 *For closed λ terms (i.e. terms which have no free variables) we have the following easily proven type assignments. You are encouraged to use the proof rules to prove these.*

1. $\vdash \mathsf{I} \triangleq \lambda x[x] : \alpha \to \alpha$. *This is the most general assignment of type to* I *with the minimum set of assumptions about the types of its components.*

2. $\vdash \mathsf{I} \triangleq \lambda x[x] : \beta \to \beta$. *This is also the most general assignment of type to* I *with the minimum set of assumptions about the types of its components. However, it may be obtained from the previous assignment by substituting β for α throughout the proof. In other words the substitution $\{\beta/\alpha\}$ throughout the previous proof gives us the proof of the current assignment. This is also most general because it is possible to obtain the previous proof $\vdash \mathsf{I} \triangleq \lambda x[x] : \alpha \to \alpha$ from the current proof $\vdash \mathsf{I} \triangleq \lambda x[x] : \beta \to \beta$ by performing the uniform substitution $\{\alpha/\beta\}$. Hence in these two cases the proofs are obtained from each other by substitutions that are merely* **renaming** *substitutions.*

3. $\vdash \mathsf{I} \triangleq \lambda x[x] : (\alpha \to \alpha) \to (\alpha \to \alpha)$. *There is a valid proof of this but it is clear that this proof may be obtained from the proof of the first case by a type-substitution $\{\alpha \to \alpha/\alpha\}$ throughout the proof. However it is not possible to obtain any of the previous proofs from the proof of this by simply performing substitutions on type-schemes. This is due to the fact that $\{\alpha \to \alpha/\alpha\}$ is not a renaming substitution, and unless it is renaming, a substitution will always yield a proof that is less general (and more specific).*

4. $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\beta \to \alpha)$. *This is again the most general type assignment. In particular, it does not assume that there exists any relationship between the types of x and y.*

5. $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\alpha \to \alpha)$. *The proof of this is less general than that of the previous assignment for* K *since it assumes that x and y are of the same type. Note that the proof of this assignment is obtained from the proof of the previous assignment by the type-substitution $\{\alpha/\beta\}$, which looks suspiciously like a renaming substitution but hides the extra condition that the types of x and y are assumed to be the same. One can easily check out that from the proof of $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\alpha \to \alpha)$ that it is impossible to obtain a proof of $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\beta \to \alpha)$ by merely performing a uniform substitution throughout.*

6. $\vdash \mathsf{K} \triangleq \lambda xy[x] : (\alpha \to \beta) \to (\alpha \to (\alpha \to \beta))$. *This is again a specific type assignment that is less general than $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\beta \to \alpha)$, but it is incomparable with the assignment $\vdash \mathsf{K} \triangleq \lambda xy[x] : \alpha \to (\alpha \to \alpha)$, because it is not possible to obtain a proof of one from the other by performing a uniform substitution.*

Example 1.3 *Here we consider terms that have free variables and terms for which it is impossible to assign a type scheme.*

1. *For open λ terms it requires a collection of assumptions for the free variables before we may assign a type to the term. A term such as $\lambda x[(x\ y)]$ where y is free variable cannot be assigned a type or a type scheme without first making an assumption of the form $y : \alpha$ which will never*

be discharged by any rules. Hence it is then possible to prove $y : \alpha \vdash \lambda x[(x\ y)] : (\alpha \to \beta) \to \beta$ *by adding the assumption* $x : \alpha \to \beta$ *and then discharging by the rule* $(\to -\mathsf{I})$*. Note that this assumption yields the most general possible assignment (subject to renaming substitutions) of a type scheme to this* λ *term.*

2. *There are terms (both closed and open) for which it is impossible to assign a type scheme with just a* single *type assumption for each* **distinct**[4] *variable.*

   (a) $\lambda x[(x\ x)]$*. With just a single assumption of a type scheme assignment for* x *it is impossible to show that* $\lambda x[(x\ x)]$ *can be assigned a type scheme. This is because of the self-referential application of* x *and the fact that the rule for application* $(\to -\mathsf{E})$ *does not permit operator and operand to have the same type scheme.*

   (b) $(x\ x)$ *is similar to the above.*

   (c) *On the other hand, we may prove that* $x : \alpha \to \beta, y : \alpha \vdash (x\ y) : \beta$ *since* x *and* y *are distinct variables.*

   (d) *Note however that* $((f\ x)\ x)$ *may be assigned a type scheme. In other words, it is possible to show that* $x : \alpha, f : \alpha \to (\alpha \to \beta) \vdash ((f\ x)\ x) : \beta$*. Here application is not self-referential.*

With the above proof system, it is possible to partition the terms of the untyped λ-calculus into those for which there exists a type assignment and those for which there is no type assignment. In general it is possible to use the rules (as patterns) and use unification to obtain the following.

1. Unification will always find the most general type assignment. This assignment is also called the **principal type-scheme** for a closed term. All other type-scheme assignments could be found by substitutions.

2. For open terms (which have free variables) it is necessary to build in a set of minimal assumptions of type-schemes such that unification may then find a suitable type-scheme assignment. For this purpose, it is necessary to ensure that distinct variables have single assignments of the form $x : \alpha$ where in each assumption the type-variables are also distinct. The result of unification in such a case is a pair $\langle \Gamma, \alpha \rangle$, where Γ si the set of assumptions made. This is called the **principal pair**, and denotes the fact that α is the principal type-scheme subject to the assumptions made in Γ.

3. Since the unification algorithm always terminates it also separates out terms for which it is impossible to do a type-scheme assignment and returns the message with the appropriate patterns that cannot be unified.

It is rather funny to end a section on a mathematical subject with a definition but here it is summarizing the above.

Definition 1.8 prinicipal type-scheme, p.t.s *Let* L *be a pure* λ *term and let* α *be a type-scheme.*

- *If* L *is is closed (i.e.* $FV(L) = \emptyset$*) then* α *is called a* **p.t.s.** *of* L *iff for every type-scheme* β

$$\vdash L : \beta$$

holds when and only when *β is a substitution-instance of α.*

[4] By *distinct* variables we mean that no amount of α-conversion can identify the two variables and make them logically the same variable. On the other two variables which have the same name could be made different by appropriate α-conversions.

- *If L is open and $FV(L) = \{x_1, \ldots, x_n\}$ for $n > 0$, a pair $\langle \Gamma, \alpha \rangle$ is called a* **principal pair (p.p.)** *of L and α a* **p.t.s.** *of L iff Γ is of the form $\Gamma = \{x_1 : \delta_1, \ldots, x_n : \delta_n\}$ (i.e. there is* exactly *one type assumption for each free variable) and for all Δ, β*

$$\Delta \vdash L : \beta$$

holds when and only when *$\langle \Delta, \beta \rangle$ is a substitution-instance of $\langle \Gamma, \alpha \rangle$.*

Notes

1. We may extend our notions to β-equality of terms as well by adding the following rule.

$$(=_\beta) \quad \frac{L : \alpha,\ L =_\beta M}{M : \alpha}$$

2. Note that it is not necessary to have a rule which says that β-reduction preserves principal types, because types are any way preserved under β-reduction[5]. But they may not in certain cases where β-equality is concerned. Why do you think this can happen?

3. The whole system be extended to include η-reductions, $\beta\eta$-reductions and their correpsonding equalities. Come up with rules for these.

1.3 Polymorphism in ML-like languages

Let us review what we have done so far. In section 1.1, by defining types explicitly, we saw that for each combinator such as I there were an infinite number of explicitly typed combinators I_α, one for each type α. This was found to be unsatisfactory and in section 1.2 we gave to the combinator I the principal type $t \to t$ where t is any type variables which may be assigned any type. In this section we combine the two approaches and give a formal notation for I as well as for every I_α and formal rules relating their type schemes.
This formalization is essential to describe polymorphism in languages such as ML where explicit typing is used (whenever the user desires it explicitly) and otherwise polymorphism is used. The following example
Such a formal system would require a substitution-rule for types so as to allow types variables to be substituted by type-schemes. Further we require a form of generalization over types. We proceed with the formalization as follows.

Definition 1.9 *Assume given some* **type-constants** *and a countably infinite set $\mathbb{T}$ of* **type-variables**. *We then have the language of* **polymorphic type-schemes** *defined as follows*

$$\tau ::= b \mid t \mid (\tau \to \tau) \mid \forall t.\tau$$

An occurrence of a type variable t in a type-scheme τ is said to be **bound** *if it occurs inside a "sub-type-scheme" of the form $\forall t.\alpha$ where α is a type-scheme. Otherwise it is said to be* **free**. *We denote the set of free type-variables of a polymorphic type-scheme τ by $FV(\tau)$. A* **polymorphic type** *is a polymorphic type-scheme in which all type-variables are bound.*

[5]This is a fact that actually needs a proof and is called the property of **subject reduction.**. It is fairly long and complex and we will not give it here.

The language in section 1.2 has been extended by the new construct $\forall t.\tau$ which denotes a generalization over type variables to arbitrary types.

Definition 1.10 *Assume given a countably infinite set $\mathbb{X}$ of* **term-variables** *(disjoint from the set $\mathbb{T}$ of type variables) a finite set $\mathbb{C}$ of* **term-constants***, with each constant $c \in \mathbb{C}$ assigned a type-scheme. Then the set of* **polymorphic λ-terms** *is defined as follows*

L	$::=$	c	*a typical term-constant with a predefined polymorphic type-scheme*
	$\mid$	x	*a typical term-variable (but without a predefined type-scheme)*
	$\mid$	$(L\ L)$	*application for both terms as well as type-schemes*
	$\mid$	$\lambda x : \alpha[L]$	*λ abstraction on terms*
	$\mid$	$(L\ \alpha)$	*application to a type-scheme α*
	$\mid$	$\Lambda a[L]$	*type-abstraction*

$FV(L)$ denotes the set of all free term-vriables and type-variables (they are distinct and therfore there is no confusion between them). $FV(L) \cap \mathbb{X}$ is the set of all free term-variables and $FV(L) \cap \mathbb{T}$ is the set of all free type-variables in a term.

Substitution. Substitution is defined in the usual fashion. However, term-variables may be replaced only by terms and type variables may be replaced only by type-schemes.

Definition 1.11 *(α-***congruence***). A change of bound variables in a type-scheme or a term is any of the following replacements*

$$\begin{array}{lcll} \forall t[\tau] & \equiv_\alpha & \forall u[\tau\{u/t\}] & \text{if } u \notin FV(\tau) \\ \Lambda t[L] & \equiv_\alpha & \Lambda u[L\{u/t\}] & \text{if } u \notin FV(L) \\ \lambda x : \tau[L] & \equiv_\alpha & \lambda y : \tau[L\{y/x\}] & \text{if } y \notin FV(L) \end{array}$$

Definition 1.12 *β-reduction is defined for the terms of this language as follows:*

$$\begin{array}{lcll} (\beta^1) & (\lambda x : \tau[L]M) & \longrightarrow_{\beta^1} & L\{M/x\} \\ (\beta^2) & (\Lambda t[L]\alpha) & \longrightarrow_{\beta^2} & L\{\alpha/t\} \end{array}$$

Note. It is also possible to define η-reductions and $\beta\eta$-reductions in a similar fashion. We leave it as a simple exercise for the reader to define rules η^1 and η^2.
The type assignment system for the polymorphic λ-calculus is defined by the following natural deduction rules.

$$(\to -\mathsf{I}) \quad \frac{x : \alpha \vdash L : \beta}{\lambda x[L] : \alpha \to \beta} \qquad x \text{ is not free in any undischarged assumption}$$

$$(\to -\mathsf{E}) \quad \frac{L : \alpha \to \beta \quad M : \alpha}{(L\ M) : \beta}$$

$$(\forall -\mathsf{I}) \quad \frac{L : \tau}{\Lambda t[L] : \forall t[\tau]} \qquad t \text{ is a type-variable which is not free in any undischarged assumption}$$

$$(\forall -\mathsf{E}) \quad \frac{L : \forall t[\tau]}{(L\sigma) : \tau\{\sigma/t\}} \qquad \beta \text{ is a type-scheme}$$

$$(\equiv_{\alpha^1}) \quad \frac{L : \tau \quad L \equiv_{\alpha^1} M}{M : \tau}$$

$$(\equiv_{\alpha^2}) \quad \frac{L : \tau \quad \tau \equiv_{\alpha^2} \sigma}{L : \sigma}$$

With the above set of rules we can prove formal statements such as

1. $\vdash \mathsf{I} \triangleq \Lambda t[\lambda x : t[x]] : \forall t[t \to t]$
2. $\vdash \mathsf{I}_{\mathtt{int}} \triangleq \lambda x : \mathtt{int}[x] : \mathtt{int} \to \mathtt{int}$
3. $\vdash (\mathsf{I}\ \ \mathtt{int}) : \mathtt{int} \to \mathtt{int}$.
4. $\vdash (\mathsf{I}\ \ \mathtt{int}) \longrightarrow_{\beta^2} \lambda x : \mathtt{int}[x]$ follows by the β-reduction rules and the type of the resulting term can be inferred as $\mathtt{int} \to \mathtt{int}$ independently.