LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Practical Polymorphism

Computation Structures Group Memo 249
July 9, 1985

Rishiyur S. Nikhil

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Practical Polymorphism

Rishiyur S. Nikhil

MIT Laboratory for Computer Science,
545 Technology Square,
Cambridge, Massachusetts 02139
U.S.A.

Abstract

Polymorphic type systems as proposed by Milner and implemented in the programming language ML offer rich types, unobtrusive compile-time type-checking, and complete type-safety in functional languages. However, straightforward addition of such a type system to languages with interactive environments such as FQL, SASL or Scheme can inhibit seriously the top-down, incremental programming style characteristically employed with them. The problems include error-recovery during type-checking, type-checking with incomplete information and incremental type-checking during program development. We describe these problems and present an integrated solution as prototyped in FQL.

1. Introduction

Polymorphic type systems as proposed by Milner in [10] and implemented in the programming language ML [8, 6] have won increasing popularity in recent years. This is evidenced by their adoption in several other languages such as FQL [3, 4, 12], Hope [5], Galileo [1] and Prolog [11]. Such type systems offer the promise of *unobtrusive* compile-time type-checking together with complete type-safety in functional languages.

However, it is not trivial to add such a type system to functional languages with interactive programming environments such as, say, SASL [16] or SCHEME [14]. If implemented naively, the use of a polymorphic type system raises practical problems that can inhibit seriously the top-down, incremental programming style characteristically employed when programming in these languages.

These problems are identified in Section 3. In Section 4 we present solutions to these problems. Our solutions have been prototyped successfully in the context of the experimental database programming language FQL [12], but are generally applicable.

Partly to set the stage for later discussion, and partly for readers not familiar with polymorphic type systems, section 2 contains a brief review of an ML-style type system and type-checker.

2. Polymorphic Type Systems and Type-Checking

This section reviews, in greatly abbreviated form, an ML-style polymorphic type system and its type-checker. The reader is invited to consult [10], [7] and [6] for detailed discussions of this subject.

2.1. Types and Polymorphism

We employ a language of *expressions* that denote values. Every expression has a value, and its *type* is described by a *Type Expression*. Type expressions are built up recursively by applying *type operators* (such as prefix "list" and infix "→") to *constant types* ("int", "bool", ...) and *type variables* ($\alpha, \beta, ...$). The following are all examples of type expressions:

int
bool
string
α
list int
(list α_1) → (list bool)

A type expression with no type variables (*i.e.* a ground type expression) is *monomorphic*; otherwise it is *polymorphic*. A value with a polymorphic type possesses all types that are monomorphic instances of that type.

For example, consider a function "filter" that returns just those elements of a list that satisfy some boolean predicate "p":

```
filter p nil      = nil
filter p (hd::tl) = if (p hd) then hd::(filter p tl)
                    else filter p tl
```

(here "nil" is the empty list and "::" is an infix "CONS" for lists). For example the expressions

```
filter  evenNumber  [1;2;3;4;5;6; ... ]

filter affirmative ["ja"; "nyet"; "oui"; "yes"; "nein"; ... ]
```

evaluate to the lists

```
[2;4;6; ... ]
["ja"; "oui"; "yes"; ... ]
```

respectively. The function "filter" is a polymorphic function. It has the polymorphic type

$$(\alpha \rightarrow \text{bool}) \rightarrow ((\text{list } \alpha) \rightarrow (\text{list } \alpha))$$

and thus it has all types

$$(\text{int} \rightarrow \text{bool}) \rightarrow ((\text{list int}) \rightarrow (\text{list int})) \ ,$$
$$(\text{string} \rightarrow \text{bool}) \rightarrow ((\text{list string}) \rightarrow (\text{list string})) \ ,$$
$$\cdot$$
$$\cdot$$

i.e. the same function may be used at any of those types.

2.2. The Type-Checker

A characteristic feature of many polymorphic type systems is that type declarations are often *optional*. The type-checker automatically *infers* a type for every definition. If the user chooses to declare its type (e.g. for documentation purposes), the type-checker makes sure that the declared type is consistent with the inferred type. The type-checker guarantees type-safety-- no run-time type-checking need be performed.

The type-checker infers a type for a definition based on an initial *Type Environment* which assigns a type to every free identifier in the definition. This type environment typically specifies a type for each built-in (pre-defined) identifier, and for all previously entered definitions.

The type-checking algorithm is very simple and efficient. It makes use of the unification algorithm [13] to make sure that certain constraints between the types of expressions are satisfied. These *Well-Typing* constraints arise primarily in ensuring that the actual type of an argument "is the same as" the type of the expected argument type of a function.

For example if "evenNumber" is a predicate on numbers and "filter" is applied to "evenNumber", the type of "evenNumber" (*i.e.* int $\rightarrow$ bool) is unified with the argument component of the type of "filter" (*i.e.* $\alpha \rightarrow$ bool). This constrains α to be equal to "int", which in turn constrains the result component of the type of "filter" (*i.e.* (list α) $\rightarrow$ (list α)) to be (list int) $\rightarrow$ (list int). Thus the value of "(filter evenNumber)" is a function that may be applied only to integer lists.

In Figure 2-1 we present the type-checking algorithm (greatly simplified) for just three syntactic forms of expressions (identifiers, applications and lambda-abstractions); it can be extended routinely to handle other syntactic forms. The inputs are: an expression to be type-checked, a type environment, and an initial *substitution* of type expressions for type variables. The outputs are: a type for the expression, and a new substitution.

The algorithm is initiated with an empty substitution. As it traverses the expression

recursively, repeated unification steps continually augment and refine this substitution. Each unification step is of the form

$$unify(typeExpr_1,\ typeExpr_2,\ subst) \Rightarrow subst'$$

i.e. it unifies two type expressions with respect to an initial substitution and produces an augmented substitution. If unification fails, the returned "substitution" will indicate that an error occurred.

```
typeCheck (expr, TypeEnv, subst) =                    {Returns (type,subst')}

  CASE expr OF
    x {identifier} :
        let type_x = lookup type of x in TypeEnv
        in
            type_x,
            subst

    (f a) {application} :
        let type_f, subst1 = typeCheck(f,TypeEnv,subst)  ;
            type_a, subst2 = typeCheck(a,TypeEnv,subst1) ;
            α  =  a new type variable ;
            subst3 = unify(type_f, type_a → α, subst2)
        in
            α,
            subst3

    λx.e {lambda-abstraction} :
        let α  =   a new type variable ;
            TypeEnv' = extend TypeEnv with (x:α) ;
            type_e, subst1 = typeCheck(e, TypeEnv', subst)
        in
            α → type_e,
            subst1
```

Figure 2-1: A Simple Polymorphic Type-checker

2.3. Type Errors

The type-checker rejects an expression when it is unable to proceed due to either of two conditions:

- an identifier for which the type environment does not specify a type, or
- unification fails

In the former case, that identifier is an unbound identifier; the latter case indicates that there was an improper application of a function to an argument.

3. Practical Problems in the Use of the Type System

These two features of ML-style types-- the economy of expression that comes with polymorphic objects, and automatic type-inference-- together hold out the promise of the *unobtrusive* introduction of a rich type system and the benefits of type-safety into an otherwise "type-less"

language such as FQL, SASL or Scheme. However, when such a type checker is introduced into these interactive programming languages, some practical problems arise which, if confronted naively, can inhibit seriously the attractive top-down, incremental program development style charecteristic of these language environments. These problems may be viewed at two levels: at the level of individual definitions, and at the "system" level (collections of top-level definitions).

3.1. At the Level of Individual Definitions

The problem here is that the type-checker as presented has no provision for error-recovery (other than reporting an error as soon as it finds one and rejecting the entire expression).

In Unix ML [6], for example, the type-checker gives up checking an expression as soon as it finds a single type error. The user then must re-enter the entire (corrected) expression, at which point a second type error may be discovered. Repeating this process several times, uncovering one type error at a time is extremely tedious. Most compilers for other languages (such as Pascal) do better, trying to discover and report as many errors as possible within a single compilation attempt. To be fair, an automatic type-checker like ML's has a more difficult job; it usually works with no user-specified type information, so that it does not have ready-made "anchor points" to which it can safely recover and then proceed.

In interactive programming, one often tests one part of a definition by giving it a specific input argument, knowing full well that another part of the definition that is not of immediate concern is still unfinished. For example, in writing a lexical analyser, one may test it for reading integers before fixing errors in another part that deals with reals. It should not then matter that the unexercised part of the definition has type-errors in it. Thus we would like to be able to type-check expressions partially.

3.2. At the System Level (Collections of Definitions)

In an interactive programming environment, we would like to operate the type-checker incrementally, normally each time we enter a definition. However, the type-checker presented earlier requires a type environment that specifies a type for each free identifier in the expression. Typically, these free identifiers include all primitive identifiers (e.g. " + ") and all other definitions used by this expression. Unfortunately, in interactive program development as in FQL one cannot usually provide such a type environment incrementally to the type-checker.

First, we prefer to design and construct programs (and to present them in text form) in a top-down order. But requiring that the type of free identifiers in a definition be known beforehand forces the user to enter his definitions in a strictly "bottom-up" order, *i.e.* an identifier must be defined before it can be used. (Mutually recursive definitions are always entered together; the details are not relevant here).

In the top-down refinement of a program, the user often wishes to test selected parts of a definition before all subsidiary definitions are available. For example, in writing the "eval" function of

an interpreter, one may wish to test it on the easy cases first (e.g. numbers) before turning ones attention to the difficult cases (e.g. closures). This means the ability to run a program knowing that certain sub-expressions will not be evaluated. It should not matter if those sub-expressions refer to identifiers that may be missing completely from the type environment.

In an interactive system, definitions may be compiled incrementally, but the resolution of identifiers defined at the top-level is usually deferred until just before the evaluation of an expression or even till run-time. This allows one to *edit* a definition and allow other expressions automatically to receive this latest value. (This is *not* to be confused with "dynamic scoping", as is alleged in [15]).

There are many reasons why the user would like to *edit* previously entered definitions, apart from fixing bugs. For example, in writing an interpreter whose top-level is a *read-eval-print* loop, one typically wishes to test the *read* and *print* functions before testing the *eval* function. We do this by initially coding *eval* as the identity function, and later editing it to fill in the details. Similarly, in writing a complex program, the user first may write all definitions in a style that prefers perspicacity, and only later edit them for efficiency.

But editing a definition may result in a change of its type, thus invalidating the type environment under which other definitions were type-checked.

3.3. Why ML does not Measure Up

The following comments apply to Unix ML [6]; we do not know if they also apply to other ML implementations [8, 2].

ML *is* interactive in the sense that a user may type in definitions from the terminal and have them evaluated incrementally. However, the system imposes certain restrictions which, while making type-checking easy, also preclude the interactive style that is so attractive in FQL.

First of all an ML program must be entirely type-correct before any part of it may be exercised. We feel that this is too restrictive. One should be able to run a program with type-incorrect regions, provided these regions are clearly marked, and any attempt to enter such a region during execution is trapped.

The scope of an identifier defined at the top-level in an ML session is always "the rest of the session". A definition is evaluated immediately, with free identifiers bound immediately in the previous environment. Thus identifiers must be defined before they are used, implying bottom-up program development; again, we find this unacceptable.

The user never *edits* any definition; he simply enters a second definition for the same identifier, and because of the scope rules, this hides the earlier one. Unfortunately, previous definitions that used that identifier are still bound to the old value; hence, to get truly the effect of an edit, all those definitions must be entered again. This is generally too tedious to do by hand; one usually runs an editor process simultaneously, reloading an edited subfile into ML. We feel that this is

a clumsy and roundabout way of obtaining the effect of an edit.

4. A More Flexible Type-Checking Discipline

For a polymorphic type-checker truly to fulfill its promise, it must therefore have some error-recovery mechanism and be able to work *without* a clear, stable set of assumptions about the types of all free identifiers in a given expression.

By appropriate modifications to the type-checker and the system-level typing discipline, we show that this is indeed possible. Programs can be developed in a top-down manner, and may be run with complete type-safety in spite of partial or missing definitions. Regions of the program with type-errors are marked automatically and can be selectively corrected so that a "finished" program is certifiably free of type-errors. We believe that this allows us to enjoy the benefits of a rich type system without sacrificing the productivity of incremental, interactive programming *ala* FQL, SASL or Scheme.

4.1. At the Level of Individual Definitions

We first turn our attention to improving the type-checker so that it gives more meaningful information when it finds a type-error, and so that it can continue operation even after it has discovered a type-error. In addition, we would like the type-checker to infer a type for an expression in spite of type-errors, if possible. For example, in the following definition,

```
f x = if (... sub-expression with type error ...)
        then x + 1
        else x - 1
```

we would like to infer the type "int → int" for "f" even though the condition part of the "if" expression has a type error. This could be of use while type-checking other functions that use "f". Of course, we must ensure that we never attempt to *evaluate* any sub-expression with a type-error.

The type-checking algorithm may be viewed as a "bottom-up" algorithm: starting from the types at the leaves (known from the type environment), it infers types of larger and larger sub-expressions. In order to pinpoint type-errors, it is more fruitful to invert this view, and re-formulate it as a "top-down" algorithm: we begin with an *expected type* for the entire expression, and recursively infer expected types for each of its subexpressions. At every stage, we check if the type of the sub-expression indeed matches the expected type (by unification). If it does not match, we report a type-error, *assume that it had matched*, and continue type-checking the rest of the expression.

At its simplest, the "expected type" may be totally unconstrained (*i.e.* just a type-variable). At various stages during the type-checking, the expected type may become more specific-- for example, if the user has declared explicitly the type of an expression, or if the argument to a function becomes constrained to be of some particular type. A sketch of the re-formulated algorithm is presented in

Figure 4-1. Note that it does not return a type anymore; it just returns a substitution. The full type of an expression may then be obtained by applying this substitution to the original expected type for the expression.

```
typeCheck (expr, expectType, TypeEnv, subst) =          {Returns  subst'}

   CASE expr OF
     x  {identifier} :
          if x is unbound in TypeEnv then
             report that x is unbound ;
             subst

          else
            let actualType = lookup type of x in TypeEnv ;
                subst1 = unify (expectType, actualType, subst) ;
            in
                if no unify-error in subst1 then
                  subst1
                else
                  report type-error(expr, expectType, actualType);
                  subst

    (f a)  {application} :
          let β  =  a new type variable ;
              subst1 = typeCheck(f,
                                 β→expectType,
                                 TypeEnv, subst) ;
              subst2 = typeCheck(a, β, TypeEnv, subst1) ;
          in
              subst2

    λx.e {lambda-abstraction} :
          let α, β  =  new type variables ;
              TypeEnv' = extend TypeEnv with (x:α) ;
              subst1 = typeCheck(e, β, TypeEnv', subst) ;
              subst2 = unify(expectType, α → β, subst1)
          in
              if no unify-error in subst2 then
                subst2
              else
                report type-error(expr, expectType, α → β) ;
                subst
```

Figure 4-1: An Improved Type-checker

At every type-error, we can identify the sub-expression that caused the type-error, and report the expected type and the actual type found. This is invaluable information for debugging.

Note that one cannot claim that the type-checker finds *the* type-error in an expression because of the difficulty of defining such a concept. For example, when an application "(f a)" is found to have a type-error, the culprit could be either "f" or "a". Our algorithm favours the function part by type-checking it first and allowing it to determine the expected type of the argument part. This is based on the observation that in most cases the function part is not a complicated expression and is less likely to be the cause of the type-error in the programmer's judgement. It also has some precedent in conventional type-checkers where the declared type of a function is treated as

"correct", and a type-error in an application is judged to be due to an incorrect argument.

Unbound identifiers are flagged and reported, but cause no problem: they are simply treated as initially having an unconstrained polymorphic type. Later (in Section 4.3) we shall see that we take some extra action with such identifiers in order that we not compromise type-safety.

The "expected type" information available on encountering a type error is very useful in compilation. The offending sub-expression is compiled as if it had been replaced by the special expression

```
typefault: expectType
```

which type-checks correctly.

The meaning of "typefault" could be to abort the program, if it is ever evaluated. This trap ensures that we never execute type-incorrect regions of the program. If a given execution of a program never evaluates "typefault", it will complete successfully (interestingly, the probability of this event increases in a lazy-evaluation regime as in FQL). This allows us to go ahead and test partially those parts of the program that do not contain type errors while deferring work on those parts which do.

In the spirit of interactive programming, however, we follow a more useful strategy: each "typefault" is tagged with the type that was expected at that point. At run time, if the evaluator encounters a "typefault", it recursively re-enters the evaluator, in a manner analogous to a "break" in Lisp systems. At this point, the user may supply explicitly an expression to be evaluated and returned as the value in place of the "typefault". We type-check the expression supplied by the user; because we know the expected type, we can reject the expression if the expected type is not a substitution of the expression's type (*i.e.* if the given expression's type is not "at least as polymorphic" as the expected type).

The "expected type" information could also be useful in a syntax-directed editor. When a change is made to some sub-expression, the type-checker may be able to use the already-known expected type to re-analyse it in isolation (and thus do it quickly). (We have not explored this possibility.)

Our experience has shown our type-checker to be quite successful at localizing type errors, catching multiple type errors in a single expression and inferring "reasonable" types for expressions in spite of the presence of type errors.

4.2. At the Level of Collections of Definitions

Armed with a more "liberal" (but still type-safe) type-checking algorithm at the level of individual definitions, we now turn our attention to collections of definitions that form a program. In particular, we wish to permit identifiers to be redefined (edited), while maintaining type-safety. The

redefinition may result in changing the type of an identifier, which in turn may invalidate the type environment for other functions.

This problem is analogous to the "module version" problem in software engineering: when a module is changed, all other modules that depend on it directly or indirectly may have to be recompiled. However, the polymorphic type system gives us a rich relationship between types that can better limit the propogation of changes. In most monomorphic type systems, the only relation (if any) between types is that of equality, so that when the type of one module changes in any way, the system must pessimistically recompile *every* module that depends on it.

In redefining "f", the change may not have affected its type at all (as, for example, in the frequent case where "f" merely is re-implemented with a better algorithm). In this case, all that is needed is the recompilation of "f" alone.

If the type of "f" indeed had changed, ML's "linear" approach may cause the recompilation of hundreds of unrelated definitions (all definitions textually following "f", whether they used "f" or not). Instead, a simple solution is to maintain a dependency graph over all definitions. There is a node in the graph for each definition. If "g" uses "f", there is an edge from "g" to "f". If there is any mutual recursion the graph will not be acyclic. This graph is built easily incrementally with information that is already available. When a definition "g" is entered, it is type-checked; it is trivial for the type-checker to collect all free identifiers in the expression, and these are precisely the definitions used by "g".

If "f" is redefined now, the dependency graph tells us which other functions may need re-analysis (type-checking). The algorithm to do this is given in Figure 4-2

```
When f is changed, re-analyze(f),  where

re-analyze(f) =

    Let fRec be
      the largest sub-graph that includes f,
      such that (for every pair of functions f1, f2 in fRec,
                 f1 and f2 are mutually recursive)

    Type-check the functions in fRec.

    For each function f in fRec,
      If its type has changed then
        For each g outside fRec that depends directly on f
           re-analyze(g)
```

Figure 4-2: Re-analysis Procedure Following a Redefinition

(At each stage, we first "grow" the candidate set of functions to be type-checked to include all mutually recursive functions because mutually recursive functions must be type-checked together).

We can strengthen further the decision whether to re-analyze the dependents of a function by

using the "<" relation between types. We say that $t_1 < t_2$ if t_1 is a substitution instance of t_2 (informally, t_1 is "less polymorphic" than t_2). For example, if

t_1 = (int → bool) → ((list int) → (list int))

t_2 = (α → bool) → ((list α) → (list α))

then $t_1 < t_2$ because there is a substitution

S = {α ⇒ int}

such that $S(t_2) = t_1$. Clearly, "<" is a transitive relation (because substitutions can be composed).

Suppose t_{fOld} is the type of "f" before it is redefined. Every function "g" dependent on "f" must use it at a more specific type $t_{fg} < t_{fOld}$. Thus if the new type t_{fNew} of "f" is no less polymorphic than t_{fOld} (*i.e.* if $t_{fOld} \leq t_{fNew}$), then the relation $t_{fg} < t_{fNew}$ must also hold by transitivity. The re-analysis of "g" when "f" changes thus need be done only if "f" has become less polymorphic, *i.e.* if $t_{fNew} < t_{fOld}$.

Extending this idea, we can narrow further the set of functions that are candidates for re-analysis by putting in more information in the dependency graph. We now *label* the edge from "g" to "f" with the set of types at which "f" is used in "g". This information is readily available anyway when "g" is type-checked.

For example, the function "filter" has a polymorphic type, but may be used at only the specific type

(int → bool) → (list int) → (list int)

within a definition of "allEvenNumbers". Thus if "filter" was redefined to be a function that operated only on integer lists, "allEvenNumbers" would still be valid.

The arrow from "g" to "f" is now labelled with the *set* of types $t_1, ..., t_n$ at which "f" is used in "g". When "f" is redefined, "g" needs to be re-analyzed only if the new type of "f" is less polymorphic than one of the types at which "g" actually uses "f". The improved procedure is given in Figure 4-3:

4.3. Unbound Identifiers

We now also have the tools to handle "unbound" identifiers encountered by the type-checker. For any identifer "x" encountered unbound during type-checking some definition, we merely introduce the following definition into the top-level environment:

val x:α = typefault

i.e. we automatically introduce a *stub* (a dummy definition) for the identifier. The dependency graph maintains all the types at which "x" is used in other functions. When "x" is defined later, this

```
When f is changed, re-analyze(f),  where

re-analyze(f) =

    Let fRec be
      the largest sub-graph that includes f,
      such that (for every pair of functions f1, f2 in fRec,
                 f1 and f2 are mutually recursive)

    Type-check the functions in fRec.

    For each function f in fRec,
      Let tOld be its old type,
      Let tNew be its new type,
      If tNew < tOld then
        For each g outside fRec that depends directly on f
          Let t1, ..., tn be the types at which g uses f
          If tNew < ti for any i in 1 .. n then
            re-analyze(g)
```

Figure 4-3: Improved Re-analysis Procedure

information is used automatically to ensure that "x" has indeed been used correctly. If "x" is invoked during testing before a definition has been supplied, the usual "typefault" trap mechanism is invoked.

This supports top-down program development well: when a definition is entered, all its free identifiers are automatically introduced into the environment, and the types of all uses of these free identifiers are remembered. When one of these lower-level identifiers is subsequently defined, provided it was used type-correctly, only the new definition will need to be type-checked.

There may be some question whether an unbound identifier should automatically be assumed to refer to some future top-level definition. For example, an unbound identifier three blocks deep could belong to any of the levels above it, not necessarily the top level. We feel that this is not a serious issue-- the system could interact with the user to determine his intent. In practice we have found that simply lifting it to the top-level is quite acceptable.

5. Conclusion

A rich type system is a useful intellectual tool in designing large programs [9]. Type-checking then leads to robust, efficient programs. Unfortunately, the use of these techniques has generally been avoided in interactive programming languages because most type systems and type-checkers are too restrictive and obtrusive, and severely inhibit the productivity of the interactive programming environment.

Milner-style polymorphic type systems with automatic type-checking offer a possible solution, but must be carefully engineered so as not to compromise the interactive environment. We have identified some practical problems that arise in doing so, and shown some solutions. Early experience with small programs in a prototype system has been favourable.

References

1. Albano,A., Cardelli,L. and Orsini,R. Galileo: a Strongly Typed Interactive Conceptual Language. Tech. Rep. 83-11271-2, Bell Laboratories, 1983.

2. Augustsson,L. A Compiler for Lazy ML. Proc. 1984 ACM Conf. on Lisp and Functional Programming, Austin, Texas, ACM, Aug., 1984, pp. 218-227.

3. Buneman,O.P., Frankel,R.E. and Nikhil,R. A Practical Functional Programming System for Databases. Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, 1981, pp. 164-186.

4. Buneman,O.P., Frankel,R.E. and Nikhil,R. An Implementation Technique for Database Query Languages. *ACM Trans. on Database Systems 7*, 2 (June 1982), 164-186.

5. Burstall,R.M., MacQueen,D. and Sanella,D. Hope: an Experimental Applicative Language. Proceedings of the Lisp Conference, Stanford, ACM, 1980, pp. 136-143.

6. Cardelli,L. ML Under Unix. Bell Laboratories, 1983.

7. Damas,L. and Milner,R. Principal Type-Schemes for Functional Programs. Proc. 9th Symp. on Principles of Programming Languages, ACM, Jan., 1982, pp. 207-212.

8. Gordon,M.J.C., Milner,R. and Wadsworth,C. *Lecture Notes in Computer Science. Vol. 78: Edinburgh LCF.* Springer-Verlag, Berlin, 1979.

9. Liskov,B.H. and Zilles,S.N. Programming with Abstract Data Types. *SIGPLAN Notices 9*, 4 (1974), 50-59. (Proc. ACM SIGPLAN Conf. on VHLL)

10. Milner,R. A Theory of Type Polymorphism in Programming. *J. Computer and System Sciences 17* (1978), 348-375.

11. Mycroft,A. and O'Keefe,R. A Polymorhic Type System for Prolog. Proc. Logic Programming Workshop 83, Portugal, 1983, pp. 107-122.

12. Nikhil,R.S. An Incremental, Strongly-Typed Database Query Language. Ph.D. Th., Moore School, University of Pennsylvania, Philadelphia, Aug., 1984. Available as Technical Report MS-CIS-85-02

13. Robinson,J.A. Computational Logic: the Unification Computation. Machine Intelligence, Vol 6, Edinburgh University Press, Edinburgh, Scotland, 1971, pp. 63-72.

14. Steele,G.L. Jr. and Sussman,G.J. The Revised Report on Scheme: A Dialect of LISP. Tech. Rep. AI Memo 452, MIT Artificial Intelligence Laboratory, Jan., 1978.

15. Steele,G.L. Jr. and Sussman,G.J. The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two). Tech. Rep. AI Memo 453, MIT Artificial Intelligence Laboratory, May, 1978.

16. Turner,D.A. *SASL Manual.* University of St.Andrews, 1976.

Table of Contents

List of Figures