Principal type-schemes for functional programs

Luis Damas* and Robin Milner
Edinburgh University

1. Introduction

This paper is concerned with the polymorphic type discipline of ML, which is a general purpose functional programming language, although it was first introduced as a metalanguage (whence its name) for conducting proofs in the LCF proof system [GMW]. The type discipline was studied in [Mil], where it was shown to be semantically sound, in a sense made precise below, but where one important question was left open: does the type-checking algorithm - or more precisely, the type <u>assignment</u> algorithm (since types are assigned by the compiler, and need not be mentioned by the programmer) - find the most general type possible for every expression and declaration? Here we answer the question in the affirmative, for the purely applicative part of ML. It follows immediately that it is decidable whether a program is well-typed, in contrast with the elegant and slightly more permissive type discipline of Coppo [Cop]. After several years of successful use of the language, both in LCF and other research and in teaching to undergraduates, it has become important to answer these questions - particularly because the combination of flexibility (due to polymorphism), robustness (due to semantic soundness) and detection of errors at compile time has proved to be one of the strongest aspects of ML.

The discipline can be well illustrated by a small example. Let us define in ML the function "map", which maps a given function over a given list - that is,

```
map f [x1;...;xn] = [f(x1);...;f(xn)]
```

The required declaration is

```
letrec map f s = if null s then nil
                 else cons(f(hd s))(map f (tl s))
```

The type-checker will deduce a type-scheme for "map" from existing type-schemes for "null", "nil", "cons", "hd" and "tl"; the term "type-scheme" is appropriate since all these objects are polymorphic. In fact, from

$$\begin{aligned}
\text{null} &: \forall\alpha(\alpha\ \text{list} \to \text{bool})\\
\text{nil} &: \forall\alpha(\alpha\ \text{list})\\
\text{cons} &: \forall\alpha(\alpha \to (\alpha\ \text{list} \to \alpha\ \text{list}))\\
\text{hd} &: \forall\alpha(\alpha\ \text{list} \to \alpha)\\
\text{tl} &: \forall\alpha(\alpha\ \text{list} \to \alpha\ \text{list})
\end{aligned}$$

will be deduced

$$\text{map} : \forall\alpha\forall\beta((\alpha \to \beta) \to (\alpha\ \text{list} \to \beta\ \text{list})).$$

* The work of this author is supported by the Portuguese Instituto Nacional de Investigacao Cientifica.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-065-6/82/001/0207 $00.75

Types are built from type constants (bool,...) and type variables ($\alpha,\beta,\ldots$) using type operators (such as infixed $\rightarrow$ for functions and postfixed "list" for lists); a type-scheme is a type with (possibly) quantification over type variables at the outermost.

Thus, the main result of this paper is that the type-scheme deduced for such a declaration (and more generally, for any ML expression) is a principal type-scheme, i.e. that any other type-scheme for the declaration is a generic instance of it. This is a generalisation of Hindley's result for Combinatory Logic [Hin].

ML may be contrasted with ALGOL 68, in which there is no polymorphism, and with Russell [DD], in which parametric types appear explicitly as arguments to polymorphic functions. The generic types of Ada may be compared with type schemes. For simplicity, our definitions and results here are formulated for a skeletal language, since their extension to ML is a routine matter. For example, recursion is omitted since it can be introduced by simply adding the polymorphic fixed-point operator

$$\text{fix} : \forall\alpha((\alpha\rightarrow\alpha)\rightarrow\alpha)$$

and likewise for conditional expressions.

2. The language

Assuming a set Id of identifiers x, the language Exp of expressions e is given by the syntax

$$e \;::=\; x \mid e\ e' \mid \lambda x.e \mid \underline{\text{let}}\ x = e\ \underline{\text{in}}\ e'$$

(where parentheses may be used to avoid ambiguity). Only the last clause extends the λ-calculus. Indeed, for type checking purposes every let expression could be eliminated (by replacing x by e everywhere in e'), except for the important consideration that in on-line use of ML declarations

$$\underline{\text{let}}\ x = e$$

are allowed, whose scope (e') is the remainder of the on-line session. As illustrated in the introduction, it must be possible to assign type-schemes to identifiers thus declared.

Note that types are absent from the language Exp. Assuming a set of type variables α and of primitive types ι, the syntax of types τ and of typeschemes σ is given by

$$\tau \;::=\; \alpha \mid \iota \mid \tau \rightarrow \tau$$

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

A type-scheme $\forall\alpha_1\ldots\forall\alpha_n\tau$ (which we may write $\forall\alpha_1\ldots\alpha_n\tau$) has generic type variables $\alpha_1,\ldots,\alpha_n$. A monotype μ is a type containing no type variables.

3. Type Instantiation

If S is a substitution of types for type variables, often written $[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n]$ or $[\tau_i/\alpha_i]$, and σ is a type-scheme, then $S\sigma$ is the type-scheme obtained by replacing each free occurrence of α_i in σ by τ_i, renaming the generic variables of σ if necessary. Then $S\sigma$ is called an instance of σ; the notions of substitution and instance extend naturally to larger syntactic constructs containing type-schemes.

By contrast, a type scheme $\sigma = \forall\alpha_1\ldots\alpha_m\tau$ has a generic instance $\sigma' = \forall\beta_1\ldots\beta_n\tau'$ if $\tau' = [\tau_i/\alpha_i]\tau$ for some types $\tau_1,\ldots,\tau_m$, and the β_j are not free in σ. In this case we shall write $\sigma > \sigma'$. Note that instantiation acts on free variables, while generic instantiation acts on bound variables. It follows that $\sigma > \sigma'$

implies $S\sigma > S\sigma'$.

4. Semantics

The semantic domain V for Exp is a complete partial order satisfying the following equations up to isomorphism, where B_i is a cpo corresponding to primitive type ι_i:

$V = B_0 + B_1 + \ldots + F + W$ (disjoint sum)

$F = V \to V$ (function space)

$W = \{.\}$ (error element)

To each monotype μ corresponds a subset of V, as detailed in [Mil]; if $v \in V$ is in the subset for μ, we write $v:\mu$. Further, we write $v:\tau$ if $v:\mu$ for every monotype instance μ of τ, and we write $v:\sigma$ if $v:\tau$ for every τ which is a generic instance of σ.

Now let $Env = Id \to V$ be the domain of environments η. The semantic function $\mathcal{E}: Exp \to Env \to V$ is given in [Mil]. Using it, we wish to attach meaning to assertions of the form

$A \models e:\sigma$

where $e \in Exp$ and A is a set of assumptions of the form $x:\sigma'$, $x \in Id$. If the assertion is <u>closed</u>, i.e. if A and σ contain no free type variables, then the sentence is said to hold iff, for every environment η, whenever $\eta[[x]]:\sigma'$ for each member $x:\sigma'$ of A, it follows that $\mathcal{E}[[e]]\eta:\sigma$. Further, an assertion holds iff all its closed instances hold. Thus, to verify the assertion

$x:\alpha,\ f:\forall\beta(\beta\to\beta) \models (f\ x):\alpha$

It is enough to verify it for every monotype μ in place of α. This example illustrates that free type-variables in an assertion are implicitly quantified over the whole assertion, while explicit quantification in a type scheme has restricted scope.

The remainder of this paper proceeds as follows. First we present an inference system for inferring valid assertions. Next we present an algorithm W for computing a type-scheme for any expression, under assumptions A. We then show that W is <u>sound</u>, in the sense that any type-scheme which it yields is derivable in the inference system. Finally we show that W is <u>complete</u>, in the sense that derivable type-scheme is an instance of that computed by W.

5. Type Inference

From now on we shall assume that A contains at most one assumption about each identifier x. A_x stands for the result of removing any assumption about x from A.

For assumptions A, expression e and type-scheme σ we write

$A \vdash e:\sigma$

if this sentence may be derived from the following inference rules:

TAUT: $A \vdash x:\sigma \qquad (x:\sigma \text{ in } A)$

INST: $\dfrac{A \vdash e:\sigma}{A \vdash e:\sigma'} \qquad (\sigma > \sigma')$

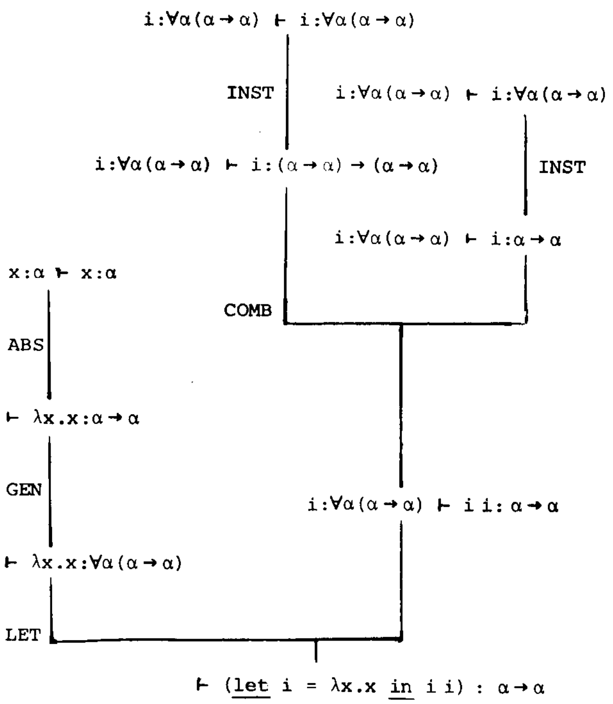
GEN: $\dfrac{A \vdash e:\sigma}{A \vdash e:\forall\alpha\sigma} \qquad (\alpha \text{ not free in } A)$

COMB: $\dfrac{A \vdash e:\tau'\to\tau \quad A \vdash e':\tau'}{A \vdash (e\ e'):\tau}$

ABS: $\dfrac{A_x \cup \{x:\tau'\} \vdash e:\tau}{A \vdash (\lambda x.e):\tau'\to\tau}$

LET: $\dfrac{A \vdash e:\sigma \quad A_x \cup \{x:\sigma\} \vdash e':\tau}{a \vdash (\underline{let}\ x=e\ \underline{in}\ e'):\tau}$

The following example of a derivation is organised as a tree, in which each node follows from those immediately above it by an inference rule.



The following proposition, stating the semantic soundness of inference, can be proved by induction on e .

Proposition 1 (soundness of inference) If $A \vdash e:\sigma$ then $A \models e:\sigma$.

We will also require later the two following properties of the inference system.

Proposition 2 If S is a substitution and $A \vdash e:\sigma$ then $SA \vdash e:S\sigma$. Moreover if there is a derivation of $A \vdash e:\sigma$ of height n then there is also a derivation of $SA \vdash e:S\sigma$ of height less or equal to n.

Proof by induction on n. ⊠

Lemma 1 If $\sigma > \sigma'$ and $A_x \cup \{x:\sigma'\} \vdash e:\sigma_0$ then also $A_x \cup \{x:\sigma\} \vdash e:\sigma_0$.

Proof We construct a derivation of $A_x \cup \{x:\sigma\} \vdash e:\sigma_0$ from that of $A_x \cup \{x:\sigma'\} \vdash e:\sigma_0$ by substituting each use of TAUT for $x:\sigma'$ with $x:\sigma$ followed by an INST step to derive $x:\sigma'$. Note that GEN steps remain valid since if α occurs free in σ then it also occurs free in σ'. ⊠

6. The type assignment algorithm W

The type inference system by itself does not provide an easy method for finding, given A and e, a type-scheme σ such that $A \vdash e:\sigma$. We now present an algorithm W for this purpose. In fact, W goes a step further. Given A and e, if W succeeds it finds a substitution S and a type τ, which are most general in a sense to be made precise below, such that

$$SA \vdash e:\tau$$

To define W we require the unification algorithm of Robinson [Rob].

Proposition 3 (Robinson) There is an algorithm U which, given a pair of types, either returns a substitution V or fails; further

(i) If $U(\tau,\tau')$ returns V , then V unifies τ and τ', i.e. $V\tau = V\tau'$.

(ii) If S unifies τ and τ', then $U(\tau,\tau')$ returns some V and there is another substitution R such that $S = RV$.

Moreover, V involves only variables in τ and τ'.

We also need to define the closure of a type τ with respect to assumptions A ;

$$\bar{A}(\tau) = \forall\alpha_1 \ldots \alpha_n \tau$$

where $\alpha_1,\ldots,\alpha_n$ are the type-variables occurring free in τ but not in A.

Algorithm W

$W(A,e) = (S,\tau)$ where

(i) If e is x and there is an assumption $x:\forall\alpha_1\ldots\alpha_n\tau'$ in A then $S = Id$ and $\tau = [\beta_i/\alpha_i]\tau'$ where the β_i's are new.

(ii) If e is e_1e_2 then
let $W(A,e_2) = (S_1,\tau_2)$
and $W(S_1A,e_2) = (S_2,\tau_2)$
and $U(S_2\tau_1,\tau_2\rightarrow\beta) = V$ where β is new;
then $S = VS_2S_1$ and $\tau = V\beta$.

(iii) If e is $\lambda x.e_1$ then let β be a new type variable
and $W(A_x \cup \{x:\beta\}, e_1) = (S_1,\tau_1)$;
then $S = S_1$ and $\tau = S_1\beta\rightarrow\tau_1$.

(iv) If e is let $x = e_1$ in e_2 then
let $W(A,e_1) = (S_1,\tau_2)$
and $W(S_1A_x\cup\{x:\overline{S_1A}(\tau_1)\},e_2) = (S_2,\tau_2)$;
then $S = S_2S_1$ and $\tau = \tau_2$.

NOTE: When any of the conditions above is not met W fails.

The following proposition proves that W meets our requirements.

Proposition 4 (Soundness of W) If $W(A,e)$ succeeds with (S,τ) then there is a derivation of $SA \vdash e:\tau$.

Proof By induction on e using proposition 2. ☒

It follows that there is also a derivation of $SA \vdash e:\overline{SA}(\tau)$. We refer to $\overline{SA}(\tau)$ as a type-scheme computed by W for e under SA.

7. Completeness of W

Given A and e, we will call σ_p a principal type-scheme of e under assumptions A iff

(i) $A \vdash e:\sigma_p$

(ii) Any other σ for which $A \vdash e:\sigma$ is a generic instance of σ_p.

Our main result, restricted to the simple case in which A contains no free type-variables, may be stated as follows:

If $A \vdash e:\sigma$, for some σ , then W computes a principal type scheme for e under A.

This is a direct corollary of the following general theorem, which is a stronger result suited to inductive proof:

Theorem (Completeness of W). Given A and e, let A' be an instance of A and σ a type-scheme such that

$$A' \vdash e:\sigma$$

Then (i) $W(A,e)$ succeeds

(ii) If $W(A,e) = (S,\tau)$ then, for some substitution R,

$A' = RSA$ and $R\,\overline{SA}(\tau) > \sigma$ ☒

In fact, from the theorem one also derives as corollaries that it is decidable whether e has any type at all under assumptions A, and that, if so, it has a principal type scheme under A.

The detailed proofs of results in this paper, and related results, will appear in the first author's forthcoming Ph.D. Thesis.

References

[LNCSn stands for Vol n, Lecture Notes in Computer Science, Springer-Verlag].

[Cop] M. Coppo, An extended polymorphic type system for applicative languages, (1980), LNCS 88, pp 194-204.

[DD] A. Demers and J. Donahue, Report on the programming language Russell, (1979), Report No. TR 79-371, Computer Science Department, Cornell University.

[GMW] M. Gordon, R. Milner and C. Wadsworth, (1979), Edinburgh LCF, LNCS 78.

[Hin] R. Hindley, The principal type-scheme of an object in Combinatory Logic, (1969), Trans AMS 146, pp 29-60.

[Mil] R. Milner, A theory of type polymorphism in programming (1978), JCSS 17,3, pp 348-375.

[Rob] J.A. Robinson, A machine-oriented logic based on the resolution principle, JACM 12,1 (1965), 23-41.