The Hindley-Milner Type Inference Algorithm

Ian Grant*

January 17, 2011

Abstract

The Hindley-Milner algorithm is described and an implementation in Standard ML is presented.

1 Introduction

The idea of *logical types* was introduced by Russell and used in *Principia Mathematica* to avoid the kinds of paradox in set-theoretic reasoning which were seen to arise as a consequence of applying concepts such as set-membership inappropriately. *Russell's paradox* is the paradox that was evident in Frege's *Begriffschrift* which was intended as the basis for a logical foundation for mathematics. The paradox comes about because of the definition of a set as the collection of objects that satisfy a certain predicate P, say. Now sets can contain other sets, and if we identify the set by just the predicate P then we can form a new predicate R, say, which defines the set of all sets that do not contain themselves. Thus R is the predicate $R(P)$ which is true if the predicate P identifies a set which does not contain itself. That is, $R(P)$ if, and only if $P(P)$ is false, which we can write as $R(P) = \neg P(P)$. Now we can consider the set R and ask if R contains itself. That is, we wish to decide the truth of the predicate $R(R) = \neg R(R)$ and so we have the paradox that $R(R)$ is true if and only if $R(R)$ is false.

Russell introduced logical types because he hoped that by restricting the types of object to which predicates can be considered to apply he would be able to avoid all kinds of inconsistency including this one. Unfortunately this made his logic considerably more complex because the way Russell defined the natural numbers, adding one to any number produced an object of a different logical type. All the predecessors of that number could also be defined at the new type, but it meant that one couldn't easily define what an arithmetic expression like $y = x - 1$ means because y has a different logical type to that of x. This complexity could be obviated somewhat by adopting a principle that one could always treat the numbers as appearing at the same type. This was called the *ramified theory of types*. The notion of logical types was used again some decades later by Church in the typed lambda calculus, which is the basis for the modern incarnations of Higher Order Logic such as HOL4 and Isabelle/HOL.

*e-mail iang@pobox.com

2 The lambda calculus

The λ-calculus is the language of expressions $e \in \text{Exp}$ in variables $x \in \text{Var}$ given by the grammar

$$e ::= x \mid e\,e' \mid \lambda x.e$$

The variables are the set $\text{Var} = \{a, b, c, \ldots, x, y, z, x', y', z', x'', y'', \ldots\}$.

The syntactic construct $\lambda x.e$ *binds* the variable x in the expression e. Variables in lambda expressions occur either *free* or *bound*. A variable is bound if it is not free, and a variable is free if it is in the set $\text{FV}(e)$ defined recursively as

$$\text{FV}(e) = \begin{cases} \text{FV}(e_1) \cup \text{FV}(e_2) & \text{if } e = e_1\,e_2 \\ \text{FV}(e_1) \backslash \{x\} & \text{if } e = \lambda x.e_1 \\ \{x\} & \text{if } e = x. \end{cases}$$

The *beta reduction* relation $\rightarrow$ is defined as the least congruence relation on expressions e such that

$$(\lambda x.e_1)\,e_2 \rightarrow [e_2/x]\,e_1$$

where $[t/x]e$ means substituting the term t for each free occurrence of x in e. If there are free variables in e_2 then bound variables in e_1 must be renamed so that substitution does not result in the free variables in e_2 becoming bound. This process of renaming bound variables is called *alpha conversion*.

A lambda expression is in *normal form* when there are no reductions that can be applied. *Normal order reduction* reduces the left-most, outer-most application of an abstraction. The weak normalisation theorem says that repeated normal order reduction always reaches a normal form, if there is one. Here are some examples of normal order beta reduction with alpha conversion.

$$\begin{aligned}
(\lambda f.\lambda x.f\,(f\,x))[\lambda f.\lambda x.f\,(f\,x)] &\rightarrow \lambda x.(\lambda f.\lambda x.f\,(f\,x))[(\lambda f.\lambda x.f\,(f\,x))\,x] \\
&\rightarrow \lambda x.\lambda x'.(\lambda f.\lambda x.f\,(f\,x))[x]\,((\lambda f.\lambda x.f\,(f\,x))\,x\,x') \\
&\rightarrow \lambda x.\lambda x'.(\lambda x''.x\,(x\,x''))[(\lambda f.\lambda x.f\,(f\,x))\,x\,x'] \\
&\rightarrow \lambda x.\lambda x'.x\,(x\,((\lambda f.\lambda x.f\,(f\,x))[x]\,x')) \\
&\rightarrow \lambda x.\lambda x'.x\,(x\,((\lambda x'''.x\,(x\,x'''))[x'])) \\
&\rightarrow \lambda x.\lambda x'.x\,(x\,(x\,(x\,x')))
\end{aligned}$$

Some lambda expressions do not have a normal form:

$$\begin{aligned}
(\lambda x.x\,x)[\lambda x.x\,x] &\rightarrow (\lambda x.x\,x)[\lambda x.x\,x] \\
&\rightarrow (\lambda x.x\,x)[\lambda x.x\,x] \\
&\rightarrow (\lambda x.x\,x)[\lambda x.x\,x] \cdots
\end{aligned}$$

Church introduced types to the lambda calculus to prevent the inconsistencies that result from Russell's paradox being expressible in the ordinary untyped lambda calculus. In fact the existence of these so-called *paradoxical combinators* is what makes the untyped lambda-calculus a fully-functional language for representing all possible computations, including those of the *partial functions* which are functions $f(x)$

which are not defined for some values of x. In computational terms, *not defined* means that the computation loops indefinitely without returning a result.

It was Church's student Haskell B. Curry who discovered that Russell's paradox could be expressed almost immediately as $R \equiv \lambda x.\neg(x\,x)$ which gives $R\,R \equiv \neg(R\,R)$. The combinator

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x)$$

generalises this for any function f. Curry's $\mathbf{Y}$ combinator has the property that for every function F, the lambda term $\mathbf{Y}\,F$ satisfies the equation $\mathbf{Y}\,F = F\,(\mathbf{Y}\,F)$. Thus the combinator $\mathbf{Y}$ finds a fixed-point $\mathbf{Y}\,F$ for any function F. Using such fixed-point combinators one may define any partial or total recursive function. $\mathbf{Y}$ has no normal form: it repeatedly applies f:

$$\begin{aligned}\lambda f.(\lambda x.f\,(x\,x))[\lambda x.f\,(x\,x)] &\to \lambda f.f\,((\lambda x.f\,(x\,x))[\lambda x.f\,(x\,x)])\\ &\to \lambda f.f\,(f\,((\lambda x.f\,(x\,x))[\lambda x.f\,(x\,x)]))\\ &\to \lambda f.f\,(f\,(f\,((\lambda x.f\,(x\,x))[\lambda x.f\,(x\,x)])))\cdots\end{aligned}$$

3 Interpreting lambda calculus

The abstract terms of the lambda calculus can be used to represent numbers and any other mathematical objects. The terms

$$\mathsf{0} \equiv \lambda f.\lambda x.x \qquad \text{and} \qquad \mathbf{S} \equiv \lambda n.\lambda f.\lambda x.n\,f\,(f\,x)$$

represent the ordinal $\mathsf{0}$ and the successor function S which produces the next ordinal. Thus the term $\mathbf{S}\,\mathsf{0}$ which we saw earlier is the number $\mathbf{1} \equiv \lambda f.\lambda x.f\,x$. In general the *Church numerals* are the lambda terms

$$\mathbf{n} \equiv \lambda f.\lambda x.[f^n]\,x$$

where $[f^n]\,x$ represents $f\,(f\cdots(f\,x)\cdots)$ i.e. the n-fold application of f to x. The Church numerals thus each 'contain' some finite amount of recursion and this is sufficient to define the usual arithmetic operations. Addition is defined as

$$\mathbf{ADD} \equiv \lambda m.\lambda n.\lambda f.\lambda x.\,m\,f\,(n\,f\,x)$$

This is a function with one argument m which evaluates to a function of one argument n which evaluates to number. This way of defining n-argument functions as n-fold compositions of abstractions is called *currying* and the resulting functions are called *curried functions*, after Curry.

A curried function can be partially applied to give another combinator. For example

$$\mathbf{ADDSIX} \equiv \mathbf{ADD}\ \mathbf{6} \to \lambda n.\lambda f.\lambda x.\mathbf{6}\,f\,(n\,f\,x))$$

is a combinator which adds six to its argument. If we apply this to 3 we get

$$\mathbf{ADDSIX}\ \mathbf{3} \to \lambda f.\lambda x.\mathbf{6}\,f\,(\mathbf{3}\,f\,x))$$

and we see that the result will be the six-fold application of f to the result of the three-fold application of f to x, and this is just the nine-fold application of f to x which is the Church numeral $\mathbf{9}$.

Multiplication and exponentiation are similar

$$\mathbf{MUL} \equiv \lambda m.\lambda n.\lambda f.\lambda x.m\,(n\,f)\,x$$
$$\mathbf{EXP} \equiv \lambda m.\lambda n.\lambda f.\lambda x.n\,m\,f\,x$$

Subtraction is less obvious. Another student of Church's, Steven C. Kleene discovered how to express it. Since writing down a Church numeral **n** sets up an n-fold application of a function, subtraction could be done if one could define a predecessor function **PRED** which was the inverse of the successor function **S**. To do this, Kleene first found a way to represent pairs of numbers using the constructor

$$\mathbf{PAIR} \equiv \lambda x.\lambda y.\lambda f.f\,x\,y$$

The operations **FST** and **SND** are the corresponding deconstructors

$$\mathbf{FST} \equiv \lambda p.p\lambda x.\lambda y.x \qquad \text{and} \qquad \mathbf{SND} \equiv \lambda p.p\lambda x.\lambda y.y.$$

Then the predecessor combinator can be defined by first defining a function g which, given a pair $(x,\cdot)$, produces the pair $(f(x),x)$ so that $g^n(x,y) = (f^n(x), f^{n-1}(x))$. Thus

$$\mathbf{G} \equiv \lambda f.\lambda p.\mathbf{PAIR}\,(f\,(\mathbf{FST}\,p))\,(\mathbf{FST}\,p)$$

then the predecessor combinator is

$$\mathbf{PRED} \equiv \lambda n.\mathbf{SND}\,(n\,(\mathbf{G}\,\mathbf{S})\,(\mathbf{PAIR}\,\mathbf{0}\,\mathbf{0}))$$

Thus **PRED n** reduces in one step to $\mathbf{SND}\,(\mathbf{n}\,(\mathbf{G}\,\mathbf{S})\,(\mathbf{PAIR}\,\mathbf{0}\,\mathbf{0}))$ and this is the second element of a pair $(\mathbf{S}^n\,\mathbf{0}, \mathbf{S}^{n-1}\,\mathbf{0})$, which in turn is the n-fold application of **G S** to the pair $(\mathbf{0},\mathbf{0})$. It is not very efficient, taking seventeen reductions to compute the predecessor of **1**. See Appendix B for the details.

Subtraction may then be defined as

$$\mathbf{SUB} \equiv \lambda m.\lambda n.n\,\mathbf{PRED}\,m.$$

4 Types

The aim of a type discipline is to constrain the statements of a formal language to just those that are well-defined according to some semantics. In the case of Russell's logical types the aim was to disallow impredicative definitions: those where a predicate refers to itself in its own definition. This is what happens when we define the Russell predicate which represents the set of all sets that do not contain themselves. It implicitly includes a reference to itself under the universally quantified 'sets that do not contain themselves' because it is a predicate which is itself defining a set.

The way the type discipline achieves this is by stratifying the expressions so that each expression may only include references to objects which are constructed at lower 'levels' of the hierarchy. In the system of *Principia Mathematica*, the levels are called *orders* and the order of an expression is one more than the highest order of any of the expressions it contains. In this way all the universally quantified statements like 'all sets that ...' are 'universally quantified' only over orders strictly lower than those at which they appear.

Russell was attempting to define mathematics from an exact and precise foundation of logic. He therefore started with the most exact language and was obliged to remain within it for the entire project. This is extremely difficult and arduous and he was not particularly successful. Church's type system [1] was very similar, but he was working two decades after Russell. Consequently he was able to use Tarski's notion of *semantic definitions* in a meta-language. This allows one to define a more precise language in terms a less precise one. This is much more efficacious. Whereas *Principia Mathematica* was published[1] as three large volumes and took decades to produce, Church's system was equally powerful and was described in one 14 page paper, produced in a matter of months.

In Church's formulation of higher-order logic, many of the axioms and theorems are infinite *schemas* indexed by meta-language type variables. This is similar to the notion of *typical ambiguity* employed in *Principia Mathematica*, but it is strictly speaking not ambiguity in the types at all. Rather, the ambiguity—if it is such—is contained entirely in the meta-language where the type variables can range over all types. This was carried through to the proofs of the theorems which were considered to be *meta-mathematical* in that they were schemas of proofs such that any one of the infinite set of theorems represented by the schema would be provable at the given type.

Church's types were syntactically bound to the lambda terms. Every term had to be typed and the types were attached to the symbols as subscripts. What I show here is a scheme due to Curry where the types are defined as a predicate which is separate from the syntax.

Given a set ι of types of individuals, we can define *simple types* τ using the grammar

$$\tau ::= \iota \mid \tau \to \tau$$

We then specify a type predicate based on a set of assumptions $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \ldots\}$ where each $x_i : \tau_i$ gives the type τ_i of the variable x_i. The type predicate $\Gamma \vdash e : \tau$ is to be read 'The expression e has type τ under assumptions Γ' and it is true if and only if it has a derivation using the rules

$$\text{TAUT: } \frac{}{\Gamma \vdash x : \tau}\ (x : \tau \in \Gamma)$$

$$\text{COMB: } \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\, e') : \tau} \qquad \text{ABS: } \frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \tau' \to \tau}$$

where Γ_x is Γ without any assumptions about the variable x.

For example here is a derivation of the type of the term $\lambda f.\lambda x.x$

$$\cfrac{\cfrac{\cfrac{}{f : \tau, x : \tau' \vdash x : \tau'}\ \text{TAUT:}}{f : \tau \vdash (\lambda x.x) : \tau' \to \tau'}\ \text{ABS:}}{\vdash (\lambda f.\lambda x.x) : \tau \to \tau' \to \tau'}\ \text{ABS:}\ (*)$$

[1] In the first instance at the authors' personal expense!

Thus for any types τ and τ' the expression $\lambda f.\lambda x.x$ is well-typed. Here τ and τ' are not *themselves* types, they are meta-language variables which are placeholders for actual types like $\iota \to \iota$ or $(\iota \to \iota) \to \iota$. Similarly ι is a placeholder for a type constant like boolean or integer in some actual language: the language of simple types is an *abstract* language. Appendix C gives the derivation for the type $(\lambda n.\lambda f.\lambda x.n\, f\, (f\, x))\, \lambda f.\lambda x.x$.

As one would hope, restricting the lambda calculus to the expressions that can be simply-typed, the fixed-point combinators are no longer available. For example, if we attempt to type the **Y** combinator

$$\text{COMB:}\ \frac{f:\tau\to\tau' \vdash (\lambda x.f\,(x\,x)):\tau''' \to \tau'' \qquad f:\tau\to\tau' \vdash (\lambda x.f\,(x\,x)):\tau'''}{\text{ABS:}\ \dfrac{f:\tau\to\tau' \vdash (\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x):\tau''}{\vdash (\lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x)):(\tau\to\tau')\to\tau''}}$$

we find we need $\tau''' = \tau''' \to \tau'' = (\tau''' \to \tau'') \to \tau'' = \cdots$ which cannot be satisfied by any simple type.

It is not just the fixed-point combinator **Y** which is un-typable. If a term t is *strongly normalisable* then this means that there are no infinite reduction sequences beginning with t. The *strong normalisation theorem* proves that every term in the simply-typed lambda calculus is strongly normalisable. Thus all terms are defined and so all functions represented in the simply-typed lambda calculus are total functions.

This discipline carries a penalty however, because whenever one applies a Church numeral to a function, as happens in the definitions of addition **ADD** and the predecessor function **PRED**, one obtains a result which is at a higher type than the operand. If we use Church's very efficient device of using α' to denote the type $(\alpha \to \alpha) \to \alpha \to \alpha$ and α'' the type $(\alpha' \to \alpha') \to \alpha' \to \alpha'$ etc. then the type of the predecessor function **PRED** is $\alpha'' \to \alpha'$. In other words, the result is at a type one level lower than that of the operand.[2]

This complicates the proof schemas considerably and Church repeatedly needs to use devious inductive applications of a type-raising operator $T : \alpha \to \alpha'$ to prove theorems like his *denouement*, the theorem schema 43^{α}:

$$N_{o\alpha'}(n_{\alpha'}) \Rightarrow P_{\alpha'\alpha'}(S_{\alpha'\alpha'}(n_{\alpha'})) = n_{\alpha'}.$$

which says[3] 'at type α', if n is a number then the predecessor of the successor of n is n.'

The difficulties arise because there is no typical ambiguity in the object language. Every instance of a meta-language proof schema indexed by a meta-language type variable α is at some definite type.

5 The ML type discipline

Every ML term has a type. The basic types are `bool`, `int`, `string` etc. Other types like lists: α `list` and pairs $\alpha * \beta$ are built using *type constructors* which are functions acting

[2] In Church's system it was *two* levels lower because his pairing operator had the type $\alpha' \to \alpha' \to \alpha''$.

[3] The subscripts are the types, where the type $\beta\alpha$ is the type of functions *from* α *to* β and the type o is the type of propositions, i.e. truth values.

on types instead of values. Here α and β are *type variables* which can be instantiated to any type. This parameterisation means that ML types are not simple types, they are *polymorphic*.

In ML the function space type constructor which takes the role of λ is written `fn` so the lambda expression $\lambda x.x$ is coded in ML as `fn x => x`. Thus we may code the Church numeral **2** as

```
val two = fn f => fn x => f (f x)
```

then ML gives it the type $\sigma_2 = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$ which means that the function `two` can be applied to any function of type $\alpha \to \alpha$ whatever the actual type α, to yield a function of type $\alpha \to \alpha$. This is the *most general type* for that expression in a sense that will be made precise later on, however note for now that the Church numeral **1** when coded in ML as

```
val one = fn f => fn x => f x
```

is given the type $\sigma_1 = \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta$ which is more general than σ_2 in that if we choose $\beta = \alpha$ in σ_1 then we have σ_2. One says that σ_2 is a *generic instance* of σ_1. The type assigned to **2** is the most general, given that the types of the domain and the codomain must be the same, because in the expression $f\,(f\,x)$ the function is applied to the values it produces. This is not the case for **1** and so there f may produce values of a different type to those in its domain.

Yet more general still is the type of the Church numeral `0`

```
val zero = fn f => fn x => x
```

which is $\sigma_0 = \forall\alpha\beta.\alpha \to \beta \to \beta$. This is because f is not applied so the type assigned to f need not be restricted to function types, it could be any individual type too.

The notation $\sigma_2 < \sigma_1 < \sigma_0$ is used to indicate that σ_2 is a generic instance of σ_1 which is itself a generic instance of σ_0. Here the letters σ_i refer to *type-schemes* rather than mere types, and they are metalanguage variables like τ_i, but the variables α, β, γ are *type variables* in the object language. The distinction is a subtle but important one.

The effect of having type variables is to allow genuine typical ambiguity in the object language itself, not just in the meta-language. In Milner's type system the infinite tower of types of the Church numerals collapses to just the generic instances of the type-scheme $\sigma_0 = \forall\alpha\beta.\alpha \to \beta \to \beta$. For example the following ML declarations directly code the function **PRED** and apply it to the Church numeral **6**:

```
fun num 0 = (fn f => fn x => x)
  | num n = (fn f => fn x => f (num (n-1) f x))

val zero = fn f => fn x => x
val succ = fn n => fn f => fn x => n f (f x)

val pair = fn x => fn y => fn f => f x y
val fst = fn p => p (fn x => fn y => x)
val snd = fn p => p (fn x => fn y => y)

val g = fn f => fn p => (pair (f (fst p)) (fst p))
val pred = fn n => (snd (n (g succ) (pair zero zero)))

val pred6 = (pred (num 6))
```

ML deduces the type-scheme[4] σ_2 for the predecessor of **6**. This is the same type-scheme as **6** itself. Thus the technical difficulties which Church had to deal with—those of numbers appearing at different types—disappear completely and the proof of Church's 43^{α} can now be carried out quite straightforwardly.[5]

ML will not type the **Y** combinator, nor any other lambda expressions that do not have a normal form. Nevertheless ML is a Turing complete language because it allows recursive function definitions which it treats as if there were a fixed-point combinator `fix` which may be written in ML as

```
fun fix f = f (fn x => fix f x)
```

and to which ML assigns a type $\forall\alpha\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$.

Although there are some 400 lines of standard ML in this document there is not one single explicit type annotation. Yet every statement is type-checked before being run. All ML compilers and many other languages use the *Hindley-Milner* algorithm [3, 5] to infer the most general types of expressions. This allows programs to be written with very few explicit type annotations, but still to benefit from type checking.

6 Unification

The type inference algorithm uses Robinson's *unification* algorithm [7] which was intended as a scheme for choosing instantiations of universally quantified variables in theorem proving with resolution which is the basis for logic programming languages like PROLOG. Unification is easier to understand in terms of functions with arguments than in terms of binary infix operators like the function type constructor $\to$ so we will interpret the latter as functions of two arguments. Type constants like ι can be considered as constant functions (i.e. ones that take no arguments and return a value). We will also be able easily to extend the language to include other type constructors such as one for polymorphic lists by introducing a unary type constructor function with an associated type-scheme.

Hence we will describe unification as working with functions like $f(x,y)$ and constants like a and b which represent type constructors like $\alpha \to \beta$ and ι respectively. In the following we will also assume that italic Roman letters x, y and z are variables.

Unification is a recursive algorithm for determining a substitution of terms for variables (i.e. a variable assignment) that makes two terms equal. For example we can unify $f(a,y)$ with $f(x,f(b,x))$ with the substitution $[a/x, f(b,a)/y]$ which should be read 'substitute a for x and then substitute $f(b,a)$ for y'. The substitution S is the composition $S = [f(b,x)/y] \circ [a/x]$ of the two separate substitutions $[a/x]$ and $[f(b,x)/y]$. Note that composition of substitutions is right to left so that we perform the rightmost first, and we apply the rightmost to the leftmost in the process of composing. The order matters. It is easy to see that the substitution S applied to $f(a,y)$ gives $f(a,f(b,a))$ and when applied to $f(x,f(b,x))$ yields the same. Thus we say S is a *unifier* of $f(a,y)$ and $f(x,f(b,x))$.

[4] Provided *value polymorphism* is not in effect. If it is, as in Moscow ML by default, then the type variable in the resulting type will not be quantified.

[5] It is still a meta-mathematical proof schema because it requires induction, but it is much simpler.

The unification algorithm finds the *most general unifier* of two terms, which is a unifier U with the property that if there is any other unifier S' of those two terms then there is some substitution T such that $T \circ U$ yields the unifier S'. In other words every other unifier S' can be seen as a particular case of the most general unifier U.

Unification is not always possible. For example one cannot unify two different functions like $f(a,b)$ and $g(x,y)$ because we treat functions and constants as different from variables and we only assign to variables. For the same reason one cannot unify two constants like a and b. Functions must also occur with the same *arity*, for example one cannot unify $f(a,b)$ and $f(x)$. Lastly, one cannot unify a variable like x and a function like $f(x,\ldots)$ which is applied to x because any substitution would not yield the same term when applied to both x and $f(x,\ldots)$. It is this fact that means that lambda expressions with no normal forms cannot be typed in ML when written as functions. Recall the type $\tau''' = \tau''' \to \tau''$ which we needed to type the **Y** combinator; putting $f(x,y)$ for the type constructor $x \to y$ we find we need x and y such that $x = f(x,y)$. In order to unify x and $f(x,\ldots)$ one would need some notion of a fixed-point operator acting on g which we could consider the solution in x to the equation $x = f(x,\ldots)$. In the absence of this we therefore need:

Algorithm (*Occurs check*)

A variable x occurs in a term t if and only if $t = f(s_1, s_2, \ldots, s_n)$ for $n > 0$ and either $s_i = x$ or x occurs in s_i for some $i = 1, 2, \ldots, n$. □

The unification algorithm is described recursively on the structure of terms.

Algorithm (*Unification*)

To find the most general unifier $U = \mathrm{MGU}(t, t')$ of terms t and t'

(i) If $t = x$ and $t' = y$ then $U = [x/y]$.

(iia) If $t = x$ and $t' = f(s_1, s_2, \ldots, s_n)$ and x does not occur in t' then $U = [t'/x]$.

(iib) If $t = f(s_1, s_2, \ldots, s_n)$ and $t' = x$ and x does not occur in t then $U = [t/x]$.

(iii) If $t = a$ and $t' = a$ then $U = []$.

(iv) If $t = f(s_1, s_2, \ldots, s_n)$ and $t' = f(s_1, s_2, \ldots, s_n)$ then

$$U = \mathrm{MGU}(f(U_1 s_2, U_1 s_3, \ldots, s_n), f(U_1 s_2', U_1 s_3', \ldots, s_n)) \circ U_1$$

where $U_1 = \mathrm{MGU}(s_1', s_1')$.

In any other circumstances the algorithm fails. □

The code in figure 1 implements unification on a datatype of terms. It consists of the datatype definition and then definitions of the operations of substitution (`subs`) and composition of lists of substitutions (`compose`), followed by the unification algorithm itself. Appendix E contains code for printing terms (`ppterm`) and substitutions (`ppsubs`).

We make meta-language copies of the variables and use ML expressions to construct terms. Here is an example of unifying $j(x,y,z)$ with $j(f(y,y),f(z,z),f(a,a))$. The MGU of which is

$$[f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)))/x, f(f(a,a),f(a,a))/y, f(a,a)/z].$$

```
datatype term = Tyvar of string
              | Tyapp of string * (term list)

fun subs [] term = term
  | subs ((t1,v1)::ss) (term as Tyvar name) =
       if name=v1 then t1 else subs ss term
  | subs _ (term as Tyapp(name,[])) = term
  | subs l (Tyapp(name,args)) =
       let fun arglist r [] = rev r
             | arglist r (h::t) =
                  arglist ((subs l h)::r) t
       in
          Tyapp(name, arglist [] args)
       end

fun compose [] s1 = s1
  | compose (s::ss) s1 =
      let fun iter r s [] = rev r
            | iter r s ((t1,v1)::ss) =
                iter (((subs [s] t1),v1)::r) s ss
      in
         compose ss (s::(iter [] s s1))
      end

exception Unify of string

fun unify t1 t2 =
   let fun iter r t1 t2 =
       let fun occurs v (Tyapp(name,[])) = false
          | occurs v (Tyapp(name,((Tyvar vn)::t))) =
                       if vn=v then true else occurs v (Tyapp(name,t))
          | occurs v (Tyapp(name,(s::t))) =
                       occurs v s orelse occurs v (Tyapp(name,t))
          | occurs v (Tyvar vn) = vn=v
       fun unify_args r [] [] = rev r
          | unify_args r [] _ = raise Unify "Arity"
          | unify_args r _ [] = raise Unify "Arity"
          | unify_args r (t1::t1s) (t2::t2s) =
              unify_args (compose (iter [] (subs r t1)
                                           (subs r t2)) r) t1s t2s
    in
       case (t1,t2) of
          (Tyvar v1,Tyvar v2) => if (v1 = v2) then [] else ((t1, v2)::r)
        | (Tyvar v,Tyapp(_,[])) => ((t2, v)::r)
        | (Tyapp(_,[]),Tyvar v) => ((t1, v)::r)
        | (Tyvar v,Tyapp _) =>
            if occurs v t2 then raise Unify "Occurs" else ((t2, v)::r)
        | (Tyapp _,Tyvar v) =>
            if occurs v t1 then raise Unify "Occurs" else ((t1, v)::r)
        | (Tyapp(name1,args1),Tyapp(name2,args2)) =>
              if (name1=name2)
                 then unify_args r args1 args2
                 else raise Unify "Const"
    end
in
   iter [] t1 t2
end
```

Figure 1: Unification

```
val x = Tyvar "x"
val y = Tyvar "y"
val z = Tyvar "z"

fun apply s l = Tyapp(s,l)

val a = apply "a" []
fun j(x, y, z) = apply "j" [x, y, z]
fun f(x, y) = apply "f" [x, y]

val t1 = j(x,y,z)
val t2 = j(f(y,y), f(z,z), f(a,a));

val U = unify t1 t2;
ppsubs U;
```

7 Hindley-Milner Type Inference

The algorithm is defined for a simple language of expressions e over variables x defined by the grammar

$$e ::= x \mid e\,e' \mid \lambda x.e \mid \texttt{let}\, x = e\, \texttt{in}\, e'$$

and another of types τ and *type-schemes* σ, over a set of *type variables* α and a set of primitive types ι.

$$\tau ::= \alpha \mid \iota \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall \alpha \sigma$$

The variables bound under universal quantifiers are called *generic type variables* and others are *free* variables.

Type derivations are with respect to the rules:

$$\text{TAUT: } \frac{}{\Gamma \vdash x : \sigma}\ (x:\sigma \in \Gamma) \qquad \text{INST: } \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma'}\ (\sigma > \sigma')$$

$$\text{GEN: } \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha \sigma}\ (\alpha \text{ not free in } \Gamma) \qquad \text{COMB: } \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\, e_2) : \tau}$$

$$\text{ABS: } \frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{LET: } \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma_x \cup \{x : \sigma\} \vdash e_2 : \tau}{\Gamma \vdash (\texttt{let}\, x = e_1\, \texttt{in}\, e_2) : \tau}$$

Note that apart from the types ι of individuals, the only other types the algorithm deals with are those constructed by the function-space type constructor $\to$. As mentioned in the previous section on unification, other *compound types* can be introduced by defining *type constructors* which are merely special functions which have as the codomain the new type. For example a type of pairs $\alpha \times \beta$ can be defined by assuming the existence of a constructor $\textbf{PAIR} : \forall \alpha\beta.\alpha \to \beta \to \alpha \times \beta$ and deconstructors $\textbf{FST} : \forall \alpha\beta.\alpha \times \beta \to \alpha$ and $\textbf{SND} : \forall \alpha\beta.\alpha \times \beta \to \beta$. The type system is therefore independent of the semantics of the language of expressions and this is how ML can accommodate higher-order functions and fixed-point operators which are completely outside the scope of the simply-typed lambda calculus.

On this view, the types ι of individuals can similarly be construed as applications of nullary type constructors like **TRUE**:`bool` and **FALSE**:`bool`.

The type inference algorithm is called W, for some reason. There are four cases, one for each of the productions in the grammar of expressions e. The four cases each correspond to one of the four rules TAUT, COMB, ABS and LET. In each case the aim is to find the most general type that will satisfy the rule for that form, and in the process to find the substitutions which, when applied to the assumptions, will make the necessary instantiations of variables in type-schemes.

Algorithm W.

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

(i) If e is x and there is an assumption $x : \forall\alpha_1, \ldots, \alpha_n\tau'$ in Γ then

$$S = [] \qquad \text{and} \qquad \tau = [\beta_i/\alpha_i]\tau'$$

where the β_is are new.

(ii) If e is $e_1\, e_2$ then let

$$W(\Gamma, e_1) = (S_1, \tau_1), \quad W(S_1\Gamma, e_2) = (S_2, \tau_2) \quad \text{and} \quad U(S_2\tau_1, \tau_2 \to \beta) = V$$

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 $W(\Gamma_x \cup \{x : \beta\}, e_1) = (S_1, \tau_1)$; then $S = S_1$ and $\tau = S_1\beta \to \tau_1$.

(iv) If e is $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ then let

$$W(\Gamma, e_1) = (S_1, \tau_1) \quad \text{and} \quad W(S_1\Gamma_x \cup \{x : \overline{S_1\Gamma}(\tau_1)\}, e_2) = (S_2, \tau_2);$$

then $S = S_2S_1$ and $\tau = \tau_2$.

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

Here $U(\tau_1, \tau_2)$ is the MGU of the types τ_1 and τ_2 as computed by Robinson's algorithm.

The simplest case is an expression which consists of a single variable x. This deduction corresponds to an instance of the rule TAUT followed by one of INST. If the assumptions include a type-scheme for x, then the result is simply the generic instantiation of the type-scheme to new variables. Otherwise the algorithm fails.

The case for abstraction $\lambda x.e_1$ is probably the next simplest. The algorithm simply picks a new variable β to represent the type of the argument x and recursively deduces the type of e_1 subject to the assumption that x has the type β. The resulting substitution is then applied to β. This corresponds to an instance of the rule ABS.

The case for application $e_1\, e_2$ is the most complex. First the most general type τ_1 for the operator e_1 is computed. The resulting substitution is applied to the assumptions and the most general type τ_2 for e_2 is computed under these new assumptions. The next step is to find the most general type for the result of the application. This is done by unifying the type τ_1 of the operator e_1 — after having applied the substitution that

produced the most general type for the operand τ_2 — with the type $\tau_2 \to \beta$, where β is a new type variable. Note that the substitution S_2 is not applied to τ_2 before the unification. This is because unification will deduce it, the necessary instantiations having been made in the type τ_1 of the operator, and the unification algorithm may be able to find a more general type for the operator than it would had the substitution been applied to the operand. Similarly, the substitution that produces the most general type τ_1 for the operator e_1 is not applied to τ_1 before unification. Substitution will always either produce a type that is as general, or less general, and if the less general types can be unified then so can the more general. It follows that only the necessary substitutions should be applied before unifying. The only necessary condition in the corresponding inference rule COMB is that the most general type τ_2 of the operand must match the type of the argument to the operator. It then follows that if all of the instantiations of type variables that were made to produce the most general type τ_2 are made to τ_1 then the resulting unification is guaranteed to respect the premisses of the inference rule and the inference will be sound. The resulting type τ of e is then just the result of applying just the unifying substitutions to β. The substitutions which make the necessary instantiations to realise the type τ are then the composition, in order, of those that were found during the derivation.

The final case of expressions of the form $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ is a little easier. The corresponding inference rule LET is almost the same as ABS except it has an additional premiss which asserts that the expression e_1 has the type-scheme σ under the assumptions. Then the type of the body e_2 is the most general under the added assumption that the bound variable x has this same type-scheme. The algorithm first finds the most general type τ_1 for the expression e_1 and makes the necessary instantiations in the assumptions. Then it *closes* the type τ_1 with respect to these new assumptions. This process consists in universally quantifying any free type variables in τ_1 which are not free in the assumptions and it corresponds to zero or more instances of the rule GEN. This results in a type-scheme σ which is added to the assumptions as the type of the bound variable x. The type of the expression e as a whole is then the most general type for the expression e_2 under these assumptions. As in all the other cases except that of variables, the resulting substitution is the composition, in order, of all the substitutions that were derived in the process of determining the type τ.

A final step after computing $(S, \tau) = W(\Gamma, e)$ is to close the resulting type computed for e under the assumptions $S\Gamma$ that result from the instantiations S. The result $\sigma_p =: \overline{S\Gamma}(\tau)$ is then called *the principal type scheme* for e under Γ and this means that any other type-scheme σ such that $\Gamma \vdash e : \sigma$ is a generic instance of σ_p.

8 ML Implementation of type inference

The implementation divides naturally into three parts

- Type-schemes
- Assumptions
- The inference algorithm.

```
datatype typescheme = Forall of string * typescheme
                    | Type of term

fun mem p l =
   let fun iter [] = false
         | iter (x::xs) = (x=p) orelse iter xs
   in iter l end

fun fbtyvars f b (Tyvar v) = if (mem v b) then (f,b) else (v::f,b)
  | fbtyvars f b (Tyapp(name,args)) =
      let fun iter r [] = r
            | iter r (t::ts) =
                let val (f,b) = fbtyvars r b t
                in
                   iter f ts
                end
      in
         let val fvs = iter f args in (fvs,b) end
      end

fun fbtsvs f b (Forall(v,s)) = fbtsvs f (v::b) s
  | fbtsvs f b (Type t) = fbtyvars f b t

fun tyvars t =
   let val (f,b) = fbtyvars [] [] t in
      f@b
   end

fun varno "" = ~1
  | varno v =
    let val vl = explode v
        val letter = (ord (hd vl)) - (ord #"a")
        fun primes r [] = r | primes r (h::t) =
              if h = #"\039" then primes (r+26) t else ~1
    in
       if letter >= 0 andalso letter <= 25
          then primes letter (tl vl) else ~1
    end

fun newvar v =
   let val nv = v+1
       fun prime v 0 = v | prime v n = prime (v^"'") (n-1)
       val primes = nv div 26
       val var = str(chr ((ord #"a") + (nv mod 26)))
   in
      (nv,prime var primes)
   end

fun lastusedtsvar nv sigma =
   let val vars = let val (f,b) = fbtsvs [] [] sigma in f@b end
       fun iter r [] = r
         | iter r (h::t) =
               let val vn = varno h in
                  if vn > r then iter vn t else iter r t
               end
   in
      (iter nv vars)
   end

fun lastfreetsvar nv sigma =
   let val (vars,_) = fbtsvs [] [] sigma
       fun iter r [] = r
         | iter r (h::t) =
               let val vn = varno h in
                  if vn > r then iter vn t else iter r t
               end
   in
      (iter nv vars)
   end
```

Figure 2: Type schemes.

```
fun tssubs nv [] sigma = (nv, sigma)
  | tssubs nv ((tvp as (t,v))::tvs) sigma =
     let val (fvs,_) = fbtyvars [] [] t
         fun iter nv rnss (tvp as (t,v)) (ts as (Forall(sv,sts))) =
           if (sv = v) then (nv, ts) else
              if mem sv fvs then
                  let val (nv,newv) = newvar nv
                      val (nv,sigma') =
                        iter nv (compose [(Tyvar newv,sv)] rnss) tvp sts
                  in
                     (nv, Forall(newv,sigma'))
                  end
              else let val (nv,sigma') = iter nv rnss tvp sts
                   in
                      (nv,Forall(sv,sigma'))
                   end
           | iter nv rnss tvp (Type term) =
              (nv,(Type (subs [tvp] (subs rnss term))))
         val (nv, sigma') = iter nv [] tvp sigma
     in
        tssubs nv tvs sigma'
     end
```

Figure 3: Substituting Types.

Figure 2 is code for type-schemes σ. The datatype extends types represented by terms to type-schemes that may include generic variables. The function `ppts` in Appendix E prints type schemes in a format that is easy to read. The function `mem p l` tests whether an element `p` occurs in a list `l`. The function `fbtsvars` returns lists of the free and bound type variables in a type-scheme. The function `tyvars` returns a list of the variables in a a type. The function `newvar` returns a new variable guaranteed not to occur in any other type-schemes. This works by defining an enumeration on variables of the form x'''. If the letter a is the `0`th letter of the alphabet and a_n is the nth, then a variable of the form $a_n^{(m)}$ where (m) is a string of m primes, has the index $26m + n$. Any variable of another form has the index -1. Thus if we know the highest index `n` of any variable in a structure then we can generate a new one by `newvar n` which simply adds one to the index of the last new variable allocated.

Finally `tssubs` in figure 3 substitutes types for the free occurrences of variables in a type-scheme, renaming the generic variables to prevent free variables being captured by bindings. It takes an integer argument `nv` which is the number of the next new variable and it returns a pair consisting of the number of the next free variable and the new type scheme.

Assumptions are lists of variables and type-schemes. In the code in figure 4 we define `assq` to implement association lists of key/value pairs. The key is the variable name and the value is a type-scheme. We then extend the operations of substitution to whole sets of assumptions Γ (`assumsubs`) and define a function (`fassumvars`) to compute the list of free variables in a set of assumptions, and for finding the greatest index of the free variables in a set of assumptions (`lastfreeassumvar`), for use in allocating new variables. Finally we define `tsclosure` to create the closure $\overline{\Gamma}(\tau)$ of a type τ with respect to a set of assumptions Γ by making generic all the free variables in τ that are not free in Γ.

```
exception Assum of string

fun assq p l =
   let fun iter [] = raise Assum p
         | iter ((k,v)::xs) = if (k=p) then v else iter xs
   in iter l end

fun fassumvars Gamma =
   let fun iter f [] = f
     | iter f ((_,ts)::Gamma') =
        let val (fvs,_) = fbtsvs f [] ts in
           iter (f@fvs) Gamma'
        end
   in
      iter [] Gamma
   end

fun assumvars Gamma =
   let fun iter f [] = f
         | iter f ((_,ts)::Gamma') =
           let val (fvs,bvs) = fbtsvs f [] ts
           in
              iter (f@fvs@bvs) Gamma'
           end
   in
      iter [] Gamma
   end

fun lastfreeassumvar Gamma =
   let fun iter r [] = r
         | iter r ((_,sigma)::Gamma') =
             iter (lastfreetsvar r sigma) Gamma'
   in
      iter ~1 Gamma
   end

fun assumsubs nv S Gamma =
   let fun iter r nv S [] = (nv, rev r)
     | iter r nv S ((v,sigma)::Gamma') =
        let val (nv, sigma') = tssubs nv S sigma
        in
            iter ((v,sigma')::r) nv S Gamma'
        end
   in
      iter [] nv S Gamma
   end

fun tsclosure Gamma tau =
   let val favs = fassumvars Gamma
       val (ftvs,_) = fbtyvars [] [] tau
       fun iter bvs [] = Type tau
         | iter bvs (v::vs) =
             if (mem v favs) orelse (mem v bvs)
                   then iter bvs vs
                   else Forall(v,iter (v::bvs) vs)
   in
      iter [] ftvs
   end
```

Figure 4: Assumptions.

```
datatype exp = Var of string
             | Comb of exp * exp
             | Abs of string * exp
             | Let of (string * exp) * exp

infixr -->
fun tau1 --> tau2 = Tyapp("%f",[tau1,tau2])

fun W nv Gamma e =
   case e of
      (Var v) =>
         let fun tsinst nv (Type tau) = (nv, tau)
               | tsinst nv (Forall(alpha,sigma)) =
                     let val (nv, beta) = newvar (lastusedtsvar nv sigma)
                         val (nv, sigma') =
                           (tssubs nv [(Tyvar beta,alpha)] sigma)
                     in
                        tsinst nv sigma'
                     end
             val (nv, tau) = tsinst nv (assq v Gamma)
         in
            (nv, ([], tau))
         end
     | (Comb(e1,e2)) =>
          let val (nv, (S1,tau1)) = W nv Gamma e1
             val (nv, S1Gamma) = assumsubs nv S1 Gamma
             val (nv, (S2,tau2)) = W nv S1Gamma e2
             val S2tau1 = subs S2 tau1
             val (nv,beta) = newvar nv
             val V = unify S2tau1 (tau2 --> Tyvar beta)
             val Vbeta = subs V (Tyvar beta)
             val VS2S1 = compose V (compose S2 S1)
         in
             (nv, (VS2S1, Vbeta))
         end
    | (Abs(v,e)) =>
         let val (nv,beta) = newvar nv
             val (nv,(S1,tau1)) = W nv ((v,Type (Tyvar beta))::Gamma) e
             val S1beta = subs S1 (Tyvar beta)
         in
            (nv, (S1,(S1beta --> tau1)))
         end
    | (Let((v,e1),e2)) =>
         let val (nv, (S1,tau1)) = W nv Gamma e1
             val (nv, S1Gamma) = assumsubs nv S1 Gamma
             val (nv, (S2,tau2)) =
                   W nv ((v,tsclosure S1Gamma tau1)::S1Gamma) e2
             val S2S1 = compose S2 S1
         in
            (nv, (S2S1,tau2))
         end

fun principalts Gamma e =
   let val (var, (S, tau)) = W (lastfreeassumvar Gamma) Gamma e
       val (_,SGamma) = assumsubs var S Gamma
   in
      tsclosure SGamma tau
   end
```

Figure 5: The algorithm W.

The code in figure 5 declares a datatype `exp` for expressions in the language and implements W. The function `W` takes an extra parameter which is the index of the next free variable in the set of assumptions and the type-scheme. It then returns the new value of this as the first element of a pair, the second of which is the pair consisting of a substitution and the type of the expression. `W` may fail with either an exception `Unify` generated in the unification or with the exception `Assum` which happens when there are no assumptions for a free variable in an expression. Finally the function `principalts` is defined which takes assumptions Γ and an expression e, finds the highest indexed free variable ocurring in the assumptions and calls `W` to find a substitution S and a type τ for e. It then applies S to the assumptions and closes the type τ with respect to the new assumptions before returning the resulting principal type-scheme for e.

The function `W` uses the right-associative function space type constructor `-->` to represent function types using the special type constant `%f`. Otherwise it assumes nothing of the types beyond those in the `Gamma` list.

The functions in figure 6 construct terms and types. The left-associative `@:` is function application. The function `num` constructs the Church numerals.

In total, excluding the code for constructing and printing the terms and types, the implementation is less than 250 lines of standard ML. The code should run when copied and pasted from the PDF file of this document, but it can also be downloaded in a single contiguous text file from `http://ian-grant.net/hm`

```
fun mk_func name args = Tyapp(name,args)
fun mk_nullary name = mk_func name []
fun mk_unary name arg = mk_func name [arg]
fun mk_binary name arg1 arg2 = mk_func name [arg1, arg2]
fun mk_ternary name arg1 arg2 arg3 = mk_func name [arg1, arg2, arg3]

fun pairt t1 t2 = mk_binary "pair" t1 t2
fun listt t = mk_unary "list" t
val boolt = mk_nullary "bool"

(* Type variables *)

val alpha = Tyvar "a"
val beta = Tyvar "b"
val alpha' = Tyvar "a'"
val beta' = Tyvar "b'"

(* Type-schemes *)

fun mk_tyscheme [] t = Type t
  | mk_tyscheme ((Tyvar v)::vs) t = Forall (v, mk_tyscheme vs t)

(* Lambda expressions with let bindings *)

fun labs (Var v) e = Abs(v,e)
fun llet (Var v) e1 e2 = Let((v,e1),e2)

infix @:
fun e1 @: e2 = Comb(e1,e2)

fun lambda [] e = e
  | lambda (x::xs) e = labs x (lambda xs e)

fun letbind [] e = e
  | letbind ((v,e1)::bs) e = llet v e1 (letbind bs e)

fun lapply r [] = r
  | lapply r (e::es) = lapply (r @: e) es

(* Variables *)

val x = Var "x"
val y = Var "y"
val z = Var "z"
val p = Var "p"
val f = Var "f"
val m = Var "m"
val n = Var "n"
val s = Var "s"
val i = Var "i"

(* Church numerals *)

fun num n =
  let val f = Var "f"
      val x = Var "x"
      fun iter r 0 = lambda [f,x] r
        | iter r n = iter (f @: r) (n-1)
  in
     iter x n
  end
```

Figure 6: Constructing Types and Terms.

Now we can use these to construct and type the expression `S0` whose type was derived in Appendix C.

```
val ZERO = num 0
val S = lambda [n,f,x] (n @: f @: (f @: x));
ppts (principalts [] (S @: ZERO));
```

This is the polymorphic let expression used as an example in [2]:

```
val polylet = letbind [(i,lambda [x] x)] (i @: i);
ppts (principalts [] polylet);
```

And we can type the `map` definition given in the same paper.

```
val condts = mk_tyscheme [alpha] (boolt --> alpha --> alpha --> alpha)
val fixts = mk_tyscheme [alpha] ((alpha --> alpha) --> alpha)

val nullts = mk_tyscheme [alpha] (listt alpha --> boolt)
val nilts = mk_tyscheme [alpha] (listt alpha)
val consts = mk_tyscheme [alpha] (alpha --> listt alpha --> listt alpha)
val hdts = mk_tyscheme [alpha] (listt alpha --> alpha)
val tlts = mk_tyscheme [alpha] (listt alpha --> listt alpha)

val pairts = mk_tyscheme [alpha, beta] (alpha --> beta --> pairt alpha beta)
val fstts = mk_tyscheme [alpha, beta] (pairt alpha beta --> alpha)
val sndts = mk_tyscheme [alpha, beta] (pairt alpha beta --> beta)

val bool_assums = [("true",Type(boolt)),("false",Type(boolt)),("cond",condts)]
val pair_assums = [("pair",pairts),("fst",fstts),("snd",sndts)]
val fix_assums = [("fix",fixts)]
val list_assums = [("null",nullts),("nil",nilts),
                   ("cons",consts),("hd",hdts),("tl",tlts)]

(* let map = (fix (λ map f s.
                  (cond (null s) nil
                        (cons (f (hd s)) (map f (tl s)))))) in map *)

val assums = bool_assums@fix_assums@list_assums

val map' = Var "map"
val fix = Var "fix"
val null' = Var "null"
val nil' = Var "nil"
val cond = Var "cond"
val cons = Var "cons"
val hd' = Var "hd"
val tl' = Var "tl"

val mapdef =
   letbind [(map',
           (fix @: (lambda [map', f, s]
                        (cond @: (null' @: s)
                              @: nil'
                              @: (cons @: (f @: (hd' @: s))
                                       @: (map' @: f @: (tl' @: s))))))))]
            map';

ppassums assums;
ppexp mapdef;
val mapdefts = principalts assums mapdef;
ppts mapdefts;
```

Here is the predecessor function **PRED**.

```
val PAIR = (lambda [x, y, f] (f @: x @: y))
val FST = (lambda [p] (p @: (lambda [x, y] x)))
```

```
val SND = (lambda [p] (p @: (lambda [x, y] y)))

val G = lambda [f,p] (PAIR @: (f @: (FST @: p)) @: (FST @: p))
val PRED = lambda [n] (SND @: (n @: (G @: S) @: (PAIR @: ZERO @: ZERO)))
val SUB = lambda [m, n] (n @: PRED @: m);

ppts (principalts [] PRED);
ppts (principalts [] (PRED @: (num 6)));
ppts (principalts [] SUB);
```

Finally, Mairson [4] constructed a pathological expression where the principal type grows exponentially with the number of repeated iterations of pairing. In standard ML this expression is

```
let fun pair x y z = z x y
    fun x1 y = pair y y
    fun x2 y = x1 (x1 y)
    fun x3 y = x2 (x2 y)
    fun x4 y = x3 (x3 y)
    fun x5 y = x4 (x4 y)
in
   x5 (fn z => z)
end;
```

This takes several seconds for Moscow ML to type and the resulting scheme is 40–50,000 lines of output. The term below is more feasible.

```
val x1 = Var "x1"
val x2 = Var "x2"
val x3 = Var "x3"
val x4 = Var "x4"
val x5 = Var "x5"
val pair = Var "pair"

val mairson =
   letbind
     [(pair,lambda [x,y,z] (z @: x @: y)),
      (x1,lambda [y] (pair @: y @: y)),
      (x2,lambda [y] (x1 @: (x1 @: y))),
      (x3,lambda [y] (x2 @: (x2 @: y))),
      (x4,lambda [y] (x3 @: (x3 @: y)))
 ] (x4 @: (lambda [x] x));

ppts (principalts [] mairson);
```

The code here can type the expression with five iterations. On my machine it takes over 50 seconds! The printing routine `ppts` is not tail-recursive, so it chokes on something of this size.

```
val mairson =
   letbind
     [(pair,lambda [x,y,z] (z @: x @: y)),
      (x1,lambda [y] (pair @: y @: y)),
      (x2,lambda [y] (x1 @: (x1 @: y))),
      (x3,lambda [y] (x2 @: (x2 @: y))),
      (x4,lambda [y] (x3 @: (x3 @: y))),
      (x5,lambda [y] (x4 @: (x4 @: y)))
 ] (x5 @: (lambda [x] x));

principalts [] mairson;
```

Appendices

A Standard ML

This is not intended as a tutorial on writing standard ML. It is just enough to let someone who has some experience with programming languages to *read* the code I present here. It would be very frustrating indeed to attempt to learn to write ML from this. There are far better texts available for that. Of particular note are Mads Tofte's very concise *Tips for Computer Scientists on Standard ML*, available from `http://www.itu.dk/people/tofte` which, in just 20 pages, covers 90% of the language, and Larry Paulson's book *ML for the Working Programmer* [6].

Variable values can be declared using expressions like

```
val p = (1,4)
```

which makes `p` a pair of integers. The expression

```
fun f x = x + 1
```

is syntactic sugar for

```
val f = fn x => op + (x, 1)
```

The prefix `op` makes an infix operator an ordinary function. Binary operators are always functions on pairs.

Recursive abstractions can be defined using `rec` like this

```
val rec len = fn n => fn l => if l = [] then n else len (n+1) (tl l);
```

An environment containing variable bindings can be established using the `let` expression

```
let val x = 4
    fun square x = x * x
    val x = 3
    val y = (square x) - 1
in
   x + (square y)
end
```

later bindings may refer to or shadow earlier ones. Functions may call themselves, but may not call others in the same let construct however, unless they are declared with `and` instead of `fun`. For example

```
let fun odd 0 = false
      | odd n = even (n-1)
    and even 0 = true
      | even n = odd (n-1)
in
   odd 15
end
```

Lists can be constructed using expressions like `1::2::3::nil` which is the same as `[1,2,3]`. Lists can be deconstructed using `hd` and `tl` but these are not used very often because ML has *pattern matching*:

```
fun len [] = 0
  | len (_::xs) = 1 + len xs
```

Here the argument of the function will be matched against one of the two mutually exclusive and exhaustive possibilities for the type of lists. The first match is with the empty list `nil` and causes the function to return `0`. If that pattern does not match then the next and last one is tried, which binds the variable `xs` to the tail of the list. The underscore _ is the *wild-card* pattern which matches anything because in this case we are not interested in the actual elements of the list, only their number.

The appearance of `=>` in the lambda notation `fn x => x` is a hint that in fact a pattern can be used in lambda definitions. Thus one may write

```
val f = fn 0 => 5
         | n => n + 1
```

This is seldom done except with simple patterns like

```
fn (x,_) => x
```

Patterns can be used in `val` declarations as well as to define any value via `case` expressions as in this contrived example

```
fun front l =
   let val (x,y) =
       case l of [] => (0,1)
              | ((x,y)::_) => (x+1,y+1)
in
   x + y
end
```

which is just

```
fun front [] = 1 | front ((x,y)::_) = x+y+2
```

Assignment to a pattern with literals like `0` or `[]` can be used to check a value because a `Bind` exception is thrown if the pattern doesn't match. It can also be used to deconstruct returned values 'automatically'. For example

```
val (x,y) = (fn m => (m-5,m)) 5
```

is the same as

```
val x = 0
val y = 5
```

Quite often one needs to deconstruct a value with a pattern and also use the whole thing. This can be done with sub-patterns using the keyword `as` and it is better than reconstructing the value from its parts. For example

```
val f =  fn (l as ((p as (t1,t2))::t)) => if t1=1 then l else p::(1,2)::t
```

The constants `true` and `false` are the type constructors for the type `bool`. The statements `andalso` and `orelse` are *lazy* in that the second argument is only evaluated if the first evaluates to `true` and `false` respectively.

Strings are enclosed in double quotes and string concatenation is via the binary operator `^`. Characters are a different type, constructed using e.g. `#"c"` or the function `chr` which takes an integer and returns the corresponding character. A character can be converted to a string using `str` and to an integer using `ord`. The built-in function `explode` takes a string a returns a list of characters.

Vectors of any types can be constructed and deconstructed using e.g.

```
val v = (1,"2",#"3");

#2 v
```

The empty vector `()` is typically called *unit*. It has a type which is called `unit`. Whereas the type *bit*, which for historical reasons is called `bool`, has two possible values, the unit type has only one.

The main use for the unit type is to delay evaluation. The ML evaluation is eager in that it evaluates all arguments before evaluating the application of a function. However, it won't 'evaluate under a lambda' which is to say that a function body will not be evaluated until it is applied to an argument value.

ML has *datatypes*. These are new types which are constructed from existing types by defining *constructors* which are functions which construct elements of the new type given the values of the constituent parts.

Polymorphic lists can only be of one uniform type α. That is we can have a list of integers or list of strings or a list of characters but we cannot have a list like `[1, "2", #"3"]`. However we can define a new *sum type*

```
datatype isc = Int of int
             | Str of string
             | Chr of char
```

Constructors like `Int`, `Str` etc. must start with an upper-case letter.

ML then allows

```
[Int 1, Str "2", Chr #"3"]
```

and we can write functions that operate on such lists using pattern matching.

```
fun homogenise r [] = rev r
  | homogenise r (x::xs) =
      let val n =
         case x of Int n => n
                 | Str s => if s = "" then 0 else ord (hd (explode s))
                 | Chr c => ord c
      in
         homogenise (n::r) xs
      end

homogenise [] [Int 1, Str "2", Chr #"3"]
```

Sum types are typically deconstructed using case analysis like this.

The constructors for datatypes must always declared all at once in a single statement otherwise the types could change as evaluation proceeds. ML *exceptions* are an exception to this. There is a datatype `exn` and declaring an exception using a statement like

```
exception Error of string
```

declares an additional constructor for the type `exn`.

The statement `raise Error "message"` can then appear anywhere a value may appear and if evaluated evaluation will continue with the most recent matching handler declared using the `handle` statement. For example

```
fun r () = raise Error "message";

(r ()) handle Error m => "Exception: Error "^m^" caught."
```

Here the expression after `=>` is a pattern which matches some but not necessarily all constructors of the `exn` datatype. If there is no matching handler then the execution of the program will terminate with a message like 'Uncaught exception.'

This is also an example of the use of the unit type to delay evaluation.

A *recursive datatype* has proper parts which are of the type being defined. For example a datatype representing simple arithmetic expressions might look like this

```
datatype expr = Sum of expr * expr
              | Prod of expr * expr
              | Neg of expr
              | Val of int
```

Given this type declaration one may construct values using the resulting type constructor functions:

```
val e = Prod(Sum(Val 1,Val 3),Neg(Val 4))
```

This is *abstract syntax* because the resulting structure is unambiguous. If this expression had been written out it would have required parentheses or precedence rules to disambiguate the order of application. For example $(1+3)\times -4$.

One may then write a function to evaluate such expressions using pattern matching

```
fun eval (Val n) = n
  | eval (Neg n) = ~(eval n)
  | eval (Prod(e1,e2)) = (eval e1) * (eval e2)
  | eval (Sum(e1,e2)) = (eval e1) + (eval e2)

eval e;
```

Note that unary minus in ML is the operator `~` not the ordinary minus sign.

ML lets you declare new operators and assign precedence and associativity. The following creates a pair of operators `^<:` and `:>^` on pairs of pairs. The puzzle is to work out what the final expression evaluates to without running it.

```
fun op :>^ ((a,b), (c,d)) = (b,d)
fun op ^<: ((a,b), (c,d)) = (c,a)

infixr 1 :>^
infix 2 ^<:;

(1,2) ^<: (3,4) :>^ (5,6) ^<: (7,8) :>^ (9,0)
```

B Computing $1-1$

Here are the seventeen reductions that get **PRED 1** to normal form:

$(\lambda n.(\lambda p.p\,\lambda x.\lambda y.y)(n\,((\lambda f.\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)(f\,((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))\,\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)))[\lambda f.\lambda x.f\,x]$
$\to(\lambda p.p\,\lambda x.\lambda y.y)[(\lambda f.\lambda x.f\,x)((\lambda f.\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)(f\,((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))\,\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)]$
$\to(\lambda f.\lambda x.f\,x)[(\lambda f.\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)(f\,((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))\,\lambda n.\lambda f.\lambda x.n\,f\,(f\,x)]((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)\,\lambda x.\lambda y.y$
$\to(\lambda x.(\lambda f.\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)(f\,((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))\,x)[(\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x]\,\lambda x.\lambda y.y$
$\to(\lambda f.\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)(f\,((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))[\lambda n.\lambda f.\lambda x.n\,f\,(f\,x)]((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)\,\lambda x.\lambda y.y$
$\to(\lambda p.(\lambda x.\lambda y.\lambda f.f\,x\,y)((\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda p.p\,\lambda x.\lambda y.x)\,p))((\lambda p.p\,\lambda x.\lambda y.x)\,p))[(\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x]\,\lambda x.\lambda y.y$
$\to(\lambda x.\lambda y.\lambda f.f\,x\,y)[(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x))]((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x))\,\lambda x.\lambda y.y$
$\to(\lambda y.\lambda f.f\,((\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)))\,y)[(\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)]\,\lambda x.\lambda y.y$
$\to(\lambda f.f\,((\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)))((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)))[\lambda x.\lambda y.y]$
$\to(\lambda x.\lambda y.y)[(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x))]((\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x))$
$\to(\lambda y.y)[(\lambda p.p\,\lambda x.\lambda y.x)((\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)]\to(\lambda p.p\,\lambda x.\lambda y.x)[(\lambda x.\lambda y.\lambda f.f\,x\,y)(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x]$
$\to(\lambda x.\lambda y.\lambda f.f\,x\,y)[\lambda f.\lambda x.x]\,(\lambda f.\lambda x.x)\,\lambda x.\lambda y.x\to(\lambda y.\lambda f.f\,(\lambda f.\lambda x.x)\,y)[\lambda f.\lambda x.x]\,\lambda x.\lambda y.x$
$\to(\lambda f.f\,(\lambda f.\lambda x.x)\,\lambda f.\lambda x.x)[\lambda x.\lambda y.x]\to(\lambda x.\lambda y.x)[\lambda f.\lambda x.x]\,\lambda f.\lambda x.x$
$\to(\lambda y.\lambda f.\lambda x.x)[\lambda f.\lambda x.x]$
$\to\lambda f.\lambda x.x$

Contrast this with the converse operation of computing $0+1$ which is the reductions in **SUCC 0**:

$(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))[\lambda f.\lambda x.x]\to\lambda f.\lambda x.(\lambda f.\lambda x.x)[f]\,(f\,x)\to\lambda f.\lambda x.(\lambda x.x)[f\,x]\to\lambda f.\lambda x.f\,x$

C Type derivation

For the term $(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))$ we have

$$\dfrac{\dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash n:(\tau\to\tau')\to\tau''\to\tau''}\,\text{TAUT:}\qquad \dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash f:\tau\to\tau'}\,\text{TAUT:}}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash n\,f:\tau''\to\tau''}\,\text{COMB:}\;\dagger$$

$$\dfrac{\dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash f:\tau\to\tau'}\,\text{TAUT:}\qquad \dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash x:\tau}\,\text{TAUT:}}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash(f\,x):\tau'}\,\text{COMB:}\;\ddagger$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash n\,f:\tau''\to\tau''}\,\dagger\qquad \dfrac{}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash(f\,x):\tau'}\,\ddagger}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau',x:\tau\vdash n\,f\,(f\,x):\tau'}\,\text{COMB:}}{n:(\tau\to\tau')\to\tau''\to\tau'',f:\tau\to\tau'\vdash\lambda x.n\,f\,(f\,x):\tau\to\tau'}\,\text{ABS:}}{n:(\tau\to\tau')\to\tau''\to\tau''\vdash\lambda f.\lambda x.n\,f\,(f\,x):(\tau\to\tau')\to\tau\to\tau'}\,\text{ABS:}}{\vdash\lambda n.\lambda f.\lambda x.n\,f\,(f\,x):((\tau\to\tau')\to\tau''\to\tau'')\to(\tau\to\tau')\to\tau\to\tau'}\,\text{ABS:}\;(**)$$

Hence

$$\dfrac{\dfrac{}{\vdash\lambda n.\lambda f.\lambda x.n\,f\,(f\,x):((\tau\to\tau')\to\tau''\to\tau'')\to(\tau\to\tau')\to\tau\to\tau'}\,(**)\qquad \dfrac{}{\vdash(\lambda f.\lambda x.x):(\tau\to\tau')\to\tau''\to\tau''}\,(*)}{\vdash(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))\,\lambda f.\lambda x.x:(\tau\to\tau')\to\tau\to\tau'}\,\text{COMB:}$$

Therefore $(\lambda n.\lambda f.\lambda x.n\,f\,(f\,x))\,\lambda f.\lambda x.x$ is well-typed whatever the actual types τ and τ'.

D Printing Types

```
fun pptsterm tau =
   let fun iter prec (Tyvar name) = ""^name
     | iter prec (Tyapp(name,[])) = name
     | iter prec (Tyapp("%f",[a1,a2])) =
         let fun maybebracket s = if prec <= 10 then s else "("^s^")"
         in
            maybebracket ((iter 11 a1)^" -> "^(iter 10 a2))
         end
     | iter prec (Tyapp(name,args)) =
         let fun arglist r [] = r
               | arglist r (h::t) =
                   arglist (r^(iter 30 h)^(if t=[] then "" else ", ")) t
         in
            if (length args) > 1 then (arglist "(" args)^") "^name
                                 else (arglist "" args)^" "^name
         end
   in
       iter 10 tau
   end

fun ppterm (Tyvar name) = name
  | ppterm (Tyapp(name,[])) = name
  | ppterm (Tyapp(name,args)) =
      let fun arglist r [] = r
            | arglist r (h::t) =
                arglist (r^(ppterm h)^(if t=[] then "" else ",")) t
      in
         name^(arglist "(" args)^")"
      end

fun ppsubs s =
   let fun iter r [] = r^"]"
         | iter r ((term,var)::t) =
             iter (r^(ppterm term)^"/"^var^(if t=[] then "" else ",")) t
    in iter "[" s end

fun ppexp e =
   let fun ppe r e =
      case e of
         (Var v) => r^v
       | (Comb(e1,e2)) => r^"("^(ppe "" e1)^" "^(ppe "" e2)^")"
       | (Abs(v,e)) => r^"(\"^v^"."^(ppe "" e)^")"
       | (Let((v,e1),e2)) => r^"let "^v^"="^(ppe "" e1)^" in "^(ppe "" e2)
   in
      ppe "" e
   end

fun ppts sigma =
   let fun iter r (Forall(sv,sts)) = iter (r^"!"^sv^".") sts
         | iter r (Type term) = r^(pptsterm term)
   in
      iter "" sigma
   end

fun ppassums Gamma =
   let fun iter r [] = r
         | iter r ((v,ts)::assums) =
             iter (r^v^":"^(ppts ts)^(if assums=[] then "" else ",")) assums
   in
      iter "" Gamma
   end
```

References

[1] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(4):56–68, June 1940.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. POPL, ACM, 1982. http://iangrant.net/hm/milner-damas.pdf.

[3] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the AMS*, 146:29–60, 1969.

[4] H.G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM symposium on principles of programming languages*, pages 382–401. POPL, ACM, 1990.

[5] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17(3):348–375, 1978.

[6] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[7] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.