Volume II, Number 1 January, 1985

Polymorphism

The ML/LCF/Hope Newsletter

Contents

Letter from the Editors

After a regrettably long absence, *Polymorphism* is back!. The fact that it has been a long time since the last issue is not a reflection of lack of activity in the ML/LCF/Hope world, however. On the contrary, 1984 was a year of significant progress, particularly in the development of Standard ML.

During the first half of 1984 reactions to the proposals published in *Polymorphism* I.3 were gathered and evaluated, and last June a meeting was held in Edinburgh to consider revising the proposals. The meeting lasted three days and was judged a great success, in that on almost all major issues it proved easy to arrive at a consensus solution. We felt that the design of Standard ML had been significantly improved and simplified. A report on the deliberations of that meeting is included in this issue.

Robin Milner incorporated the proposed changes in a revised version of the the Standard ML core proposal, and David MacQueen completely rewrote the Module proposal in response to suggestions made at the June meeting. We expect to present these documents to you in the next issue of *Polymorphism,* due out this April.

ML is getting a good deal of exposure in the wider programming language community. For instance, there was a session devoted to ML at the ACM Symposium on Lisp and Functional Programming last August in Austin Texas. The papers presented were "A Proposal for Standard ML" by Robin Milner, "Modules for Standard ML" by David MacQueen, "Compiling a Functional Language" by Luca Cardelli, and "A Compiler for Lazy ML" by Lennart Augustsson. Judging by the number of papers mentioning ML at the last two ACM POPL conferences, ML is already becoming something of a benchmark among languages.

This year we expect that the focus will begin to turn from language design to language implementation. Work will continue and will intensify in Edinburgh with the start of an SERC funded ML project. A workshop devoted to implementation of functional languages is to be held this February in Gothenburg. At Cambridge, Dave Matthews has written a Standard ML compiler in his language Poly (see *Polymorphism* I.2). We hope to be able to report on these developments in future issues. In fact, we invite contributions for a special issue of *Polymorphism* devoted to techniques and experiences relevant to the implementation of ML and related languages.

Another possible special issue topic would be ML derivatives or spinoffs, such as Gothenburg's LML and Luca Cardelli's Amber. We would be happy to receive reports on any such efforts.

Also, don't forget to send us address updates. As you know, we are keeping the burden of producing and distributing *Polymorphism* low by sending copies to one individual representing each institution or research group. We depend on those individuals to propagate the newsletter to their local communities.

Luca Cardelli
David MacQueen

AT&T Bell Laboratories
Murray Hill, NJ 07974

Report on the Standard ML Meeting
Edinburgh, 6-8 June 1984

David MacQueen

AT&T Bell Laboratories

Robin Milner

University of Edinburgh

Introduction

A three day meeting on Standard ML was held in Edinburgh from 6 June to 8 June 1984. The meeting was convened by Robin Milner with the goal of further refining the definition of the core language and integrating the proposals for modules and stream I/O with the core design. The meeting was also intended to plan for the further development of the language, including its implementation, the prospective programming environment and tools, formal definition, documentation, and distribution. We feel we made considerable progress toward agreement on the language we intend to IMLPEMENT, FORMALLY DEFINE and PROMULGATE over the next few years.

Present at the meeting were:

Rod Burstall	Edinburgh
Guy Cousineau	INRIA/Paris VII
Jim Hook	Cornell/Edinburgh
Dave MacQueen	Bell Labs
Robin Milner	Edinburgh
Kevin Mitchell	Edinburgh
Larry Paulson	Cambridge
Don Sannella	Edinburgh
John Scott	Edinburgh

Comments were also received from Alan Mycroft and Kent Karlsson summarizing the views of people at Chalmers Institute of Technology in Gothenburg. Regretfully, Luca Cardelli of Bell Labs, Gerard Huet of INRIA, and Mike Gordon of Cambridge were not able to attend the meeting, but they were represented by their colleagues who were present.

This report will be organized by topics, with some indications of the chronological order of discussions. The often repeated phrases such as "it was agreed that ... " and "it was felt that ..." are meant to indicate the consensus opinion of the participants. The term "Core language" refers to the basic language described in "A Proposal for Standard ML", without modules or I/O.

Modules [Wednesday AM (June 6), Thursday PM]

1. Module tutorial

Dave MacQueen gave a tutorial on the module proposal, setting forth the basic principles behind the design and explaining the need for some of its special features, particularly inheritance and sharing specifications.

2. Instance terminology

It was agreed that the term "instance" used in the first draft of the Modules proposal was unsatisfactory and should be replaced. The term is not descriptive (instance of what?) and conflicts with the commonly used terminology, "instance of a polytype". Rod noted (see item 3 below) that the notion of an instance should be considered the primary one, with modules playing a secondary role. This also argues against the word "instance", with its connotations of being derivative.

The term "environment" would be aptly descriptive, but it is too long for convenience and is commonly used to refer to other, more general concepts. There was a consensus for provisionally adopting the term "structure" as a replacement for "instance", despite its use in Algol 68 and C to denote records. One can think of the term as short for the qualified phrase "environment structure". This choice proved satisfactory as the meeting went on, so we will adopt this new terminology throughout the remainder of this record of the meeting.

3. Structure expressions

Rod Burstall advocated thinking of structures as the primary concept and keeping in mind the analogy

STRUCTURE	-	value
SIGNATURE	-	type
MODULE	-	function

in deriving the module syntax. This analogy should induce harmony even though we have to impose certain constraints (such as no higher order modules) that weaken it.

One should think of large EXPRESSIONS, composed of applications of modules and denoting structures, as defining environments in which to evaluate small expressions denoting values.

4. Sharing

4.1 Sharing specifications restricted to signatures

It was agreed that sharing specifications can indeed be confined to signatures, as suggested by Robin, rather than allowing them also to occur in module bodies. The idea is to regard the parameter specifications of a module as a special form of signature, namely an anonymous signature consisting of structure specifications (the formal parameter specs) and sharing specifications that indicate inter-parameter sharing. Sharing specifications that were previously part of the module body become part of the parameter signature.

4.2 Checking of sharing specifications (propagation of sharing constraints)

Parameter sharing specifications may be checked dynamically when a module is applied to actual structures by inspecting the parameter structures themselves. When defining a module, it must be statically verified that the sharing specs of the result signature are implied by the sharing specs of the parameter signature. This can be done by building a dummy structure hierarchy graph and applying a congruence closure algorithm to it, but the algorithm has not yet been worked out in detail. The question of how to handle sharing that involves globally referenced structures needs to be considered; perhaps this problem can be dealt with by referring to a structure hierarchy graph for the global structure environment. A similar problem arises from globally referenced modules in structure expressions used, for instance, in substructure declarations. These points should be addressed in the second draft of the module proposal.

5. Nongenerative declarations

Nongenerative declarations are those that simply bind a name to a preexisting entity, as in

```
    val x = 3+2
```

Most value bindings are nongenerative, since we think of typical data values as existing independently of their being denoted by an expression. A generative declaration, on the other hand, causes a name to be bound to a new, freshly created object. A typical generative declaration is

```
        val x = ref 3
```

Note that it is really the generative nature of the expression on the right hand side that makes this declaration generative, so we can think of all declarations as being simple identity declarations (in the Algol 68 terminology), and then consider whether the defining expression is generative or not.

Currently all type and exception declarations are generative (we can imagine that the current exception declaration syntax is short for something like

```
        exception e:ty = new_exception(ty)
```

where new_exception is a generator of exceptions). But to build structures on existing types, we need nongenerative type declarations within structure and module definitions. The solution proposed is that we make all type declarations be identity declarations of one of the following forms

```
        type id = ty             {nullary type constructor}

        type tvars id = ty     {1st-order type constructor}
```

where ty can be any type expression, including the tagged union type construct, which will be the one generative form of type expression (and which can only occur as the top-level defining expression in type declarations). The syntax for the tagged union was initially to be the same as before, *i.e.*

```
        con [of ty] {| con [of ty]}*
```

This had the unfortunate effect that the degenerate form

```
        type foo = empty
```

was ambiguous, because empty might be either a single constant constructor in a tagged union expression (in which case neither "of" nor "|" appears), or it might be a type constant. The proposed resolution of this problem was to always assume the second interpretation, making it impossible to define a new union type with a single nullary constructor.

There was still some unease about the syntax of unions, depending as it did on one or the other of two infixed keywords. This seemed to be bad syntactic hygiene. After the meeting, Robin proposed adding the initial keyword "data" to signal the union construct. Thus

```
        type foo = data empty

        type rec 'a list = data nil
                           | cons of ('a * 'a list)
```

This proposal entails the restriction that in (mutually) recursive type declarations, all defining type expressions must be unions. This could be seen as roughly analogous to the restriction that in (mutually) recursive function declarations, all the defining expressions must by fun- (*i.e.* lambda-) expressions.

This treatment of type declarations restores the former type abbreviation facility. We can define types like

```
        type 'a pair = 'a * 'a
```

and then int pair and int * int will match. A pragmatic matter is how to propagate names such as pair when printing types. One simple solution is to let the named forms remain intact unless and until they must be expanded for the purpose of unification with another type during type checking.

Robin also felt that a nongenerative form of exception declaration such as

```
        exception e = e'
```

would be useful to overcome certain scoping problems. These arise (*e.g.*) from certain weaknesses of the *local* ... in ... declaration construct, where one might want to declare an exception in the local part so that it could be raised in functions defined there, and also export the exception so that it could be trapped latter. This is not possible with the "`local ... in ...`" construct. A similar problem concerns local declarations of several mutually recursive concrete (*i.e.* union) types, some of which are to be exported.

6. Local structure and module declarations.

We could see no clear reason why one could not locally declare structures and modules and signatures locally within expressions and declarations. [This would intermix structure and module bindings with ordinary value and type bindings in the environment, but this is already the case at top level.] These local bindings would presumably be subject to the usual scoping rules and would not be persistent. As with top-level structure and module declarations, they would not be allowed to contain free type and value identifiers.

However, such local declarations would probably be cumbersome and inefficient to use, and would probably be rarely used (just as type declarations within functions are rare in the core language). The matter requires further study.

7. Mutual recursion in signatures, structures and modules

Don gave an example illustrating that, without mutually recursive signatures and structures, modules involving families of mutually recursive types and functions are likely to be rather large; he showed how mutual recursion could permit the resolution of such modules into several smaller modules.

It was pointed out that it is not easy or natural to extend the constraints on the rhs's of mutually recursive function declarations (*i.e.* that they be explicit lambda-abstractions) to cover mutually recursive structures -- particularly as it is not manifest in a module declaration that it will be used to define mutually recursive structures. It was agreed to omit (mutual) recursion from the initial modules proposal, but the question should be born in mind for the future (possible minor experiment).

8. Matching structures to signatures

It was agreed that the matching of candidate to target signatures be relaxed in two ways:

(1) the target need only match a subset of the candidate,

(2) ordinary functions in the target can be matched by constructors in the candidate.

When matching the body signature of a module declaration (the candidate) against its explicit result signature (the target), point (1) allows us to screen out local bindings, thus eliminating many uses of "local" or "export". When matching the signature of a parameter structure (the candidate) against the formal parameter signature of a module (the target), point (1) amounts to a kind of implicit (forgetful) coercion. The implementation of this coercion will probably involve the creation of a restricted version of the parameter structure.

Point (2) is seen as a natural rule that avoids the necessity for declarations like

```
type 'a stack = {data} empty' | push' of 'a
val empty = empty'
val push = push'
```

whose purpose is to allow constructors to be exported as ordinary functions.

9. Inheritance

Suppose a module

```
M(S: SIG1): SIG2 = ...
```

specifies that S should be inherited by the result structure. Don suggested that when M is applied to a structure Str, the resulting structure M(Str) should inherit all of Str, not just the restricted substructure that matches the formal parameter signature SIG1.

There were a couple of objections to this proposal

(1) SIG2 would no longer accurately specify the effective signature of the result of applying M. If declared signatures are not accurate, the process of accessing components of structures will be much more complex, and correspondingly less efficient.

(2) The suggestion appears to violate principles of information hiding. The environment of a program fragment should be limited to that which is explicitly specified to be necessary and sufficient. Implicit inheritance of extraneous bindings makes it difficult to determine what the current environment consists of, because it will depend on the dynamic history of structure creation rather than the static information in the explicit signatures.

After some discussion of these points, the suggestion was not accepted.

10. Open

It was felt that the present meaning of "open" in structures and signatures was a bit too delicate. It was proposed that, whatever is adopted, the meaning of open should be expressible -- and expressed -- as a derived form, and should be as simple as possible. In particular, the "open" declaration construct should obey the usual scoping rules. However, the general intention of open was accepted as necessary.

[There should probably be two different keywords for the two uses of "open" as a declaration construct ("open Str") and as a flattening modifier of instance specifications in signatures ("open instance S"). These uses are related, but it is not clear that either can be derived from the other.]

11. Modified form of module and structure declarations

In note distributed before the meeting, Robin proposed a simplified syntax for structure (formerly instance) and module declarations (RM 23/5/84). These suggestions were adopted with minor changes. The new syntax is as follows

```
struct ::= struct dec end          {immediate structure}
           struct_path               {qualified name}
           mod_id(struct_seq)        {module application}

stb    ::= struct_id{: sig} = struct  {structure bindings}
           stb1 and ... and stbn         {simultaneous}

dec    ::= structure stb

mod    ::= module mod_id({param_sig}){: sig} = struct

param_sig ::= param_spec_seq {, share_spec_seq}
```

(the parameter sharing specifications have been made part of the anonymous parameter signature, which still provides for multiple structure parameters). In addition, immediate structure expressions may contain free signature, structure, and module identifiers. Note that a structure binding may involve a coercion if an explicit signature specification on the lhs is a restriction of the signature of the rhs struct expression.

As noted in the Module proposal (Section 3.5, Views), immediate structure expressions can be used to succinctly express coercions of a given structure into a new signature. It was agreed that special syntactic forms for expressing such coercions were probably unnecessary.

Stream I/O [Wednesday PM]

On Wednesday afternoon we turned our attention to Luca Cardelli's proposal on character stream I/O.

1. Bipolar vs unipolar streams

A major goal of Luca's proposal is to deal uniformly with i/o to and from files and "filter" processes (*i.e.* processes that transform a single input character stream to a single output character stream).

We felt that having all streams be bipolar might be excessively general. Unipolar (input only or output only) streams seem more basic, or at least are more familiar, and it appears that a "matched pair" of such streams could usually perform the same functions as a single bipolar stream. Furthermore, unipolar streams could have two different types, say instream for input streams and outstream for output streams, and then the type checker could catch certain errors, such as an attempt to write to an input stream.

With unipolar streams, Luca's primitives would become

```
stream  : unit -> outstream
file    : string -> instream
save    : string -> outstream -> unit
channel : string -> instream * outstream
terminal: instream * outstream
```

Note that these modified primitives are weaker than Luca's, since they do not provide for

- appending characters to an existing file (either output or input)
- writing and reading an internal buffer file

We could either decide to introduce additional primitives to recover this functionality, or to forgo these capabilities because they are not felt to be necessary. Another possibility is to have a third type of bipolar streams, called bistream, with Luca's original primitives (suggested by Gothenburg).

2. Stream attributes

Several people were concerned with how to support unusual I/O requirements associated with window management or screen editing on character oriented terminals.

These applications should be within the scope of the character stream I/O model (as has been amply demonstrated by existing screen-oriented systems under Unix and VMS). The problem is to control certain attributes of streams connected to terminal devices, such as buffering, editing, flow control, and control character interpretation. These features of terminal I/O are generally controlled by the operating system, which provides commands or system calls to allow the user to change them. We need a relatively transparent, operating system independent way of managing these "external" characteristics of terminal streams from within ML.

One suggestion was that there should be a stream-status "record" that could be inspected to determine the properties of a stream and updated to change these properties (problem: how would the operating system be informed of these updates in the status record). There is the question of what type this status record would have, and whether it would be operating system dependent.

Another suggestion was that there should be a set of status reading and setting functions on streams. These might fail if applied to streams not associated with a terminal-like device. The signature of this set of functions might, or might not be operating system dependent.

[This raises the general question of accessing operating system services from within ML. Certain "core" services could possibly be packaged as a structure with a fixed, operating system independent signature. More exotic services, and more exotic forms of I/O, could be packaged as specialized structures.]

3. Buffering and lookahead

Does the lookahead primitive have to be as elaborate as that described in the proposal? It was suggested that a simpler version with type

```
    lookahead: stream -> int -> string
    (return a string of the next n characters to be read)
```

would suffice. There was some discussion of the advisability of using the input stream for buffered character processing as opposed to using integer or character or byte arrays as buffers for such processing.

A phone call to Luca gained his provisional approval for these points, and it was agreed that Robin and Kevin would draft a revised I/O proposal and circulate it (particularly to Luca) for comment.

The Core language [Thursday AM, Friday AM]

1. Use of period

It was agreed to replace the period in matches, freeing it for use in qualified names. Robin's proposal to require varstructs to be atomic in matches, and replace the ensuing period by nothing at all, was not favored. ".." and "=>" were proposed as replacements for the period. [Later in the meeting, after considering some example program texts, "=>" came to be preferred. It was considered to be particularly appropriate for case expressions, and not too annoying in fun expressions.]

2. Comment convention

The use of comment brackets, rather than "comment to end-of-line" was favored by the majority, though some felt that having both forms would be convenient. It was tentatively agreed to replace "{ }" by "(* *)" or some such pair of decorated brackets to free "{ }" for possible use elsewhere in the language. Luca was reported still to favor single character brackets of some kind.

[Later in the meeting it was suggested that "{ }" might be used to delimit the tagged union construct in type declarations, but this suggestion was not adopted. No other immediate use of "{ }" was found, so it is still open to us to retain "{ }" for comments.]

3. Escape sequences in strings

It was decided that string escape sequences were not in need of reinvention and that the best course was to adopt the widely known conventions of the Unix system and the C language. Therefore the following escape sequences were adopted:

\n	a single character that the system will interpret as an end-of-line (linefeed in Unix)
\t	tab
\^c	control character c (for any printing character c for which there is a corresponding control character)
\ooo	single character with ASCII octal code ooo (3 octal digits)
\c	the character c (in all other cases)

Later Luca commented that using octal codes for characters was obsolete and that perhaps ASCII character codes should be given in decimal.

4. Infix directives

Larry proposed that infixes be introduced by a restricted form of declaration

```
    infix precedence id' = id
```

where id has already been declared as a function (of pairs) using noninfix syntax. This would have the desirable effect of associating infix status with a binding, rather than with an identifier, and would render nonfix superfluous.

This was (reluctantly) not accepted, mainly because it would forbid the use of infix syntax for a function within its (often recursive) declaration. It was also pointed out that the normal use of an infix directive, which would just precede the declaration of the function to which is was intended to refer, is quite natural.

But it was noted that some care will be needed in specifying infix status in signatures. In particular, it was agreed that if id is declared with infix status in (the signature of) a structure S, then S.id will not possess infix status (because it is a compound identifier distinct from the identifier S).

5. Labelled products

It was agreed that Dave MacQueen would propose an extension to the core language incorporating labelled products, in harmony with the current unlabelled products (an perhaps yielding them as a derived form). The core language definition will not include the extension at first, but the extension will be circulated for comment and adopted with fairly high priority, since ML-in-ML can use labelled products with advantage. The main design decision is between an unqualified, free-standing labelled product construct, and a qualified construct which is somehow a derived form reducing to unlabelled products in combination with constructors. It may be that two alternative proposals will be circulated for comments.

Meanwhile, Robin's derived form for selectors in the core language (Section 6.2) will be dropped, together with the standard selectors hd and tl. This is to insure no conflict with the labelled product extension.

6. Real numbers

It was accepted that a primitive type "real" would be included in the language and that

(a) standard functions like "+" are overloaded, with two types: "`int * int -> int`" and "`real * real -> real`". [This exceptional admission of ad hoc overloading for the arithmetic operators is an acknowledgement of the strength of conventional usage. It is not expected to set a precedent.]

(b) no coercion is allowed; one must write "`sqrt(3.0) + 1.0`", not "`sqrt(3) + 1`". But explicit coercion functions such as

```
real: int -> real
trunc: real -> int
```

will be provided.

(c) real constants will include floating point forms like "`~35.7E~1`".

(d) the usual standard functions like sin, cos, arctan, log, exp, sqrt will be included.

Reals will be incorporated into the core language document, but details relegated to an Appendix as far as possible. [Luca later asked whether the IEEE floating point standard might be used.]

7. Definition of div and mod

The core proposal states that "div and mod are defined as in PASCAL". It was pointed out by Gothenburg that these operators are not well defined in Pascal. It was agreed that Robin would find an appropriate definition to include in the core definition.

8. Layered patterns and "as"

We reaffirmed the importance of layered patterns and the appropriateness of the keyword "as" for their formation, despite a skeptical query from Gothenburg.

9. Where

Guy urged that the deletion of the where construct be reconsidered (Bob Constable has expressed a similar sentiment). The reasons for discarding "where" were reviewed; they are

(a) If infix directives are allowed in where declarations as they are in let declarations, then the directive would textually follow the occurrences of the identifier to which it applied, causing rather severe difficulties for the parser.

(b) Experience has shown that the "where" construct is very error prone because of the difficulty of determining its scope at a glance, particularly when "where"s are nested or intermixed with "let"s.

It was decided (reluctantly) that it was best to leave the "where" construct out.

Friday, AM resumed discussion of remaining core language issues.

10. abstype

It was recognized that the current abstype will become redundant -- or at most a derived form -- when the modules proposal is enriched (or even in the present proposal). [See Appendix on structures and abstraction.]

In view of the fact that the Core language will be adopted as standard before the modules proposal, and that the latter must be allowed to develop without prejudice, it was felt that abstype should not be dropped from the Core (nor replaced there by an ad hoc form involving signatures and structures), but that in the Core language document there should be a clear indication that it will become redundant.

We were also optimistic that abstype constructs would be *automatically* convertible to forms involving structures and signatures, when these forms are fully defined; this is because the compiler, when reporting the bindings exported by and abstype effectively computes its signature.

11. Abstraction

Dave pointed out that abstraction will be obtainable in two ways, when modules are present: (a) by hidden constructors (abstype), and (b) by signatures and structure abstraction. The realization that the special generative nature of union types (with their constructors) is not needed to support data abstractions raises the intriguing possibility of making do with a simple, nongenerative labelled-union construct. This would simplify the type system by not unnecessarily conflating the notions of type abstraction and labelled-union, and would mean that purely structural type matching could be used. However, the question of whether special constructor functions would still be necessary to mediate the unwinding of recursive types remains to be settled.

It was recognized that this is a research matter, and that it is better to proceed with the current use of constructors because such an overhaul would involve a deep re-design of the language. [A possible Major Experiment.]

12. Nonlinear varstructs (patterns)

Guy pointed out that, with repeated variables in varstructs, such pleasant declarations as

```
        val MP(x ==> y, x) = y
```

would be possible (where "==>" is an infix constructor representing implication, and `MP` is a function encoding the rule Modus Ponens). It was recognized that such repeated variables would only be admissible at types admitting equality, but this would be no barrier in principle.

This feature was felt to be a bit too complex for the present, particularly as its impact on optimized implementations of pattern matching had not been explored. However, it was agreed that it would be a valuable subject for an experiment, noting that Guy has implemented it (with a nonoptimal matching algorithm).

13. Multiple patterns per rule

Another extension to the pattern matching facilities that was mentioned was a sort of alternative construct that would allow one to preface the same rhs with several patterns. All these patterns would have to bind the same variables, presumably. The purpose of such a construct would be to avoid having to make several copies of a large rhs expression in a match, as in

```
case exp of
    red => large-expression1
    green => large-expression2
    blue => large-expression2
    orange => large-expression1
    yellow => expression3
```

which would become

```
case exp of
    red or orange => large-expression1
    green or blue => large-expression2
    yellow => expression3
```

It was pointed out that this problem could be ameliorated by abstracting the repeated expression as a function. There was not enough interest to pursue this issue, but it might be future minor experiment. It doesn't appear to present any problems for merged pattern matching.

14. "local" declarations

It was noted that Luca's "export ... from ... end" declaration construct avoids some of the limitations of "local ... in ... end" [generative bindings that must be both exported and visible to the local declarations], but that it should not be added to the Core language at the moment. Its primary use has been to support the prototype, substandard module facility, and it was liable to be made redundant by signature matching rules in the revised module proposal.

15. Exception raising and handling

Larry complained about the three forms "handle", "trap", and "?" for catching exceptions, and he was supported. Robin recalled various people suggesting the keyword "with" (or some such keyword), as in

```
handle exid with match
```

Adopting this suggestion, it was agreed that we demolish escape and trap and allow the forms

```
raise exid with exp
handle exid with match
```

with derived forms for the unit case

```
raise exid
handle exid exp
```

standing for (respectively)

```
raise exid with ()
handle exid with () => exp
```

Furthermore, Kevin remarked that the wild card handler ("?") sometimes produced unintended effects, given that it traps all exceptions including the user generated exception "interrupt" produced by ^C. It was felt that this relatively "undisciplined" exception handling construct should not be favored with a particularly simple syntactic form, though some felt reluctant to abandon this convenient notation. As an elegant and agreeable compromise it was suggested that we add the special wild card handler form

```
handle ? exp
```

See the Appendix: "Afterthoughts on exceptions" for further development of these ideas.

16. Varstruct terminology

We agreed to adopt the word "pattern" for "varstruct".

17. Printing values

As a rudimentary aid to debugging, and for other purposes, it would be helpful to be able print more or less arbitrary ML values.

It was proposed that two "functions" be included:

```
print: 'a -> 'a
makestring: 'a -> string
```

The effect of "makestring(exp)" is to yield a string which corresponds exactly to what would be printed if exp was evaluated at top level (without its type); however, the type assigned to this particular occurrence of exp will determine the string produced. This is because makestring, like =, must interpret the type of its argument (either during compilation or dynamically) to decide what the result will be. [We could call such functions "type-driven".]

The function print is also type-driven, and has the effect of printing the representation string (the same as returned by makestring) on the standard output stream while returning its argument as its result (to facilitate inserting print commands for debugging with minimal perturbation of the code). When the I/O primitives are defined, it will turn out that "print exp" will be nearly equivalent to something like

```
output outstream (makestring exp)
```

but "print" is still needed as a primitive, because the general polymorphic definition

```
val print v = (output outstream (makestring v); v)
```

will not work because the type context to drive makestring is not available.

Rod suggested that the "functions" print and makestring could be made into syntax rather than function applications; at first this seemed nice (in view of their nonstandard type-driven semantics), but on closer inspection this would be confusing due to the different syntactic binding precedence. It would also preclude their being passed as parameters (with appropriate type qualification) as is possible with "=".

Unlike "=", makestring and print should not fail when applied to arguments with polymorphic or abstract types. Rather, like top level printing, they should print a reasonable representation of whatever structure is revealed by the type. It would also be very desirable that a prettyprinter be used to lay out the printed representation. [In fact, since ultimate layout constraints (*e.g.* width of line, current indentation) are probably not available when generating a string using makestring, it might be better to generate a list of strings which could be used as input to a layout routine as in an Oppen-style prettyprinter.]

Another variant would be a (type-driven) function

```
typrint: 'a -> 'a
```

that would print out both the value and type of its argument.

Plans [Friday PM]

I. Projects and Documents

1. We would like to produce two journal articles -- one on the Core, and one on Modules. It would be appropriate to build them around a series of examples.
2. We would like to produce an abstract syntax -- or probably two: (1) the bare syntax, and (2) the syntax with derived forms included. There might also be justification for having both a clean reference abstract syntax, and a more elaborate, implementation-oriented abstract syntax. The reference syntax would serve as a skeleton for the implementation syntaxes.
3. We need to investigate formal semantics, both operational and denotational. The former should be related to an eval function. We could also challenge Peter Mosses to apply his abstract semantic algebras to ML.
4. We want a tutorial text, or possibly more than one. Luca's manual is already written and can serve as a first approximation. The new book on Scheme sets a standard to aim at.
5. A collection of these documents, including a tutorial text and reference manual might be collected together and published as a book at the appropriate time.

II. Implementation

1. We need a strategy for building portable implementations. This probably means settling on a particular FAM code and a form for its external presentation. The bootstrapping strategy for bringing up new implementations needs to be carefully designed. [We might use the Smalltalk 80 strategy as a pattern. This would involve a common representation of a "system image" that could be brought up on various implementations of the abstract machine.]
2. As a different exercise, we need a fast implementation, to convince the world of ML's viability. It is particularly important that ML not gain a reputation as an impractical, toy language. This goal probably cannot be compatible with full portability, at least initially.
3. We would like a profiling tool, or set of tools, as an aid in tuning the implementation.

III. The ML Programming Environment

1. At least the following ingredients of an environment need to be considered

   Editors
     structure, or syntax directed, and text (screen-oriented)
   Debugger
     selective tracing, break points, facilities for examining the state of a "live" computation, metalevel access
   Configuration or system management
     library facilities (relating modules and structures to files)
     version consistency, automatic recompilation
   Graphics
   Operating System interface
     interprocess communication
     low-level I/O

2. Can we (should we) introduce concurrency in ML? How do we establish communication between a newly created process and its creator? Is this to be done by shared structures (structures as monitors)? For interprocess communication, how much do we rely on communication mechanisms provided by an existing operating system. Note also that Kevin is already experimenting with CCS style processes in ML.
3. What is the ML "top level"? Normally one will work within an ML "system", which will constitute a (persistent) environment of signatures, modules, and structures(?). New module and structure declarations will extend or augment a system, but how will such extensions be

made permanent and persistent? How does this relate to the lower level at which we evaluate ML expressions; for example, at which level are structures declared? These questions concern the interface between the module proposal and "configuration management".

IV. Distribution

We agree that ML is "public domain". Rod proposed that we should impose a distribution fee of around 100 pounds, and also suggest to clients that they make an ex gratia payment. This was thought to be about the right level of financial involvement; the distribution fees and ex gratia payments could provide a useful expense fund. We should distinguish this distribution mechanism from agreements (with ICL for example) to help make ML run on particular machines.

An alternate, low overhead means of distribution would be to allow access to ML implementations and tools via FTP over networks (SERCnet and ARPA). News distribution lists could be used to disseminate announcements and bug fixes. Users could report bugs by network mail.

V. Agenda of Future Tasks

0. ML Meeting report. [Dave MacQueen - June 84]
1. Abstract syntax (reference) definition. [Jim Hook - Summer 84]
2. Revised core definition. [Robin Milner - Summer 84]
3. Revised character stream I/O proposal. [Robin Milner, Kevin Mitchell - Summer 84]
4. Revised module proposal. [Dave MacQueen - Summer 84]
5. Labelled product proposal. [Dave MacQueen, Robin Milner - Summer 84]
6. Parser for Core language. [John Scott - Summer 84]
7. Collection of ML lecture notes and tutorial examples. [Larry Paulson - Ongoing]

VI. Future Experiments

1. Minor
   - nonlinear patterns
   - recursive structures
   - exceptions as (pseudo)constructors
2. Major
   - persistent structures
   - concurrency
   - meta/object interface
   - theories as structures

Appendix: Structures and abstraction

It was noted that the abstype construct could eventually be superseded by the used of structures, particularly with the liberal signature matching rules [Modules, item 8]. For instance, consider the abstype declaration of sequences:

```
abstype rec 'a seq = data null | prefix of 'a * 'a seq
    with val empty = null
         val rec concat null s = s
                 | concat (prefix(x,s1)) s2 = prefix(x, concat s1 s2)
    end
```

This would be equivalent to defining a named structure Sequence and then opening it:

```
structure Sequence: sig type 'a seq
                        val empty: 'a seq
```

```
                                    val concat: 'a seq -> 'a seq -> 'a seq
                                end
                    = struct
                          type rec 'a seq = data null | prefix of 'a * 'a seq
                          val empty = null
                          val rec concat null s = s
                                | concat (prefix(x,s1)) s2 = prefix(x, concat s1 s2)
                      end
    open Sequence               {optional, to avoid qualified names}
```

One could also have introduced the signature of Sequence as a named signature before declaring Sequence. One could introduce a derived form that would avoid the need to introduce a name for the structure and open it; this might amount to little more than replacing "structure Sequence" by some keyword like "abstraction". However, this derived form may not be worth introducing, since the above syntax is not too cumbersome and the name of the abstraction structure serves a useful role as documentation.

In the Sequence example, abstraction depends on the generative nature of the union type declaration along with the screening function of signature matching. Suppose we wanted to define an abstract type whose representation was an existing type, as in

```
    signature POINT =
        sig type point
            val x_coord: point -> int
            val y_coord: point -> int
            val eqpoints: point * point -> bool
            val mkpoint: int * int -> point
        end

    structure Cartesian: POINT =
        struct
           type point = int * int
           val x_coord(x,_) = x
           and y_coord(_,y) = y
           and eqpoint(p1,p2) = p1=p2
           and mkpoint(x,y) = x,y
        end
```

The problem with this example is that Cartesian is not "abstract", because the default type propagation rules mean that Cartesian.point will continue to match int*int. In order to use Cartesian as an abstraction, we need to conceptually abstract "the rest of the program" with respect to the signature POINT (and type check it with respect to that signature), and then supply Cartesian as the actual parameter. This requires a new syntactic form, such as

```
    abstraction Cartesian: POINT =
        struct
           ...
        end
```

Appendix: Afterthoughts on Exceptions

After the meeting, Robin thought about the problem implied by Kevin's remark: how do we program something that will handle *all exceptions except a given exception such as "interrupt"*? We cannot write

```
    exp handle interrupt raise interrupt
        handle ? exp'
```

```
                         val concat: 'a seq -> 'a seq -> 'a seq
                       end
           = struct
                    type rec 'a seq = data null | prefix of 'a * 'a seq
                    val empty = null
                    val rec concat null s = s
                          | concat (prefix(x,s1)) s2 = prefix(x, concat s1 s2)
             end

      open Sequence                      {optional, to avoid qualified names}
```

One could also have introduced the signature of Sequence as a named signature before declaring Sequence. One could introduce a derived form that would avoid the need to introduce a name for the structure and open it; this might amount to little more than replacing "structure Sequence" by some keyword like "abstraction". However, this derived form may not be worth introducing, since the above syntax is not too cumbersome and the name of the abstraction structure serves a useful role as documentation.

In the Sequence example, abstraction depends on the generative nature of the union type declaration along with the screening function of signature matching. Suppose we wanted to define an abstract type whose representation was an existing type, as in

```
      signature POINT =
         sig type point
             val x_coord: point -> int
             val y_coord: point -> int
             val eqpoints: point * point -> bool
             val mkpoint: int * int -> point
         end

      structure Cartesian: POINT =
         struct
            type point = int * int
            val x_coord(x,_) = x
            and y_coord(_,y) = y
            and eqpoint(p1,p2) = p1=p2
            and mkpoint(x,y) = x,y
         end
```

The problem with this example is that Cartesian is not "abstract", because the default type propagation rules mean that Cartesian.point will continue to match int*int. In order to use Cartesian as an abstraction, we need to conceptually abstract "the rest of the program" with respect to the signature POINT (and type check it with respect to that signature), and then supply Cartesian as the actual parameter. This requires a new syntactic form, such as

```
      abstraction Cartesian: POINT =
         struct
            ...
         end
```

Appendix: Afterthoughts on Exceptions

After the meeting, Robin thought about the problem implied by Kevin's remark: how do we program something that will handle *all exceptions except a given exception such as "interrupt"*? We cannot write

```
      exp handle interrupt raise interrupt
          handle ? exp'
```

because the attempt to transmit the interrupt exception is foiled by the second wild-card handle clause. What seems to be needed is a non-nested, simultaneous form for combining exception handlers. A possible syntax would be

```
ematch ::= eclause || ... || eclause
eclause ::= exid with match
            exid exp        {a derived form}
            ? exp           {wild-card form}
```

and then handle expressions have the syntax "exp handle ematch". The above interrupt example is then

```
exp handle interrupt raise interrupt
           || ? exp'
```

Although this was not discussed at the meeting, Robin proposes to add it to the revised exception facility since it also facilitates the writing of multiple handling and goes very little beyond what was discussed.

In describing the above proposal to Dave, Robin noted a kind of pun that could be used to motivate the use of "with" as the keyword in both the raise and handle constructs. One could think of the with as an operator (or infix constructor?) that binds an exception and its value into a single compound object. In the phrase

```
raise  ( exid with exp )
```

the "with" is constructing this exception-value object, while in

```
handle ( exid with x ) => exp'
```

the "with" is being used for destructive pattern matching (the pattern would always have to contain an explicit, constant exception as its first component, of course(?)). In the case of a multirule match in a handler, one could preserve the analogy by replicating the "exid with " part of the pattern for each rule. Then

```
handle exid with nil => exp1
                 | x::l => exp2
```

would become

```
handle exid with nil => exp1
     | exid with x::l => exp2
```

This pleasing parallel between exception raising/handling and construction/pattern-matching breaks down in the special derived forms for the unit case where we have gotten rid of "with". We could extend the analogy to the unit case, simplify the syntax somewhat, and harmonize with our general principle of preferring constants over nullary functions by always omitting "with" and regarding the exceptions themselves as playing the role of the constructor [this revives in a new form an earlier suggestion of Rod's that exceptions be replaced by constructors]. The above handler would then become

```
handle exid(nil) => exp1
     | exid(x::l) => exp2
```

The unit case syntax would be modeled on match rules with constant constructors as patterns, thus

```
handle interrupt => exp
```

Having integrated the exception and the associated match, one is tempted to merge the simultaneous handlers construct into one big match, so that

```
handle ex1 with nil => exp1
              | x::l => exp2
    || ex2 with u,true => exp3
```

```
                    | u,false => exp4
```

would become

```
        handle ex1(nil) => exp1
             | ex1(x::l) => exp2
             | ex2(u,true) => exp3
             | ex2(u,false) => exp4
```

We could even replace the wild-card handler "?" by a top-level wild-card pattern "_", as in

```
        handle interrupt => exp1
             | _ => exp2
```

The problem with this generalization is that it doesn't necessarily agree with the implicit "two level matching" semantics of the simultaneous handler construct, in which we first match the exception and then match the value. The single level exception match would appear to simultaneously match the exception constructor and its value argument, so that the handler

```
        handle exc(nil) => exp1
             | _ => exp2
```

would handle "raise exc([2;3])" by evaluating exp2 instead of generating a match failure after trapping exc. (It is not entirely clear which of these behaviors is to be preferred in the abstract, but for the sake of continuity with the past we should prefer the latter). Permutation of the clauses, as in

```
        handle ex1(nil) => exp1
             | ex2(u,false) => exp4
             | ex1(x::l) => exp2
             | ex2(u,true) => exp3
```

would not necessarily cause difficulties, given that merged pattern matching would automatically regroup patterns with the same exception name together.

A compromise syntax that would acknowledge the special two level matching semantics of handlers would be to require clauses with the same exception to be grouped together and separated by "||". Then the previous example becomes

```
        handle ex1(nil) => exp1
             | ex1(x::l) => exp2
             || ex2(u,true) => exp3
             | ex2(u,false) => exp4
```

And we could also require "?" to be used instead of "_" as the exception wild-card, as in

```
        handle interrupt => exp1
             || ? => exp2
```

The corresponding syntax for raising exceptions would again use the exception as a constructor:

```
        raise interrupt

        raise ex2(3,true)
```

(Note that in both raise expressions and handlers, the exception would have to be the top-level operator of the "expression" or "pattern".)

Whether it would be a good thing to push this analogy between exceptions and constructors is not yet clear. We might regard this as the subject of a future experiment.

Basic polymorphic typechecking

Luca Cardelli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Polymorphic typechecking has its foundations in a type system devised by Hindley [Hindley 69], and later rediscovered and extended by Milner [Milner 78]. As implemented in ML [Gordon 79, Milner 84], this type system shares with Algol 68 properties of compile-time checking, strong typing and higher-order functions, but it is more flexible in allowing polymorphism, i.e. the ability to define functions which work uniformly on arguments of many types.

Milner's polymorphic typechecking algorithm has proved very successful: it is sound, efficient, and supports a very rich and flexible type system. It can also be used to infer the types of untyped or partially typed programs.

However, the pragmatics of polymorphic typechecking has so far been restricted to a small group of people. The only published description of the algorithm is the one in [Milner 78] which is rather technical, and mostly oriented towards the theoretical background.

In the hope of making the algorithm accessible to a larger group of people, we present an implementation, in the form of an ML program, which is very close to the one used in LCF, Hope and ML [Gordon 79, Burstall 80, Milner 84]. Although clarity has sometimes been preferred to efficiency, this implementation is reasonably efficient and quite usable in practice for typechecking large programs.

Only the basic cases of typechecking are considered in the program presented below, and many extensions to common programming language constructs are fairly obvious. The major non-trivial extensions which are known so far (and not discussed here) concern overloading, abstract data types, exception handling, updatable data, and labeled records and union types. Many other extensions are being studied, and there is a diffuse feeling that important discoveries are yet to be made, both in the theory and the practice of typechecking.

This paper presents two views of typing, as a system of type equations and as a type inference system, and attempts to relate them informally to the implementation.

2. A simple applicative language

The language considered here is a simple typed lambda calculus with constants, constituting what can be considered the kernel of the ML language. The evaluation mechanism (call-by-name or call-by-value) is immaterial for the purpose of typechecking.

The concrete syntax of expressions is given below; the corresponding abstract syntax is given by the type "Term" in the program at the end of this paper (parsers and printers are not provided).

```
Term ::=
    Identifier |
    'if' Term 'then' Term 'else' Term |
    'fun' Identifier '.' Term |
    Term Term |
    'let' Declaration 'in' Term |
  '(' Term ')'

Declaration ::=
  Identifier '=' Term |
  Declaration 'and' Declaration |
  'rec' Declaration |
  '(' Declaration ')'
```

Data types can be introduced into the language simply by having a predefined set of identifiers in the initial environment; this way there is no need to change the syntax or, more importantly, the typechecking program when extending the language.

For example, the following program defines the factorial function and applies it to zero:

```
let rec factorial n =
        if zero n
        then succ 0
        else (times (pair n (factorial (pred n))))
in factorial 0
```

3. Types

A type can be either a type variable α, β, etc., standing for an arbitrary type, or a type operator. Operators like **int** (integer type) and **bool** (boolean type) are nullary type operators. Parametric type operators like → (function type) or × (cartesian product type) take one or more types as arguments. The most general forms of the above operators are $\alpha \to \beta$ (the type of any function) and $\alpha \times \beta$, (the type of any pair of values); α and β can be replaced by arbitrary types to give more specialized function and pair types. Types containing type variables are called *polymorphic*, while types not containing type variables are *monomorphic*. All the types found in conventional programming languages, like Pascal, Algol 68 etc. are monomorphic.

Expressions containing several occurrences of the same type variable, like in $\alpha \to \alpha$, express contextual dependencies, in this case between the domain and the codomain of a function type. The typechecking process consists in matching type operators and instantiating type variables. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same variable must be instantiated to the same value: legal instantiations of $\alpha \to \alpha$ are **int** → **int**, **bool** → **bool**, $(\beta \times \xi) \to (\beta \times \xi)$, etc. This contextual instantiation process is performed by *unification*, [Robinson 1] and is at the basis of polymorphic typechecking. Unification fails when trying to match two different type operators (like **int** and **bool**) or when trying to instantiate a variable to a term containing that variable (like α and $\alpha \to \beta$, where a circular structure would be built). The latter situation arises in typechecking self-application (e.g. **fun x. (x x)**), which is therefore considered illegal.

Here is a trivial example of typechecking. The identity function **I = fun x. x** has type $\alpha \to \alpha$ because it maps any type onto itself. In the expression **(I 0)** the type of **0** (i.e. **int**) is matched to the domain of the type of **I**, yielding **int → int** as the specialized type of **I** in that context. Hence the type of **(I 0)** is the codomain of the type of **I**, which is **int** in this context.

In general, the type of an expression is determined by a set of type combination rules for the language constructs, and by the types of the primitive operators. The initial type environment could contain the following primitives for booleans, integers, pairs and lists (where → is the function type operator, **list** is the list operator, and × is cartesian product):

true, false	: **bool**
0, 1, ...	: **int**
succ, pred	: **int → int**
zero	: **int → bool**

pair	: $\alpha \to \beta \to (\alpha \times \beta)$
fst	: $(\alpha \times \beta) \to \alpha$
snd	: $(\alpha \times \beta) \to \beta$
nil	: α **list**
cons	: $(\alpha \times \alpha$ **list**$) \to \alpha$ **list**
hd	: α **list** $\to \alpha$
tl	: α **list** $\to \alpha$ **list**
null	: α **list** $\to$ **bool**

4. The type of 'length'

Before describing the typechecking algorithm, let us discuss the type of a simple recursive program which computes the length of a list:

```
let rec length =
        fun l.
                if null l
                then 0
                else succ(length(tl l))
in ...
```

(we write **length = fun l. ...** instead of the equivalent but more elegant **length l = ...** for convenience in the discussion below).

The type of **length** is α **list** $\to$ **int**; this is a polymorphic type as **length** can work on lists of any kind. The way we deduce this type can be described in two ways. In principle, typechecking is done by setting up a system of type constraints, and then solving it with respect to the type variables. In practice, typechecking is done by a bottom-up inspection of the program, matching and synthesizing types while proceeding towards the root; the type of an expression is computed from the type of its subexpressions and the type constraints imposed by the context, while the type of the predefined identifiers is already known and contained in the initial environment. It is a deep property of the type system and of the typechecking algorithm that the order in which we examine programs and carry out the matching does not affect the final result and solves the system of type constraints.

The system of type constraints for **length** is:

[1]	**null**	: α **list** $\to$ **bool**
[2]	**tl**	: β **list** $\to \beta$ **list**
[3]	**0**	: **int**
[4]	**succ**	: **int** $\to$ **int**
[5]	**null l**	: **bool**
[6]	**0**	: γ
[7]	**succ(length(tl l))**	: γ
[8]	**if null l then 0 else succ(length(tl l))**	: γ
[9]	**null**	: $\delta \to \epsilon$
[10]	**l**	: δ
[11]	**null l**	: ϵ
[12]	**tl**	: $\phi \to \xi$
[13]	**l**	: ϕ
[14]	**tl l**	: ξ

[15]	**length**	$: \theta \to \iota$
[16]	**tl l**	$: \theta$
[17]	**length(tl l)**	$: \iota$
[18]	**succ**	$: \kappa \to \lambda$
[19]	**length(tl l)**	$: \kappa$
[20]	**succ(length(tl l))**	$: \lambda$
[21]	**l**	$: \mu$
[22]	**if null l then 0** **else succ(length(tl l))**	$: \nu$
[23]	**fun l. if null l then 0** **else succ(length(tl l))**	$: \mu \to \nu$
[24]	**length**	$: \pi$
[25]	**fun l. if null l then 0** **else succ(length(tl l))**	$: \pi$

Lines [1-4] express the constraints for the predefined global identifiers, which are already known. The conditional construct imposes [5-8], that the result of a test must be boolean, and the two branches of the conditional must have the same type γ, which is also the type of the whole conditional expression. The four function applications in this program determine [9-20]; in each case the function symbol must have a functional type (e.g. $\delta \to \epsilon$ in [9]); its argument must have the same type as the domain of the function (e.g. δ in [10]), and the result must have the same type as the codomain of the function (e.g. ϵ in [11]). The lambda-expression [23] has a type $\mu \to \nu$, given that its parameter has type μ [21] and its body has type ν [22]. Finally the definition construct imposes that the variable being defined (**length** [24]) has the same type as its definition [25].

Typechecking **length** consists in (i) verifying that the above system of constraints is consistent (e.g. it does not imply **int** = **bool**), and (ii) solving the constraints with respect to π. The expected type of **length** (π = β **list** $\to$ **int**) can be inferred as follows:

$\pi = \mu \to \nu$	**by [25, 23]**
$\mu = \phi = \beta$ **list**	**by [21, 13, 12, 2]**
$\nu = \gamma =$ **int**	**by [22, 8, 6, 3]**

Considerably more work is needed to show that β is completely unconstrained, and that the whole system is consistent. The typechecking algorithm described in the next section systematically performs this work, functioning as a simple deterministic theorem prover for systems of type constraints.

Here is a bottom-up derivation of the type of **length** which is closer to what the typechecking algorithm really does; the consistency of the constraints (i.e. the absence of type errors) is also checked in the process:

[26]	**l**	$: \delta$	**[10]**
[27]	**null l**	**: bool**	
	ϵ = **bool**; $\delta = \alpha$ **list**;		**[11, 9, 1]**
[28]	**0**	**: int**	
	γ = **int**;		**[6, 3]**
[29]	**tl l**	$: \beta$ **list**	
	$\phi = \beta$ **list**; $\xi = \beta$ **list**; $\beta = \alpha$;		**[26, 27, 12-14, 2]**
[30]	**length(tl l)**	$: \iota$	
	$\theta = \beta$ **list**;		**[15-17, 29]**

[31]	**succ(length(tl l))**	**: int**	
	$\iota = \kappa =$ int;		**[18-20, 4, 30]**
[32]	**if null l then 0**		
	else succ(length(tl l))	**: int**	**[5-8, 27, 28, 31]**
[33]	**fun l. if null l then 0**		
	else succ(length(tl l))	**: β list → int**	
	$\mu = \beta$ list; $\nu =$ int;		**[21-23, 26, 27, 32]**
[34]	**length**	**: β list → int**	
	$\pi = \beta$ list → int;		**[24, 25, 33, 15, 30, 31]**

Note that recursion is taken care of: the types of the two instances of **length** in the program (the definition and the recursive function call) are compared in [34].

5. Typechecking

The basic algorithm can be described as follows.

1. When a new variable **x** is introduced by a lambda binder, it is assigned a new type variable α meaning that its type must be further determined by the context of its occurrences. The pair **<x,α>** is stored in an environment (called **TypeEnv** in the program) which is searched every time an occurrence of **x**, is found, yielding α (or any intervening instantiation of it) as the type of that occurrence.

2. In a conditional, the **if** component is matched to **bool**, and the **then** and **else** branches are unified in order to determine a unique type for the whole expression.

3. In an abstraction **fun x. e** the type of **e** is inferred in a context where **x** is associated to a new type variable.

4. In an application **f a**, the type of **f** is unified against a type **A** → β, where **A** is the type of **a** and β is a new type variable. This implies that the type of **f** must be a function type whose domain is unifiable to **A**; β (or any instantiation of it) is returned as the type of the whole application.

In order to describe the typechecking of **let** expressions, and of variables introduced by let binders, we need to introduce the notion of *generic* type variables. Consider the following expression:

```
fun f. pair (f 3) (f true)                    [Ex1]
```

In Milner's type system this expression cannot be typed, and the algorithm described above will produce a type error. In fact the first occurrence of **f** determines a type **int** → β for **f**, and the second occurrence determines a type **bool** → β for f, which cannot be unified with the first one.

Type variables appearing in the type of a lambda-bound identifier like **f** are called *non-generic* because, as in this example, they are shared among all the occurrences of **f** and their instantiations may conflict.

One could try to find a typing for **Ex1**, for example by somehow assigning it (α → β) → (β × β); this would compute correctly in situations like **(Ex1 (fun a. 0))** whose result would be **(pair 0 0)**. However this typing is unsound in general: for example **succ** has a type that matches α → β and it would be accepted as an argument to **Ex1** and wrongly applied to **true**. There are sound extensions of Milner's type system which can type **Ex1**, but they are beyond the scope of this discussion.

Hence there is a basic problem in typing heterogeneous occurrences of lambda-bound identifiers. This turns out to be tolerable in practice, because expressions like **Ex1** are not extremely useful or necessary, and because a different mechanism is provided. We are going to try and do better in typing heterogeneous occurrences of let-bound identifiers. Consider:

```
let f = fun a. a                              [Ex2]
in pair (f 3) (f true);
```

It is essential to be able to type the previous expression, otherwise no polymorphic function could be applied to distinct types in the same context, making polymorphism quite useless. Here we are in a better position than **Ex1**, because we know exactly what f is, and we can use this information to deal separately with its occurrences.

In this case f has type $\alpha \rightarrow \alpha$; type variables which, like α, occur in the type of let-bound identifiers (and that moreover do not occur in the type of *enclosing* lambda-bound identifiers) are called *generic*, and they have the property of being able to assume different values for different instantiations of the let-bound identifier. This is achieved operationally by making a copy of the type of f for every distinct occurrence of f.

In making a copy of a type, however, we must be careful not to make a copy of non-generic variables, which must be shared. The following expression for example is as illegal as **Ex1**, and g has a non-generic type which propagates to f:

```
fun g. let f = g                              [Ex3]
       in pair (f 3) (f true)
```

Again, it would be unsound to accept this expression with a type like $(\alpha \rightarrow \beta) \rightarrow (\beta \times \beta)$ (consider applying **succ** so that it is bound to g).

The definition of generic variables is:

> A type variable occurring in the type of an expression **e**
> is generic iff it does not occur in the type of the binder
> of any lambda-expression enclosing **e**.

Note that a type variable which is found to be non-generic while typechecking within a lambda expression, may become generic outside it. This is the case in **Ex2** where **a** is assigned a non-generic α, and f is assigned $\alpha \rightarrow \alpha$ where α is now generic.

To determine when a variable is generic we maintain a list of the non-generic variables at any point in the program: when a type variable is not in the list it is generic. The list is augmented when entering a lambda; when leaving the lambda the old list automatically becomes the current one, so that that type variable becomes generic. In copying a type, we must only copy the generic variables, while the non-generic variables must be shared. In unifying a non-generic variable to a term, all the type variables contained in that term become non-generic.

Finally we have to consider recursive declarations:

```
let rec f = ... f ...
in ... f ...
```

which are treated as if the **rec** where expanded using a fixpoint operator **Y** (of type $(\alpha \rightarrow \alpha) \rightarrow \alpha$):

```
let f = Y fun f. ... f ...
in ... f ...
```

it is now evident that the instances of (the type variables in the type of) f in the recursive definition must be non-generic, while the instances following **in** are generic.

5. Hence, to typecheck a **let** we typecheck its declaration part, obtaining an environment of identifiers and types which is used in the typechecking of the body of the **let**.

6. A declaration is treated by checking all its definitions $x_i = t_i$, each of which introduces a pair $<x_i,T_i>$ in the environment, where T_i is the type of t_i. In case of (mutually) recursive

declarations $x_i = t_i$ we first create an environment containing pairs $<x_i,\alpha_i>$ for all the x_i being defined, and where the α_i are new non-generic type variables (they are inserted in the list of non-generic variables for the scope of the declaration). Then all the t_i's are typechecked in that environment, and their types T_i are again matched against the α_i (or their instantiations).

6. A digression on models, inference systems and algorithms

There are two basic approaches to the formal semantics of types. The most fundamental one is concerned with devising mathematical models for types, normally by mapping every type expression into a set of values (the values having that type); the basic difficulty here is in finding a mathematical meaning for the $\rightarrow$ operator [Scott 76] [Milner 78] [MacQueen 84].

The other, complementary, approach is to define a formal system of axioms and inference rules, in which it is possible to prove that an expression has some type. The relationship between models and formal systems is very strong. A semantic model is often a guide in defining a formal system, and every formal system should be self-consistent, which is often shown by exhibiting a model for it.

A good formal system is one in which we can prove nearly everything we "know" is true (according to intuition, or because it is true in a model). Once a good formal system has been found, we can "almost" forget the models, and work in the usually simpler, syntactic framework of the system.

Typechecking is more strictly related to formal systems than to models, because of its syntactic nature. A typechecking algorithm, in some sense, implements a formal system, by providing a procedure for proving theorems in that system. The formal system is essentially simpler and more fundamental than any algorithm, so that the simplest presentation of a typechecking algorithm is the formal system it implements. Also, when looking for a typechecking algorithm, it is better to first define a formal system for it.

Not all formal type systems admit typechecking algorithms. If a formal system is too powerful (i.e. if we can prove many things in it), then it is likely to be undecidable, and no decision procedure can be found for it. Typechecking is usually restricted to decidable type systems, for which typechecking algorithms can be found. However in some cases undecidable systems could be treated by incomplete typechecking *heuristics* (this has never been done in practice, so far), which only attempt to prove theorems in that system, but may at some point give up. This could be acceptable in practice because there are limits to the complexity of a program: its meaning could get out of hand long before the limits of the typechecking heuristics are reached.

Even for decidable type systems, all the typechecking algorithms could be exponential, again requiring heuristics to deal with them. This has been successfully attempted in Hope [Burstall 80] for the treatment of overloading in the presence of polymorphism.

The following section presents an inference system for the kind of polymorphic typechecking we have described. We have now two distinct views of typechecking: one is "solving a system of type equations", as we have seen in the previous sections, and the other is "proving theorems in a formal system", as we are going to see now. These views are interchangeable, but the latter one seems to provide more insights because of its connection with type semantics on one side and algorithms on the other.

7. An inference system

In the following inference system, the syntax of types is extended to type quantifiers $\forall\alpha.\ \tau$. In Milner's type system, all the type variables occurring in a type are intended to be implicitly quantified at the top level. For example, $\alpha \rightarrow \beta$ is really $\forall\alpha.\ \forall\beta.\ \alpha \rightarrow \beta$. However, quantifiers cannot be nested inside type expressions.

A type is called *shallow* if it has the form $(\forall\alpha_1. \cdots \forall\alpha_n.\ \tau)$ where $n \geqslant 0$ and no quantifiers occur in τ. Our inference system allows the construction of non-shallow types: unfortunately we do not have typechecking algorithms able to cope with them. Hence, we are only interested in inferences which involve only shallow types. We have chosen to use type quantifiers in the inference system because this helps explain the behavior of generic/non-generic type variables, which correspond exactly to free/quantified type variables. For a slightly different inference system which avoids non-shallow types, see [Damas 82].

Here is the set of inference rules. [VAR] is an axiom scheme, while the other rules are proper inferences. The horizontal bar reads "implies". The notation $A \vdash e : \tau$ means that given a set of assumptions A, we can deduce that the expression e has type τ. An assumption is a typing of a variable which may be free in the expression e. The notation $A.x : \tau$ stands for the union of the set A with the assumption $x : \tau$; and $\tau[\sigma/\alpha]$ is the result of substituting σ for all the free occurrences of α in τ.

[VAR] $\quad A.x : \tau \vdash x : \tau$

[COND] $$\frac{A \vdash e : bool \qquad A \vdash e' : \tau \qquad A \vdash e'' : \tau}{A \vdash (\mathit{if}\ e\ \mathit{then}\ e'\ \mathit{else}\ e'') : \tau}$$

[ABS] $$\frac{A.x : \sigma \vdash e : \tau}{A \vdash (\lambda x.\ e) : \sigma \rightarrow \tau}$$

[COMB] $$\frac{A \vdash e : \sigma \rightarrow \tau \qquad A \vdash e' : \sigma}{A \vdash (e\ e') : \tau}$$

[LET] $$\frac{A \vdash e' : \sigma \qquad A.x : \sigma \vdash e : \tau}{A \vdash (\mathit{let}\ x = e'\ \mathit{in}\ e) : \tau}$$

[REC] $$\frac{A.x : \tau \vdash e : \tau}{A \vdash (\mathit{rec}\ x.\ e) : \tau}$$

[GEN] $$\frac{A \vdash e : \tau}{A \vdash e : \forall\alpha.\ \tau} \qquad (\alpha \text{ not free in } A\ )$$

[SPEC] $$\frac{A \vdash e : \forall\alpha.\ \tau}{A \vdash e : \tau[\sigma/\alpha]}$$

As a first example, we can deduce the most general type of the identity function: $(\mathit{fun}\ x.\ x) : \forall\alpha.\ \alpha \rightarrow \alpha$.

$$\cfrac{\cfrac{x : \alpha \vdash x : \alpha}{\vdash (\mathit{fun}\ x.\ x) : \alpha \rightarrow \alpha}}{\vdash (\mathit{fun}\ x.\ x) : \forall\alpha.\ \alpha \rightarrow \alpha} \qquad \begin{array}{l} [\mathit{VAR}] \\ [\mathit{ABS}] \\ [\mathit{GEN}] \end{array}$$

A specialized type for the identity function can be deduced either from the general type:

$$\frac{\vdash (\mathit{fun}\ x.\ x) : \forall\alpha.\ \alpha \rightarrow \alpha}{\vdash (\mathit{fun}\ x.\ x) : int \rightarrow int} \qquad [\mathit{SPEC}]$$

or more directly:

$$\frac{x : int \vdash x : int}{\vdash (\mathit{fun}\ x.\ x) : int \rightarrow int} \qquad \begin{array}{l} [\mathit{VAR}] \\ [\mathit{ABS}] \end{array}$$

We can extend the above inference to show *(fun x.x)*(3): *int*;

$$\dfrac{\dfrac{x: int,\ 3: int \vdash x: int}{3: int \vdash (fun\ x.\ x): int \to int}\ [VAR]\,[ABS] \qquad 3: int \vdash 3: int\ \ [VAR]}{3: int \vdash (fun\ x.\ x)(3): int}\ [COMB]$$

Here is an example of a *forbidden* derivation using non-shallow types, which can be used to give a type to *(fun x. x x)*, which our algorithm is not able to type (here $\phi = \forall\alpha.\ \alpha \to \alpha$):

$$\dfrac{\dfrac{\dfrac{x{:}\phi \vdash x{:}\phi}{x{:}\phi \vdash x{:}\phi\to\phi}\ [VAR]\,[SPEC] \qquad x{:}\phi \vdash x{:}\phi\ \ [VAR]}{x{:}\phi \vdash x\ x: \phi}\ [COMB]}{\vdash (fun\ x.\ x\ x): \phi\to\phi}\ [ABS]$$

Note how $\forall\alpha.\alpha\to\alpha$ gets instantiated to $(\forall\alpha.\alpha\to\alpha) \to (\forall\alpha.\alpha\to\alpha)$ by [SPEC], substituting $\forall\alpha.\alpha\to\alpha$ for α.

We want to show now that *(let f = fun x.x in pair (f 3)(f true))*: *int*×*bool*. Take $A = \{3{:}\ int,\ true{:}\ bool,\ pair{:}\ \forall\alpha.\ \forall\beta.\ \alpha\to\beta\to\alpha\times\beta\}$ and $\phi = \forall\alpha.\ \alpha \to \alpha$.

$$\dfrac{\dfrac{A.f{:}\phi \vdash f{:}\phi}{A.f{:}\phi \vdash f: int\to int} \qquad A.f{:}\phi \vdash 3{:}\ int}{A.f{:}\phi \vdash f\ 3{:}\ int}$$

$$\dfrac{\dfrac{A.f{:}\phi \vdash f{:}\phi}{A.f{:}\phi \vdash f: bool\to bool} \qquad A.f{:}\phi \vdash true: bool}{A.f{:}\phi \vdash f\ true: bool}$$

$$\dfrac{\begin{array}{l} A.f{:}\phi \vdash f\ 3{:}\ int \\ A.f{:}\phi \vdash f\ true: bool \\ \underline{A.f{:}\phi \vdash pair: \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha\times\beta} \\ \qquad\qquad \cdots \end{array}}{A.f{:}\phi \vdash pair\ (f\ 3)(f\ true){:}int\times bool}$$

$$\dfrac{A \vdash fun\ x.x{:}\ \phi \qquad A.f{:}\phi \vdash pair\ (f\ 3)(f\ true){:}int\times bool}{A \vdash (let\ f = fun\ x.x\ in\ pair\ (f\ 3)(f\ true)){:}\ int\times bool}$$

Note that from the assumption $f{:}\ \forall\alpha.\alpha\to\alpha$, we can independently instantiate α to *int* and *bool*; i.e., *f* has a generic type. Instead, in *(fun f. pair (f 3)(f true))(fun x. x)*, which is the function-application version of the above let expression, no shallow type can be deduced for *(fun f. pair (f 3)(f true))*.

A variable is generic if it does not appear in the type of the variables of any enclosing lambda-binder. Those binders must occur in the set of assumptions, so that they can be later discarded by [ABS] to create those enclosing lambdas. Hence a variable is generic if it does not appear in the set of assumptions. Therefore, if a variable is generic, we can apply [GEN] and introduce a quantifier. This determines a precise relation between generic variables and quantifiers.

There is a formal way of relating the above inference system to the typechecking algorithm presented in the previous sections. It can be shown that if the algorithm succeeds in producing a type for an expression, then that type can be deduced from the inference system (see [Milner 78] for a result involving a closely related inference system). We are now going to take a different, informal approach to intuitively justify the typechecking algorithm. We are going the show how an algorithm can be extracted from an inference system. In this view a typechecking algorithm is a *proof heuristic*; i.e. it is a strategy to determine the order in which the inference rules should be applied. If the proof heuristic succeeds, we have determined that a type can be inferred. If it fails, however, it may still be possible to infer a type. In particular our heuristic will be unable to cope with expressions which require some non-shallow type manipulation, like in the deduction of (*fun x. x x*)(*fun x. x*): $\forall\alpha.\ \alpha\rightarrow\alpha$.

There are two aspects to the heuristic. The first one is how to determine the sets of assumptions, and the second is the order in which to apply the inference rules. If a language requires type declarations for all identifiers, it is trivial to obtain the sets of assumptions, otherwise we have to do *type inference*.

In carrying out type inference, lambda-bound identifiers are initially associated to type variables, and information is gathered during the typechecking process to determine what the type of the identifier should have been in the first place. Hence, we start with these initial broad assumptions, and we build the proof by applying the inference rules in some order. Some of the rules require the types of two subexpressions to be equal. This will not usually be the case, so we *make* them equal by unifying the respective types. This results in specializing some of the types of the identifiers. At this point we can imagine repeating the same proof, but starting with the more refined set of assumptions we have just determined: this time the types of the two subexpressions mentioned above will come out equal, and we can proceed.

The inference rules should be applied in an order which allows us to build the expression we are trying to type from left to right and from the bottom up. For example, earlier we wanted to show that (*fun x. x*): $\forall\alpha.\ \alpha\rightarrow\alpha$. Take $x:\alpha$ as our set of assumptions. To deduce the type of (*fun x. x*) bottom-up we start with the type of x, which we can obtain by [VAR], and then we build up (*fun x. x*) by [ABS].

If we proceed left to right and bottom-up then, with the exception of [GEN] and [SPEC], at any point only one rule can be applied, depending on the syntactic construct we are trying to obtain next. Hence the problem reduces to choosing when to use [GEN] and [SPEC]; this is done in conjunction with the [LET] rule.

Before applying [LET], we derive $A \vdash e':\sigma'$ (refer to the [LET] rule) and then we apply all the possible [GEN] rules, obtaining $A \vdash e':\sigma$, where σ can be a quantified type. Now we can start deriving $A.x:\sigma \vdash e:\tau$, and every time we need to use [VAR] for x and σ is quantified, we immediately use [SPEC] to strip all the quantifiers, replacing the quantifier variable by a fresh type variable. These new variables are then subject to instantiation, as discussed above, which determines more refined ways of using [SPEC].

As an exercise, one could try to apply the above heuristic to infer the type of **length**, and observe how this corresponds to what the typechecking algorithm does in that case. Note how the list of non-generic variables corresponds to the set of assumptions and the application of [GEN] and [SPEC] rules.

8. The program

The following ML program implements the polymorphic typechecking algorithm, and also illustrates how polymorphism is used in ML. ML syntax and semantics are described in [Milner 84]. Here are some comments on the program and the ML language; they refer to the program code.

Keywords are in boldface, identifiers in roman, and data constructors in italic. Data constructors are used in expressions to create data, and in pattern matching to analyze and select data components. Type variables are roman identifiers starting with a quote: ''a' is a type variable and is normally pronounced "alpha".

• The program begins with standard list manipulation routines, defined by pattern matching; note that no types are declared here.

• Pointers, in the sense of assignable references to values, are not predefined in ML. They can be built by 'ref' (built-in updatable references) and 'Option'. A pointer is an updatable reference to an optional value; if the value is missing ('none') we have a null pointer, otherwise we have a non-null pointer ('some'). 'Void' creates a new null pointer, 'Access' dereferences a pointer, and 'Assign' updates a pointer. The type ''a Pointer' is parametric, but it will only be used here as a 'Type Pointer'.

● Time stamps are used to uniquely identify variables. This is an abstract type (so that time stamps cannot be faked) with an own variable 'Counter' which is incremented every time a new time stamp is needed.

● The types 'Ide', 'Term' and 'Decl' form the abstract syntax of our language. A type expressions 'Type' can be a type variable or a type operator. A type variable, identified by a unique time stamp, is *uninstantiated* when its type pointer is null, or *instantiated* otherwise. An instantiated type variable behaves like its instantiation. A type operator (like 'bool' or '→') has a name and a list of type arguments (none for 'bool', two for '→').

● The function 'Prune' is used whenever a type expression has to be inspected: it will always return a type expression which is either an uninstantiated type variable or a type operator; i.e. it will skip instantiated variables, and will actually prune them from expressions to remove long chains of instantiated variables.

● The function 'OccursInType' checks whether a type variable occurs in a type expression.

● The type 'NGVars' is the type of lists of non-generic variables. 'FreshType' makes a copy of a type expression, duplicating the generic variables and sharing the non-generic ones.

● Type unification is now easily defined. Remember that when unifying a non-generic variable to a term, all the variables in that term become non-generic. This is handled automatically by the lists of non-generic variables, and no special code is needed in the unification routine.

● Type environments are then defined. Note that 'RetrieveTypeEnv' always creates fresh types; some of this copying is unnecessary and could be eliminated.

● Finally we have the typechecking routine, which maintains a type environment and a list of non-generic variables. Recursive declarations are handled in two passes. The first pass 'AnalyzeRecDeclBind' simply creates a new set of non-generic type variables and associates them with identifiers. The second pass 'AnalyzeRecDecl' analyzes the declarations and makes calls to 'UnifyType' to ensure the recursive type constraints.

```
{ ------ List Manipulation (standard library routines) ------ }

val rec
        map f nil = nil |
        map f (head :: tail) = (f head) :: (map f tail);

val rec
        fold f nil x = x |
        fold f (head :: tail) x = f (head, fold f tail x);

val rec
        exists p nil = false |
        exists p (head :: tail) = if p head then true else exists p tail;

{ ------ Options ------ }

type ´a Option = data none | some of ´a;

{ ------ Pointers ------ }

type ´a Pointer = data pointer of ´a Option ref;
val Void() = pointer(ref none);
val Access(pointer(ref(some V))) = V;
val Assign(pointer P, V) = P := some V;

{ ----- Time Stamps ----- }

abstype Stamp = data stamp of int;
with  local Counter = ref 0
        in      val NewStamp() = (Counter := !Counter+1; stamp(!Counter));
                val SameStamp(stamp S, stamp S´) = (S = S´)
        end
end;
```

```
{ ----- Identifiers ----- }

type Ide = data symbol of string;

{ ----- Expressions ----- }

type rec Term = data
        ide of Ide |
        cond of Term * Term * Term |
        lamb of Ide * Term |
        appl of Term * Term |
        block of Decl * Term

and Decl = data
        defDecl of Ide * Term |
        andDecl of Decl * Decl |
        recDecl of Decl;

{ ----- Types ----- }

type rec Type = data
        var of Stamp * Type Pointer |
        oper of Ide * Type list;

val NewTypeVar() = var(NewStamp(),Void());
val NewTypeOper(Name,Args) = oper(Name,Args);

val SameVar (var(Stamp,_),var(Stamp',_)) =
        SameStamp(Stamp,Stamp');

val rec Prune (Type: Type): Type =
        case Type of
                var(_,Instance) =>
                        (case Instance of
                                pointer(ref none). Type |
                                pointer(_).
                                        let val Pruned = Prune(Access Instance)
                                        in (Assign(Instance,Pruned); Pruned) end
                        ) |
                oper(_) => Type;

val rec OccursInType(TypeVar: Type, Type: Type): bool =
        let val Type = Prune Type
        in case Type of
                var(_) => SameVar(TypeVar,Type) |
                oper(Name,Args) =>
                        fold (fun Arg,Accum => OccursInType(TypeVar,Arg) orelse Accum) Args false
        end;

val OccursInTypeList(TypeVar: Type, TypeList: Type list): bool =
        exists (fun Type => OccursInType(TypeVar,Type)) TypeList;
```

```
{ ----- Generic Variables ----- }

type NGVars = data nonGenericVars of Type list;

val EmptyNGVars = nonGenericVars [];

val ExtendNGVars(Type: Type, nonGenericVars NGVars): NGVars =
        nonGenericVars(Type :: NGVars);

val Generic(TypeVar: Type, nonGenericVars NGVars): bool =
        not(OccursInTypeList(TypeVar,NGVars));

{ ----- Copy Type ----- }

type CopyEnv = (Type * Type) list;

val FreshType (Type: Type, NGVars: NGVars): Type =
        let val rec Fresh (Type: Type, Env: CopyEnv ref): Type =
                let val Type = Prune Type
                in case Type of
                        var(_) =>
                                if Generic(Type,NGVars) then FreshVar(Type,!Env,Env) else Type |
                        oper(Name,Args) =>
                                NewTypeOper(Name, map (fun Arg => Fresh(Arg,Env)) Args)
                end
        and FreshVar (Var: Type, Scan: CopyEnv, Env: CopyEnv ref): Type =
                if null Scan
                then  let val NewVar = NewTypeVar()
                      in (Env := (Var,NewVar)::(!Env); NewVar) end
                else  let val (OldVar,NewVar)::Rest = Scan
                      in if SameVar(Var,OldVar) then NewVar else FreshVar(Var,Rest,Env) end
        in Fresh(Type,ref []) end;

{ ----- Basic Type Operators ----- }

val BoolType =
        NewTypeOper(symbol "bool",[]);

val FunType (From: Type, Into: Type): Type =
        NewTypeOper(symbol "fun",[From;Into]);
```

```
{ ----- Type Unification ----- }

val rec UnifyType (Type: Type, Type´: Type): unit =
        let val Type = Prune Type and Type´ = Prune Type´
        in case Type of
                var(Stamp,Instance) =>
                        if OccursInType(Type,Type´)
                        then case Type´ of var(_) => () | oper(_) => escape "Unify"
                        else Assign(Instance,Type´) |
                oper(Name,Args) =>
                        case Type´ of
                                var(_) => UnifyType(Type´,Type) |
                                oper(Name´,Args´) =>
                                        if Name=Name´ then UnifyArgs(Args,Args´) else escape "Unify"
        end

and     UnifyArgs ([], []) = () |
        UnifyArgs (Hd::Tl, Hd´::Tl´) = (UnifyType(Hd, Hd´); UnifyArgs(Tl, Tl´)) |
        UnifyArgs (_) = escape "Unify";

{ ----- Environments ----- }

type TypeEnv = data typeEnv of (Ide * Type) list;

val EmptyTypeEnv = typeEnv [];

val ExtendTypeEnv (Bind: Ide, Type: Type, typeEnv TypeEnv): TypeEnv =
        typeEnv((Bind,Type)::TypeEnv);

val RetrieveTypeEnv (Ide: Ide, typeEnv TypeEnv, NGVars: NGVars): Type =
        let val rec
                Retrieve ([]: (Ide * Type) list): Type = escape "Undefined identifier" |
                Retrieve ((Bind,Type)::Rest: (Ide * Type) list): Type =
                        if Ide=Bind then FreshType(Type,NGVars) else Retrieve Rest
        in Retrieve TypeEnv end;
```

```
{ ----- Typechecking ----- }

val rec
        AnalyzeTerm (ide Ide, TypeEnv, NGVars): Type =
                RetrieveTypeEnv(Ide,TypeEnv,NGVars) |
        AnalyzeTerm (cond(If,Then,Else), TypeEnv, NGVars): Type =
                let     val () = UnifyType(AnalyzeTerm(If,TypeEnv,NGVars),BoolType);
                        val TypeOfThen = AnalyzeTerm(Then,TypeEnv,NGVars);
                        val TypeOfElse = AnalyzeTerm(Else,TypeEnv,NGVars)
                in (UnifyType(TypeOfThen,TypeOfElse); TypeOfThen) end |
        AnalyzeTerm (lamb(Bind,Body), TypeEnv, NGVars): Type =
                let     val TypeOfBind = NewTypeVar();
                        val BodyTypeEnv = ExtendTypeEnv(Bind,TypeOfBind,TypeEnv);
                        val BodyNGVars = ExtendNGVars(TypeOfBind,NGVars);
                        val TypeOfBody = AnalyzeTerm(Body,BodyTypeEnv,BodyNGVars)
                in FunType(TypeOfBind,TypeOfBody) end |
        AnalyzeTerm (appl(Fun,Arg), TypeEnv, NGVars): Type =
                let     val TypeOfFun = AnalyzeTerm(Fun,TypeEnv,NGVars);
                        val TypeOfArg = AnalyzeTerm(Arg,TypeEnv,NGVars);
                        val TypeOfRes = NewTypeVar()
                in (UnifyType(TypeOfFun,FunType(TypeOfArg,TypeOfRes)); TypeOfRes) end |
        AnalyzeTerm (block(Decl,Scope), TypeEnv, NGVars): Type =
                let val DeclEnv = AnalyzeDecl(Decl,TypeEnv,NGVars)
                in AnalyzeTerm(Scope,DeclEnv,NGVars) end

and     AnalyzeDecl (defDecl(Bind,Term), TypeEnv, NGVars): TypeEnv =
                ExtendTypeEnv(Bind,AnalyzeTerm(Term,TypeEnv,NGVars),TypeEnv) |
        AnalyzeDecl (andDecl(Left,Right), TypeEnv, NGVars): TypeEnv =
                AnalyzeDecl(Right,AnalyzeDecl(Left,TypeEnv,NGVars),NGVars) |
        AnalyzeDecl (recDecl Rec, TypeEnv, NGVars): TypeEnv =
                let val TypeEnv,NGVars = AnalyzeRecDeclBind(Rec,TypeEnv,NGVars)
                in AnalyzeRecDecl(Rec,TypeEnv,NGVars) end

and     AnalyzeRecDeclBind (defDecl(Bind,Term), TypeEnv, NGVars) : TypeEnv * NGVars =
                let val Var = NewTypeVar()
                in ExtendTypeEnv(Bind,Var,TypeEnv), ExtendNGVars(Var,NGVars) end |
        AnalyzeRecDeclBind (andDecl(Left,Right), TypeEnv, NGVars) : TypeEnv * NGVars =
                let val TypeEnv,NGVars = AnalyzeRecDeclBind(Left,TypeEnv,NGVars)
                in AnalyzeRecDeclBind(Right,TypeEnv,NGVars) end |
        AnalyzeRecDeclBind (recDecl Rec, TypeEnv, NGVars) : TypeEnv * NGVars =
                AnalyzeRecDeclBind(Rec,TypeEnv,NGVars)

and     AnalyzeRecDecl (defDecl(Bind,Term), TypeEnv, NGVars): TypeEnv =
                (UnifyType(RetrieveTypeEnv(Bind,TypeEnv,NGVars),
                           AnalyzeTerm(Term,TypeEnv,NGVars));
                 TypeEnv) |
        AnalyzeRecDecl (andDecl(Left,Right), TypeEnv, NGVars): TypeEnv =
                AnalyzeRecDecl(Right,AnalyzeRecDecl(Left,TypeEnv,NGVars),NGVars) |
        AnalyzeRecDecl (recDecl Rec, TypeEnv, NGVars): TypeEnv =
                AnalyzeRecDecl(Rec,TypeEnv,NGVars);
```

9. Conclusions and acknowledgements

This paper presented some of the pragmatic knowledge about polymorphic typechecking, trying to relate it informally to the theoretical background. These ideas have been developed by a number of people over a number of years, and have been transmitted to me by discussions with Luis Damas, Mike Gordon, Dave MacQueen, Robin Milner and Ravi Sethi.

References

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: "Hope: an Experimental Applicative Language", Conference Record of the 1980 LISP Conference, Stanford, August 1980, pp. 136-143.

[Damas 82] L.Damas, R.Milner: "Principal type-schemes for functional programs", Proc. POPL 82, pp.207-212.

[Gordon 79] M.J.Gordon, R.Milner, C.P.Wadsworth: "Edinburgh LCF", Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[Hindley 69] R.Hindley: "The principal type scheme of an object in combinatory logic", Transactions of the American Mathematical Society, Vol. 146, Dec 1969, pp. 29-60.

[MacQueen 84] D.B.MacQueen, R.Sethi, G.D.Plotkin: "An ideal model for recursive polymorphic types", Proc. Popl 84.

[Milner 78] R.Milner: "A theory of type polymorphism in programming", Journal of Computer and System Sciences, No. 17, 1978.

[Milner 84] R.Milner: "A proposal for Standard ML", Proc. of the 1984 ACM Symposium on Lisp and Functional Programming, Aug 1984.

[Robinson 65] J.A.Robinson: "A machine-oriented logic based on the resolution principle", Journal of the ACM, Vol 12, No. 1, Jan 1965, pp. 23-49.

[Scott 76] D.S.Scott: "Data types as lattices", SIAM Journal of Computing, 4, 1976.

Annotated Bibliography on LCF

Compiled by Lawrence Paulson
with contributions from the authors

. Computer Laboratory
University of Cambridge
September 1984

L. Aiello, M. Aiello, R. Weyhrauch (1977), Pascal in LCF: semantics and examples of proof, *Theoretical Computer Science* 5, pages 135–177.

A. J. Cohn (1979), High level proof in LCF, Fourth conference on Automated Deduction, pages 73–80.

The use of theories and tactics to specify and verify a simple compiler.

A. J. Cohn (1980), *Machine Assisted Proofs of Recursion Implementation*, Report CST-6-79, PhD. Thesis, University of Edinburgh.

Several case studies of proof, ranging from the correctness of recursion removal transformations to a simple compiler.

A. J. Cohn and R. Milner (1982), On using Edinburgh LCF to prove the correctness of a parsing algorithm, Report CSR-113-82, Dept. of Computer Science, University of Edinburgh.

An introduction to the use of Edinburgh LCF, illustrated by a proof of the correctness of a parser for expressions. It also demonstrates the use of Milner's structural induction package.

A. J. Cohn (1982), The correctness of a precedence parsing algorithm in LCF, Report 21, Computer Lab., University of Cambridge.

A. J. Cohn (1983), The equivalence of two semantic definitions: a case study in LCF, *SIAM Journal of Computing* 12, pages 267–285.

R. L. Constable and J. L. Bates (1984), The nearly ultimate PEARL, Report TR 83-551, Cornell University, Ithaca, New York.

Describes a descendant of LCF, where PPLAMBDA has been replaced by a constructive theory of types.

D. A. Giles (1978), The theory of lists in LCF, Report CSR-31-78, Dept. of Computer Science, University of Edinburgh.

M. J. C. Gordon, R. Milner, and C. Wadsworth (1978), A metalanguage for interactive proof in LCF, Fifth Principles of Programming Languages, pages 119–130.

A tutorial on the language ML, emphasizing polymorphism and abstract types.

M. J. C. Gordon, R. Milner, and C. Wadsworth (1979), *Edinburgh LCF*, Springer.

The manual for Edinburgh LCF, both ML and PPLAMBDA.

M. J. C. Gordon (1982), Representing a logic in the LCF metalanguage, *in:* D. Néel, editor, *Tools and Notions for Program Construction*, Cambridge University Press, pages 163–185.

A tutorial paper on the LCF methodology for interfacing object languages to ML (illustrated with propositional calculus, not PPLAMBDA).

M. J. C. Gordon (1983a), LCF_LSM: A System for Specifying and Verifying Hardware, Report 41, Computer Lab., University of Cambridge.

Describes a system built on top of Cambridge LCF by adding terms loosely based on CCS to the logic PPLAMBDA.

M. J. C. Gordon (1983b), Proving a Computer Correct with the LCF_LSM Hardware Verification System, Report 42, Computer Lab., University of Cambridge.

M. J. C. Gordon (1984), Higher Order Logic: description of the HOL proof generating system, Report (in preparation), Computer Lab., University of Cambridge.

S. Holmström, Experiments and experiences with Edinburgh LCF, Report LPM-6, Dept. of Computer Sciences, Chalmers University, Göteborg.

F. Jensen (1981a), Inductive inference in reflexive domains, Report CSR-86-81, Dept. of Computer Science, University of Edinburgh.

F. Jensen (1981b), An LCF-system for automatic creation of theories for 1-constructible data types, Report CSR-87-31, Dept. of Computer Science, University of Edinburgh.

J. Leszczyłowski (1980a), An experiment with 'Edinburgh LCF,' *in:* W. Bibel and R. Kowalski, editors, *Fifth Conference on Automated Deduction*, Springer LNCS 87, pages 170–181.

A proof due to Boyer and Moore, the termination of a normalization function for conditional expressions, is verified in LCF.

J. Leszczyłowski (1980b), Theory of FP systems in Edinburgh LCF, Report CSR-61-80, Dept. of Computer Science, University of Edinburgh.

R. Milner (1979), LCF: a way of doing proofs with a machine, Eighth Math. Foundations of Comp. Sci.

R. Milner, L. Morris, M. Newey (1975), A logic for computable functions with reflexive and polymorphic types, IRIA Conference on Proving and Improving Programs, pages 371–394.

R. Milner, R. Weyhrauch (1972), Proving compiler correctness in a mechanized logic, *in:* B. Meltzer, D. Michie, editors, *Machine Intelligence 7*, Wiley, pages 51–70.

K. Mulmuley (1984a), The mechanization of existence proofs of recursive predicates, *in:* R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170, pages 460–475.

Describes Edinburgh LCF theories and tactics for proving the existence of inclusive predicates, which play a role in compiler verification. Theories of the universal domain and of finitary projections allow quantification over domains within PPLAMBDA. The system reduces the existence of a predicate to several goals, and proves most of them automatically. It can handle several examples from the literature.

K. Mulmuley (1984b), *Full Abstraction and Semantic Equivalence*, PhD thesis (in preparation), Carnegie-Mellon University, 1984?.

A fuller description of the system of Mulmuley (1984a), and its application to his discovery of a fully abstract model of the lambda-calculus.

L. Paulson (1983a), Recent developments in LCF: examples of structural induction, Report 34, Computer Lab., University of Cambridge.

An introduction to performing structural induction in LCF, including proofs of two simple theorems about substitution.

L. Paulson (1983b), Rewriting in Cambridge LCF, Report 35, Computer Lab., University of Cambridge.

A preliminary version of Paulson (1983f).

L. Paulson (1983c), The revised logic PPLAMBDA: a reference manual, Report 36, Computer Lab., University of Cambridge.

Describes the axioms, standard and derived inference rules, and syntactic functions for the Cambridge version of PPLAMBDA. This version includes the logical connectives $\vee$, $\exists$, and $\leftrightarrow$.

L. Paulson (1983d), Tactics and tacticals in Cambridge LCF, Report 39, Computer Lab., University of Cambridge.

Describes the primitive and derived tactics and tacticals for breaking apart goals and assumptions, and for applying rewriting or resolution to goals. Illustrates them via a sample interactive session.

L. Paulson (1983e), Structural induction in LCF, Report 44, Computer Lab., University of Cambridge.

States axioms for defining recursive data types, and derives the rule of structural induction. Constructor functions may be lazy or strict in any arguments, and may satisfy equational constraints. The resulting types are related to initial algebras.

L. Paulson (1983f), A higher-order implementation of rewriting, *Science of Computer Programming* 3, pages 119–149.

Describes in detail the implementation of rewriting in Cambridge LCF, in successive layers of simple, modular functions. Rewriting functions are built up from primitives by means of higher-order functions.

L. Paulson (1984a), Verifying the unification algorithm in LCF, Report 50, Computer Lab., University of Cambridge. (To appear in *Science of Computer Programming*.)

Describes how Manna and Waldinger's proof of the unification algorithm was formalized in LCF, and gives a detailed proof of a theorem about substitution.

L. Paulson (1984b), Deriving structural induction in LCF, *in:* G. Kahn, D. B. MacQueen, G. Plotkin, *International Symposium on Semantics of Data Types*, Springer LNCS 173, pages 197–214.

A shortened and revised version of Paulson (1983e).

L. Paulson (1984c), Lessons learned from LCF, *in:* D. Djørner, editor, *Workshop on Formal Software Development Combining Specification Methods.* Springer. (Also Report 54, Computer Lab., University of Cambridge.)

An introductory, survey paper on the concepts and history of LCF. Introduces the metalanguage ML, logic PPLAMBDA, inference rules, and tactics. Describes LCF proofs in denotational semantics, functional programming, and digital circuits, and discusses the evolution of ideas from this work.

K. Petersson (1982), A programming system for type theory, Report LPM-21, Dept. of Computer Sciences, Chalmers University, Göteborg.

D. Sannella, R. Burstall (1983), Structured theories in LCF, Report CSR-129-83, Dept. of Computer Science, University of Edinburgh.

A proposal to provide CLEAR-like primitives for building theories, providing information hiding, parametrization, and renaming.

D. Schmidt (1983b), Natural deduction theorem proving in set theory, Report CSR-142-83, Dept. of Computer Science, University of Edinburgh.

D. Schmidt (1984), A programming notation for tactical reasoning, *in:* R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170, pages 445–459.

S. Sokołowski (1983a), A note on tactics in LCF, Report CSR-140-83, Dept. of Computer Science, University of Edinburgh.

Describes tactics that allow certain variables in goals to be instantiated by unification.

S. Sokołowski (1983b), An LCF proof of the soundness of Hoare's logic, Report CSR-146-83, Dept. of Computer Science, University of Edinburgh.

A detailed account of the verification of the Hoare rules for an if-while language defined by a direct denotational semantics. Infinite programs are allowed — the while statement is defined to be an infinite nest of if statements.

ABSTRACTS

Univ. of Edinburgh, Computer Science Department

Stephan Sokolowski
A Note on Tactics in LCF
Internal Report CSR-140-83
August 1983

During my experiments in LCF I came across the situations where the LCF tactical approach as described in the "Edinburgh LCF" by Gordon, Milner and Wadsworth turns out to be unsatisfactory. The difficulties that I encountered seem hardly inherent to my particular area of applications, therefore I put forward to reformulate some basic concepts of LCF rather than to look for an *ad hoc* solution. In what follows I give some justification for the changes and I describe how to use new definitions of proof, tactic and new standard tacticals from the package UTAC.COD.

Stephan Sokolowski
An LCF Proof of Soundness of Hoare's Logic -- A Paper without a Happy Ending
Internal Report CSR-146-83
October 1983

This paper describes the proof of soundness of Hoare's logic carried out by the Edinburgh LCF theorem prover. It illustrates the way one can construct a general proof strategy for a particular problem area and then apply it to prove theorems in this area. The paper shows how much can be done using Edinburgh LCF and also what cannot be done.

Univ. of Cambridge, Computer Laboratory

Lawrence Paulson
Verifying the Unification Algorithm in LCF
Tech. Rep. No. 50
March 1984

Manna and Waldinger's verification of the unification algorithm, which includes a substantial theory of substitutions, has been performed in the interactive theorem-prover LCF. The problems and results are surveyed, with references to papers that give details.

The LCF formalization differs from Manna and Waldinger's, particularly since LCF proves theorems in Scott's Logic of Continuous Functions. Tedious reasoning about termination appears everywhere, and the final well-founded induction is reformulated as two nested structural inductions. A simpler data structure for expressions shortens the proofs.

The paper demonstrates interaction with LCF, defining expressions as a recursive type, and introducing functions to search for an occurrence of one expression inside another, and to apply a substitution to an expression. Substitutuion is proved to be monotonic relative to the occurrence ordering. The formalization of unification and its properties is presented.

The exercise has produced a better understanding of how structural induction, substitutions, and finite sets are used in mechanical theorem-proving. Numerous improvements have been made to Edinburgh LCF, resulting in a new version, Cambridge LCF.

Jon Fairbairn
A New Type-Checker for a Functional Language
Tech. Rep. No. 53
August 1984

A polymorphic type checker for the functional programming language Ponder [Fairbairn 82] is described. The initial sections give an overview of the syntax of Ponder, and some of the motivation behind the design of the type system. This is followed by a definition of the relation of 'generality' between these types, and of the notion of type-validity of Ponder programmes. An algorithm to determine whether a Ponder programme is type-valid is then presented. The final sections give examples of useful types which may be constructed within the type system, and describe some of the areas in which it is thought to be inadequate.

Lawrence Paulson
Lessons Learned from LCF
Tech. Rep. No. 54
August 1984

The history and future prospects of LCF are discussed. The introduction sketches basic concepts such as the language ML, the logic PPLAMBDA, and backwards proof. The history discusses LCF proofs about denotational semantics, functional programs, and digital circuits, and describes the evolution of ideas about structureal induction, tactics, logics of computation, and the use of ML. The bibliography contains thrity-five references.

Lawrence Paulson
Constructing Recursion Operators in Intuitionistic Type Theory
Tech. Rep. No. 57
October 1984

Martin-Löf's Intuitionistic Theory of Types is becoming popular for formal reasoning about computer programs. To handle recursion schemes other than primitive recursion, a theory of well-founded relations is presented. Using primitive recursion over higher types, induction and recursion are formally derived for a large class of well-founded relations. This includes < (on natural numbers) and relations formed by inverse images, addition, multiplication, and exponentiation of other relations. The theory is compared with work in the field of ordinal recursion over higher types.

University of Umea, Institute of Information Processing

Lennart Edblom
Implementation of ML on a Lisp Machine
Report UMINF-11483, ISSN 0348-0542

This report describes an implementation of the functional language ML on a Lisp Machine. An implementation in Franz Lisp was translated to the Lisp dialect of the Lisp Machine, Zetalisp.

Following brief introductions to ML and the Lisp Machine, the main part of the report describes the changes necessary to move the Franz Lisp programs to the Lisp Machine. The main differences were found to be the functions for character handling, input/ouput, and file handling, otherwise only minor changes were needed.

The Zetalisp functions implementing ML input and file handling, along with an example of ML programming are shown in the appendices. An elementary introduction to ML on the Lisp Machine, written in Swedish, is attached as appendix F.

Lambda Calculus Models of Typed Programming Languages

John Clifford Mitchell

Submitted to the MIT Department of Electrical Engineering and Computer Science on August 24, 1984 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Abstract

The first part of this thesis studies the *second-order lambda calculus*, developed independently by Girard and Reynolds. In this typed language, featuring polymorphic functions and abstract data type declarations, types play an important role in defining the set of well-formed terms. We discuss the features of second-order lambda calculus that correspond to common programming language constructs, demonstrating some natural extensions to Ada, Alphard, CLU, ML and related programming languages. In particular, we describe a simple approach to passing representations of abstract data types as parameters and returning representations as results of function calls.

The semantics of second-order lambda calculus is studied using a slightly more general *higher-order lambda calculus* that makes it possible to treat type-building operations like the product-space constructor or the tagged union constructor as optional constants of the language. We define semantic models for the general language $\mathcal{H}\Lambda$ and prove a completeness theorem, extending previous work on the second-order lambda calculus. A formal axiomatization of models of $\mathcal{H}\Lambda$ is then given using a *higher-order type theory* $\mathcal{HT}$. This axiomatization is similar in spirit to the first-order combinatory characterization of models of untyped lambda calculus.

The second part of the thesis is concerned with type inference, the problem of finding types for untyped expressions. We study two type inference systems for untyped lambda calculus. The first system combines a relatively simple language of types with some simple postulates about relationships between types. The main results here are a complete axiomatization for all valid typing statements and a decision procedure for a natural class of typing statements. In practical terms, the decision procedure is a typing algorithm that may be used to add simple coercions to programming languages like ML.

The second type inference system includes the more complicated universally quantified type expressions of second-order lambda calculus. A general definition of the semantics of typing statements with universally quantified types is proposed, generalizing previous work by MacQueen, Sethi and Plotkin. These *inference models* are models of untyped lambda calculus with extra structure similar to models of second-order lambda calculus. We show that the GR_{eq} axiom system, an extension of the typing rules for second-order lambda calculus, is complete for all typing statements valid over all inference models. A more specialized set of type inference rules, the GRS_{eq} rules, characterize the more specialized *simple semantics*. We also study containments between types by reformulating the inference rules so that containments play a central role.

Thesis Supervisor: Albert R. Meyer, Professor of Computer Science

Author's present address: AT&T Bell Labs, 600 Mountain Ave., Murray Hill, NJ 07974.

Addenda to the Mailing List

Makoto Amamiya
Research Section 2,
Musashino Electrical Comm. Lab
3-9-11 Midoricho Musashino-shi
Tokyo 180
Japan

S. Arun-Kumar
NCSDCT
Tata Institute of Fundamental Research
Homi Bhabha Road
Bombay 400 005
India

Egidio Astesiano
Istituto di Matematica
Via Leon B. Alberti 4
16132 Genova
Italy

E. K. Blum
Dept of Mathematics
University of Southern California
Los Angeles, CA 90089
USA

Pierre Casteran
Universite de Bordeaux I
Mathematiques et Informatique
351 cours de la Liberation
33405 Talence Cedex
France

Paul Chisholm
Heriot-Watt University
Department of Computer Science
79 Grassmarket
Edinburgh EH1 2HJ
Scotland

Tony Davie
University of St. Andrews
John Honey Building
North Haugh
St. Andrews KY16 9SR
Scotland

Rocco DeNicola
Istituto di Elaborazione dell'Informazione
via S.Maria 46
56100 Pisa
Italy

Lennart Edblom
Institute of Information Processing
University of Umea
S-901 87 Umea
Sweden

Marie-Claude Gaudel
Jean-Claude Raoult
LRI, Bat 490
Univ. Paris-Sud
91405 Orsay-cedex
France

Mark Gerhardt
Raytheon Co.
Submarine Signal Division
P.O.Box 360
Portsmouth, RI 02871
USA

Susan L. Graham
Computer Science Division-EECS
Univ. of California
Berkeley, CA 94720
USA

Brent Hailpern
IBM T.J.Watson Research Center
P.O.Box 218
Yorktown Heights, NY 10598
USA

Roger Hindley
Mathematics Dept
University College
Swansea SA22 8PP
Wales

Kiyoshi Ishihata
Department of Information Science
University of Tokyo
7-3-1 Hongo
Bunkyo-Ku, Tokyo 113
Japan

Sam Kamin
Computer Science Dept.
Univ. of Illinois
1304 W. Springfield
Urbana, IL 61801
USA

Butler Lampson
320 Lombard St
Philadelphia, Penn.
USA

Barbara Liskov
MIT/LCS
545 Technology Square
Cambridge, MA 02-39
USA

Steven Litvintchouk
Mitre
Burlington Road
Bedford, MASS 01730
USA

John McCarthy
Computer Science Dept.
Stanford University
Stanford CA 94305
USA

Robert Neely
Network Development Centre
International Computers Limited
PO Box 8
Icknield Way, Letchworth
Hertfordshire SG6 1ER
England

Erich J. Neuhold
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
USA

Malcolm Newey
Computer Science Department
Australian National University
Canberra ACT 2600
Australia

Per-Olov Nilsson
Dept of Computer Science
Lund University
Box 725
S-22007 Lund
Sweden

K. V. Nori
Tata Research, Development
and Design Centre
1 Mangaldas Road
PUNE 411 011
India

E. R. Olderog
Institut fur Informatik
Olshausenstr 40-60
D-2300 Kiel
W.Germany

William Samson
Dept of Mathematics
and Computer Studies
Dundee College of Technology
Bell Street
Dundee DD1 1HG
Scotland

David Shorter
Systems Designers Ltd.
Systems House
105 Fleet Road
Fleet
Hampshire GU13 8NZ
England

M.R. Sleep
School of Computing Studies
and Accountancy
University of East Anglia
Norwich NR4 7TJ
England

Stefan Sokolowski
Uniwersytet Gdanski
Instytut Matematyki
u. Wita Stwosza 57
8–952 Gdansk
Poland

Herbert Stoyan
Institut fur Mathematische Maschinen
und Datenverarbeitung (VI)
der Universitat Erlangen-Nurnberg
Martensstrabe 3
D-8520 Erlangen
W. Germany

Norihisa Suzuki
Dept of Math Eng
The University of Tokyo
Bunkyo-ku, Tokyo 113
Japan

Satoro Takasu
Research Inst. for Math. Sciences
Kyoto University
Sakyoku, Kyoto 606
Japan

Prof. Boris Trakhtenbrot
School of Mathematical Science
Tel Aviv University
Ramat Aviv
Tel Aviv
Isreal

Greg Vesonder
AT&T Bell Laboratories
Whippany Road
Whippany, NJ 07981
USA

Friedrich von Henke
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
USA

Prof R. Wilhelm
FB 1o - Informatik
Universität
D-66 Saarbrücken 11
W. Germany

John Williams
K51-282
IBM San Jose Research Center
5600 Cottle Road
San Jose, CA 95193
USA

Martin Wirsing
Fakultat fur Informatik
Universitat Passau
Innstr. 53
D-8390 Passau
W.Germany

Akinori Yonezawa
Tokyo Institute of Technology
Dept of Information Science
Oookayama, Meguro-ku
Tokyo 152
Japan

Mailing Changes

Old	New
Hans Boehm	Andrew P. Black
Dept of Computer Science, FR-35	"
University of Washington	"
Seattle, WA 98195	"
USA	
John Cartmell	Iain D. Craig
Software Sciences Limited	"
London & Manchester House Park Street	"
Macclesfield Cheshire SK11 6SR	"
England	"