Volume I, Number 2 April, 1983

Polymorphism

The ML/LCF/Hope Newsletter

Contents

Letter from the Editors

This third issue of Polymorphism is devoted to work from Cambridge University. There are two reports by Dave Matthews on the programming language Poly, a systems programming language derived from Pascal and Russell but strongly influenced by ML. "Ponder and its Type System" by Jon Fairbairn describes a type system related to that of ML but incorporating explicit universal type quantification. Larry Paulson contributes three papers describing some of the LCF research being done at Cambridge. Finally, we have a note from C. Kitchen and B. Lynch from Trinity College, Dublin with advice on running ML under the Eunice system (a Unix emulator running on top of VAX VMS).

We have had some questions about the subscription or distribution policy of Polymorphism. Currently we are distributing one copy to each group or location where ML/LCF/Hope activity (or at least interest) exists, with the expectation that local duplication and distribution will follow. The mailing list therefore contains only one individual per site or institution, and this individual is responsible for local redistribution. A group may change its representative on the mailing list by simply informing us.

In order for us all to get a better appreciation of the scope of ML/LCF/Hope style research, and who is doing it, we encourage each group to send in a short summary of its activities and a list of participants. We will then publish these summaries in future issues of Polymorphism.

In the next issue, probably forthcoming in June, we hope to publish a new manual for Luca's ML (ML 81?), a Hope manual, and a tutorial note on ML typechecking, including a simplified treatment of references. The Franz lisp based Hope compiler should also be ready for distribution by June.

Luca Cardelli
David MacQueen

Bell Laboratories
Murray Hill, NJ 07974
USA

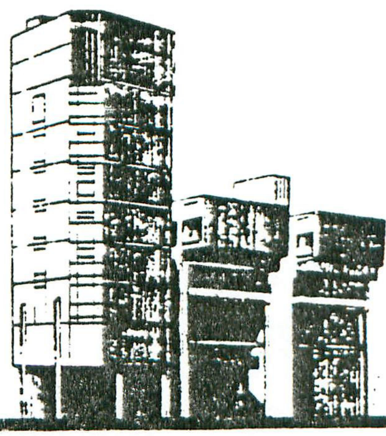
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 28

POLY REPORT

by

D. C. J. Matthews



University of Cambridge
Computer Laboratory

POLY REPORT

D.C.J. Matthews, August 1982

Computer Laboratory,

University of Cambridge

Abstract

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time oveheads. The type system, based on that of Russell allows polymorphic operations to be used to manipulate abstract objects, but with all the type checking being done at compile-time. Types may be passed explicitly or by inference as parameters to procedures, and may be returned from procedures. Overloading of names and generic types can be simulated by using the general procedure mechanism. Despite the generality of the language, or perhaps because of it, the type system is very simple, consisting of only three classes of object. There is an exception mechanism, similar to that of CLU, and the exceptions raised in a procedure are considered as part of its 'type'. The construction of abstract objects and hiding of internal details of the representation come naturally out of the type system.

POLY REPORT

D.C.J. Matthews, August 1982

Computer Laboratory,

University of Cambridge

Abstract

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time oveheads. The type system, based on that of Russell allows polymorphic operations to be used to manipulate abstract objects, but with all the type checking being done at compile-time. Types may be passed explicitly or by inference as parameters to procedures, and may be returned from procedures. Overloading of names and generic types can be simulated by using the general procedure mechanism. Despite the generality of the language, or perhaps because of it, the type system is very simple, consisting of only three classes of object. There is an exception mechanism, similar to that of CLU, and the exceptions raised in a procedure are considered as part of its 'type'. The construction of abstract objects and hiding of internal details of the representation come naturally out of the type system.

1. INTRODUCTION

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time oveheads. The type system, based on that of Russell [1, 2] allows polymorphic operations to be used to manipulate abstract objects, but with all the type checking being done at compile-time. Types may be passed explicitly or by inference as parameters to procedures, and may be returned from procedures. Overloading of names and generic types can be simulated by using the general procedure mechanism. Despite the generality of the language, or perhaps because of it, the type system is very simple, consisting of only three classes of object. There is an exception mechanism, similar to that of CLU [3], and the exceptions raised in a procedure are considered as part of its 'type'. The construction of abstract objects and hiding of internal details of the representation come naturally out of the type system.

1.1 Syntax

The description of the syntax in this report follows the Russell and Alphard reports in the use of superscripts # + * to denote respectively optional items, repetition with at least one occurrence, and repetition with possibly no occurrence. Subscript symbols occur as separators between occurrences of the item. Braces { and } are used as meta-brackets.

For instance

```
const# <name> is equivalent to
const <name> | <name>
and
<identifier>+, is equivalent to
<identifier> | <identifier>,<identifier>|
<identifier>,<identifier>,<identifier> | ....
```

Reserved words, shown underlined in this report, may be written in upper, lower or mixed case. In identifiers the case of letters is significant and two identifiers are distinct if any of their letters are written in different cases.

```
<identifier> ::= <letter> | <letter><letter or digit>* | <symbol>*
<letter> ::= a|b|c|d|....|z|A|B|C|D|....|Z
<digit>  ::= 0|1|2|3|4|5|6|7|8|9|.
<letter or digit> ::= <letter>|<digit>|_
<symbol> ::= +|-|*|/|\|!|#|$|%|&|=|<|>|?|_|~|^
```

Comments are written by enclosing them in braces ({ and }). Words are separated by one or more spaces, newlines or comments. The following words have special meaning and cannot be used as ordinary identifiers.

==	:	any	begin	catch
const	do	early	else	end
extends	if	inline	infix	let
letrec	prefix	proc	raise	raises
record	then	type	union	while

2. SPECIFICATION CHECKING

Every object in Poly has both a value and a specification. The value is what is used when the object is used, the specification describes what can be done with it. There are three main classes of objects; constants, procedures and types. Exceptions (signals) could be regarded as a fourth class though they cannot be used in the same way as the other classes.

Constants are simple values which can be manipulated but have no visible structure of their own. A constant is the implementation of an abstract object.

Procedures are operations which can manipulate constants, other procedures or types. They may return objects which may be constants, procedures or types or they may raise exceptions. A procedure implements an abstract operation.

Types are simply sets of named objects. They may, like conventional types, have values belonging to them or they may be simply modules. A type implements an abstract set of co-operating operations which can together manipulate objects.

```
<specification> ::= <constant specification>
                  | <procedure specification>
                  | <type specification>
```

The specification of an object is checked when it is used in some context, either as a parameter to a procedure, or when an identifier is declared with an explicit specification. The object and the context must be of the same class (constant, procedure or type) and must satisfy the rules for that class. The rules themselves are given in the following sections.

2.1 Constants

```
<constant specification> : = const# <name>

<name> ::= <name>$<identifier> | <identifier>
```

A constant is a simple unstructured value. It has no properties of its own and can only be manipulated by certain operations. All constants belong to a named type, which is the set of operations which can correctly interpret it. New procedures can be written which operate on a constant but they will always be written using existing operations from the type.

The specification of a constant is T (or const T) where T is some type name. It is then said to have type T. A value with type T can only be used in a context requiring a value of type T. This rule is similar to the name equivalence rule for type checking in other languages. Two different type names are incompatible even if they are derived from the same declaration.

e.g. let S,T == integer

creates S and T with all the operations of integer but values of types S and T cannot be combined with integer values or with each other.

2.2 Procedures

```
<procedure specification> ::= proc <operator mode>#
                   <implied argument list>#
                   <explicit argument list>
                   <result specification>#
                   [raises <exception list>]#

<implied argument list> ::= [ <argument list> ]
<explicit argument list> ::= ( <argument list> )

<argument list>           ::= <argument specification>*;

<argument specification>  ::= <identifier list> : <specification>
                              | <specification>

<operator mode>           ::= infix | prefix

<exception list>          ::= any | <identifier>*,
```

Procedures constitute the most complex class of objects. In general a procedure takes objects as parameters, alters the global state and either returns a result or raises an exception. The specification of a procedure contains the specifications of its arguments and of its result. It also indicates whether the procedure is to be used as an operator, and what exceptions it may raise.

A procedure may have two argument lists, the explicit arguments and the implied arguments. The implied arguments must all be types and must be referred to by the explicit arguments. As far as checking the specification of a procedure in a context is concerned the two argument lists are considered as one, the only difference comes when the procedure is called. Only the arguments listed in the explicit argument list need be given, the ones in the implied argument list will be inferred from the explicit arguments.

Any arguments which are types may be used in the specifications of subsequent arguments or in the result. This allows polymorphic procedures. For instance

proc (t : type end ; t)t

is the specification of a procedure which takes any type together with a value of that type and returns as result a value of the type. An alternative representation, using an implied parameter would be

proc [t : type end] (t)t

which would be compatible with the previous specification but when called only the constant would be supplied.

A procedure matches a given context if corresponding arguments in the value and the context have the same specifications and the result specifications are the same. For the specifications to match they must be the same except that where a specification refers to a preceeding type name in one argument list the corresponding specification in the other list must refer to the corresponding name. Apart from this the names of the arguments are ignored in the matching process. For instance

proc (tt : type end ; x : tt)tt

is the same as the examples above and would match them correctly. The exception lists have to match in that every exception listed with the procedure value must appear in the exception list of the context. An exception list with the word any is considered being the set of all possible exceptions.

2.3 Types

```
<type specification> ::= type [(<identifier>)]#
                         [<identifier list>:<specification>]*;
                         end
```

A type is a collection of attributes: procedures, constants or types. Its specification is the list of the names of the attributes, together with their specifications. The specification of a type is type (T) x : A; y : B; z : C;... end where t, the internal name, represents the type within the specifications A;B;C... The ordering of the attributes is irrelevant. A type value matches a context if every attribute in the specification of the context appears in the specification of the value. In other words, attributes may be lost from a type value to make it match a context, but if any required attribute is not present or has the wrong specification then the value will not match. For instance a type value with specification

type (i) zero: i; succ: proc(i)i end

would match a context with specification

type (i) succ: proc(i)i end

but not the other way round. The type specification of a context can be regarded as a filter which removes all attributes apart from those listed.

2.4 Coercions

There is one circumstance in which a coercion may be applied when an expression appears to break the above rules. It is included to allow the usual syntax of expressions using variables, when a variable is used to denote its current value. A variable in Poly is a type with two attributes, both procedures (see sections 5.4 and 5.5). 'assign' gives a new value to the variable, and 'content' returns its current value. If a type 't' is used in a context requiring a constant, 't' is eplaced by 't$content()' if 't' has such an attribute. (i.e. the 'conten' procedure of the type is called to return the current value).

3. STATEMENTS AND EXPRESSIONS

```
<expression>  ::= <if expression>
                | <while expression>
                | <infix expression>
                | <raise expression>
```

An expression describes a computation which returns a result and possibly has side-effects. All expressions in Poly return results. If a result is not returned explicitly then a value of 'void$empty' is returned. It is also returned from expressions like the 'while loop' which cannot return a general value.

3.1 Declarations

```
<declaration> ::= let <identifier>+, {: <specification>}#
                  == <expression>
                | letrec <identifier> {: <specification>}#
                  == <expression>
```

A declaration associates a name with a value. The name can then be used to represent the value in the block which contains the declarations and any inner blocks. Declarations can occur in compound expressions or type constructors. A declaration may contain a specification for the value bound to the name. This may be necessary to simplify checking when a complex expression is being bound or when the specification of the name is not the same as the expression.

Declaring a name in an inner scope will hide an outer declaration of the name; there is no overloading in Poly. Names may not be declared twice in the same scope. Names belonging to operations of a type are not automatically available in a scope where the type is available. The effect of overloading can often be achieved by overloading.

<u>let</u> and <u>letrec</u> differ in that <u>letrec</u> declares the identifier before the expression, making it available inside it, while <u>let</u> declares the identifiers afterwards. <u>letrec</u> must therefore be used for declaring recursive procedures. A declaration has scope from the point of declaration to the end of the block containing it. An identifier cannot be referred to before it is declared.

3.2 If Statement and If Expression

```
<if expression> ::= if <expression> then <expression>
                    else <expression>
                  | if <expression> then <expression>
```

The if expression causes an expression to be executed depending on the value of the "guard" expression. The guard, which must have a result type of boolean, is evaluated and if it returns "true" the expression following the <u>then</u> is executed. If the guard is "false" the expression following the <u>else</u> is executed. The specifications of the values produced by the then-part and the else-part must be capable of being converted to a single specification, that of the result. The second form of the if expression, without the else-part, may only be used if the then-part returns a value of void$empty.

The ambiguity in the syntax of nested if expressions is resolved by requiring that an if-expression without an else-part may not be followed by <u>else</u>. An else-part is thus paired with the nearest unpaired <u>then</u>.

3.3 Compound Expression

```
<compound expression>  ::= begin
                            <expression block>
                            end
                          | ( <expression block> )

<expression block>  ::= {<declaration> | <expression>}*;
                        <catch expression>°
                        end

<catch expression>  ::= catch <expression>
```

The compound expression is used to introduce new identifiers and to group expressions together. All the expressions except the last must return a value of void$empty. The specification of the compound expression is the specification of the last expression. An empty compound expression or a compound expression containing only declarations returns void$empty.

The catch expression is used to trap any exceptions which may be raised in the expressions or declarations in the block. If an exception is raised within the block and not caught in an inner block, it may be caught at this level. The expression following the word <u>catch</u> must yield a procedure whose argument must have type 'string'. Its result must be similar to the result of the last expression in the compound expression (i.e. they must both be capable of being converted to a single specification, that of the result). When an exception is caught the name of the exception is passed as the parameter to this procedure and the result of the procedure is returned as the result of the compound expression. If there is no catch expression or a further exception is raised in the catch expression then it is propagated to the next level out where it may be caught or propagated further.

3.4 Operators

```
<infix expression>  ::= <infix expression> <operator>
                        <prefix expression>
                      | <prefix expression>

<prefix expression>  ::= <operator> <prefix expression>
                       | <application>
```

The specification of a procedure can indicate that it is to be used as a prefix or infix operator. This allows a more convenient notation for some expressions than the procedure application, but does not affect the semantics. The precedence rules for operators are simple: All operators of the same mode (prefix or infix) have the same precedence, prefix operators being more binding than infix.

3.5 Procedure Application

```
<application> ::= <application> (<expression>*,)
                | <basic expression>

<basic expression> ::= <compound expression>
                     | <name>
                     | <manifest>
                     | <procedure constructor>
                     | <type constructor>
                     | <union type>
                     | <record type>
```

A procedure application causes the expression associated with the routine returned from the expression to be executed. The expressions in brackets, if any, provide values for the explicit formal parameters of the routine. The expressions must have specifications which match the specifications of the formal parameters There must be the same number of expressions as formal parameters in the explicit parameter list. The specifications of the parameters are matched from left to right, starting with the implied parameters. Each parameter is tested for a match using the rules in chapter 2. If it matches then any subsequent use of the formal parameter is renamed with the matched parameter value. The specification of the result of a procedure call is the result specification of the procedure, except that any formal parameters used are renamed with the actual parameter value. If 'plus' has specification

```
proc (inttype: type (i) + : proc (i; i)i end;
     x, y: inttype)inttype
```

and 'a' and 'b' have type integer then it can be correctly used as

```
plus(integer, a, b)
```

and the result will have type integer.

3.6 Names

```
<name> ::= <identifier> | <name>$<identifier>
```

A name yields the value given to it by its declaration. Names may be simple identifiers or a sequence of identifiers separated by the $ symbol. The first identifier must always have been declared in one of the currently open scopes. Subsequent names must be an attribute of the type referred to by the previous name, so all but the last must refer to a type. E.g. if 'atype' has specification

```
type (a)
x, y: proc(a)a;
z: type (z)
   p: proc (z)
   end
end
```

then

atype   atype$x   atype$y   atype$z   atype$z$p

are all valid names.

3.7 Manifests

```
<manifest> ::= <number>
             | <single-quoted sequence>
             | <double-quoted sequence>
<number>   ::= <digit> <alphanumeric>*

<single-quoted sequence>   ::= '<any char>*'

<double-quoted sequence>   ::= "<any char>*"
```

Manifest constants are values which stand for themselves. There are three forms of manifest, the number and the single and double-quoted sequences.

```
0   9999   0x6f83   9xz   'X'   'hello'   '\n'   ''
""   "A message"
```

are examples of manifests. They can be converted to values of any type by defining a procedure "convertn", "convertc" or "converts" to return a value of the appropriate type. For instance "convertn" for integer is defined as

```
convertn: proc(string)integer raises conversionerror
```

The compiler will act as though a call to the appropriate routine had been written and the conversion will be made. Prefixing a manifest with a type selector causes the compiler to use convertc, convertn or converts from that type.

3.8 Procedure Constructor

```
<procedure constructor> ::=
        proc <operator mode>
        [[<argument list>]]* (<argument list>)
        <specification>* [raises <exception list>]*
        <compound expression>
```

The procedure constructor creates procedure values. If a result specification is given the expression must return a value which satisfies it. If the result specification is omitted then the compound expression must return void$empty. The exceptions which may be raised in the compound expression must be equal to or a subset of those listed in the raises list. However omitting the raises list is taken to mean that a list should be made from the exceptions which may be raised in the compound expression.

3.9 Type Constructor

```
<type constructor> ::= type [(<identifier>)]*
                       [<declaration>;]*\
                       [extends <expression>;]*
                       <declaration>; end
```

The type constructor makes a new type by collecting together a set of declarations. The declarations will usually be of procedures which provide additional operations to an existing type.

The "extends" clause defines an existing type as the basis for the new type. Any new operations can be written in terms of operations available on this type. For instance

```
let newint == type (int) extends integer;
                 let cube == proc(i: int)int
                     (i int$* i int$* i)
              end;
```

declares "newint" to be like integer but with the new operation "cube" added. Its specification includes all the operations available for integer together with the new operation, however it is a completely separate type from integer. Values can be converted between the original type and the new type by means of two operations, 'up' and 'down' which are created when 'extends' is used. In this example they have specifications

```
up: proc (integer)int;        down: proc (int)integer
```

Within the declarations the identifier in parentheses, in this case "int", represents the type being created.

Existing operations on the base type may be overridden by declaring a new operation with the same name. If let is used to declare a new operation then it can be written using the existing one since the new operation will not replace the existing one until the end of the declaration. All the operations of the base type, together with any newly declared operations, are returned by the type constructor. It is possible to hide operations by binding the result to a type name with a specification with fewer operations.

3.10 Union Type

<union type> ::= union ({<identifier>†:<specification>}$^{*}_{;}$)

The union type returns a type which is the union of the specifications listed. For each identifier x with specification T there are three operations on the union type. inj_x creates a union from a value of specification T and proj_x extracts a value of specification T from the union. is_x is a predicate which is true only if the union was created with inj_x. proj_x is only valid if is_x is true, otherwise "projectionerror" will be raised.

For instance the specification of the union created by union(x: S; y: T) is

```
type (U)
   is_x, is_y : proc (U)boolean;
   inj_x :      proc (S)U;
   inj_y :      proc (T)U;
   proj_x :     proc (U)S raises projectionerror;
   proj_y :     proc (U)T raises projectionerror
end
```

Two different identifiers listed with the same specification (e.g. union(x,y:T)) create different variants, so proj_y is not allowed on a union created with inj_x.

3.11 Record Type

<record type> ::= record ({<identifier>†:<specification>}$^{*}_{;}$)

The record type returns a type which is the Cartesian product of the specifications listed. The identifiers are declared as fields of the record and can be used as selecting procedures. The selecting procedures take a value of the record type as argument and return a value of the field specification as result. There is also a constructor procedure "constr" which makes a record out of values with the field specifications. For instance the record created by record(x: S; y: T); has specification

```
type (U)
   x      : proc(U)S;
   y      : proc(U)T;
   constr : proc(S; T)U
end
```

Because the fields may have any specification a record may be used to make types whose basic values are procedures or types. Hence procedure or

type variables can be used.

3.12 Raise Statement

<raise expression> ::= raise <identifier>

The raise expression causes the named exception to be raised. This causes further processing to be halted until the exception is caught. Working from the raise expression outwards each compound expression is examined until one is found which contains a catch phrase. If the exception is caught the corresponding procedure is executed and processing continues.

If the exception is not caught within the immediately enclosing procedure and it has not been included in its "raises" list then the program is in error. Otherwise the exception is raised at the point of call of the procedure, and the exception propagated further.

The raise expression may form part of an expression even though it does not return a value. For the purpose of specification checking it appears to have the required specification for the context.

3.13 While Statement

<while expression> ::= while <expression> do <expression>

The while expression causes the expression after the do to be executed repeatedly until the expression, which must have a result type boolean, returns "false". The expression is evaluated before the expression and if it is "false" on entry the expression is never executed. The body of the while statement must return void$empty. and the while statement itself returns void$empty.

4. STANDARD DECLARATIONS

The chapter describes the declarations that should be provided by the compiler or standard library for any implementation.

4.1 Void

```
Specification
type (v) empty: v end
```

The type "void" has only one value, "empty". "empty" is returned as the result of operations which do not otherwise return a result.

4.2 Boolean

```
Specification
type (bool)
   true, false: bool;
   &, | : proc infix (bool; bool) bool;
   ~ : proc prefix (bool) bool;
   print: proc(bool)
end
```

The type "boolean" is one of the few types which are actually built into the language. It is required because various constructions in the language such as if and while expressions use values of specification "boolean".

4.3 Integer

```
Specification
type (i)
   convertn: proc(string)i raises conversionerror;
   +,-,* : proc  infix (i; i) i
                    raises rangeerror;
   div, mod: proc infix (i; i) i
                    raises divideerror;
   succ,pred,neg,abs: proc(i) i raises rangeerror;
   =,<>,>=,<=,>,< : proc(i; i) boolean;
   print: proc(i)
end
```

Integer represents the positive and negative integers. "convertn" is invoked automatically to convert a number (i.e. a sequence of letters or digits beginning with a digit) into an integer value. It raises "conversionerror" if the characters do not form a valid integer.

4.4 New

Specification

```
proc [base: type end] (initialval: base)
   type
   assign:  proc (base);
   content: proc ()base
   end
```

"new" is a procedure which creates and initialises variables. A variable in Poly is a type containing a pair of procedures, one of which (cont) extracts the value currently held, and the other (assign) stores a new value in it.

4.5 Vector

Specification

```
proc [base: type end] (size: integer; initial: base)
    proc (index: integer)
        type
        assign:  proc (base)
        content: proc ()base
        end
    raises subscripterror
raises rangeerror
```

Vector constructs one dimensional arrays of values which can be indexed by an integer value. The value of index must be in the range from 1 to size (inclusive) otherwise 'subscripterror' will be raised. The result of indexing the vector is a variable so that the element can either be read or updated. All the elements are initialised to the value 'initial' when the array is constructed. The size of the vector must be at least 1 otherwise rangeerror will be raised.

4.6 Char

Specification

```
type (c)
   print: proc (c);
   succ, pred: proc (c)c raises rangeerror;
   =, <> : proc (c;c)boolean;
   convertc: proc (string)c;
end
```

The type 'char' represents the characters used to form readable text.

4.7 String

Specification

```
type (str)
   sub: proc infix (str; integer)char raises subscripterror;
   + : proc infix (str; str)str;
   =, <> : proc (str; str)boolean;
   length: proc (str)integer;
   converts: proc (str)str;
   print: proc (str);
   mk: proc(char)str
end
```

String is the type used for arguments to converts, convertn and convertc, and in a catch phrase. A constant of this type is regarded as a sequence of characters of unspecified length. The 'length' procedure gives the number of characters in a string, and 'sub' can be used obtain a particular character. Strings can be concatenated using '+' and compared using '=' and '<>'.

5. REFERENCES

[1]
Demers A. and Donahue J. Report on the Programming Language Russell
TR79-371 Department of Computer Science, Cornell University, 1979

[2]
Demers A. and Donahue J. Revised Report on Russell
TR 79-389 Department of Computer Science, Cornell University, 1979

[3]
Liskov B. et al. CLU Reference Manual
Lecture Notes in Computer Science No. 114, Springer-Verlag, 1981

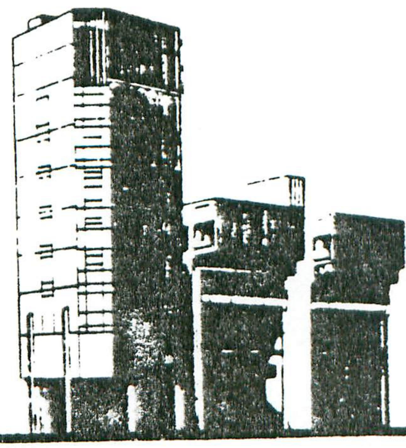
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 29

INTRODUCTION TO POLY

by

D. C. J. Matthews



University of Cambridge
Computer Laboratory
Corn Exchange Street

INTRODUCTION TO POLY

D.C.J. Matthews, May 1982

Computer Laboratory,

University of Cambridge

Abstract

This report is a tutorial introduction to the programming language Poly. It describes how to write and run programs in Poly using the VAX/UNIX implementation. Examples given include polymorphic list functions, a double precision integer package and a subrange type constructor.

INTRODUCTION TO POLY

Poly is a programming language which supports polymorphic operations. This document explains how it is used on the VAX.

1. Commands and Declarations

The system is entered by running the appropriate program (e.g. /mnt/dcjm/poly at Cambridge). The compiler will then reply with a prompt (>). To exit from Poly at any time type ctrl-D (end-of-text) or ctrl-C (interrupt). There are three types of instructions which can be typed to Poly; declarations of identifiers, statements (commands), or expressions. An example of a command and the output it produces is

```
> print("Hello");
Hello
```

Note the closing semicolon which must be present to indicate the end of the command. If you forget it the compiler will print a # as a prompt to indicate that the command is not yet complete.

An example of an expression is

```
> "Hi";
Hi
```

Poly prints the value of an expression without the need to type the word 'print'.

Commands can be grouped by enclosing them with the bracketting symbols **begin** and **end** or (and). For instance

```
> begin
#   print("Hello");
#   print(" again")
# end;
Hello again
```

Any object in Poly can be bound to an identifier by writing a declaration. For instance

```
> let message == "Hello ";
```

declares an identifier 'message' to have the value of the string 'Hello '. It can be printed in the same way as the string constant.

```
> message;
Hello
```

Names can be either a sequence of letters and digits starting with a letter, or a sequence of the special characters + - * = < > etc. Certain names are reserved to have special meanings and cannot be used in declarations. Those words can be written in upper, lower or mixed case, all other words are considered to be different if written in different cases. When declaring a name made up of the special characters remember to put a

space between the name and the == or colon which follows it. Comments are enclosed in curly brackets { and }. They are ignored by the compiler and are equivalent to a single space or newline between words.

2. Procedures

Statements or groups of statements can be declared by making them into procedures.

```
> let printmessage ==
#    proc()
#       (print("A message "));
```

A procedure consists of a procedure header (in this case the word **proc** and parentheses (and)) and a body. The procedure body must be enclosed in bracketting symbols (in this case ('and ')') even if there is only one statement.

This is simply another example of a declaration. Just as previously 'message' was declared to have the value "Hello ", 'printmessage' has been declared with the value of the procedure.

The procedure is called by typing the procedure name followed by ().

```
> printmessage().
A message
```

The effect of this is execute the body of the procedure and so print the string.

Procedures can take arguments so that values can be passed to them when they are called.

```
> let pmessage ==
#    proc(m : string)
#       begin
#       print("The message is :");
#       print(m)
#       end;
```

This can be called by typing

```
> pmessage("Hello");
The message is :Hello
```

or by typing

```
> pmessage("Goodbye");
The message is :Goodbye
```

3. Specifications

As well as having a value all objects in Poly have a specification, analogous to a type in other languages. It is used by the compiler to ensure that only meaningful statements will be accepted. You can find the specification of a declared name x by typing ? "x";.

```
> ? "message";
message : string
```

This means that message is a constant belonging to the type 'string'.

```
> ? "pmessage";
pmessage : PROC(string)
```

This means that pmessage is a procedure taking a value of type string as its argument. Since message has that specification the call

```
> pmessage(message);
The message is :Hello
```

will work. Likewise the call

```
> pmessage("Hi");
The message is :Hi
```

will work because "Hi" also belongs to type string. However

```
> pmessage(pmessage);
Error - specifications have different forms
```

will fail because 'pmessage' has the wrong specification. Incidently, the specification of the procedure is the same as the header used when it was declared, ignoring the differences in the case of some of the words.

4. Integer and Boolean

So far the only constants used have been those belonging to the type string. Another type, **integer** provides operations on integral numbers.

```
> print(42);
42
```

The usual arithmetic operations +, -, *, div, mod, succ and pred are available.

```
> 42+10-2;
50
```

However, unlike other languages all infix operators have the same precedence so

```
> 4+3*2;
14
```

prints 14 rather than 10. Also - is an infix operator only, there is a procedure neg which complements its argument.

Another 'standard' type is **boolean** which has only two values true and false. Its main use is in tests for equality (the = operator), inequality

(<>) and magnitude (> < >= <=).

```
> let two == 2;
> 1 = two;
false
> 2 = two;
true
> 3 <> 4;
true
> 4 >= 5;
false
```

The expression '1 = two' has type boolean. Identifiers can be declared to have boolean values in the same way as integers and strings.

```
> let testtwo == two > 1;
```

declares testtwo to be 'true' since 'two' is greater than 1. There are three operators which work on boolean values, &, | and ~. ~ is a prefix operator which complements its argument (i.e. if its argument was false the result is true, and vice-versa). & is an infix operator which returns true only if both its arguments are true. | is also an infix operator which returns true if either of its arguments is true.

5. If-Statement

Boolean values are particularly useful since they can be tested using **if**. The if-statement causes different statements to be obeyed depending on a condition.

```
> if two = 2
# then print("It is two")
# else print("It isn't two");
It is two
```

tests the value of the expression 'two = 2' and executes the statement after the word **then** if it is true, and the statement after the word **else** if it is false. This could be written as a procedure,

```
> let iszero ==
#   proc(i: integer)
#       (if i = 0 then print("It is zero")
#       else print("It isn't zero"));
```

which could then be called to test a value.

```
> iszero(4);
It isn't zero
```

since 4 is not zero. If-statements can return values as well as perform actions in the then and else parts. An alternative way of writing 'iszero' could have been

```
> let iszero ==
# proc(i: integer)
#    (print(
#      if i = 0
#      then "It is zero"
#      else "It isn't zero"
#        ));
```

This version tests the condition, and returns one or other of the strings for printing. This can only be used if both the then and else parts return values with similar specifications (in this case both sides return string constants). The version of the if-statement which does not return a value can be written with only a then-part. If the then-part returns a value there must be an else-part (otherwise what value would be returned if the condition were false?).

6. More on Procedures

Procedures can be written which return results. For instance a further way of writing 'iszero' would be to allow it to return the value of the string.

```
> let iszero ==
# proc(i: integer)string
#   (if i = 0 then "It is zero"
#    else "It isn't zero");
> ? "iszero";
iszero : PROC(integer)string
```

Calling it would then cause it to return the appropriate string which would then be printed.

```
> iszero(0);
It is zero
```

Another example is a procedure which returns the square of its argument.

```
> let sqr ==
#   proc(i: integer)integer (i*i);
```

declares sqr to be a procedure which takes an argument with type integer and returns a result with type integer. The body of the procedure evaluates the square of the argument i, and the result is the value of the expression. The call

```
> sqr(4);
16
```

will therefore print out the value 16.

Procedures in Poly can be written which call themselves, i.e. recursive procedures. These are declared using letrec rather than let.

```
> letrec fact ==
#   proc(i: integer)integer
#     (if i = 1 then 1
#      else i*fact(i-1));
```


This is the recursive definition of the factorial function. The procedure can be called by using

```
> fact(5);
120
```

which prints the result. **letrec** has the effect of making the name being declared available in the expression following the ==, whereas **let** does not declare it until after the closing semicolon.

7. Variables

Constants are objects whose value cannot be changed. There are also objects whose value can change, these are variables. Variables are created by declarations such as

```
> let v == new(0);
```

The procedure 'new' returns a variable whose initial value is the argument.

```
> v;
0
```

A new value can be given to v by using the assignment operator.

```
> v := 3;
> v;
3
```

Thus v now has the value 3. The new value can depend on the old value.

```
> v := (v+2);
```

Sets the value to be 5. The parentheses are necessary because otherwise the order of evaluation would be strictly left-to-right. Variables can be of any type.

```
> let sv == new("A string");
```

declares sv to be a string variable. The specification of a variable is not as simple as it may seem and will be dealt with later.

8. The While Loop

It is often necessary to repeat some statements more than once. This can be done using the **while** statement. For instance

```
> let x == new(10);
> while x <> 0
# do
#   begin
#   print(x*x);
#   print(" ");
#   x := pred(x)
#   end;
100 81 64 49 25 16 9 4 1
```

prints the square of all the numbers from 10 down to 1. The body of the loop (the statement after the word **do**) is executed repeatedly while the condition (the expression after the word while) is true. The condition is tested before the loop is entered, so

```
> while false
# do print("Looping");
```

will not print anything.

9. Operators

We have already seen examples of operators such as + and &. In Poly operators are just procedures whose specifications include the words infix or prefix. They are declared in a similar way to procedures, for instance

```
> let sq == proc prefix (i : integer)integer (i*i);
```

has declared sq as a prefix operator. It can be used like any other prefix operator:

```
> sq 3;
9
```

The difference between a prefix operator and other procedures is that the argument to a prefix operator does not need to be in parentheses. Infix operators can be defined similarly.

10. The Specifications of Types

All objects in Poly have specifications. This includes types such as string, integer and boolean.

```
> ? "boolean";
boolean : TYPE (boolean)
   & : PROC INFIX (boolean; boolean)boolean;
   false : boolean;
   print : PROC (boolean);
   true : boolean;
   | : PROC INFIX (boolean; boolean)boolean;
   ~ : PROC PREFIX (boolean)boolean
END
```

Types in Poly are regarded as sets of "attributes". These attributes are usually procedures or constants but could be other types. The attributes of a type can be used exactly like ordinary objects with the same specification. However, since different types may have attributes with the same name, it is necessary to prefix the name of the attribute with the name of the type separated by $.

```
> integer$print(5);
5
```

This invokes the attribute 'print' belonging to integer and prints the number. Most types have a print attribute which prints a value of that type in an appropriate format. $ acts a selector which finds the attribute belonging to a particular type. It is not an operator so operators always

work on the selected name rather than the type name.

```
> ~ boolean$true;
false
```

11. Records

Poly allows new types to be created in the same way as new procedures, constants or variables. One way of creating a new type is by making a record. A record is a group of similar or dissimilar objects.

```
> let rec == record(a, b: integer);
```

This declares 'rec' to be a record with two components, a and b, both of type integer.

```
> ? "rec";
rec : TYPE (rec)
  a : PROC(rec)integer;
  b : PROC(rec)integer;
  constr : PROC(integer;integer)rec
END
```

'constr' is a procedure which makes a record by taking two integers, and 'a' and 'b' are procedures which return the 'a' and 'b' values of the record.

```
> let recv == rec$constr(3, 4);
```

creates a new record with 3 in the first field (a) and 4 in the second field (b). The result is given the name 'recv'.

```
> rec$a(recv);
3
> rec$b(recv);
4
```

show that the values of the individual fields can be found by using 'a' and 'b' as procedures. They must of course be prefixed by 'rec$' to show the type they belong to.

Records can be made with fields of any specification, not just constants.

```
> let arec ==
# record(x:integer; p: proc(integer)integer);
```

declares a record with fields x and p, x being an integer constant and p a procedure.

```
> let apply ==
#   proc(z : arec)integer
#      begin
#      let pp == arec$p(z);
#      pp(arec$x(z))
#      end;
```

is a procedure which takes a constant of this record type and applies the procedure p to the value x and returns the result. In fact, it is not necessary to declare pp in the body of the procedure. An alternative way of writing apply is

```
> let apply ==
#   proc(z : arec)integer
#      (arec$p(z)(arec$x(z)));
```

12. Unions

Another way of constructing a type is using a 'union'. A union is a type whose values can be constructed from the values of several other types. For instance a value of a union of integer and string could be either an integer or a string.

```
> let un == union(int: integer; str: string);
```

This has created a type which is the union of integer and string. A value of the union type can be constructed by using an injection function. This union type has two such functions, their names made by appending 'int' and 'str' onto the letters 'inj_', making 'inj_int' and 'inj_str'. ('int' and 'str' were the 'tags' given in the declaration, in a similar way to fields in a record).

```
> let intunion == un$inj_int(3);
```

This has created a value with type 'un' containing the integer value 3.

```
> let stringunion == un$inj_str("The string");
```

creates a value, also with type 'un', but this time containing a string. Given a value of a union type it is often useful to be able to decide which of its constituent types it was made from. For each of the 'tags' there is a procedure whose name is made by prefixing with the letters 'is_', which returns 'true' or 'false' depending on whether its argument was made from the corresponding injection function.

```
> un$is_int(intunion);
true
```

prints 'true' because intunion was made from 'inj_int'. However

```
> un$is_str(intunion);
false
```

Values of the original types can be obtained by using 'projection' functions, which are the reverse of the 'injection' functions. Their names are made by prefixing the tags with 'proj_' to make names like 'proj_str' and 'proj_int'.

```
> un$proj_int(intunion);
3
> un$proj_str(stringunion);
The string
```

print the original values. It is possible to write

```
> un$proj_str(intunion);
Exception projecte raised
```

because 'intunion' has type 'un', just like 'stringunion'. However, 'proj_str'

is expected to return a value with type string so when this is run it will cause an error. The effect will be to raise an 'exception' called 'projecterror' which means that a projection procedure was given an argument constructed using a different injection procedure.

```
> let unprojstr == un$proj_str;
> ? "unprojstr";
unprojstr : PROC(un)string RAISES projecterror
```

shows that 'proj_str' may raise 'projecterror'. Exceptions will be dealt with in more detail later on.

13. The Type-Constructor

It is often useful to be able to construct a type which is similar to an existing one but with additional attributes. This can be done by using the type-constructor.

```
> let nrec ==
#   type (r) extends rec;
#   let print ==
#      proc(v : r)
#         begin
#         print(r$a(v));
#         print(",");
#         print(r$b(v))
#         end
#   end;
> ? "nrec";
nrec : TYPE (nrec)
   a : PROC (nrec)integer;
   b : PROC (nrec)integer;
   constr : PROC (integer; integer)nrec;
   print : PROC (nrec)
   END
```

This declares 'nrec' to be a new type which is an 'extension' of an existing type 'rec'. It then lists the new attributes, in this case just the procedure 'print', which are declared just as though they were ordinary declarations. The name 'r' in parentheses which follows the word 'type' is the name for the new type within the body of the type constructor, so the argument of the procedure 'print' is given the type 'r'. It is important to remember that the new type is a completely separate type from 'rec'. Values can be changed from the old to the new type and vice versa, but they cannot be used interchangeably. The specification of nrec is similar to that of rec except that there is now an extra procedure 'print'.

```
> let nrecv == nrec$constr(5,6);
> nrec$print(nrecv);
5,6
```

makes a value with type nrec, and prints it using the new 'print' attribute. It is possible to write simply

```
> print(nrecv);
5,6
```

because there is a procedure 'print' which looks for the 'print' attribute of the type of the value given, and then calls it. This is the way integers and strings are printed (they both have 'print' attributes). Many of the other operations such as ':=' and '+' work in a similar way. A further alternative is to write an expression.

```
> nrecv;
5.6
```

In this case the compiler looks for the 'print' attribute and applies it.

14. A Further Example

This record could be extended in a different way, to make a double-precision integer. Suppose that the maximum range of numbers which could be held in a single integer was from -9999 to 9999. Then a double-precision number could be defined by representing it as a record with two fields, a high and low order part, and the actual number would have value (high)*10000 + (low). This can be implemented as follows.

```
> let dp ==
#   type (d) extends record(hi, lo: integer);
#   let succ ==
#      proc(x:d)d
#        begin
#        if d$lo(x) = 9999
#        then d$constr(succ(d$hi(x)), 0)
#        else if (d$hi(x) < 0) & (d$lo(x) = 0)
#        then d$constr(succ(d$hi(x)), neg(9999))
#        else d$constr(d$hi(x), succ(d$lo(x)))
#        end;
#   let pred ==
#      proc(x:d)d
#        begin
#        if d$lo(x) = neg(9999)
#        then d$constr(pred(d$hi(x)), 0)
#        else if (d$hi(x) > 0) & (d$lo(x) = 0)
#        then d$constr(pred(d$hi(x)), 9999)
#        else d$constr(d$hi(x), pred(d$lo(x)))
#        end;
```



```
#   let print ==
#      proc(x:d)
#        begin
#        if d$hi(x) <> 0
#        then
#           begin
#           print(d$hi(x));
#           if abs(d$lo(x)) < 10
#           then print("000")
#           else if abs(d$lo(x)) < 100
#           then print("00")
#           else if abs(d$lo(x)) < 1000
#           then print("0");
#           print(abs(d$lo(x)))
#           end
#        else print(d$lo(x))
#        end;
#   let zero == d$constr(0,0);
#   let iszero ==
#      proc(x:d) boolean
#        ((d$hi(x) = 0) & (d$lo(x) = 0))
#   end;
```

This is sufficient to provide the basis of all the arithmetic operations, since +,-,* etc. can all be defined in terms of succ, pred, zero and iszero.

15. Exceptions

In the section on union types above mention was made of exceptions. In the case of the projection operations of a union type an exception is raised when attempting to project a union value onto a type which was not the one used in the injection. An exception is simply a name and any exception can be raised by writing 'raise' followed by the name of the exception.

```
> raise somefault;
Exception somefault raised
```

raises an exception called 'somefault'.

```
> let procraises
#   == proc(b: boolean)
#         (if b then raise afault);
```

has specification

```
PROC(b: boolean) RAISES afault
```

Various operations, as well as projection, may raise exceptions. For instance many of the attributes of integer, such as 'succ' raise the exception 'rangeerror' if the result of the operation is outside the range which can be held in an integer constant. 'div' will raise 'divideerror' if it is asked to divide something by 0.

As well as being raised exceptions can also be caught, which allows a program to recover from an error. A group of statements enclosed in brackets or 'begin' and 'end' can have a 'catch phrase' as the last item. A catch phrase is the word catch followed by a procedure. e.g. 'catch p' will catch any exception raised in the group of statements and apply p to its name.

```
>let procoatches ==
#   proc(excp: string) (print(excp));
> begin
# procraises(true);
# catch procoatches
# end;
afault
```

'procoatches' has been declared as a procedure which takes a argument of type string. The exception is raised by 'procraises' and, since it is not caught in that procedure it propagates back to the point at which 'procraises' was called. The catch phrase catches the exception and calls the procedure with the name of the exception as the argument. The catching procedure can then look at the argument and decide what to do.

```
> begin
# procraises(false);
# catch procoatches
# end;
```

does not print anything because an exception has not been raised and so the procedure is not called.

If the block containing the catch phrase returns a value, then the catching procedure must return a similar value.

```
> let infinity == 99999;
> let divi ==
# proc infix(a, b: integer)integer
#    begin
#    a div b
#    catch proc(string)integer (infinity)
#    end;
```

This declares 'divi' to be similar to 'div' except that instead of raising an exception it returns a large number. Since 'a div b' returns an integer value the catch phrase must also return an integer.

16. The Specification of Variables

The specification of a variable in Poly is not, as one might expect, a constant of some reference type or a separate kind of specification, but each variable is in fact a separate type. Since a type in Poly is simply a set of constants, procedures or other types, a type can be used simply as a way of conveniently grouping together objects.

```
> let intpair ==
# type
# let first == 1;
# let second == 2
# end;
```

This has declared 'intpair' to be a pair of integers containing the values

1 and 2. 'intpair$first' and 'intpair$second' can be used as integer values directly.

The specification of an integer variable is

```
TYPE
assign: PROC(integer);
content: PROC()integer
END
```

A variable is a pair of procedures, 'assign' which stores a new value in the variable, and 'content' which extracts the current value from it. The standard assignment operator ':=' simply calls 'assign' on the variable. The compiler inserts a call to 'content' automatically when a variable is used when a constant is expected. 'assign' and 'content' can both be called explicitly.

```
> let vx == new(5);
> vx$assign(vx$content() + 1);
> vx$content();
6
```

As an example of a more complicated variable, suppose we wanted to write a subrange variable, similar to a subrange in Pascal, which could hold values between 0 and 10.

```
> let sr ==
#   begin
#   let varbl == new(0);
#     type
#     let content == varbl$content;
#     let assign ==
#        proc(i: integer)
#          (if (i < 0) | (i > 10)
#           then raise rangeerror
#           else varbl$assign(i))
#     end
#   end;
```

'varbl' is an integer variable which is initially set to 0. 'assign' checks the value before assigning it to 'varbl', and raises an exception if it is out of range. 'content' is just the 'content' procedure of the variable. It can be used in a similar way to a simple variable.

```
> sr := 2;
> sr;
2
> sr := 20;
Exception rangeerror raised
> sr;
2
```

17. Specifications in Declarations

The double-precision type declared above has one drawback. The specification contains the 'hi', 'lo' and 'constr' attributes in the specification of the type which would allow someone to construct a value which had the type 'dp', but had, for instance, fields outside the range -9999 to 9999 or with different signs. This could make some of the operations fail to work. We need a way of hiding details of the internals of a type declaration so that they do not appear in the specification, and so cannot be used outside. In Poly a specification can be given to something explicitly as well as having it inferred from the declaration.

```
> let aconst: integer == 2;
```

declares 'aconst' and forces it to have type 'integer'. The specification is written in the same way as the specification of the argument of a procedure.

```
> let quote : proc(string)
#     == proc(x: string)
#           begin
#           print("`");
#           print(x);
#           print("'")
#           end;
```

is another example of explicitly giving a specification to a value. An explicitly written specification is the specification of the name which is being declared. It need not be identical to the specification of the value following the '=='. However it must be possible to convert the specification of the value to the explicit specification (the 'context').

```
> let avar == new(3);
> let bconst: integer == avar;
```

declares 'avar' to be an integer variable and 'bconst' to be an integer constant. In the latter case the specification is necessary, otherwise 'bconst' would have been a variable and would have been another name for 'avar'. The conversion of a variable to a constant in order to match a given specification is one example of a 'coercion' of a value to match a 'context'. There are several others which can be applied depending on the particular specification. For instance the specification of a procedure may be changed from an operator to a simple procedure or vice versa.

```
> let plus:
#   proc(integer;integer)integer raises rangeerror
#   == integer$+ ;
```

declares 'plus' as a procedure which is the same as the '+' attribute of integer except that it is not an infix operator.

```
> plus(3,4);
7
```

The list of exceptions raised by the procedure must be included in the specification. The exception list in the specification given must include all the exceptions which may be raised, but may include others as well. A

special exception name any can be used to indicate that a procedure can raise any exception. Any exception list will match a context with exception list 'raises any'.
The specifications of the arguments and result must all match.

```
> let dble:
#    type (d)
#    succ, pred: proc(d)d raises rangeerror;
#    print: proc(d) raises rangeerror;
#    zero: d;
#    iszero: proc(d)boolean;
#    end
#    == dp;
```

creates a new type 'dble' with the specification given. The specification is the same as that of 'dp' but with some of the attributes of dp missing.

In the case of types the specification of the value must possess all the attributes of the explicit specification, but the explicit specification need not include all the attributes of the value. If a type is regarded as a set of named attributes then it is possible to take a subset of them and make them into a new type, simply by giving the new type the required specification. The specification of each attribute must itself match the specification that is given for it.

This mechanism provides a way of 'hiding' internal operations from the specification of a type. The specification of 'dble' above has only those attributes which are necessary to use it, and none of the operations which are used internally.

18. Types as Results of Procedures

So far we have considered procedures which take constants as arguments or return constants as results. In Poly values of any specification can be passed to or returned from a procedure. For instance

```
> let subrange
#   == proc(min, max, initial: integer)
#         type (s)
#         content: proc()integer;
#         assign: proc(integer) raises outofrange
#         end
#      begin
#         type
#         let varbl == new(initial);
#         let content == varbl$content;
#         let assign ==
#            proc(i: integer)
#              (if (i < min) | (i > max)
#               then raise outofrange
#               else varbl$assign(i))
#         end
#      end;
```

This procedure is similar to the definition of the subrange type 'sr' previously. However the bounds of the type are now arguments of a procedure so their values can be supplied when the program is run. Also new subrange variables can be created by calling the procedure.

```
> let sv == subrange(0,10,0);
```

This creates 'sv' as a variable of this subrange type. As with any procedure the arguments can be arbitrary expressions provided they return results with the correct specification.

19. Types as Arguments to Procedures

Types can be passed as arguments as well as being returned from procedures.

```
> let copy ==
#   proc(atype: type end)
#      type (t)
#      into: proc(atype)t;
#      outof: proc(t)atype
#      end
#   begin
#      type (t) extends atype;
#      let into == t$up
#      let outof == t$down
#      end
#   end;
```

This procedure takes a type and returns a type with two operations 'into' and 'outof'. 'up' and 'down' are procedures which are created when 'extends' is used, and provide a way of converting between the original and the resulting types. The specification of 'atype' merely says that it must be passed a type as an argument, but since it does not list any attributes then any type can be used as an actual argument (this is effectively saying that the empty set is a subset of every set). The procedure can be called, giving it an actual type as argument.

```
> let copyint == copy(integer);
```

The specification of the result is

```
   TYPE (copyint)
   into: PROC(integer)copyint;
   outof: PROC(copyint)integer
   END;
```

The specification of copyint allows mapping between integer and copyint since the type integer has been included in the specification.

```
> let copy5 == copyint$into(5);
> copyint$outof(copy5);
5
```

has mapped the integer constant 5 into and out of 'copyint'.

```
> let copychar == copy(char);
```

creates a similar type which maps between char and copychar.

20. Polymorphic Procedures

There are often cases where, in addition to passing a type as a argument, one or more values of that type are passed as well. For instance a procedure to find the second successor of a value might be written as

```
> let add2 ==
#   proc(atype:
#         type (t)
#         succ: proc(t)t raises rangeerror
#         end;
#       val: atype)
#     (atype$succ(atype$succ(val)));
```

The specification of 'val' is that it must be a constant, and its type is 'atype'. However 'atype' is also an argument to the procedure so the specification really means that this procedure could be called by giving it any type with the required attributes, and a constant which must be of the same type as the first argument.

```
> add2(integer, 2);
4
```

Similarly

```
> add2(char, 'A');
C
```

However

```
> add2(integer, 'A');
```

and

```
> add2(string, "A string");
```

both fail, in the first case because 'A' is not integer, and in the second because string does not have a successor function.

21. Implied Arguments

Many types have a 'print' attribute which prints a constant of the type.

```
> let pri ==
#   proc(printable: type (t) print(t) end; val: printable)
#      (printable$print(val));
```

declares 'pri' as a procedure which takes as arguments a type and a constant of that type and prints the constant using the 'print' attribute. This can be called by writing

```
> pri(integer, 3);
or
> pri(char, 'a');
```

since both 'integer' and 'char' have a 'print' attribute. Having to pass the type explicitly is really unneccessary, since it is possible for the system to find the type from the specification of the constant. It would be possible for the system to convert 'pri(3)' into 'pri(integer,3)' since '3' has type integer. In Poly types which can be deduced from the specifications of other arguments can be declared as 'implied' arguments. A argument list written in square brackets, [and], can preceed the normal argument list and those parameters, which must be all be types, are inferred from the other actual arguments when the procedure is called.

```
> let prin ==
#   proc [printable: type (t) print: proc(t) end]
#         (val: printable)
#       (printable$print(val));
```

This can now be called by writing

```
> prin(3);
or
> prin("hello");
```

and is in fact the definition of 'print' in the standard library. Alternatively 'prin' could have been declared by giving it an explicit specification and using 'pri'.

```
> let prin: proc[printable: type (t) print: proc(t) end]
#                 (printable)
#       == pri;
```

This is another form of conversion which can be made using an explicit specification. Using implied parameters can simplify considerably the use of procedures with types as arguments, and allow infix or prefix operators to be used in cases where they could not otherwise be used. For instance, consider an addition operation defined as

```
> let add ==
#   proc(summ: type (s) + : proc infix (s;s) raises rangeerror
#                end;
#          i, j: summ)summ
#           (i + j);
```

would be used by writing

```
> add(integer, 1, 2);
3
```

However, by writing

```
> let +
# : proc infix [summ: type(s)
#                       + : proc infix (s;s)raises rangeerror
#                        end]
#              (i, j: summ)summ raises rangeerror
#    == add;
```

'+' can become an infix operator, since it has only two actual arguments. Similar definitions are used for many of the other declarations in the library.

22. Literals

We have already seen how constants can be written as "Hello" or 42. These are known as literal constants, because their values are given by the characters which form them, rather than by some previous declaration. They are however, only sequences of characters, it is only by convention that "Hello" is a string constant and 42 an integer constant. This is only important when we wish to use some other definition than the 'standard' one. For instance, if the type integer were restricted to the range -9999 to 9999 then the constant 100000 would be an error if it were treated as an integer. The definition of double-precision integer above, would, however, be able to represent it.

In Poly, therefore, literals have no intrinsic type, they must be converted into a value by the use of a conversion routine. The compiler recognises certain sequences of characters as literals rather than names or special symbols. The three forms of literal constants recognised by the compiler are 'numbers', 'double-quoted sequences' and 'single-quoted sequences'. 'Numbers' begin with a digit and may consist of numbers or letters.

42 0H3F6A 3p14159

are examples of 'numbers'. 'Double-quoted sequences' are sequences of characters contained in double-quotes. A double-quote character inside the sequence must be written twice.

"Hello"   ""   "He said ""Hello"""

'Single-quoted sequences' are similar to double-quoted sequences but single rather than double-quotes are used.

'Hello'   ''   'He said ''Hello'''

When the compiler recognises one of these literals it tries to construct a call to a conversion routine which can interpret it as a value of some type. For instance, the standard library contains a definition of 'convertn' which the compiler calls if it finds a 'number'. That definition has specification

PROC(string)integer

All conversion routines must have similar specifications, but the result type will differ and some exceptions may be raised. The literal is supplied as a constant of type 'string'. The conversion routine can examine the characters which form the literal and return the appropriate value. It may of course raise an exception if the characters do not form a valid value, if either the value would be out of range or if the literal contains illegal characters.

There are also two other conversion routines in the standard library, 'converts' which converts double-quoted sequences into string values, and 'converto' which converts single-quoted sequences into values of the type 'char'. These definitions can be overridden by preceeding the literal by the name of a type and a $ sign. For instance

```
> let int == integer;
> let one == int$1;
```

applies the 'convertn' routine belonging to 'int', so that 'one' has type int rather than integer.

23. Lists

Lists are a convenient example for polymorphic operations. List types can be constructed by the following procedure.

```
> let list ==
# proc(base: type end)
#    type (list)
#    car : proc(list)base raises nil_list;
#    cdr : proc(list)list raises nil_list;
#    cons: proc(base; list)list;
#    nil : list;
#    null: proc(list)boolean
#    end
#  begin
#     type (list)
#     let node == record(cr: base; cd: list);
#     extends union(nl : void; nnl : node);
#
#     let cons ==
#       proc(bb: base; ll: list)list
#         (list$inj_nnl(node$constr(bb, ll)));
#
#     let car ==
#       proc(ll: list)base
#         begin
#         node$cr(list$proj_nnl(ll))
#         catch proc(string)base (raise nil_list)
#         end;
#
#     let cdr ==
#       proc(ll: list)list
#         begin
#         node$cd(list$proj_nnl(ll))
#         catch proc(string)list (raise nil_list)
#         end;
#
#     let nil == list$inj_nl(void$empty);
#
#     let null == list$is_nl
#     end
#   end;
```

'void' is a standard type which has only one value (empty), and is used to represent the 'nil' value of the list. The list structure is made using a recursive union with each node containing a value of the 'base' type and the next item of the list, or containing a nil value. 'cons' makes a new node of the list, 'car' and 'cdr' find the 'base' and 'list' parts of a node respectively, and 'null' tests for the value 'nil'. 'car' and 'cdr' both trap the exeception which would be raised if a projection error occured and

raise 'nil_value' in its place.

A particular list type can now be created, for instance a list of integers.

```
> let ilist == list(integer);
> let il == ilist$cons(1, ilist$cons(2, ilist$cons(3, ilist$nil)));
```

A polymorphic 'cons' function could be declared to work on lists of any base type.

```
> let cons ==
#    proc[base: type end;
#         list: type (l) cons: proc(base; l)l end]
#        (bb: base; ll: list)list
#           (list$cons(bb, ll));
```

It is now possible to write simply

```
> let il == cons(1, cons(2, cons(3, ilist$nil)));
```

Polymorphic 'car', 'cdr' and 'null' functions can be written similarly. As further examples some other polymorphic list functions are given.

```
> letrec append ==
# proc[base: type end;
#      list: type (l)
#              car: proc(l)base raises nil_list;
#              cdr: proc(l)l raises nil_list;
#              cons: proc(base; l)l;
#              null: proc(l)boolean end]
#     (first, second: list)list
#     ( if null(first) then second
#       else cons(car(first), append(cdr(first), second)) );
> letrec reverse ==
# proc[base: type end;
#      list: type (l)
#              car: proc(l)base raises nil_list;
#              cdr: proc(l)l raises nil_list;
#              cons: proc(base; l)l;
#              nil: l;
#              null: proc(l)boolean end]
#     (ll: list)list
#     ( if null(ll) then list$nil
#       else append(reverse(cdr(ll)), cons(car(ll), list$nil)) );
```

A useful function would be one which would print the data part of a list if the base type could be printed.

```
> letrec pr ==
# proc [base: type(b) print: proc(b) end;
#       list: type(l) car: proc(l)base raises nil_list;
#                     cdr: proc(l)l raises nil_list;
#                     null: proc(l)boolean
#             end ]
#      (ll: list)
#      begin
#      if null(ll)
#      then print("nil")
#      else
#         begin
#         print("( ");
#         print(list$car(ll));
#         print(". ");
#         pr(list$cdr(ll));
#         print(") ")
#         end
#      catch proc(string) ()
#      end;
```

The list created above can now be printed.

```
> pr(il);
( 1. ( 2. ( 3. nil) ) )
```

Other polymorphic functions on lists can be declared in a similar way.

24. Conclusion

This document is intended as an introduction to Poly and to give some idea of the ways in which it can be used. It is not a rigorous description and various details, such as the precise checking rules for specifications, have been deliberately skated over in order to explain the language simply. A companion document, the Poly Report, is the reference for the precise details of the language.

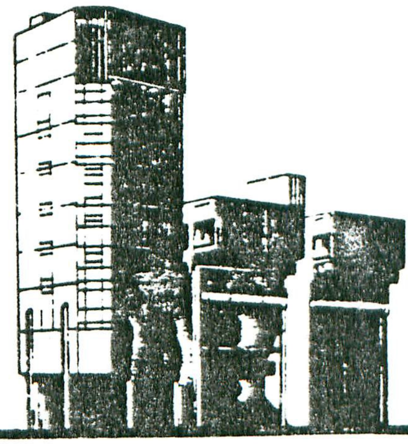
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 31

PONDER AND ITS TYPE SYSTEM

by

J. Fairbairn



University of Cambridge
Computer Laboratory

Ponder and its Type System

J. Fairbairn

November 1982

Ponder and its Type System

Abstract

This note describes the programming language "Ponder", which is designed according to the principles of referencial transparency and "orthogonality" as in [vWijngaarden 75]. Ponder is designed to be simple, being functional with normal order semantics. It is intended for writing large programmes, and to be easily tailored to a particular application. It has a simple but powerful polymorphic type system.

The main objective of this note is to describe the type system of Ponder. As with the whole of the language design, the smallest possible number of primitives is built in to the type system. Hence for example, unions and pairs are not built in, but can be constructed from other primitives.

different routine must be written for each type of array that is to be sorted. For example, in the notation of Algol68, one might need both a

PROC ([] REAL, PROC (REAL, REAL) BOOL) [] REAL

and a

PROC ([] STRING, PROC (STRING, STRING) BOOL) [] STRING.

This leads to the idea that the type of an object should reflect any polymorphism that is inherent in it. Thus the type of 'sort' should indicate that its first argument is an array of objects of any type, its second is a function taking two objects of that type, and that it returns an array of that type. One might extend the type system of Algol68 to include some kind of type parameters, so that the type of 'sort' might be:

PROC (AMODE M, [] M, PROC (M, M) BOOL) [] M,

the first argument being the type that is to be used throughout.

Type parameters of this nature occur in some other type systems, for example that of Russell [Demers 79], but are accompanied by some philosophical objections: the appearance of a type as a variable suggests that types should be "first class objects", with various operations to split them apart, and to make new ones. If this is the case, then surely types should also be given some sort of type (Indeed in Russell, types do have 'signatures'), but then one can ask why can one not pass the type of the type as a parameter, and so on? Furthermore, the ability of a programme to manipulate its data types in an unpredictable way means that some type checking would be necessary at run time. This is due to the problem that if types may be computed in a perfectly general manner, then the compiler would have to be able to compare functions to see if they always give the same result (which is not generally possible). In Russell types are restricted so that computation of this nature is not done at all, which gives rise to unpleasant effects, such as **'int_array (1+3)'** being a different type from **'int_array (3+1)'**.

One alternative would be to abandon the idea of totally static (compile time) type checking and rely on some dynamic (run time) checking of types. I feel that this is undesirable, since the idea of a type can be either extended to encompass all

programming errors (at one extreme), or restricted to check none (at the other). The first is too general to be really useful, and corresponds for example, to including the fact that divide may not be used with a second argument of zero in its type, and obviously cannot be totally checked before running the programme. The second extreme corresponds to no type checking at all (and equally obviously may be totally checked even before writing the programme!). Thus it is useful to restrict the idea of type errors to that which may be checked for at before running the programme.

My approach has more in common with that in ML, in that type parameters do not really occur. Taking the example of 'sort' again, what one wants to say is that, for all types **'M'**, **'sort'** has type

PROC ([] M, PROC (M, M) BOOL) [] M,

and have the compiler decide what M should be whenever the function is applied. The notation for types in Ponder also has more in common with ML than with Algol68, so that the type of **'sort'** might be written:

∀T. ARRAY [T] × (T × T → BOOL) → ARRAY [T]

2.3 Comparison with ML

The type system of ML answers many of the problems indicated above, but includes more predefined mechanisms than are necessary, and can fail to give a type for some useful functions. In ML a type may include free variables, represented as **'*'**, **'*1'**, **'*2'**, etc, so that the type of **'sort'** in ML is:

(* array) × (* × * → bool) → (* array)

where **'t1 × t2'** is the type of a pair whose left hand element is of type **'t1'**, and whose right hand element is of type **'t2'**, and **'t1 → t2'** is the type of a function requiring its argument to have type **'t1'** and which returns an object of type **'t2'**. My notation differs for type generators (see below for a description), in that for an array of booleans ML has **'(bool array)'** where Ponder has **'ARRAY [BOOL]'**, since I find the prefix notation more readable when there are several applications of generators.

Section 1: Introduction

Ponder is functional programming language with normal order semantics and a strong polymorphic type checking system. The language is not yet defined in full, and hence this document is open to suggestion, and may not describe the final version of the language.

Ponder is designed with a small set of primitives for declarations and the syntax includes very few "built in" constructs. Normal order semantics were chosen in order that new constructs may be introduced by the programmer, without the problem of unexpected evaluation of arguments, as when attempting to define 'IF', for example. That the language is functional also allows implementations to take advantage of new developments in processor design. Thus no concession is made to conventional machine architectures, since it is hoped that programmes may be efficiently run on (for example) combinator [Clarke 82] or graph reducing machines.

Section 2: Motivation

This section describes the motivation behind the language and its type system, and examines some aspects of other languages. The reader who is unfamiliar with other languages mentioned is asked to bear with me. Some of the notation which occurs is explained later.

Programming languages like Algol68, (and lately Ada), include too much that is built in to the language. This has the effect that they are difficult to remember in full, so that there is a tendency for programmers to only take advantage of a small subset of the language. Furthermore, the excessive size of the definition tends to make implementation difficult, and prone to modification by the implementor in order to make it "efficient".

2.1 Type Checking

Experience has shown that statically checked types in programming languages aid the production of correct programmes in several ways. The most important aspect is that more mistakes in the logic of the programme are detected at compile time, and hence do not need to be found by "debugging". This is particularly

important, since it is often very difficult to test a programme in such a way as to sufficiently 'exercise' all of its individual parts.

A further aspect is that the programmer may take advantage of type checking when making changes to the data structures used within a programme, since the structure of an object is reflected in its type. Thus any incorrect accesses to an object will be reported as type errors, so that any that are missed when the programmer makes a change will be reported by the compiler, rather than lurking to cause a disaster when some part of the programme runs.

2.2

Compile time type checking, then, both protects the programmer from itself*, and provides it with extra facilities which make the writing and maintainance of programmes easier.

Important constructors in data types are STRUCTures (as in Algol68, or records in Pascal), UNIONs (as in Algol68 or as in ML [Gordon 79]), arrays, lists, pairs, and so on, and some form of encapsulation (as in Abstract types in ML). One of the aims of the Ponder type checking system is to provide a mechanism for creating such constructors, and the minimum number of primitives to make them out of.

Unfortunately, the strong type checking as in Algol68 or Pascal is rather <u>too</u> strict, in that it tends to prevent one from doing things which do make sense, as well as things which do not. This tends to make one write the same thing more than once. A commonly quoted example of this is of a routine to sort an array. If one wishes to write a function which given an array and a function saying whether or not two elements are in order, returns a sorted version of the array, one can use the same algorithm regardless of the data type of the elements of the array. In languages such as Pascal or Algol68, this <u>polymorphism</u> (that is to say the ability of a function to work for many "forms" of argument) of the sorting function can not be exploited, and a

*Unfortunately this English usage tends to discriminate against humans. However, the discrimination is less than that against machines suggested by the use of 'him/her'.

expression_1 expression_2

which means apply '**expression_1**' to '**expression_2**'. The result of applying a lambda function to an argument is the body of the lambda function with all the ocurrences of its parameter replaced by the argument. Hence

(λx. on rainy x) days

evaluates to

on rainy days

The parameter '**x**' having been replaced with the value '**days**'. Note that parentheses are used purely for grouping, and that application associates to the left, so that

the monk ryokan

means the same as

(((the) monk) ryokan)

It is intended that a lambda function should always mean the same thing regardless of the name used for the bound variable, since '**λx. x**' clearly behaves the same as '**λy. y**'. Thus any lambda expression may be replaced by another one which is the same except for the names of bound variables, without change of meaning. When an expression contains nested lambda functions, variables bind to the nearest textually enclosing lambda, so that

λx. (λx.x)

means the same as

λx. (λz.z)

Note that if we allow the simplification of the insides of lambda functions, the process of substitution of a value for a parameter may cause a confusing state to arise. Consider:

λx. ((λy. (λx. x y)) x)

which in all respects behaves the same as

λx. (λz. z x)

In that

λx. ((λy. (λx.x y)) x) p

becomes

((λy. (λx.x y)) p)

and then

λx.x p

which is what you get from

λx. (λz. z x) p

(try it)

If we try to simplify the inside of

λx. ((λy. (λx. x y)) x)

We might try

λx. (λx. x x)

but the last '**x**' is clearly meant to be bound to the first one. In these situations the proper thing to do is to change the names so that we can see what is happening, so

λx. ((λy. (λx. x y)) x)

may first be changed to

λx. ((λy. (λz. z y)) x)

and from there we get the correct answer. One final example:

(λx. (λy. x y)) (λy. y y) beep

evaluates to

(λy. (λx. x x) y) beep

then to

(λx. x x) beep

and finally (since '**beep**' is unbound, and cannot be evaluated) to

ML is, however, unable to express the types of certain sensible functions. Consider the function:

```
let imposs = λselect. λi. λb.
        pair (select (i, 1)) (select (b, true))
```

which takes a selector function 'select' (which returns either its first or second argument), and returns a pair constructed of a selected integer, and a selected boolean. In ML parameters to functions are required to have the same type at every application within the function, and the type inference mechanism cannot detect that 'select' must be a polymorphic function, nor can ML types express this fact.

This also has the unpleasant effect that names bound by 'let' are not treated in the same way as names bound by application. Thus:

```
let select = λa. λb. a in
let fs = λi. λb.
      pair (select (i, 1)) (select (b, true))
```

is valid, although 'fs' is equivalent to '**imposs select**'.

As will be shown, the type system for Ponder can express a type for such functions, and Ponder preserves the equivalence of binding by declaration and by application. An advantage of this is that type constructors such as pairs and disjoint unions do not have to be built in to the type system, since they may be expressed in terms of simpler constructors. A minor objection to ML types is that the type variables are declared implicitly rather than explicitly. In Ponder all type variables must have explicit declarations, but it is the use of this which provides the extra expressive power.

There is also a slight disadvantage, however. In ML types are arranged so that there is always a most general type for a well typed expression, and so that the compiler can infer what that type is. In Ponder, the type system is somewhat richer, and expressions do not always have a most general type, so of course the compiler cannot be expected to find it. Thus in Ponder the programmer must always specify the types of the parameters of a function. I feel that this is no real loss, since it encourages the programmer to think more about the types of things, and probably improves style. It is also my belief that the programming language should ensure that as much of the

information about a function as possible may be discovered by reading a small amount of the definition of the function. Thus, although it may be difficult at first to get used to function definitions which appear to be cluttered with type information, it is worth while in the end. Programmers used to languages like Algol68 will find that Ponder requires rather less repetition of type information.

Section 3: Semantics

To define the semantics of a programming language in complete formal detail is quite a complicated process, and is beyond the scope of this description. It is, however, desirable that when different people read a Ponder programme they should come to the same conclusion about what it means. To this end, some form of universally understood model is needed (this is clearly impossible). Hence I will describe the semantics of Ponder as transformations into the lambda calculus (see [Hindley 72]).

3.1 Lambda Calculus

I will now give a brief description of λ-calculus, so readers who are already familiar with it should skip to the next heading. Lambda expressions are abstract representations of functions (in the sense of methods of computation). The simplest form of lambda expression is the <u>name</u> (Normally called a variable, but 'variable' has unfortunate connotations of 'variable storage' in computer circles). Here is a lambda expression, then:

days

It does not mean very much, since there is no value associated with '**days**'. Names which occur without defining ocurrences are said to be free. Names may be used as parameters to functions by means of 'λ', like this:

λparameter_name. body

Where '**body**' may be any form of lambda expression, and '**parameter_name**' is a name for the parameter of the function. It may help to think of '**λx. body**' as "That function **f**, such that **f(x) = body**". I will refer to these as lambda functions. The only other form of expression is the application:

$ % & + - * / = ~ | \ < > : . ? @

(It was decided to allow an arbitrary number of these characters to be made into symbols, because it is clear that only allowing one at a time would give too few, and because allowing two at a time seems rather arbitrary and disallows some nice combinations like <=> and =*=. However, it is obviously not a good idea to use long combinations, since it is not easy to read them. For example using both '*******' and '********' would reduce readability.)

Note however that the following symbols already have built in meanings, and may not be used in any other way:

( ) [ ] , ; : . ≡ → ∀

When the characters '≡', '→' and '∀' are not available, they will be represented by '==', '->' and '!' respectively, in which case, these combinations of characters may not be redefined.

The remaining class of symbol is the icon. Icons are symbols whose values and types are determined solely from the text of the symbol. There are currently three types of icon for Ponder. The simplest is the character icon, which is written as an apostrophe, followed by a representation for a particular character. Thus

'a

is a character icon, and stands for the letter a. To enable the representation of non-printing and other difficult characters, character representations include escaped characters, which are written as a further apostrophe followed by another character. Some standard escaped characters are:

'" for "
'' for '
's for space
't for tab

so that

'''

is the character icon for apostrophe.

To simplify the inputting of text, there are string icons. A string icon is represented as a quotation mark, followed by a series of character representations, and ending with a further quotation mark. Character representations in strings are the same as in character icons, with the addition that an apostrophe followed by layout is removed up to the next non layout character. This is to allow long strings to be printed on more than one line. Hence

"A string icon with a '"line'
break'" in it"

is the string **'A string icon with a "linebreak" in it'**. Note that all layout is removed, so that to insert a space at a line break it is necessary to leave the space before the apostrophe, as in:

"Another icon with a line '
break in it"

which means the same as **'"Another icon with a line break in it"'**.

The only other icon is for digit sequences, which are represented by combinations of digits, for instance:

42

4.2 Declarations

There are two ways of giving a value a name. The first is as the parameter of a function (which is obviously necessary), but the second is by means of the 'LET' declaration, for example

LET three ≡ 3;

Which declares the name **'three'** and binds it to the value 3, and to have the type **'INT'**, so that further ocurrences of the name 'three' mean exactly the same as 3. The general form is

LET name ≡ expression; ...

Note that the name is declared AFTER the expression, so that, for example

beep beep

Normally descriptions of lambda calculus go on to an extended notation including numbers and so on, but that is unnecessary here.

3.2 Evaluation Order

All objects in Ponder are intended to be capable of representation as lambda expressions. The only other question of semantics which needs answering is "In what order are things evaluated?" In Ponder, function applications are evaluated function first (this is called normal order) as opposed to arguments first, which is usually the case in more conventional languages (and which is called applicative order). To see what this means, consider

(λx. (λy. x)) nice nasty

If we were to evaluate the arguments first, we would have to work out **'nasty'** first, and if **'nasty'** does something nasty, we're in trouble. Normal order evaluation gives us first

(λy. nice) nasty

and then

nice

since **'nasty'** was bound to 'y' which did not appear in the result, which means that the value of **'nasty'** never even gets considered.

Normal order evaluation corresponds approximately to call by name in ALGOL 60, and has a reputation for being inefficient. However, the absence of side effects in functional programming makes it possible to use a scheme of lazy evaluation, in which arguments are evaluated at most once, the first time they are used.

This has the fortunate consequence that it is easy to represent infinite structures, since they are never evaluated in full.

Section 4: Syntax

The syntax of Ponder is intended to be both rich and simple (this is, of course, impossible). The appendix contains a formal description of the syntax of Ponder.

4.1 Symbols

Programmes in Ponder are represented by sequences of symbols. Symbols are divided into four classes, the first being object names. The name of an object should be written in italic letters and numerals, with optional (italic) hyphens to separate individual words within the name. The second class is bold names. These are used for things which are purely syntactic, like the names of data types, and keywords. Bold names are written in bold face roman letters and numerals, with optional (bold face roman) hyphens to separate words within a name.

Unfortunately, current terminals tend not to have italics, and do not preserve the distinctions between hyphens, dashes and minus signs. For this reason it is necessary to adopt the convention that bold names are written in upper case, and names in lower case, using underlines instead of hyphens. hence

feels_sorry_for_himself

is a name, and

ZEN_KOAN

is a bold name. Note that since it is often difficult to distinguish between one and two underlines, consecutive underlines are treated as meaning the same as one.

The third class of symbol is called (for want of a better name) special. Special symbols are either one character, like '(' and ')' (so that '((' is two symbols), or are combinations of the more unusual characters available. The characters which are symbols on their own (and cannot be combined) are:

( ) [ ] { } , ; !

the following characters are the ones normally available in ASCII for making special symbols:

Types in Ponder are constructed from other types by means of 'constructors' and 'generators', and by the introduction of named types.

5.1 Function Types

The most obvious constructor is that for functions, which is written '↦' (cf Scott [Stoy 77]). Thus if **'LEFT'** and **'RIGHT'** are both types, then **'LEFT → RIGHT'** is the type of a function taking objects of type **'LEFT'** to objects of type **'RIGHT'**. Note that '↦' associates to the right, so that **'A → B → C'** means the same as **'A → (B → C)'**, and that as with expressions, parentheses serve only to group things together, so that **'(D → E) → F'** is equivalent to **'(((D → E)) → F)'**, but is probably easier to read.

5.2 Polymorphic Types

One way of introducing a named type is to allow it to be any type at all. This is done by means of the quantifying constructor '∀' (pronounced "for all"), which introduces a name within the rest of a type or expression. Hence **'∀I. I → I'** is a type, which means "for all types 'I', take an object of type 'I', to another object of type 'I'." This is the type of the identity function, which I will now declare as an example:

```
LET identity ≘ ∀I. I i → I: i;
```

The name introduced in this way is known as the variable in which the type is quantified, and things like **'∀T.'** are known as quantifiers. (This type corresponds roughly to the ML type '* → *'.) However, the function which takes a selector function as an argument could be written:

```
LET f ≘ (∀S. S → S → S) select →
        INT i → BOOL b →
        pair (select  1 i) (select true b)
```

A further note about binding:

The variable introduced by a quantifier exists as far to the right in its type as is possible, so **'∀T. T → BOOL'** means the same as **'∀T. (T → BOOL)'**, and takes any argument, whereas **'(∀T. T) → BOOL'** demands that its argument has type **'∀T. T'** and hence cannot be expressed in ML. Note that it is mainly in this

respect in which Ponder types differ from the types in ML, in that a quantifier introduces a variable locally within a particular type, whereas in ML all type variables are effectively bound at the outermost level. MacQueen & Sethi's system also allows local type quantifiers.

5.3 Type Generators

The final kind of type constructor is the the type generator. These are declared using declarations similar to:

```
TYPE IDENTITY ≘ ∀I. I → I;
```

This declares a generator **'IDENTITY'** which has no parameters, and which means the same as **'∀I. I → I'**. Generators may also have parameters:

```
TYPE ARROW [LEFT, RIGHT] ≘ LEFT → RIGHT;
```

so that **'ARROW [BOOL, BOOL]'** means the same as **'BOOL → BOOL'**, and **'ARROW [INT, REAL]'** is the same as **'INT → REAL'**, and so on.

Finally Ponder allows types to be recursive, so we can have declarations like

```
RECTYPE INFINITE_LIST [THING] ≘ PAIR [THING, INFINITE_LIST [THING]];
```

which means that **'INFINITE_LIST [INT]'** means the same as **'PAIR [INT, PAIR [INT, PAIR [INT, ...]]]'** (except that it is easier to see where the recursion goes than deciding what '...' should mean, (or writing it out in full, which would take too long!)).

Note, however, that **'RECTYPE'** is intended to declare generators for recursive types rather than recursive functions returning types, hence applications of the generator being declared are restricted to be to the parameters declared within the declaration, and parameters may not be generators. For example:

```
RECTYPE WRONG [T] ≘ WRONG [T → T];
```

is invalid. These restrictions are necessary to make mechanical compile time type checking possible, since without them type generators would be as powerful as the lambda calculus, and hence comparing generated types would be as difficult (undecidable, in fact) as comparing two functions.

```
LET x = 3;
LET x = x + 1;
x
```

has the value 4, and not infinity.

__Semantics__

LET declarations may be transformed into lambda expressions:

LET name = expression_1; expression_2

becomes

(λname. expression_2) expression_1

4.3 Functions

As in lambda calculus, expressions in Ponder may be names, or applications or functions (but see below for more). Names we have seen already, and applications are exactly as described for lambda calculus above.

Functions are represented differently, however. Since Ponder is a typed language, all function parameters must have type definitions, so to simplify the form of functions we have a different syntax.

BOOL b → not b

is a function, and specifies that the parameter, which is called 'b' is to be of type '**BOOL**', and has the value '**not b**'. (It may help to pronounce '→' as "returning".) Within the body of a function

TYPE name → body

'**name**' may be used where any expression of type '**TYPE**' could be used. Hence we may declare named functions as in:

```
LET and = BOOL a → BOOL b →
          IF a
          THEN b
          ELSE false
          FI
```


__Semantics__

To transform a Ponder function into a lambda function:

TYPE name → expression

becomes

λname. expression

4.4 Casts

A further form of expression is the cast:

TYPE: old_expression

is an expression which has the type '**TYPE**', and the same value as '**old_expression**'. Note that this is purely an operation on types, and has no effect on the value of the expression, so the value of the expression must be suitable as an object of type '**TYPE**'. For example

FUNGUS: a_mushroom

is an expression whose type is '**FUNGUS**', and is valid if every value that '**a_mushroom**' can be is also a '**FUNGUS**'

__Semantics__

TYPE: expression

becomes

expression

Section 5: Types

Ponder types are similar to the types of MacQueen and Sethi, but are restricted in order to make mechanical type checking possible, and include fewer built in type constructors. The reader is referred to [MacQueen 82] for a more formal treatment of a similar type theory.

R4. Generalisation

$$\frac{(K \geq G\ [\underline{T}])\ \&\ \underline{T} \text{ not free in } K}{K \geq \forall \underline{T}.\ G\ [\underline{T}]}$$

R5. Function

$$\frac{(K3 \geq K1)\ \&\ (K2 \geq K4)}{K1 \rightarrow K2 \geq K3 \rightarrow K4}$$

R6. Result

$$\forall T.\forall \underline{T1}.G\ [\underline{T1}] \rightarrow G1\ [\underline{T1}, T] \geq \forall \underline{T1}.G\ [\underline{T1}] \rightarrow \forall T.G1\ [\underline{T1}, T]$$

If T is not free in 'G'

R7. Recursion

$$\frac{K1 \geq K2 \Rightarrow G\ [K1] \geq K2\ \&\ K3 = G\ [K3]}{K3 \geq K2}$$

('a => b' is used here to mean 'b' may be deduced from the assumption 'a')

R8. Expansion

$$\frac{G\ [\underline{K}] = K}{G\ [\underline{K}] \geq K\ \&\ K \geq G\ [\underline{K}]}$$

(where '$\underline{K}$' may involve 'K')

R1 simply states that a specifier for a type represents a type which is more general or the same as itself

R2 indicates that the relation is transitive.

R3 notes that a function which works for all types is more general than one which only works for one type.

R4 states that if a type 'K' is more general than some function of 'K1', for all types 'K1', then 'K' is also more general than the generalised version of that function.

R5 is perhaps a little more difficult. At first one might believe that functions requiring more general parameters might me more general. However, this corresponds to a greater restriction on the applicability of the function. For example, if we have:

```
LET u = X x -> ...;
LET v = Y y -> ...;
```

with '$Y \geq X$' Within the body of 'u' 'x' may be used in any situation where an object as general as 'X' is needed. Similarly 'v' may use its argument as an object of type 'Y'. Hence 'u' will only work if its argument is more general than 'X', and 'v' will only work for arguments more general than 'Y'. But '$Y \geq X$', so all the objects to which 'v' may be applied are also objects to which 'u' may be applied, hence 'u' is more general than 'v'. Thus R5 states that a function which requires less of its argument is more general.

R6 shows that a quantifier which does not appear in the parameter specifier of a function can be moved to be in the result (and so on until it 'drops off the end' if it does not appear in the specifier at all).

R7 allows the comparison of recursive types.

R8 gives a slightly more formal description of the meaning of definition.

6.1 Validity of Application

Using the definition of '$\geq$' we can now specify that an argument of type 'K' is acceptable to a function whose type is 'Ks' if

$$Ks \geq K \rightarrow Kr,$$

for some (see Section 6) Kr.

since this is true if 'K' is more general than the specifier for the parameter of 'Ks' (if it has one).

Section 6: Relationships Between Ponder Types

It is now necessary to consider which combinations of functions and arguments "make sense" and should be allowed (and if so, what is the type of the result?). One would like to be allowed to do the following:

```
LET identity = ∀I. I i -> I: i;
identity something
```

and it is clear that it does not matter what the type of '**something**' is, and also that whatever it is, the type of the expression '**identity something**' is going to be the same. Hence if '**true**' has type '**BOOL**' then '**identity true**' is valid, and also has type '**BOOL**'. If we define 'y' as in:

```
RECTYPE GENY [T] = GENY [T] -> (T -> T) -> T;
LET half_y = ∀T. GENY [T] part_y -> (T -> T) f -> T:
             f (part_y part_y f);
LET y = half_y half_y;
```

then it is not easy to see whether

```
y identity
```

is valid, and if so, what is the type of '**y identity**'?

A more helpful example is

```
LET apply_boolean_operation_to_true = (BOOL -> BOOL) op -> BOOL:
                                       op true;
apply_boolean_operation_to_true identity
```

since it is clear that although 'op' is specified as being of type '**BOOL → BOOL**', '**identity**' will work just as well. The notion being used here is that of the 'generality' of a function. If we require a function of type '**BOOL → BOOL**' in some situation, it is always safe to use the function 'identity' instead, but we can use '**identity**' in situations where a function from '**BOOL**' to '**BOOL**' will not work, so that '**identity true**' has type '**BOOL**' but '**identity 3**' has type integer, and expressions such as '**not 3**' are invalid. We can hence say that the function '**identity**' is <u>more general than</u> any function of type '**BOOL → BOOL**', or that the type '**∀T. T → T**' is more general than the type '**BOOL → BOOL**'. This relationship is defined more rigorously below.

Thus an object is an acceptable argument to a function if the type of the object is more general than the type of the parameter specified in the declaration of the function.

The non-mathematical reader may like to skip to the end of this section.

For the definition:
The notation

$$\frac{p}{q}$$

means If 'p' is proven, deduce 'q'.
'Tn' are names of types, either in quantifiers or bound at an outer level,
'Kn' are specifiers of types,
'Gn' are generators,
and '∀T, U.' is the same as '∀T. ∀U.'
(X means X1, X2, ...)
'≥' is read 'more general (or the same as)', and '=' means 'is defined as'

The following rules define the relation:

R1. Reflexivity

$$K1 \geq K1$$

R2. Transitivity

$$\frac{(K1 \geq K2) \ \& \ (K2 \geq K3)}{K1 \geq K3}$$

R3. Instantiation

$$\forall T1.\ G\ [T1] \geq G\ [K]$$

By induction:

K1 ≥ B & K2 ≥ B & ... & G [K] ≥ B => G3 [G [K]] ≥ B

Hence:

K1 ≥ B & K2 ≥ B & ... => (G [K] ≥ B => G3 [G [K]] ≥ B)

Hence by R7:

K1 ≥ B & K2 ≥ B & ... => (G3 [G [K]] ≥ B)

QED.

Note that if 'G' is parameterless, we have

G ≥ B

since 'K' is empty.

It is clear that, since any object may be applied to an object of type '∀T.T', and if it returns at all, it will return an object with the same property, any object is acceptable for 'B'

Section 8: Result Types

It now remains to answer the question "What is the type of the result of applying some function to some argument when the application is valid?".

We begin by noticing that if there are no quantifiers on the outside of the type of the function, i.e. it is of the form '**Ks → Kr**', then the type returned is clearly the result type '**Kr**'. For instance, if we have:

```
LET true = BOOL:...;
LET bint = BOOL b -> INT:
           IF b
           THEN 1
           ELSE 0
           FI;
```

then 'bint (true)' has type '**INT**'.

We may also notice that if a type is compared with one in which there are free names, the relationship may be dependant upon some constraints on the values of the variables. Hence if we compare '**BOOL → BOOL**' with the parameter type of '**∀T. (T → T) → T → T**' (that is '**(T → T)**') then '**BOOL → BOOL ≥ (T → T)**' is true if both '**BOOL ≥ T**' and '**T ≥ BOOL**'. We could consider that the result type would be the most general substitution of variables into the result type within these constraints. This seems to work, since for example in

```
LET identity = ∀I. I i -> I: i;
LET true = BOOL: ...;
identity true
```

we have '**BOOL ≥ I**', and hence the most general type for the result would be '**BOOL**'. Unfortunately this does not always make sense. An example is:

```
LET f = ∀T. ((T -> T) -> T) thing -> T: something non obvious;
LET x = (INT -> BOOL) ib -> BOOL: something less odd;
f x
```

since '**(BOOL → INT) → BOOL ≥ (T → T) → T**' is true if both '**BOOL ≥ T**' and '**INT ≥ T**', and it is possible that the only '**T**' for which this is satisfied is '**B = (∀T. T) → B**'. However, since it is unlikely that such applications were intended, the solution here is to add the condition that an application is only valid if the type which satisfies the constraints is in the set of constraining types. Hence the above example would be rejected.

There are still some situations in which there is no most general type satisfying the conditions. An example of this is:

```
LET f = ∀T. ((T -> T) -> T) thing -> T -> T: something odd;
LET x = (A -> B) bi -> A: something equally odd;
f x
```

In this case we have both '**A ≥ T**' and '**T ≥ B**', and hence '**A ≥ B**', but the result types satisfying the conditions are '**A → A**', and '**B → B**' and everything between, and none of these types is the most general. In such cases as this the result can be said to have every such type. Thus the type checker should wait until the result of such an application is used before deciding which of the types was intended. (The present implementation of the type checker does not do this, however, since such applications appear to be rare.

6.2

The following properties are straightforward consequences of the rules:

Proposition 1

$$\forall \underline{T1}, \underline{T2}.\ G\ [\underline{T1}, \underline{T2}] \equiv \forall \underline{T2}, \underline{T1}.\ G\ [\underline{T1}, \underline{T2}]$$

(where 'K1 ≡ K2' means 'K1 ≥ K2 & K2 ≥ K1'). i.e. The order in which quantifiers appear in a type specifier is irrelevant to its meaning.

Proposition 2

$$\forall \underline{T}, T1. G1[\underline{T}] \to G2[\underline{T}, T1] \equiv \forall \underline{T}.\ G1[\underline{T}] \to \forall T1.\ G2[\underline{T}, T1]$$

Section 7: Properties

We may now examine the properties of some types.

7.1 Lemma 1:

(∀T. T) ≥ K for all K

Proof:

Follows immediately from R3

Hence an object of this type is acceptable as an argument to any function, and no object other than one of this type is acceptable for this type. Thus the only way of creating an object of this type is to declare something like:

LET bottom = y identity;

(With 'y f' reducing to 'f (y f)'; hence 'y **identity**' reduces to '**identity (y identity)**', and to 'y **identity**' again, and so on.) which corresponds to a non terminating computation (i.e. there are no values of this type).

7.2 Lemma 2:

If TYPE B = (∀T. T) → B
Then K1 ≥ B & K2 ≥ B & ... => G [K] ≥ B for all G [K]

Proof:

By structural induction on G [K]

Case 1:

G [K] = K1
Follows immediately from the assumptions.

Case 2:

G [K] = G1 [K] → G2 [K]

By Lemma 1, ∀T. T ≥ G1 [K]
By induction, K1 ≥ B & K2 ≥ B & ... => G2 [K] ≥ B,

Hence by R5,
K1 ≥ B & K2 ≥ B & ... => G1 [K] → G2 [K] ≥ (∀T. T) → B

by R8, R2
K1 ≥ B & K2 ≥ B & ... => G1 [K] → G2 [K] ≥ B

Case 3:

G [K] = ∀T. G2 [K, T]

By induction
K1 ≥ B & K2 ≥ B & ... & T ≥ B => G2 [K, T] ≥ B

Hence by R2, R4:
K1 ≥ B & K2 ≥ B & ... & T ≥ B => ∀T. G2 [K, T] ≥ B

Case 4:

G [K] = G3 [G [K]]

Similarly there are no primitive data structures, and we might declare:

9.2 PAIR

```
CAPSULE TYPE PAIR [T1, T2] = ∀U. (T1 → T2 → U) → U;
LET pair = ∀T1, T2. T1 t1 → T2 t2 → PAIR [T1, T2]:
           ∀U. (T1 → T2 → U) u → U:
           u t1 t2;

LET left = ∀T1, T2. PAIR [T1, T2] p → T1:
           p (∀T1, T2. T1 t1 → T2 t2 → t1);

LET right = ∀T1, T2, PAIR [T1, T2] p → T2:
            p (∀T1, T2. T1 t1 → T2 t2 → t2);
SEAL PAIR;
```

representing 'pair's as functions which take unpacking functions as arguments. Thus for example:

```
left (pair a b)
Reduces to:
pair a b (∀T1, T2. T1 t1 → T2 t2 → t1)
to:
(∀T1, T2. T1 t1 → T2 t2 → t1) a b
to:
(∀T2. T2 t2 → a) b
to:
a
```

The rules for comparing capsules are straightforward: if the names of the two capsules identify with different generators, then they are incomparable. If the names identify with the same generator, then compare the arguments as if comparing the body of the generator with the two sets of arguments substituted.

We now have means of representing **'true'** and **'false'**, and **'PAIR'**s of things, so it should not be too difficult to represent bit patterns, but what about **'UNION'**s?

9.3 UNION

```
CAPSULE TYPE UNION [L, R] = ∀E. PAIR [(L → E), (R → E)] → E;
LET inject_l = ∀L. L l → ∀R. UNION [L, R]:
                ∀E. PAIR [(L → E), (R → E)] p → E:
                left p l;

LET inject_r = ∀R. R r → ∀L. UNION [L, R]:
                ∀E. PAIR [(L → E), (R → E)] p → E:
                right p r;

LET choose = ∀L, R. UNION [L, R] u →
             ∀E. PAIR [(L → E), (R → E)] p →
             u p;
SEAL UNION;
```

The above implements disjoint unions of two types (which may either or both be **'UNION'**s of other types), as functions which remember an object of one of the two types, and take a **'PAIR'** of functions, one for each type. Thus if **'left_thing'** has type **'LEFT'**, **'right_thing'** has type **'RIGHT'**, and 'u_l_r' has type **'UNION [LEFT, RIGHT]'**,

```
inject_l left_thing
```

creates a **'∀R. UNION [LEFT, R]'**, i.e. a union of **'LEFT'** and anything,

```
inject_r right_thing
```

creates a **'∀L. UNION [L, RIGHT]'**, and

```
choose u_l_r
       (pair function_to_apply_if_left
             function_to_apply_if_right)
```

will apply to the element **'function_to_apply_if_left'** or **'function_to_apply_if_right'** as appropriate.

9.4 LIST

As a final example of the types of functional data structures, here is one version of lists:

Instead it selects one of the possible types by using a similar scheme to that of ML. It is at present not known whether there are any meaningful programmes in which this strategy selects an inappropriate type.)

An alternative approach is found in [MacQueen & Sethi 1982], and is to allow conjunctions of types. This, however tends to result in very complicated types, with little gain for practical applications.

One final problem is that having chosen the result type in this way, it is possible that the bounds on a type variable were in terms of that variable, for example 'A ≥ A → A', which would be solved by putting 'A ≐ A → A'. As far as is known, most of the examples in which this situation occurs are nonsensical, and those that make sense may be expressed in ways which avoid the problem (and are usually clearer, for example 'y' above). Given this, it was decided that the type checker should not 'invent' generators as solutions to (in)equations like this, and should reject function applications which produce them (as does ML).

An example of this case is:

```
LET f = ∀A. ((A → A) → A) thing → A: ...;
LET z ≐ ∀T. (T → T) g → T → T: ...;
f z
```

which could be solved by putting 'A ≐ A → A' in the result, but is instead rejected.

Section 9: Useful Types

The reader may have noticed that I have said nothing about 'primitive' types in Ponder. This is because there are none. Since it is possible to represent any kind of object with an appropriate functional data structure (as in Gedanken [Reynolds 70]), it is not strictly necessary to make any type primitive. For example, we might represent booleans as:

```
TYPE BOOL = ∀B. B → B → B;
LET true ≐ ∀B. B t → B f → B: t;
LET false ≐ ∀B. B t → B f → B: f;
LET if = BOOL b → ∀T. T then_part → T else_part → T:
        b then_part else_part;
```


Note that the definition of 'if' relies on normal order evaluation, which is the order of evaluation defined for Ponder (otherwise the 'then' and 'else' parts would always be evaluated). Thus 'true' is the function which retains its first argument and discards its second, whereas 'false' retains its second.

It would however be desirable to have types such as boolean built in to the implementation, so that programmes may take advantage of properties of the hardware. Unfortunately 'BOOL' as represented above has slightly different properties from booleans, since it includes several objects which fail to terminate if applied. Also 'if' may be given arguments other than (functions which evaluate to) one of the objects 'true' or 'false'. Since this criticism applies to functional data structures in general, it is clearly necessary to provide a mechanism to prohibit unintended applications. The proposed mechanism is to allow the programmer to state that a particular generator is to be sealed, so that it is no longer equivalent to the type which it generates. Thus 'BOOL' might be declared:

9.1 BOOL

```
CAPSULE TYPE BOOL ≐ ∀B. B → B → B;
LET true = BOOL: ∀B. B t → B f → B: t;
LET false = BOOL: ∀B. B t → B f → B: f;
LET if = BOOL b → ∀T. T then_part → T else_part → T:
        b then_part else_part;
SEAL BOOL;
```

which means that 'true' and 'false' are the only objects of type 'BOOL', and that 'if' is the only function which is allowed to take advantage of the representations of 'true' and 'false'. Note that only objects which have been declared to have type 'BOOL' explicitly retain the type 'BOOL' after 'BOOL' is sealed, so that if the declarations included

```
LET spurion = ∀B. B b → B f → B: y i;
```

'spurion' would only have type '∀B. B → B → B'. Conversely, objects which have been stated to be of type 'BOOL' lose their relationship with the representation of 'BOOL' after the 'SEAL', so that one may no longer apply 'true' to anything.

```
PREFIX - = minus;
```

so that '-1' means 'minus 1'.

Prefix operators differ from functions in that they may be overloaded, and that they bind differently. Thus '**minus minus 1**' means '(**minus minus**) 1', whereas '- - 1' means '- (- 1)'.

10.3 Bracketing Operators

The final kind of operator goes round the outside:

```
LET identity = ∀T. T t -> t;
BRACKET BEGIN END = identity;
BRACKET IF FI = if_thing;
```

so that '**BEGIN expression END**' means the same as '**identity (expression)**'. Note that for a function to be infixed, it must have a type which, when it is applied once, yields another function.

10.4 Overloading

It also would be inconvenient if one had to have a different name for every type of equality function (like '**equals_string_string**' or '**equals_int_int**'), and indeed, programmes would be difficult to read. Hence any kind of operator may be overloaded on the the type(s) of its argument(s). Hence we might have:

```
LET equal_bool_bool = BOOL a -> BOOL b -> if a b (not b);
INFIX = = equal_bool_bool;
INFIX = = INT a -> INT b -> ...;
INFIX = = STRING a -> STRING b -> ...;
PREFIX - = INT i -> ...;
PREFIX - = REAL r -> ...;
```

After which all of '**"s" = "s"**', '**true = true**' and '**3 = 3**' are valid. Note that the overloading is purely syntactic, and that special symbols are not objects, merely syntactic marks.

An overloaded operator application identifies with the most recently declared version of the operator for which an application is valid.

10.5 Pairs

Pairs are however, known more intimately to the compiler. This is to allow "Colateral Declarations" which have proven to be very useful in functional languages. Hence

```
LET a, b = some_pair;
```

means the same as

```
LET Invisible_Name = some_pair;
LET a = left Invisible_Name;
LET b = right Invisible_Name;
```

(Where '**Invisible_Name**' is intended to be some name which will not be visible to the rest of the programme).

Although ',' could have been declared as an infix operator for 'pair', it was decided to build this one facility into declarations, since the more general notion of declarations of which this is a special case does not fit readily with the semantics of the lambda calculus, and hence would require something too complicated to be included in Ponder.

Similarly if the argument to a function is a pair, the parts of the pair may be given names, as in

```
LET swap = ∀T1, T2. T1 a, T2 b -> b, a;
```

Note that ',' associates to the left, so that '**a, b, c**' means '**pair (pair a b) c**'. This is hardly important, since the declarations work the same way, so that

```
LET one, two, three = 1, 2, 3;
```

has the obvious effect.

10.6 Recursive Objects

A further form of syntactic sugar is the recursive object declaration:

```
LET_REC factorial = INT n -> INT: IF (n <= 1)
                                  THEN 1

                                  ELSE n * (factorial (n - 1))
                                  FI
```

```
LET abort = y identity;
CAPSULE RECTYPE LIST [T] = ∀R. (BOOL → T → LIST [T] → R) → R;

LET nil = ∀T. LIST [T]: ∀R. (BOOL → T → LIST [T] → R) f → R:
          f true abort abort;

LET cons = ∀T. T new_head → LIST [T] tail → LIST [T]:
           ∀R. (BOOL → T → LIST [T] → R) f → R:
           f false new_head tail;

LET head = ∀T. LIST [T] l → T:
           l (BOOL null → T h → LIST [T] tail → h);

LET tail = ∀T. LIST [T] l → LIST [T]:
           l (BOOL null → T h → LIST [T] tail → tail);

LET null = ∀T. LIST [T] l → BOOL:
           l (BOOL null → T h → LIST [T] tail → null);

SEAL LIST;
```

In which a list is represented as a function which applies its first argument to a boolean (which is **'true'** if the list is 'nil'), the head of the list, and to another list, which is the tail.

Note that **'CAPSULE'** and **'SEAL'** are not brackets, so that the areas where two capsules are open may overlap. However, if you leave the seal off a capsule, you can expect the compiler to complain, since its contents are likely to spill out and ruin your programme.

Section 10: Syntactic Sugar

The language so far described is rather dry, and some things would be a little tedious to do. Hence Ponder includes some mechanisms for introducing new syntactic forms.

10.1 Infix Operators The simplest form of syntactic sugar is to allow infix operators. First the symbol to be used must be declared as an operator, and given a precedence over other operators and a direction for association.


```
PRIORITY n symbol ASSOCIATES direction;
```

Here 'n' is a nonzero digit with '1' being the most binding, **'symbol'** is either a bold name other than a keyword, or is a "Special symbol", and **'direction'** is either **'LEFT'** or **'RIGHT'**. Then we can associate an expression with the symbol.

```
INFIX symbol = expression;
```

and

```
TYPEINFIX symbol = NAME_OF_GENERATOR;
```

After such a declaration, the symbol may be used as an infix version of the expression or type. So we might have:

```
PRIORITY 5 - ASSOCIATES LEFT;
PRIORITY 3 * ASSOCIATES RIGHT;
INFIX - = subtract;
INFIX * = times;
```

after which 'a - b - c' means **'subtract (subtract a b) c'**, 'a * b * c' means **'times (a (times b c))'** and 'a * b - c * d' means **'subtract (times a b) (times c d)'**.

Further examples:

```
PRIORITY 5 >< ASSOCIATES LEFT;
TYPEINFIX >< = PAIR;

LET or = BOOL a → BOOL b → if b true a;
INFIX OR = or;
LET implies = BOOL a → BOOL b → (not a) OR b;
```

It would be better if operators were not given priorities, but that their binding power were expressed in relation to other operators, but it is not yet clear what the notation for this should be.

10.2 Prefix Operators

Similarly we can have prefix operators:

'CAPSULES' do not provide all the facilities of other forms of abstract type, in that everything declared within a 'CAPSULE' is visible from outside. Hiding is, however already provided within the normal block structure, and is made more palatable with the use of the syntactic sugar for declarations, for example:

```
CAPSULE TYPE BOOL = ∀T. T -> T -> T;
LET true, false, if = BEGIN LET true = BOOL: ∀T. T t -> T f -> T: t;
                            LET false = BOOL: ∀T. T t -> T f -> T: f;
                            LET spurion = ∀T. T t -> T f -> T:
                                          y identity;
                            LET if = BOOL b -> ∀T. PAIR [T, T] te ->
                                     T: b (left te) (right te);

                           true, false, if
                      END;
SEAL BOOL;
```

so that 'spurion' does not even appear in the outside world. This can be improved even more with the use of suitable combining forms for declaration [Milne 76].

The author has implemented a parser and a type checker for Ponder.

Section 15: Acknowledgment

The author wishes to thank his supervisor, Dr M.J.C. Gordon for useful direction, and Dr A.C. Norman for his helpful introduction to functional programming in general.

Appendix: Reference Grammar

The grammar is given as a two level grammar [vWijngaarden 75], but does not attempt to describe the type checking or scope rules.

```
{Metaproductions}
EMPTY:: .

ALPHA::  a; b; c; d; e; f; g; h; i; j; k; l; m;
         n; o; p; q; r; s; t; u; v; w; x; y; z.

NOTION::  ALPHA; NOTION ALPHA.

LEVEL::  i; ii; iii; iiii; iiiii; iiiii i;
         iiiii ii; iiiii iii; iiiii iiii.

PRIO::  prior LEVEL ty.

ASSOC::  left; right.

BRACKET::  name; parenthesis.

{General Hyperrules}
NOTION list:  NOTION;
              NOTION, comma symbol, NOTION list.

{Predicates}
provided that PRIO1 greater than or equal PRIO2:
              provided that PRIO1 greater than PRIO2;
              provided that PRIO1 equal PRIO2.

provided that PRIO1 equal PRIO1: true.

provided that prior LEVEL1 LEVEL2 ty
         greater than prior LEVEL3 ty:  true.

true: EMPTY.

{Productions}
programme:  unit.
```

declares the factorial function.

Semantics

```
LET_REC name = expression_1; expression_2
```

means the same as:

```
(λname. expression_2) (y (λname.expression_1))
```

where

```
y = λf. (λg. f (g g)) (λg. f (g g))
```

Section 11: Separate Compilation

In the first versions of the Ponder compiler it will be necessary to allow the programmer to split a programme into several pieces. The mechanism intended for this is similar to that of Algol68c, in that a programme may optionally begin with **'USING "some-definition-file"'** which causes the compiler to read definitions from the file, and may include expressions of the form **'TYPE: ENVIRON "some-other-definition-file"'**, which would cause it to output all the preceeding definitions into the file.

Section 12: Standard environment

In order to relieve the programmer of some of the initial definitions, a standard environment file will be provided. This will include definitions of the types **'BOOL, INT, STRING, CHAR,'** and of the generators **'PAIR'**, **'LIST'**, and possibly some others, together with definitions of some useful infix operators, such as '+', '-', and so on.

Section 13: Example

I now give a complete definition of the construction for **'IF...THEN...ELSE...FI'**:


```
CAPSULE TYPE IF_FI [T] = T;
CAPSULE TYPE BOOL = ∀T. T → T → T;
CAPSULE TYPE TE [T1, T2] = PAIR [T1, T2];
BRACKET IF FI = ∀T. IF_FI [T] if_fi → T: if_fi;

PRIORITY 9 THEN ASSOCIATES RIGHT;
INFIX THEN = BOOL b → ∀T. TE [T, T] te → IF_FI [T]:
            BEGIN LET then_part, else_part = te;
                  b then_part else_part
            END;

PRIORITY 9 ELIF ASSOCIATES RIGHT;
INFIX ELIF = ∀T1. T1 then → ∀T2. TE [T2, T2] te → TE [T1, T2]:
            pair then IF te
                      FI;
                      .....

PRIORITY 9 ELSE ASSOCIATES RIGHT;
INFIX ELSE = ∀T. T1 then → ∀T2. T2 else → TE [T1, T2]:
            pair then else;

SEAL IF_FI;
SEAL BOOL;
SEAL TE;
```

Note the use of **'CAPSULES'** to ensure that only objects constructed using the operators may be passed to **'IF ... FI'**, and the use of pairs to ensure that the various results may have different types, but have some least general type in common.

Section 14: Conclusion

This note has shown that locally quantified polymorphic types with parameterised generators and capsules provide almost all the facilitates required of a type system.

Some useful kinds of types appear at first to be missing from the system, STRUCTures being a notable example. However, much of this can be solved by the use of the mechanism for the overloading of operators, so that a STRUCTure can be represented by a capsule with the appropriate number of **'PAIR's**, and overloadable functions for field selectors.

```
representation:  function rep;
                 cast;
                 prior iiiii iiii ty ASSOC application.

function rep:  quantified named map;
               named map.

cast:  type, colon symbol, representation.

quantified named map:  quantifier, function rep.

named map:  named parameters, arrow symbol, representation.

named parameters:  typed name;
                   named parameters, comma symbol, typed name.

typed name:  solid type, name symbol.

PRIO1 ASSOC1 application:
     prefix application;
     PRIO2 left application, PRIO2 ASSOC1 operator,
     PRIO2 right application,
     provided that PRIO1 greater than or equal PRIO2.

prefix application:  operator, prefix application;
                     function application.

function application:  expression;
                       function application, expression.

expression:  name symbol;
             character icon;
             string icon;
             integer icon;
             bracketed expression.

bracketed expression:  opening BRACKET, representation,
                       closing BRACKET.
```

{end of productions}

{The representation of a NOTION-symbol is usually obvious from NOTION. In the case of PRIO ASSOC operator-symbols, however, PRIO and ASSOC are determined by the priority declaration for the symbol in question}

```
unit:  declaration, semicolon symbol, unit;
       representation.

declaration:  type dec;
              capsule dec;
              seal capsule;
              operator dec;
              name dec.

operator dec:  priority dec;
               infix declaration;
               type infix declaration;
               prefix declaration;
               bracket declaration.

priority dec:  priority symbol, integer icon, new operator,
               associates symbol, direction.

infix declaration:  infix symbol, new operator,
                    is defined as symbol, representation.

type infix declaration: type infix symbol, new operator,
                        is defined as symbol, bold name symbol.

prefix declaration:  prefix symbol, new operator,
                     is defined as symbol, representation.

bracket declaration:  bracket symbol, opening BRACKET symbol,
                      closing BRACKET symbol,
                      is defined as symbol, representation.

new operator:  PRIO ASSOC operator.

PRIO ASSOC operator:  PRIO ASSOC operator symbol;
                      PRIO ASSOC bold name symbol.

type dec:  type symbol, generator name,
           is defined as symbol, type;

           rectype symbol, generator name,
           is defined as symbol, type.
```



```
capsule dec:  capsule symbol, type dec.

seal capsule:  seal symbol, bold name symbol.

generator name:  bold name symbol, bound types;
                 bold name symbol.

bound types:  left bracket symbol, bold name list,
              right bracket symbol.

type: quantified type;
      prior iiiii iiii ty ASSOC map.

quantified type:  quantifier, type.

quantifier:  for all symbol, bold name list, dot symbol.

PRIO1 ASSOC1 map: solid type;
                  PRIO2 left map, PRIO2 ASSOC1 operator,
                  PRIO2 right map,
                  provided that PRIO1 greater than or equal PRIO2.

solid type:  type name; type pack; applied type generator.

applied type generator:  bold name symbol,
                         solid type list square pack.

solid type list square pack:  left bracket symbol,
                              solid type list,
                              right bracket symbol.

type name:  bold name symbol.

type pack:  opening parenthesis symbol, type,
            closing parenthesis symbol.

name dec:  let symbol, name list,
           is defined as symbol, representation.
```

References:

[Clarke 82]: T.J.W. Clarke,
Proceedings of the 1980 Lisp Conference,
The Lisp Company, 1982

[Demers 79]: A.J. Demers & J.E. Donahue,
Revised Report on Russell,
Department of Computer Science Cornell University, 1979

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

[Hindley 72]: Hindley & Seldin,
Introduction to Combinatory Logic,
Cambridge University Press, 1972

[MacQueen 82]: MacQueen & Sethi,
A Semantic Model of Types for Applicative Languages,
Bell Laboratories, 1982

[Milne 76]: R. Milne & C. Strachey,
A Theory of Programming Language Semantics (1.9.3),
Chapman and Hall, 1976

[Reynolds 70]: J.C. Reynolds,
GEDANKEN — A Simple Typeless Language Based on the Principle
of Completeness and the Reference Concept,
CACM Vol. 13 No. 5, 1970

[Stoy 77]: J.E. Stoy,
Denotational Semantics: the Scott-Strachey Approach to
Programming Language Theory,
MIT Press, 1977

[vWijngaarden 75]: van Wijngaarden et. al.,
The Revised Report on the Algorithmic Language Algol 68,
Springer Verlag, 1975

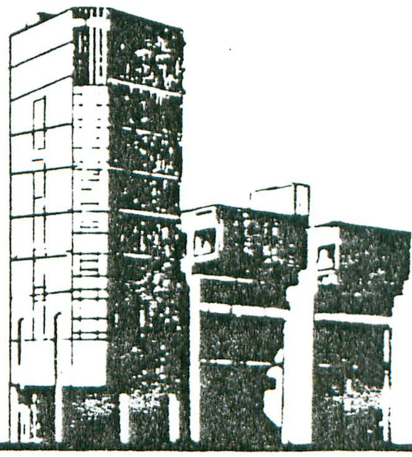
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 34

RECENT DEVELOPMENTS IN LCF: EXAMPLES OF STRUCTURAL INDUCTION

by

Larry Paulson



University of Cambridge
Computer Laboratory

Recent Developments in LCF: Examples of Structural Induction

by Larry Paulson

University of Cambridge

January 1983

Abstract

Manna and Waldinger have outlined a large proof that probably exceeds the power of current automatic theorem-provers. The proof establishes the unification algorithm for terms composed of variables, constants, and other terms. Two theorems from this proof, involving structural induction, are performed in the LCF proof assistant. These theorems concern a function that searches for an occurrence of one term inside another, and a function that lists the variables in a term.

Formally, terms are regarded as abstract syntax trees. LCF automatically builds the first-order theory, with equality, of this recursive data structure.

The first theorem has a simple proof: induction followed by rewriting. The second theorem requires a cases split and substitution throughout the goal. Each theorem is proved by reducing the initial goal to simpler and simpler subgoals. LCF provides many standard proof strategies for attacking goals; the user can program additional ones in LCF's meta-language, ML. This flexibility allows the user to take ideas from such diverse fields as denotational semantics and logic programming.

Recent Developments in LCF: Examples of Structural Induction

by Larry Paulson

University of Cambridge

January 1983

1. Introduction

An interactive proof assistant should be able to reason about a variety of data structures and algorithms, and automatically perform simple proofs. It should be flexible, allowing experimentation with different ways of expressing and performing proofs. These are the goals of the proof assistant Edinburgh LCF (Gordon, Milner, Wadsworth [1979]).

In view of the success of the Boyer and Moore [1979] theorem-prover, why should a proof assistant excite any interest? Manna and Waldinger [1981, page 47] manually prove the Unification Algorithm and remark, "Although the above proof may be beyond the power of current automatic systems, a partially interactive system could be used to produce it with known techniques. This approach requires more human effort, but it still would convey many of the benefits of automatic synthesis." This paper presents two theorems from Manna and Waldinger's theory.

2. Essential Background

LCF relies on one fundamental principle: proofs are conducted in a meta-language, ML. ML is a general-purpose functional program-

4. The Occurrence Relation

Our proofs concern the ordering relation "t OCCS u", an infix function that tests whether t occurs in u as a sub-structure. This requires a theory of the infix equality function, =. The structure package can prove theorems describing the outcome of the equality test[1] for various arguments; for instance, if v, t, u, t', u' are all defined, then

```
(COMB t u)=(COMB t' u')  ==  (t=t') AND (u=u')
(VAR v)=(COMB t u)  ==  FF
```

Figure 2 shows how to axiomatise the OCCS and OCCS_EQ functions, binding the axioms to ML names. The function OCCS_EQ tests "equals or occurs in." The function OCCS is defined as a set of clauses, one for each possible input:[2] a term cannot occur in a constant or variable, and occurs in COMB t1 t2 exactly if it equals or occurs in t1 or t2. This style of defining functions, reminiscent of Prolog (Clocksin and Mellish [1981]) or HOPE (Burstall et al. [1981]), eliminates the need for destructor and discriminator functions.

---

[1] Do not confuse the function = with the predicate ==. The formula "x==y", which may be proved using inference rules, asserts that x and y are equal. The term "x=y" represents a computable equality test applied to x and y. Likewise, do not confuse the truth-valued functions AND, OR, and NOT, with the logical connectives /\, \/, and ~.

[2] The first clause asserts that OCCS is strict; the other uses of UU confine the clauses to defined values only. Our theorems contain similar definedness hypotheses, a reflection that they were originally formulated for a first-order logic, not for PPLAMBDA.


```
let OCCS_EQ =
new_axiom (`OCCS_EQ`,
   "!t t2. t OCCS_EQ t2  ==  (t=t2) OR (t OCCS t2)");;

let OCCS_CLAUSES =
new_axiom (`OCCS_CLAUSES`,
"!t.  t OCCS UU == UU
   /\
   (!c. ~ c==UU  ==>
      t OCCS (CONST c) == FF)
   /\
   (!v. ~ v==UU  ==>
      t OCCS (VAR v) == FF)
   /\
   (!t1 t2. ~ t1==UU  ==>  ~ t2==UU  ==>
      t OCCS (COMB t1 t2) == (t OCCS_EQ t1) OR (t OCCS_EQ t2))");;
```

Figure 2. Axioms for the infix functions OCCS_EQ and OCCS.

---

5. The Variables Proposition

Our first theorem concerns a function VARS_OF, which computes a list of all the variables in a term. (We use a theory of lists, with constructors NIL and CONS, and infix operators APP for append, MEM for membership test.)

```
let VARS_OF_CLAUSES =
new_axiom (`VARS_OF_CLAUSES`,
   "   VARS_OF UU == UU
     /\
     (!c. ~ c == UU  ==>
        VARS_OF(CONST c) == NIL)
     /\
     (!v. ~ v == UU  ==>
        VARS_OF(VAR v) == CONS v NIL)
     /\
     (!t u. ~ t == UU  ==>  ~ u == UU  ==>
        VARS_OF(COMB t u) == (VARS_OF t) APP (VARS_OF u))");;
```

Let us prove that a variable v occurs in a term t exactly when v is a member of the list VARS_OF(t). We give LCF the goal:

ming language whose data values include terms, formulas, and theorems of the logic. Inference rules are ML functions that map theorems to theorems. The only way an ML program can prove a theorem is by applying inference rules to axioms or previously proved theorems.

LCF's logic, PPLAMBDA, uses Scott's theory of continuous partial orders (Stoy [1977]). PPLAMBDA has the usual introduction and elimination rules for each connective. Please note its ASCII representation of logical formulas (Figure 1).

PPLAMBDA Terms

```
c       constant, where c is a constant symbol
x       variable
\x.t    lambda-abstraction over a term
t u     combination (application of function to argument)

UU      "bottom" or "undefined" element
TT      truth-value "true"
FF      truth-value "false"
```

PPLAMBDA Formulas

```
t == u      equality of t and u
!x.A        universal quantifier
?x.A        existential quantifier
A /\ B      conjunction
A \/ B      disjunction
A ==> B     implication
A <=> B     if-and-only-if
~A          negation
etc.
```

Figure 1. Syntax of the logic PPLAMBDA

---

3. Axiomatising a Structure: Combinator Terms

Let us formalise Manna and Waldinger's [1981] theory of the Unification Algorithm, which concerns substitution over combinator terms. These are like PPLAMBDA terms without lambda-abstraction. For proofs we are only concerned with their abstract syntax:

```
term = CONST const      |
       VAR var          |
       COMB term term
```

To axiomatise this structure in LCF, we introduce the abstract types "const", "var", and "term", then axiomatise them using LCF's structure package. LCF stores the types and axioms on a theory file, which can become part of a theory hierarchy.

```
%percent signs enclose comments%

new_type 0 `var`;;        %declare the types var, const, term%
new_type 0 `const`;;
new_type 0 `term`;;

struct_axm (":term",      %build the theory of terms%
     `strict`,
     [`CONST`, ["c:const"];
      `VAR`,   ["v:var"];
      `COMB`,  ["t1:term"; "t2:term"]] );;
```

The resulting theory includes the constructor functions CONST, VAR, and COMB, and axioms stating that these are distinct, one-to-one, etc. The constructor functions are all strict: for instance, CONST UU == UU. To build a theory that includes infinite and partially defined structures, call struct_axm with argument `lazy` instead of `strict`.

```
"v MEM (VARS_OF(COMB t1 t2)) == (VAR v) OCCS_EQ (COMB t1 t2)"
     [ "~ v == UU" ]
     [ "v MEM (VARS_OF t1) == (VAR v) OCCS_EQ t1" ]
     [ "v MEM (VARS_OF t2) == (VAR v) OCCS_EQ t2" ]
     [ "~ t1 == UU" ]
     [ "~ t2 == UU" ]

"v MEM (VARS_OF(VAR v')) == (VAR v) OCCS_EQ (VAR v')"
     [ "~ v == UU" ]
     [ "~ v' == UU" ]

"v MEM (VARS_OF(CONST c)) == (VAR v) OCCS_EQ (CONST c)"
     [ "~ v == UU" ]
     [ "~ c == UU" ]

"v MEM (VARS_OF UU) == (VAR v) OCCS_EQ UU"
     [ "~ v == UU" ]
```

Figure 3. Subgoals after applying induction.

The most interesting case involves terms of the form (COMB t1 t2), with induction hypotheses for t1 and t2. The left and right sides converge:

```
v MEM (VARS_OF (COMB t1 t2))
        unfolding the definition of VARS_OF    --->
v MEM ((VARS_OF t1) APP (VARS_OF t2))
        by a theorem about MEM, APP, and OR    --->
(v MEM (VARS_OF t1)) OR (v MEM (VARS_OF t2))


(VAR v) OCCS_EQ (COMB t1 t2)
        unfolding the definition of OCCS_EQ    --->
(VAR v)=(COMB t1 t2) OR
(((VAR v) OCCS_EQ t1) OR ((VAR v) OCCS_EQ t2))
        since any VAR is distinct from any COMB    --->
(((VAR v) OCCS_EQ t1) OR ((VAR v) OCCS_EQ t2))
        by the induction hypotheses   --->
(v MEM (VARS_OF t1)) OR (v MEM (VARS_OF t2))
```

The other goals converge similarly. To perform such reasoning, LCF provides the tactic ASM_REWRITE_TAC. This rewrites the goal using its assumptions and a list of theorems furnished by the user. The symbols AND_CLAUSES, OR_CLAUSES, etc., denote axioms

and theorems from parent theories.

```
expand (ASM_REWRITE_TAC
          [AND_CLAUSES; OR_CLAUSES;
           TERM_EQUAL_ALL;
           MEM_CLAUSES; MEM_SINGLE; MEM_APP;
           VARS_OF_CLAUSES; VARS_OF_TOTAL;
           OCCS_EQ_CLAUSES] );;
```

LCF reports that the subgoal is solved and prints those that remain. The above call of ASM_REWRITE_TAC includes enough theorems to solve any of the four subgoals.

5.3. Summarising the proof

Now that the interactive proof is complete, let us combine the tactics we used into a composite one that performs the entire proof. For combining tactics, LCF provides functions called *tacticals*. The basic ones are THEN, ORELSE, and REPEAT.

```
TAC1  THEN  TAC2
        calls TAC1, then applies TAC2 to all resulting subgoals

TAC1 ORELSE TAC2
        calls TAC1, if it fails then calls TAC2

REPEAT TAC
        calls TAC repeatedly on the goal and its subgoals
```

The tactic that proves the Variables theorem is

```
TERM_INDUCT_TAC  "t"    THEN
ASM_REWRITE_TAC [AND_CLAUSES; OR_CLAUSES; etc.]
```

In words, the proof is induction followed by rewriting. Many proofs have this simple form -- for instance, properties of list

```
set_goal ([],
   "!v. ~ v==UU  ==>
        !t. v MEM (VARS_OF t) == (VAR v) OCCS_EQ t"  );;
```

We will work backwards from the goal, by applying subgoaling functions, called <u>tactics</u>, to it. A tactic returns a list of subgoals, paired with a <u>proof function</u> that maps proofs of the subgoals to a proof of the original goal. By applying further tactics we reduce all the subgoals to trivial ones. Then we assemble the complete proof from the proof functions.

One simple tactic is GEN_TAC, which reasons that to prove !x.A(x) it suffices to choose a new variable x' and prove A(x'), since this theorem can then be <u>generalised</u> over x'. Another tactic is DISCH_TAC, which reasons that to prove A==>B, it suffices to prove B under the assumption A, since this assumption can then be <u>discharged</u>. Notation: the double bar means "suffices to prove"; assumptions are enclosed in [square brackets]; other assumptions of the goal are implicitly passed to the subgoals.

```
    !x.A(x)
=============        GEN_TAC
    A(x')


    A==>B
=============        DISCH_TAC
    [A] B
```

For interactive proof, LCF's <u>subgoal package</u> is convenient: it stacks pending and solved subgoals, displays the current goal, and applies the proof functions in the correct order. You can apply tactics and back up from faulty steps.

5.1. The structural induction tactic

The structure package provides a tactic, TERM_INDUCT_TAC, to perform <u>structural induction</u> on a goal !t.A(t). This produces four subgoals: t may be a COMB (the step case); t may be a VAR or CONST (the base cases); t may be UU (the undefined case). The subgoals include induction hypotheses and assumptions that the sub-structures are defined.

```
                                     !t. A(t)            TERM_INDUCT_TAC
=================================================================
[ A(t1) ; A(t2) ; ~ t1==UU ; ~ t2 ==UU ]      A(COMB t1 t2)
                             [ ~ v == UU ]       A(VAR v)
                             [ ~ c == UU ]       A(CONST c)
                                                 A(UU)
```

Let us ask the subgoal package to <u>expand</u> the current goal using TERM_INDUCT_TAC. The induction variable t is submerged inside the goal, so the tactic calls GEN_TAC and DISCH_TAC before applying induction.

```
expand (TERM_INDUCT_TAC "t");;
```

5.2. The rewriting tactic

LCF prints the four resulting subgoals and their assumptions (Figure 3). We can prove each one by <u>rewriting</u>: if we have a theorem t==u, change the goal by replacing every instance t' of t by the corresponding instance u'. For <u>implicative</u> rewrites, a theorem A==>(t==u) may be used to rewrite an instance t' by u' if the antecedent A' can be proved.

The inference rule MP_CHAIN, given a list of implications, recursively modifies a theorem using Modus Ponens on it and its parts. With the above theorems, it can change our antecedent to a disjunction of equalities.

```
before:
((tb = t1) OR (tb OCCS t1)) OR
((tb = t2) OR (tb OCCS t2))        == TT

after:
tb==t1  \/  tb OCCS t1 == TT  \/
tb==t2  \/  tb OCCS t2 == TT
```

This theorem has the proper form for the tactic SUBST_CASES_TAC. This splits the goal into four cases, and substitutes each equality through the corresponding goal and its assumptions. Figure 4 shows the first two cases; the other two are similar. An asterisk (*) marks those assumptions which, altered by the substitution, match part of the goal. Now ASM_REWRITE_TAC can finish each case.

```
"((ta = t1) OR (ta OCCS t1)) OR ((ta = t2) OR (ta OCCS t2)) == TT"
       [ "~ ta == UU" ]
       [ "ta OCCS tb == TT" ]
  *    [ "TT == TT   ==>  ta OCCS t1 == TT" ]
       [ "tb OCCS t2 == TT   ==>  ta OCCS t2 == TT" ]
       [ "~ t1 == UU" ]
       [ "~ t2 == UU" ]

"((ta = t1) OR (ta OCCS t1)) OR ((ta = t2) OR (ta OCCS t2)) == TT"
       [ "~ ta == UU" ]
  *    [ "ta OCCS t1 == TT" ]
       [ "t1 OCCS t1 == TT   ==>  ta OCCS t1 == TT" ]
       [ "t1 OCCS t2 == TT   ==>  ta OCCS t2 == TT" ]
       [ "~ t1 == UU" ]
       [ "~ t2 == UU" ]
```

Figure 4. Two cases after substitution in the assumptions.

In the goal A==>B, how do we grab hold of the antecedent A, to put it through MP_CHAIN and SUBST_CASES_TAC? During the interactive search for a proof, we might use DISCH_TAC to put A on the assumption list, then use one of LCF's tacticals for manipulating assumptions. But we have a slick way of expressing the completed proof. Instead of the tactic DISCH_TAC, we can use the tactical DISCH_THEN, which binds the antecedent A to a variable for further use.

```
TERM_TAC "tc"   THEN
ASM_REWRITE_TAC [OCCS_CLAUSES; OCCS_EQ]
THEN
%solves all base cases, but the COMB case remains%
DISCH_THEN      %binds antecedent to a variable%
  (\ante. SUBST_CASES_TAC  (MP_CHAIN [OR_EQ_TT; EQUAL_TT] ante))
THEN
%splits into four cases%
ASM_REWRITE_TAC [OR_CLAUSES; OR_R_TT; OR_TOTAL;
                 TERM_EQUAL_TOTAL; OCCS_TOTAL]
```

In the jargon of denotational semantics (Stoy [1977]), the argument to DISCH_THEN is a continuation that tells what to do with the antecedent. Only time will tell whether such a high-powered approach can be justified; flexibility to try different styles is the hallmark of LCF.

7. Postscript

We can prove many similar theorems. The ordering relation OCCS is anti-reflexive; it is also monotonic with respect to substitution.

utilities (append, map, membership), and totality of recursively defined functions.

Given a set of theorems, ASM_REWRITE_TAC strips off universal quantifiers, splits apart conjunctions, and decides which of the resulting pieces are useful for rewriting or for solving implicative rewrites. It accepts not only term rewrites, t==u, but also <u>formula rewrites</u>, A<=>B. After rewriting, it removes tautologies from the goal -- perhaps solving it completely, returning an empty subgoal list.

Edinburgh LCF provided a similar tactic, SIMPTAC, consisting of seven inscrutable pages of ML. Since SIMPTAC was impossible to modify, Avra Cohn [1982] spent considerable effort adapting her proofs to its limitations. In contrast, ASM_REWRITE_TAC has a modular construction. Its apparently baroque strategy is controlled by a twelve-line ML function that calls tactics, pattern matchers, canonical form translators, rewriting functions, and tautology checkers. These components can easily be changed to suit individual needs or correct shortcomings.

6. Transitivity of the Occurrence Relation

Let us prove a more difficult theorem, that the ordering relation OCCS is transitive:

```
!ta. ~ ta==UU  ==>
!tb. ta OCCS tb == TT  ==>
!tc. tb OCCS tc == TT  ==>  ta OCCS tc == TT
```


If we induct on the variable tc, rewriting solves only three of its four subgoals.[3] The COMB case remains:

```
"((tb = t1) OR (tb OCCS t1)) OR
 ((tb = t2) OR (tb OCCS t2))        == TT
 ==>
 ((ta = t1) OR (ta OCCS t1)) OR
 ((ta = t2) OR (ta OCCS t2))        == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
    [ "tb OCCS t1 == TT  ==>  ta OCCS t1 == TT" ]
    [ "tb OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
    [ "~ t1 == UU" ]
    [ "~ t2 == UU" ]
```

The goal has the form A==>B, so we can use DISCH_TAC to attempt proving B by assuming the antecedent A. Since B has the form b1 OR b2 OR b3 OR b4 == TT, it suffices to prove that one of b1, b2, b3, or b4 equals TT. The antecedent A has the form a1 OR a2 OR a3 OR a4 == TT, and, by studying the assumptions, we realise that if any ai equals TT, then the corresponding bi must equal TT.

If A were a disjunction A1 \/ A2 \/ A3 \/ A4, then we could split into four subgoals, proving B in each case of whether A1, A2, A3, or A4 held. Now A uses the truth-valued functions OR and =, rather than the logical connectives \/ and ==, but we have theorems to correct this, using a weaker kind of formula rewriting:

```
OR_EQ_TT    !p q. p OR q == TT    ==>    p==TT  \/  q==TT
EQUAL_TT    !x y.    x=y == TT    ==>    x==y
```

[3] These three hold vacuously, by contradicting the antecedent tb OCCS tc == TT.

```
!t. ~ t OCCS t == TT

!s1. ~ s1==UU  ==>
!t. ~ t==UU  ==>
!u. t OCCS u == TT  ==>
    (t SUBST s1) OCCS (u SUBST s1) == TT
```

Most of these proofs are straight-forward inductions, but some reveal weaknesses in LCF. For instance, we plan to extend the backwards-chaining primitives to handle existential implications such as (?x.A)==>B. This would be executing PPLAMBDA theorems as a Prolog program (Clocksin and Mellish [1981]).

LCF's methods apply to any logic. The logic PPLAMBDA can complicate first-order problems, adding cases about undefined elements. However, the extra cases are usually trivial. PPLAMBDA is essential for proofs about denotational semantics, compiler correctness, lazy evaluation, and higher-order functional programs.

In such a short paper it is impossible to document, motivate, or even mention all the techniques -- particularly experimental ones. You may not see how LCF helped to discover the proofs shown here, since I have omitted the fruitless first attempts. Look again at the subgoals in Figures 3 and 4, which LCF printed. Imagine writing them out by hand. With computer assistance, we can hope to prove theorems involving increasingly complex data structures.

Acknowledgments. I would like to thank Gerard Huet for porting the Lisp sources, and Mike Gordon for daily discussions. Robin Milner wrote the first structure package.

References

R. Boyer, J. Moore. A Computational Logic. Academic Press, 1979.

R. Burstall, D. MacQueen, D. Sannella. "HOPE: An Experimental Applicative Language." Technical Report CSR-62-80, University of Edinburgh, 1981.

W. Clocksin, C. Mellish. Programming in Prolog, Springer-Verlag, 1981.

A. Cohn, R. Milner. "On using Edinburgh LCF to prove the correctness of a parsing algorithm." Technical Report CSR-113-82, University of Edinburgh, 1982.

A. Cohn. "The correctness of a predecence parsing algorithm in LCF." Technical Report No. 21, University of Cambridge, 1982.

M. Gordon, R. Milner, C. Wadsworth. Edinburgh LCF. Springer-Verlag, 1979.

Z. Manna, R. Waldinger. "Deductive Synthesis of the Unification Algorithm." Science of Computer Programming, 1981, pages 5-48.

J. Stoy. Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.

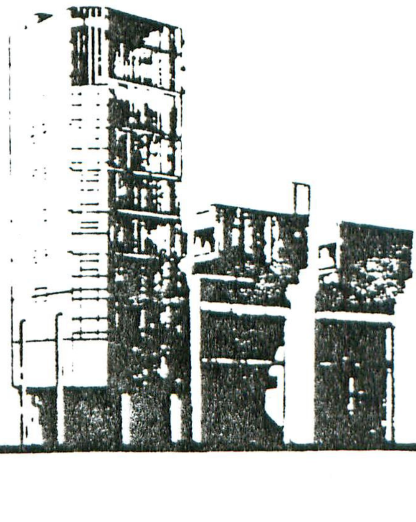
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 35

REWRITING IN CAMBRIDGE LCF[1]

by

Lawrence Paulson



University of Cambridge
Computer Laboratory

Rewriting in Cambridge LCF[2]

Lawrence Paulson

Cambridge University

February 1983

1. Introduction

In LCF, proof strategies are implemented as functions, called tactics, that reduce goals to sub-goals. Tactics are combined into larger tactics using other functions, called tacticals. The basic tacticals, THEN, ORELSE, and REPEAT, correspond to notions that occur both in programming languages and in regular grammars: the notions of sequencing, alternation, and repetition.

Now consider a different task — systematically rewriting a PPLAMBDA term into an equivalent term. Since we want a proof that the new term is equivalent, we are concerned with functions that map a term t into a theorem |-"t==u", from which the new term u can be extracted. We shall see that such conversion functions, like tactics, can be combined using operators for sequencing, alternation, and repetition.

We will examine a succession of functions: to match patterns, instantiate theorems, rewrite terms and formulas, simplify tautologies, search by backwards chaining, and translate to canonical forms. In a few lines of code, these operators can build many simplifiers comparable to the one in

---

[2] Cambridge LCF is a descendant of Edinburgh LCF. The name change is necessary because the new system is incompatible with Edinburgh LCF, which had remained stable for several years.

Rewriting in Cambridge LCF[1]

Lawrence Paulson

Cambridge University

February 1983

Abstract

Most theorem provers provide some means of simplifying a goal by rewrite rules. The LCF proof assistant provides a family of rewriting functions, and operators to combine them. A succession of functions is described, from pattern matching primitives to the proof tactic that performs most inferences in LCF. The functions offer a wide choice of rewriting methods for implementing tactics. The approach is an example of programming with higher-order functions in the language ML.

[1] Research supported by S.E.R.C. Grant number GR/B67766

Table of Contents

Edinburgh LCF (Gordon, Milner, Wadsworth [1979]).

This paper is only superficially concerned with rewriting. Its real purpose is to illustrate functional programming in action, using the language ML (Gordon et al. [1978]). It presents examples of ML code from the implementation of LCF, though sometimes in an idealised form that ignores efficiency. In the production version of LCF, certain critical inferences are wired in, though I have kept a record of the original code that derives these inferences from PPLAMBDA axioms.

Notation: if a theorem A has been proved, then LCF prints it as |-"A". This notation appears in ML code below, as shorthand for a variable containing the theorem A, or for an expression that would prove the theorem A. You cannot use this notation in real ML programs, because LCF provides no way of creating theorems except to prove them. Figure 1 describes some of the ML functions that this paper uses. Figure 2 shows the syntax of LCF's logic, PPLAMBDA. I assume that you have some knowledge of LCF.

In examples of terminal sessions, the lines beginning with # denote input to LCF, and the other lines denote LCF's response. Comments are enclosed in percent signs. These sessions have actually been run.

2. Pattern Matching Primitives

A quantified theorem such as |-!x.A stands for an infinity of theorems, one for each x. In a proof, you are likely to need some of these instances,


```
flat ll
     flattens the list of lists, ll, into a list

itlist f [x1;...;xn] y
     computes (f x1 (f ... (f xn y) ...)), like APL's reduce operator

mapfilter f l
     maps the function f over the list l, ignoring any elements on which f
     fails, and returning a list of f's results

f o g
     function composition, \x.f(g x)

dest_equiv, dest_imp, dest_conj, ...
     destructor functions to take apart  equivalences, implications, con-
     junctions, ...

ASSUM_LIST (\asl. tac)
     a tactic that maps ASSUME over a goal's assumptions, and binds this
     list of theorems to the variable asl.
```

Figure 1. Some ML and LCF functions

PPLAMBDA Terms

```
c          constant, where c is a constant symbol
x          variable
\x.t       lambda-abstraction over a term
t u        combination (application of function to argument)
p => t | u conditional expression (actually "COND p t u")
t,u        ordered pair (actually "PAIR t u")

UU         "bottom" or "undefined" element
TT         truth-value "true"
FF         truth-value "false"
```

PPLAMBDA Formulas

```
TRUTH()    standard tautology
FALSITY()  standard contradiction

t == u     equality of t and u
t << u     inequality of t and u, Scott partial ordering
!x.A       universal quantifier
?x.A       existential quantifier
A /\ B     conjunction
A \/ B     disjunction
A ==> B    implication
A <=> B    if-and-only-if
~A         negation (actually "A ==> FALSITY()")
```

Figure 2. Syntax of the logic PPLAMBDA

rather than the general form. LCF's pattern matching primitives relieve you of the tedium of instantiating theorems.

2.1. Matching Terms and Formulas

Most rewriting functions depend on the matching functions term_match and form_match. These match types as well as variables, to exploit PPLAMBDA's polymorphism. If pattern and object are terms, the call

```
term_match pattern object
```


returns a pair of lists, describing how to instantiate the pattern's variables and type variables to make it alpha-convertable[3] to the object. If no match exists, term_match fails. The analogous function for formulas is form_match. Figure 3 shows examples.

2.2. Instantiating Theorems by Matching

Since term_match and form_match are too primitive for most applications, LCF provides functions for instantiating theorems. Calling

```
PART_TMATCH (partfn,A) t
```

matches the term t to some part of the theorem A obtained by the function partfn, and returns A with its types and variables instantiated. The partfn is typically lhs or rhs, and is applied to the theorem after stripping its outer universal quantifiers.

PART_TMATCH makes it easy to define inference rules that involve matching. For instance, PPLAMBDA includes an axiom stating that the bottom element is smaller than any other element:

```
MINIMAL:      |- !x. UU << x
```

Since axioms are sometimes inconvenient to use, LCF provides an inference rule MIN that maps any term t to the theorem |-UU<<t. This rule can be expressed using PART_TMATCH and the destructor function rhs:

[3] Two terms are alpha-convertable if they can be made identical by renaming bound variables.

```
#let tm_obj = "TT=> (FF,TT) | (TT,FF)";;
tm_obj = "(TT => (FF,TT) | (TT,FF))" : term

#term_match "x:*" tm_obj;;
["(TT => (FF,TT) | (TT,FF))","x"],
[":tr # tr",":*"]
: ((term # term) list # (type # type) list)

#term_match "p=>x|y:*" tm_obj;;
["TT,FF","y"; "FF,TT","x"; "TT","p"],
[":tr # tr",":*"]
: ((term # term) list # (type # type) list)

#term_match "(f:*->**)x" tm_obj;;        %p=>x|y denotes COND p x y%
["TT,FF","x"; "COND TT(FF,TT)","f"],
[":tr # tr",":**"; ":tr # tr",":*"]
: ((term # term) list # (type # type) list)

#term_match "p=>FF|y" tm_obj;;
evaluation failed     term_match

#term_match "\p.p" tm_obj;;
evaluation failed     term_match


#let fm_obj = "!x:*. (x,TT) == UU";;
fm_obj = "!x. x,TT == UU" : form

#form_match "!x:*. (x,x) == UU" fm_obj;;
evaluation failed     form_match

#form_match "!y:*. (x,y) == UU" fm_obj;;
evaluation failed     form_match

#form_match "?x. (x,TT) == UU" fm_obj;;
evaluation failed     form_match

#form_match "!y:*. (y,z:**) == UU" fm_obj;;
["TT","z"],
[":tr",":**"]
: ((term # term) list # (type # type) list)
```

Figure 3. Examples of term_match and form_match

```
let MIN = PART_TMATCH (rhs,MINIMAL);;
```

This matches a term t against the right-hand-side of the axiom MINIMAL, returning the instance |-"UU<<t".

The function PART_FMATCH is analogous, but matches some sub-formula of the theorem rather than a sub-term. One of its applications is a simple resolution rule, MATCH_MP. This is a Modus Ponens that matches an implication to an antecedent.[4]

```
let MATCH_MP impth =
    let match = PART_FMATCH (fst o dest_imp) impth
    in
    \th. MP (match (concl th)) th;;
```

Calling MATCH_MP (|-"!x1...xn. A ==> B") (|-"A'"), where A' is an instance of A, returns the corresponding instance of B, |-"B'". One of my proofs (Paulson [1983]) involves a total function VARS_OF and a theory of strict lists. The function MAP, which maps any function f over a list, produces a total function if f is total. The rule MATCH_MP can prove that the function (MAP VARS_OF) is total (Figure 4).

3. Term Conversions

Let a term conversion be any function of type "term->thm" that maps a term t to a theorem |-"t==u". This converts the term t to another term u, and proves that the two are equal. Since ML allows us to take theorems apart, we can extract the term u from the theorem |-"t==u". Several conversions were built into Edinburgh LCF, where they were called axiom schemes. Cambridge LCF provides only a few conversions, but many functions for creating and combining them.

[4] The composite function (fst o dest_imp) takes the first part of an implication, which is the antecedent.

```
#load_theory `VARS_OF`;;
Theory VARS_OF loaded
() : void

# % now get two theorems from this theory %
#let VARS_OF_TOTAL = theorem `VARS_OF` `VARS_OF_TOTAL`;;
VARS_OF_TOTAL = |-"!t. ~ t == UU  ==>  ~ VARS_OF t == UU" : thm

#let MAP_TOTAL = theorem `list_fun` `MAP_TOTAL`;;
MAP_TOTAL =
|-"!f.
    (!x. ~ x == UU  ==>  ~ f x == UU)  ==>
    (!l. ~ l == UU  ==>  ~ MAP f l == UU)"
: thm

#let TOTAL1 = MATCH_MP MAP_TOTAL VARS_OF_TOTAL;;
TOTAL1 = |-"!l. ~ l == UU  ==>  ~ MAP VARS_OF l == UU" : thm

#let TOTAL2 = MATCH_MP MAP_TOTAL TOTAL1;;
TOTAL2 = |-"!l. ~ l == UU  ==>  ~ MAP(MAP VARS_OF)l == UU" : thm

#let TOTAL3 = MATCH_MP MAP_TOTAL TOTAL2;;
TOTAL3 = |-"!l. ~ l == UU  ==>  ~ MAP(MAP(MAP VARS_OF))l == UU" : thm
```

Figure 4. Examples of MATCH_MP

3.1. Basic Conversions

Beta conversion, BETA_CONV, is standard in LCF. As Figure 5 shows, BETA_CONV performs exactly one beta-conversion, not two or zero. If x is a variable, and u, v are terms, and u[v/x] denotes the substitution of v for x in u, then

```
BETA_CONV "(\x.u)v"    returns    |-"(\x.u)v == u[v/x]"
```

Another basic conversion is to rewrite according to a theorem that states an equivalence. Such theorems are called rewrites or term rewrites.

```
REWRITE_CONV (|-"!x1...xn. t==u")
```



```
#   %test data used also in later figures%
#let [abs1; abs2; condu; condt; condf; condfst] = terms;;
abs1 = "(\fun.(fun(TT,FF) => x | y))FST" : term
abs2 = "(\t.(\u.t,u)FF)TT" : term
condu = "(UU => (TT,FF,p) | (q,TT,q))" : term
condt = "(TT => x | y)" : term
condf = "(FF => f x | f y)" : term
condfst = "(FST(TT,FF) => x | y)" : term

#BETA_CONV abs1;;
|-"(\fun.(fun(TT,FF) => x | y))FST == (FST(TT,FF) => x | y)" : thm

#BETA_CONV abs2;;        %two beta-conversions possible%
|-"(\t.(\u.t,u)FF)TT == (\u.TT,u)FF" : thm

#BETA_CONV condt;;       %no beta-conversions possible%
evaluation failed        BETA_CONV
```

Figure 5. Examples using the conversion BETA_CONV

is a conversion that takes any instance of t, such as t', and instantiates the variables x1 ... xn, to return the theorem |-"t'==u'". As Figure 6 shows, REWRITE_CONV fails on all terms that do not match t. It is implemented using the matching function PART_TMATCH:

```
let REWRITE_CONV = PART_TMATCH (fst o dest_equiv);;
```

3.2. Sequential Conversions

These conversions apply only to a narrow class of terms. We need some way of combining conversions so that if one fails, we can try another. The operator ORELSEC provides this notion of alternation. For functions f and g, and argument t,

```
# % some PPLAMBDA axioms, used also in later figures %
#let [COND_UU; COND_TT; COND_FF;
#     MIN_COMB; MIN_ABS;
#     MK_PAIR; FST_PAIR; SND_PAIR]
#  = rewrites;;

COND_UU = |-"(UU => x | y) == UU" : thm
COND_TT = |-"(TT => x | y) == x" : thm
COND_FF = |-"(FF => x | y) == y" : thm
MIN_COMB = |-"UU x == UU" : thm
MIN_ABS = |-"\x.UU == UU" : thm
MK_PAIR = |-"FST x,SND x == x" : thm
FST_PAIR = |-"FST(x,y) == x" : thm
SND_PAIR = |-"SND(x,y) == y" : thm

#let COND_TT_CONV = REWRITE_CONV COND_TT;;
COND_TT_CONV = - : conv

#COND_TT_CONV condt;;
|-"(TT => x | y) == x" : thm

#COND_TT_CONV condfst;;
evaluation failed     term_match

#let FST_CONV = REWRITE_CONV FST_PAIR;;
FST_CONV = - : conv

#FST_CONV "FST (TT,FF)";;
|-"FST(TT,FF) == TT" : thm

#FST_CONV "SND (TT,FF)";;
evaluation failed     term_match
```

Figure 6. Examples using the conversion REWRITE_CONV

(conv1 ORELSEC conv2) t   is defined to be   conv1 t ? conv2 t

It tries conv1; if that fails, then it tries conv2.

We can also implement the notion of sequencing, defining an operator THENC. For two term conversions conv1 and conv2, the conversion

(conv1 THENC conv2) t

derives

```
|-"t==t1"       by conv1
|-"t1==t2"      by conv2
and returns
|-"t==t2"       by transitivity
```

The conversion fails if either conv1 or conv2 does. Both ORELSEC and THENC have identity elements. The conversion NO_CONV applies to no terms, and the conversion ALL_CONV applies to all (Figure 7); these are defined

```
let NO_CONV t = failwith `NO_CONV`;;

let ALL_CONV t = REFL t;;
```

Ignoring differences in failure tokens, the following equivalences hold for all conversions conv:

```
NO_CONV ORELSEC conv = conv ORELSEC NO_CONV = conv

ALL_CONV THENC conv = conv THENC ALL_CONV = conv
```

It is interesting to compare these with tacticals. The tactic ALL_TAC,[5] which passes on its goal unchanged, is the identity for THEN. The tactic NO_TAC, which fails on all goals, is the identity for ORELSE. The tactical ORELSE is implemented exactly like ORELSEC. However, the tactical THEN is entirely different from THENC.

[5] called IDTAC in Edinburgh LCF.

```
#let BETA_BETA_CONV = BETA_CONV THENC BETA_CONV;;
BETA_BETA_CONV = - : conv

#BETA_BETA_CONV abs1;;  %can only do one beta-conversion%
evaluation failed     BETA_CONV

#BETA_BETA_CONV abs2;;
|-"(\t.(\u.t,u)FF)TT == TT,FF" : thm

#let COND_TF_CONV =      %handles both TT and FF%
#    (REWRITE_CONV COND_TT) ORELSEC
#    (REWRITE_CONV COND_FF);;
COND_TF_CONV = - : conv

#COND_TF_CONV condf;;
|-"(FF => f x | f y) == f y" : thm

#COND_TF_CONV condt;;
|-"(TT => x | y) == x" : thm

#COND_TF_CONV condfst;;
evaluation failed     term_match

#ALL_CONV condfst;;
|-"(FST(TT,FF) => x | y) == (FST(TT,FF) => x | y)" : thm

#NO_CONV condt;;
evaluation failed     NO_CONV
```

Figure 7. Examples of THENC, ORELSEC, ALL_CONV, NO_CONV

For combining a list of conversions, it is convenient to use FIRST_CONV, defined using itlist and NO_CONV, where

```
FIRST_CONV [conv1; ...; convn] =
    conv1 ORELSEC ... ORELSEC convn
```

Once we have sequencing, alternation, and identities, we can define repetition for conversions exactly as it is defined on tactics. Figure 8 shows more examples.


```
letrec REPEATC conv t =
    ((conv THENC (REPEATC conv)) ORELSEC ALL_CONV) t;;
```

This definition involves a fine point. Without the abstraction over t, REPEATC would always loop, because ML evaluates in applicative order rather than normal order (lazy evaluation). We will see this in all the recursive conversions below.

3.3. Depth Conversions

```
#let BETA_N_CONV = REPEATC BETA_CONV;;
BETA_N_CONV = - : conv

#BETA_N_CONV abs1;;      %one beta-conversion%
|-"(\fun.(fun(TT,FF) => x | y))FST == (FST(TT,FF) => x | y)" : thm

#BETA_N_CONV abs2;;      %two beta-conversions%
|-"(\t.(\u.t,u)FF)TT == TT,FF" : thm

#BETA_N_CONV condt;;     %no beta-conversions%
|-"(TT => x | y) == (TT => x | y)" : thm

#let COND_CONV =         %any of TT, FF, UU%
#    FIRST_CONV (map REWRITE_CONV [COND_TT; COND_FF; COND_UU]);;
COND_CONV = - : conv

#COND_CONV condu;;
|-"(UU => (TT,FF,p) | (q,TT,q)) == UU" : thm

#COND_CONV condt;;
|-"(TT => x | y) == x" : thm

#COND_CONV condfst;;     %still cannot simplify the condition%
evaluation failed     FIRST_CONV
```

Figure 8. Examples of FIRST_CONV and REPEATC

These conversions and operators cannot traverse terms recursively; they can only rewrite top-level terms. LCF provides functions for converting sub-terms: COMB_COMB handles combinations, while ABS_COMB handles abstractions. They fail on terms that do not have the corresponding form.

The conversion (COMB_CONV conv "f t") derives

```
|-"f==g"        by conv
|-"t==u"        by conv
and returns
|-"f t == g u"  by substitution
```

The conversion (ABS_CONV conv "\x.t") derives

```
|-"t == u"           by conv
|-"\x.t == \x.u"     by extensionality, possibly renaming x
```

Let us combine COMB_CONV and ABS_CONV into a conversion that rewrites a term's top-level sub-terms. A term can be a constant, variable, combination, or abstraction.

```
let SUB_CONV conv =
    (COMB_CONV conv)  ORELSEC
    (ABS_CONV conv)   ORELSEC
    ALL_CONV;;
```

Now it is simple to write a conversion DEPTH_CONV that recursively rewrites all sub-terms of a term:

```
letrec DEPTH_CONV conv t =
   (SUB_CONV (DEPTH_CONV conv) THENC (REPEATC conv))
   t;;
```


DEPTH_CONV may return a term that is not in simplest form. A more sophisticated conversion (Figure 9) resimplifies the term every time it is rewritten.

```
letrec REDEPTH_CONV conv t =
   (SUB_CONV (REDEPTH_CONV conv) THENC
    ((conv THENC (REDEPTH_CONV conv)) ORELSEC ALL_CONV))
   t;;
```

DEPTH_CONV and REDEPTH_CONV rewrite sub-terms before rewriting the top-level term. You may prefer a conversion that tries to rewrite the term before its sub-terms. In a theory of lazy lists, this would simplify NULL(CONS x l) to FF without simplifying x or l, which could be quicker.

```
#let MANY_CONV =        %try all the top-level conversions%
#    FIRST_CONV (map REWRITE_CONV rewrites)   ORELSEC
#    BETA_CONV;;
MANY_CONV = - : conv

#let D_CONV = DEPTH_CONV MANY_CONV;;
D_CONV = - : conv

#D_CONV abs2;;       %does both beta-conversions%
|-"(\t.(\u.t,u)FF)TT == TT,FF" : thm

#D_CONV condfst;;    %simplifies the condition, then uses it%
|-"(FST(TT,FF) => x | y) == x" : thm

#D_CONV abs1;;       %result could be simplified further%
|-"(\fun.(fun(TT,FF) => x | y))FST == (FST(TT,FF) => x | y)" : thm

#let RD_CONV = REDEPTH_CONV MANY_CONV;;
RD_CONV = - : conv

#RD_CONV abs1;;
|-"(\fun.(fun(TT,FF) => x | y))FST == x" : thm
```

Figure 9. Examples of DEPTH_CONV and REDEPTH_CONV

```
letrec TOPDEPTH_CONV conv t =
   (let RTOP_CONV = (conv THENC (TOPDEPTH_CONV conv)) ORELSEC ALL_CONV in
    RTOP_CONV  THENC
    SUB_CONV (TOPDEPTH_CONV conv)  THENC
    RTOP_CONV)
   t;;
```

4. Formula Conversions

The same ideas apply to the rewriting of formulas. Let a formula conversion be any function of type "form -> thm", mapping formulas A to theorems |-"A <=> B". This converts A to B and proves the two equivalent. LCF provides a family of formula conversions, and operators to combine them, as for term conversions.

4.1. Analogs of Term Conversions

The formula conversion REWRITE_FCONV allows the rewriting of a formula according to a theorem

```
|-"!x1...xn. A <=> B"
```

Such theorems are called formula rewrites. The conversion is implemented like REWRITE_CONV, using the instantiation function PART_FMATCH:

```
let REWRITE_FCONV = PART_FMATCH (fst o dest_iff);;
```

This is useful for expanding out the definition of a predicate, such as

```
|- !rel. TRANSITIVE rel  <=>
      !x y z. rel x y == TT  /\  rel y z == TT  ==>  rel x z == TT
```


LCF provides the identity conversions NO_FCONV, which always fails, and ALL_FCONV, which maps A to "|-"A <=> A"." For sequencing, the conversion fconv1 THENFC fconv2 is defined in terms of Modus Ponens. The operators ORELSEFC, REPEATFC, and FIRST_FCONV are implemented like their term counterparts.

If P is a predicate, then (PRED_FCONV conv "P(t)") is a formula conversion that derives

```
|-"t==u"                 by conv
|-"P(t) ==> P(u)"        by substitution
|-"P(u) ==> P(t)"        by symmetry and substitution
and returns
|-"P(t)  <=>  P(u)"
```

Similarly, PPLAMBDA's inference rules allow us to implement conversion operators for the quantifiers and logical connectives. The conversion SUB_FCONV applies a conversion to all top-level terms and formulas of a formula:

```
let SUB_FCONV conv fconv =
   (CONJ_FCONV fconv) ORELSEFC
   (DISJ_FCONV fconv) ORELSEFC
   (IMP_FCONV fconv) ORELSEFC
   (FORALL_FCONV fconv) ORELSEFC
   (EXISTS_FCONV fconv) ORELSEFC
   (PRED_FCONV conv);;
```

For mapping a conversion over all sub-formulas of a formula, the conversions DEPTH_FCONV and REDEPTH_FCONV are defined like their term analogs. Figure 10 offers more examples.

```
letrec DEPTH_FCONV conv fconv w =
   (SUB_FCONV conv (DEPTH_FCONV conv fconv) THENFC
    (REPEATFC fconv))
   w;;
```

```
letrec REDEPTH_FCONV conv fconv w =
   (SUB_FCONV conv (REDEPTH_FCONV conv fconv) THENFC
    ((fconv THENFC (REDEPTH_FCONV conv fconv)) ORELSEFC ALL_FCONV))
   w;;
```

4.2. Eliminating Propositional Tautologies

LCF includes conversions that recognise propositional tautologies. For instance, TAUT_CONJ_FCONV can derive

```
TRUTH()  /\  A          <=>     A
A  /\  TRUTH()          <=>     A
FALSITY()  /\  A        <=>     FALSITY()
A  /\  FALSITY()        <=>     FALSITY()
```

Most of the tautology conversion functions are hand-coded to treat a particular class of formulas. But ones for quantifiers are implemented directly in terms of formula conversions. LCF has stored the theorems

```
FORALL_TRUTH      |- "(!x.TRUTH())  <=>  TRUTH()"

FORALL_FALSITY    |- "(!x.FALSITY())  <=>  FALSITY()"
```

Using these, the "forall" tautology conversion can simplify !x.TRUTH() and !x.FALSITY(). Its ML code is


```
#PRED_FCONV RD_CONV "P (^abs2, ^cond fst)";;
|-"P (\t.(\u.t,u)FF)TT,(FST(TT,FF) => x | y)  <=>  P (TT,FF),x" : thm

#let [P_Q; LESS_UU] = frewrites;;
P_Q = |-"!x. P (x,x)  <=>  Q x" : thm
LESS_UU = |-"!x. x << UU  <=>  x == UU" : thm

#let [impl; conj1; equiv1] = forms;;
impl =
"!x y. P (TT => y | z),SND(y,y)  ==>  Q (\p.(p => v | y))FF"
: form
conj1 = "?x. x << UU TT  \/  SND(x,TT) == (UU => TT | FF)" : form
equiv1 =
"!x. ?p. (FST(p,p) => x | UU) == (p => (\z.SND(x,z))x | (\r.r)UU)"
: form

#let MANY_FCONV =
    FIRST_FCONV (map REWRITE_FCONV [P_Q; LESS_UU]) ;;
MANY_FCONV = - : fconv

#let D_FCONV = DEPTH_FCONV (REDEPTH_CONV MANY_CONV) MANY_FCONV;;
D_FCONV = - : fconv

#D_FCONV impl;;
|-"(!x y. P ((TT => y | z), SND(y,y))  ==>  Q (\p.(p => v | y))FF)  <=>
   (!x y. Q y  ==>  Q y)"
: thm

#D_FCONV conj1;;
|-"(?x. x << UU TT  \/  SND(x,TT) == (UU => TT | FF))  <=>
   (?x. x == UU  \/  TT == UU)"
: thm

#D_FCONV equiv1;;
|-"(!x. ?p. (FST(p,p) => x | UU) == (p => (\z.SND(x,z))x | (\r.r)UU))  <=>
   (!x. ?p. (p => x | UU) == (p => x | UU))"
: thm
```

Figure 10. Examples of formula conversions

```
let TAUT_FORALL_FCONV =
    REWRITE_FCONV FORALL_TRUTH
    ORELSEFC
    REWRITE_FCONV FORALL_FALSITY;;
```

The family of conversions is modular. To improve the tautology test, just write a better version of FORALL_TAUT_FCONV. Perhaps it should simplify

!x.A to A for any formula A that does not contain x.

LCF provides the conversion BASIC_TAUT_FCONV, which tries all the tautology tests in turn, failing if none apply.

```
let BASIC_TAUT_FCONV =
    FIRST_CONV [
        TAUT_CONJ_FCONV;
        TAUT_DISJ_FCONV;
        TAUT_IMP_FCONV;
        TAUT_IFF_FCONV;
        TAUT_FORALL_FCONV;
        TAUT_EXISTS_FCONV;
        TAUT_PRED_FCONV];;
```

There are many ways of building simplifiers from these conversions. The standard one, BASIC_FCONV, uses REDEPTH_CONV to simplify terms, DEPTH_FCONV to simplify formulas, and BASIC_TAUT_FCONV to find tautologies in the resulting formulas. This is a pragmatic choice. The REDEPTH conversions are slow but thorough; terms generally need them but formulas do not. Figure 11 shows it solving the tautologies that DEPTH_FCONV missed.

```
let BASIC_FCONV conv fconv =
    DEPTH_FCONV (REDEPTH_CONV conv) (fconv ORELSEFC BASIC_TAUT_FCONV);;
```

5. Anatomy of a Rewriting Tactic

In studying all these functions, let us remember their original purpose: to help us prove theorems. LCF provides a tactic, ASM_REWRITE_TAC, which simplifies a goal by rewriting it and detecting tautologies. This tactic is imperfect and is likely to be modified from time to time. Yet its struc-


```
#let B_FCONV = BASIC_FCONV MANY_CONV MANY_FCONV;;
B_FCONV = - : fconv

#B_FCONV impl;;
|-"(!x y. P ((TT => y | z), SND(y,y))  ==>  Q (\p.(p => v | y))FF)  <=>
   TRUTH ()"
: thm

#B_FCONV conjl;;
|-"(?x. x << UU TT  \/  SND(x,TT) == (UU => TT | FF))  <=>
   (?x. x == UU)"
: thm

#B_FCONV equivl;;
|-"(!x. ?p. (FST(p,p) => x | UU) == (p => (\z.SND(x,z))x | (\r.r)UU))  <=>
   TRUTH ()"
: thm
```

Figure 11. Examples of BASIC_FCONV solving tautologies

ture illustrates both the use of conversion functions and the power of functional programming in general.

5.1. Implicative Rewrites

The above rewriting conversions accept rewriting theorems of the form |- "t==u" or |-"A<=>B". However, there are many weaker theorems where an equivalence holds only if, for example, some function is strict or some variable is defined. Consider a theory of lists with strict CONS and MAP functions. The following theorem would not hold if x could be undefined, because if the function f were not strict, then the right-hand-side would be defined, the left undefined.

```
~ x==UU  ==>   MAP f (CONS x l) == CONS (f x) (MAP f l)
```

In general, such implicative rewrites have the form:

```
A1 ==> (... ==> (An ==>  t==u) ...)
A1 ==> (... ==> (An ==>  B<=>C) ...)
(where n may be zero)
```

How can a conversion use such a theorem, given some instance t' of the left hand term t? If it can prove the instances of the antecedents A1' to An', then, by Modus Ponens, it can return the theorem |-"t'==u'". How should it try to prove the antecedents? The simplifiers in both Edinburgh LCF and the Boyer and Moore [1979] Theorem Prover solve antecedents by recursively invoking the simplifier. However, there is no need to commit ourselves; we can pass the proof tactic as an argument. Conversions that attempt to prove instances of the antecedents using a tactic tac are

```
IMP_REW_CONV tac  (|-"A1 ==> (... ==> (An ==>  t==u) ...)")
IMP_REW_FCONV tac  (|-"A1 ==> (... ==> (An ==>  B<=>C) ...)")
```

The standard tactic ASM_REWRITE_TAC uses IMP_REW_CONV and IMP_REW_FCONV, so these conversions perform more checking than REWRITE_CONV and REWRITE_FCONV. They reject rewrites that would obviously loop: any rewrite of the form t==t', where t' is an instance of t. The classic example is the commutative law, x+y==y+x. Unfortunately, the general problem of detecting loops in rewriting systems is difficult (Huet and Oppen [1980]).

5.2. Backwards Chaining

The tactic ASM_REWRITE_TAC does not invoke itself to prove the antecedents of implicative rewrites, since this was slow and often looped in Edinburgh LCF. Instead it uses backwards chaining -- a proof technique that resembles the execution of PPLAMBDA implications as a PROLOG program (Clocksin and Mellish [1981]). Currently it is weaker than PROLOG; it uses one-way matching rather than unification.

We will not examine the implementation of backwards chaining.[6] It is about to be replaced, and depends on advanced tacticals that are beyond the scope of this paper. Let us just see how backwards chaining can solve antecedents of implicative rewrites.

Typically, an antecedent will assert that a list l is defined: ~ l==UU. LCF provides a first-order theory of lists, with NIL, a strict CONS, and an infixed operator APP to append lists. The theory includes theorems asserting that these constants create defined lists:

```
|- ~ NIL == UU
|- ~ a==UU  ==>  (~ l==UU  ==>  ~ CONS a l == UU)
|- ~ l1==UU  ==>  (~ l2==UU  ==>  ~ l1 APP l2 == UU)
```

In a particular proof, you may have assumptions that some terms t1, ..., tn, and some lists l1, ..., lm, are defined. The tactic IMP_SEARCH_TAC, performs a depth-first search using such theorems. It matches a goal B' with the consequent of a theorem |-A==>B, instantiates the theorem to |-A'==>B', and calls itself recursively to prove the antecedent A', proving

[6] However, let me mention that it, too, uses PART_TMATCH — to match the consequent of an implication.

the goal B' by Modus Ponens. In this example, it can establish that any list is defined, provided it is constructed from the constants NIL, CONS, APP, and the terms t1, ..., tn, and the lists l1, ..., lm.

```
|- ~ NIL == UU
|- ~ CONS t1 l2 == UU)
|- ~ (CONS t2 l1) APP (CONS t1 (CONS t2 (CONS t3 NIL))) == UU)
```

5.3. Canonical Forms

A predicate logic such as PPLAMBDA allows many different ways of saying the same thing. For instance, (A/\B)==>C is logically equivalent to A==>(B==>C), though IMP_REW_CONV and IMP_SEARCH_TAC expect the latter. Rather than clutter them with code to handle various forms of input, LCF provides functions to put theorems into a canonical form.[7] These functions are *inference rules*. They do not simply manipulate data structures, but prove their output theorems from their input theorems.

The function IMP_CANON converts a theorem into a list of implications. If its argument is |-A==>B, then it calls itself to convert A|-B to [A|-B1; ...; A|-Bn], then discharges A to return [A==>B1; ...; A==>Bn]. Otherwise it calls itself recursively using the rules

```
A /\ B          --->    A, B
(A/\B) ==> C    --->    A==>(B==>C)
(?x.A) ==> B    --->    A[x'/x] ==> B    (a variant of x not used elsewhere)
!x.A            --->    A[x'/x]
```

[7] Edinburgh LCF provided a standard canonical form. For example, if you attempted to construct the formula "A/\TRUTH" or "TRUTH==>A", it would return just A. This "formula identification" made it difficult to compute reliably with formulas. In Cambridge LCF, the use of canonical forms is under the user's control.

Figure 12 shows IMP_CANON converting several different theorems into the same canonical form. It strips off quantifiers and splits apart conjunctions, returning a list of theorems of the shape

```
|- "A1 ==> (... ==> (An ==> B) ...)"
```

Such implications are fine for IMP_REW_CONV and IMP_SEARCH_FCONV, but the formula conversion IMP_REW_FCONV expects B to have the form C<=>D, which does not appear often in practice. The inference rule FCONV_CANON alters the consequent B using the rules

```
P(x)     --->    P(x) <=> TRUTH()        P a predicate symbol
~ P(x)   --->    P(x) <=> FALSITY()
C<=>D    --->    unchanged
else fail
```

Thus FCONV_CANON proves logical equivalences that rewrite predicates to TRUTH() and negated predicates to FALSITY(). It passes on any formula

```
#IMP_CANON (ASSUME "!x. ~ x==UU:* ==> !y. ~ y==UU:* ==> ~ f x y ==UU:*");;
[.|-"~ x == UU ==> ~ y == UU ==> ~ f x y == UU"] : thm list

#IMP_CANON (ASSUME "!x y. ~ x==UU:* ==> ~ y==UU:* ==> ~ f x y ==UU:*");;
[.|-"~ x == UU ==> ~ y == UU ==> ~ f x y == UU"] : thm list

#IMP_CANON (ASSUME "(~ x==UU:* /\ ~ y==UU:*) ==> ~ f x y ==UU:*");;
[.|-"~ x == UU ==> ~ y == UU ==> ~ f x y == UU"] : thm list

#IMP_CANON (ASSUME "(~ x==UU:* /\ ~ y==UU:*) ==>
#                  !f g. ~ f x y ==UU:* /\ ~ g x y == UU:*");;
[.|-"~ x == UU ==> ~ y == UU ==> ~ f x y == UU";
 .|-"~ x == UU ==> ~ y == UU ==> ~ g x y == UU"]
: thm list
```

Figure 12. IMP_CANON putting theorems into canonical form

rewrites it encounters.

5.4. The Primitive Conversion Tactic

Though there are many different ways of building a formula conversion, there are only a few ways to reduce a goal using a conversion. The tactic (FCONV_TAC fconv) uses fconv to convert a goal A to a new one B, leaving its assumptions unchanged. If the goal B is just TRUTH(), then FCONV_TAC has achieved the goal A, and returns an empty sub-goal list.

5.5. The Rewriting Tactic

The tactic REWRITE_TAC requires all the above pieces. It puts its input, a list of theorems, into canonical form using IMP_CANON and FCONV_CANON. It handles implicative rewrites using the conversions IMP_REW_CONV and IMP_REW_FCONV. It combines these conversions, along with BETA_CONV, using FIRST_CONV, FIRST_FCONV, and BASIC_FCONV. It solves antecedents of implicative rewrites by backwards chaining, using the tactic IMP_SEARCH_TAC. To solve trivial sub-goals that arise during backwards chaining, it adds the reflexivity axiom EQ_REFL (!x.x==x) to the list of input theorems.

The tactic ASM_REWRITE_TAC calls REWRITE_TAC, adding the goal's assumptions to the input list of theorems. Figure 13 shows the definitions of both tactics.


```
let REWRITE_TAC thl =
    let thms = flat (map IMP_CANON thl) in
    let chain_tac = IMP_SEARCH_TAC (EQ_REFL . thms) in
    let conv =
        FIRST_CONV (mapfilter (IMP_REW_CONV chain_tac) thms)
        ORELSEC  BETA_CONV
    and fconv =
        FIRST_FCONV
        (mapfilter ((IMP_REW_FCONV chain_tac) o FCONV_CANON) thms)
    in
    FCONV_TAC (BASIC_FCONV conv fconv);;

let ASM_REWRITE_TAC thl =
    ASSUM_LIST (\asl. REWRITE_TAC (asl @ thl));;
```

Figure 13. Definitions of the rewriting tactics

6. Examples of Solving Goals by Rewriting

To see ASM_REWRITE_TAC in use, consider a proof of mine (Paulson [1983]). It uses a theory of combinator expressions and concerns infixed functions OCCS and OCCS_EQ. The relation "t1 OCCS t2" searches t2 for an occurrence of t1, returning TT if it finds one. The relation OCCS_EQ is the reflexive closure of OCCS. Figure 14 shows their definitions in LCF.

We will see how ASM_REWRITE_TAC helps to prove that the relation OCCS is transitive:

```
!ta. ~ ta==UU  ==>
!tb. ta OCCS tb == TT  ==>
!tc. tb OCCS tc == TT  ==>  ta OCCS tc == TT
```

Inducting on the variable "tc" yields four sub-goals (Figure 15). Compare these with the axiom defining OCCS; three of them contradict the antecedent, tb OCCS tc == TT. Using the axioms OCCS_CLAUSES and OCCS_EQ,

```
let OCCS_EQ =
new_axiom (`OCCS_EQ`,
   "!t t2. t OCCS_EQ t2  ==  (t=t2) OR (t OCCS t2)");;


let OCCS_CLAUSES =
new_axiom (`OCCS_CLAUSES`,
"!t. t OCCS UU == UU
   /\
   (!c. ~ c==UU  ==>
      t OCCS (CONST c) == FF)
   /\
   (!v. ~ v==UU  ==>
      t OCCS (VAR v) == FF)
   /\
   (!t1 t2. ~ t1==UU  ==> ~ t2==UU  ==>
      t OCCS (COMB t1 t2) == (t OCCS_EQ t1) OR (t OCCS_EQ t2))");;
```

Figure 14. Axioms for the infix functions OCCS_EQ and OCCS

the tactic ASM_REWRITE_TAC solves the three easy goals. First, it splits the axiom OCCS_CLAUSES into its components. While rewriting the goal involving CONST, it notices the assumption ~c==UU, and rewrites tb OCCS (CONST c) to FF. Then it rewrites the antecedent FF==TT to FALSITY(). Similarly, it rewrites the consequent to FALSITY(), yielding the tautologous goal FALSITY() ==> FALSITY(). ASM_REWRITE_TAC solves the goals involving UU and VAR in the same way.

The fourth goal is harder to solve, but the tactic advances it considerably (Figure 16). My paper (Paulson [1983]) describes the rest of the proof, involving a cases split followed by a further call to ASM_REWRITE_TAC.


```
#expand (TERM_TAC "tc");;
OK..
4 subgoals
"tb OCCS (COMB t1 t2) == TT  ==>  ta OCCS (COMB t1 t2) == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
    [ "tb OCCS t1 == TT  ==>  ta OCCS t1 == TT" ]
    [ "tb OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
    [ "~ t1 == UU" ]
    [ "~ t2 == UU" ]

"tb OCCS (VAR v) == TT  ==>  ta OCCS (VAR v) == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
    [ "~ v == UU" ]

"tb OCCS (CONST c) == TT  ==>  ta OCCS (CONST c) == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
    [ "~ c == UU" ]

"tb OCCS UU == TT  ==>  ta OCCS UU == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
```

Figure 15. Subgoals after structural induction

```
"((tb = t1) OR (tb OCCS t1)) OR
 ((tb = t2) OR (tb OCCS t2))      == TT
 ==>
 ((ta = t1) OR (ta OCCS t1)) OR
 ((ta = t2) OR (ta OCCS t2))      == TT"
    [ "~ ta == UU" ]
    [ "ta OCCS tb == TT" ]
    [ "tb OCCS t1 == TT  ==>  ta OCCS t1 == TT" ]
    [ "tb OCCS t2 == TT  ==>  ta OCCS t2 == TT" ]
    [ "~ t1 == UU" ]
    [ "~ t2 == UU" ]
```

Figure 16. The sub-goal that remains after calling ASM_REWRITE_TAC

7. Conclusions

I can imagine you thinking, "This is all very elegant, but it must be hopelessly inefficient." In fact, conversions are efficient enough to help perform complex proofs. ASM_REWRITE_TAC can simplify a typical goal, involving twenty rewrite rules, in twenty to forty seconds on a VAX 750 computer.

The efficiency could be improved, though not easily. Fast simplifiers (Boyer and Moore [1979]) simplify a term relative to an environment of variable bindings, to minimise the number of substitutions. Implementing this in LCF would require that the tactic for solving antecedents of implicative rewrites could also see this environment. I have been experimenting with discrimination nets for simultaneous pattern matching (Charniak, Riesbeck, McDermott [1980]). We could gain efficiency all round by implementing an ML compiler that generated machine instructions instead of Lisp (Cardelli [1983]).

Conversion functions have many advantages over Edinburgh LCF's simplifier, a seven-page ML program. The various operators, PART_FMATCH, REWRITE_CONV, TAUT_CONJ_CONV, IMP_CANON, carry out small, well-defined tasks. They have simple specifications and implementations. Together they can express the rewriting tactic, ASM_REWRITE_TAC, in only a dozen lines.

Its modularity makes ASM_REWRITE_TAC easy to extend, and, more importantly, easy to comprehend. This tactic performs the vast majority of inferences in a proof. Proofs are supposed to make sense to a human reader as a summary of the formal manipulations. It is essential that ASM_REWRITE_TAC should denote a uniform, simple proof strategy, not an ad-hoc bunch of tricks.

Morris [1981] has asked whether real programming in functional languages is possible. The answer must be "yes", since LCF contains five thousand lines of functional code, written in ML. This includes a few imperative functions, to print on the terminal and to declare constants and axioms. There are functions to interactively manage a "proof state" -- a stack of unsolved sub-goals. While functional programming means relying less on the state, it does not seem necessary or desirable to do away with the state completely.

These rewriting tools illustrate the power of higher-order functions. Because ML treats functions as first-class data, we can implement rewriting tools as functions and even write functions, such as THENC, to combine them. Proof tactics are functions too; the conversion IMP_REW_CONV generates and proves sub-goals using a tactic passed to it as an argument. The instantiation function PART_TMATCH accepts a function argument that tells it what part of a theorem to match.

This flexibility is essential in an experimental system like LCF. If the techniques for proving theorems were thoroughly understood, then we would know which tactic IMP_REW_CONV required, and would not need to pass a tactic as an argument. But we know little about proving theorems. Using a functional language we can postpone design decisions and make our proof strategies as general as possible.

References

R. Boyer, J. Moore. A Computational Logic. Academic Press, 1979.

L. Cardelli. "The Functional Abstract Machine." Polymorphism: the ML/LCF/HOPE Newsletter. Bell Laboratories, Murray Hill, New Jersey, January 1983.

E. Charniak, C. Riesbeck, D. McDermott. Artificial Intelligence Programming. Lawrence Erlbaum Associates, 1980.

W. Clocksin, C. Mellish. Programming in Prolog. Springer-Verlag, 1981.

M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth. "A Metalanguage for Interactive Proof in LCF." Pages 119-130, Fifth ACM Symposium on Principles of Programming Languages, 1978.

M. Gordon, R. Milner, C. Wadsworth. Edinburgh LCF. Springer-Verlag, 1979.

G. Huet, D. Oppen. "Equations and Rewrite Rules: A Survey." in R. Book (Editor), Formal Languages: Perspectives and Open Problems. Academic Press, 1980.

J. Morris. "Real Programming in Functional Languages." Technical Report CSL-81-11. Xerox Palo Alto Research Center, 1981.

L. Paulson. "Recent Developments in LCF: Examples of Structural Induction." Technical Report No. 34, Computer Laboratory, University of Cambridge, 1983.

UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

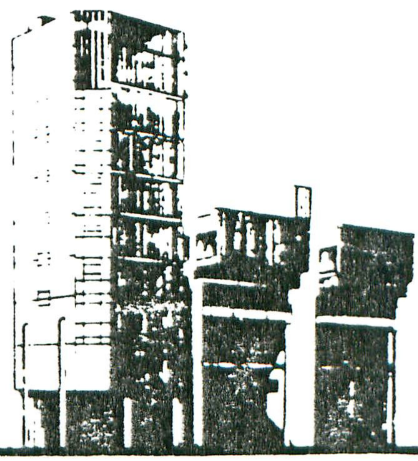
Technical Report No. 36

THE REVISED LOGIC PPLAMBDA

A REFERENCE MANUAL

by

Lawrence Paulson



The Revised Logic PPLAMBDA[1]

A Reference Manual

Lawrence Paulson

Cambridge University

March 1983

Abstract

PPLAMBDA is the logic used in the Cambridge LCF proof assistant. It allows Natural Deduction proofs about computation, in Scott's theory of partial orderings. The logic's syntax, axioms, primitive inference rules, derived inference rules, and standard lemmas are described, as are the LCF functions for building and taking apart PPLAMBDA formulas.

PPLAMBDA's rule of fixed-point induction admits a wide class of inductions, particularly where flat or finite types are involved. The user can express and prove these type properties in PPLAMBDA. The induction rule accepts a list of theorems, stating type properties to consider when deciding whether to admit an induction.

---

[1] Research supported by S.E.R.C. Grant number GR/B67766.

Table of Contents

The Revised Logic PPLAMBDA

A Reference Manual

Lawrence Paulson

Cambridge University

March 1983

1. Introduction

The proof assistant LCF is an interactive computer program that helps a user prove theorems and develop theories about computable functions, using a logic called PPLAMBDA. It can reason about non-terminating computations, arbitrary recursion schemes, and higher-order functions, by virtue of Scott's theory of continuous partial orders (Stoy [1977]). PPLAMBDA uses standard natural deduction rules (Dummet [1977]).

The version known as Edinburgh LCF (Gordon, Milner, Wadsworth [1979]) has been used for many projects, for example, Cohn [1982, 1983]. Cambridge LCF (Paulson [1983]) is a descendant of Edinburgh LCF. Though based on the same principles, the new system is quite different from the old one. In particular, the logic PPLAMBDA has been revised to include disjunction, existential quantifiers, and predicates.

Some notes of caution: Cambridge LCF is still in a state of flux. The revised PPLAMBDA has been stable for only a few months. This report is largely self-contained, but you may wish to refer to Gordon et al. [1979] for background information. Please notify me of any major errors you discover, particularly in the section on fixed-point induction.

I would like to thank Mike Gordon for his many comments and corrections regarding this paper.

2. Syntax

In this paper, syntactic meta-variables obey the following conventions, possibly subscripted:

name	PPLAMBDA construct
x,y,z	variables
t,u,v	terms
A,B,C	formulas
P,Q	predicate symbols
ty	types

Standard types

void	type containing only one element
tr	type of truth-values: TT, FF, UU
ty1 # ty2	Cartesian product of ty1 and ty2 -- actually ":(ty1,ty2)prod"
ty1 -> ty2	continuous functions from ty1 to ty2 -- actually ":(ty1,ty2)fun"

Terms

c	constant, where c is a constant symbol
x	variable
\x.t	lambda-abstraction over a term
t u	combination (application of function to argument)
p => t | u	conditional expression -- actually "COND p t u"
t,u	ordered pair -- actually "PAIR t u"

Standard constants

UU:*	bottom element for partial ordering
TT:tr	truth-value "true"
FF:tr	truth-value "false"
FIX:(* -> *) -> *	fixed-point operator
COND:(tr-> * -> * -> *)	function for making conditional expressions
PAIR:(* -> ** -> (*#**))	function to construct an ordered pair
FST:(* # **) -> *	selector for the first element of a pair
SND:(* # **) -> **	selector for the second element of a pair
(): void	sole element of the type ":void"

Formulas

TRUTH()	tautology
FALSITY()	contradiction
t == u	equality of t and u -- actually "equiv(t,u)"
t << u	Scott partial ordering -- actually "inequiv(t,u)"
P t	where P is a predicate symbol
!x.A	universal quantifier
?x.A	existential quantifier
A /\ B	conjunction
A \/ B	disjunction
A ==> B	implication
A <=> B	if-and-only-if
~A	negation -- actually "A ==> FALSITY()"

3. Functions for Manipulating PPLAMBDA Objects

3.1. Abstract Syntax Primitives

LCF provides functions to construct, test the form of, and take apart PPLAMBDA terms, formulas, and types. These use standard naming conventions.

Prefixes:

mk	make an object (term, formula, type)
is	test that an object has a given top-level constructor
dest	take apart an object, yielding its top-level parts

Suffixes for terms

const	constant
var	variable
abs	abstraction
comb	combination
pair	ordered pair
cond	conditional expression

Suffixes for formulas

equiv	equivalence of terms
inequiv	inequivalence of terms
forall	universal quantifier
exists	existential quantifier
conj	conjunction
disj	disjunction
imp	implication
iff	if-and-only-if
pred	predicate

For example, there are three basic functions for manipulating universal quantifiers:

```
mk_forall : (term # form) -> form
is_forall : form -> bool
dest_forall : form -> (term # form)
```

3.2. [illegible]

[illegible] functions involving lists. Unless stated otherwise, n denotes any non-negative integer.

Constructors:

```
list_mk_abs     ["x1";...;"xn"], "t"    --->  "\x1 ... xn.t"
list_mk_comb    "t", ["u1";...;"un"]    --->  "t u1 ... un"
list_mk_conj    ["A1";...;"An"]   --->  "A1 /\ ... /\ An",     n>0
list_mk_disj    ["A1";...;"An"]   --->  "A1 \/ ... \/ An",     n>0
list_mk_imp     ["A1";...;"An"], "B"    --->  "A1 ==> ... ==> An ==> B"
list_mk_forall  ["x1";...;"xn"], "A"    --->  "!x1 ... xn.A"
list_mk_exists  ["x1";...;"xn"], "A"    --->  "?x1 ... xn.A"
```

Destructors:

```
strip_abs       "\x1 ... xn. t"    --->  ["x1";...;"xn"], "t"
strip_comb      "t u1 ... un"    --->  "t", ["u1";...;"un"]
conjuncts       "A1 /\ ... /\ An"    --->  ["A1";...;"An"]
disjuncts       "A1 \/ ... \/ An"    --->  ["A1";...;"An"]
strip_imp       "A1 ==> ... ==> An ==> B"  --->  ["A1";...;"An"], "B"
strip_forall    "!x1 ... xn. A"    --->  ["x1";...;"xn"], "A"
strip_exists    "?x1 ... xn. A"    --->  ["x1";...;"xn"], "A"
```

3.3. Functions Concerning Substitution

These functions are similar to those that Appendix 7 of Gordon et al. [1979] describes in detail. This summary is for the sake of completeness.

Choosing a variant of a variable

variant : (term list) -> term -> term

Generating a new variable (distinct from any already in use)

genvar : type -> term

Returning all variables in a PPLAMBDA object

```
term_vars: term -> term list
form_vars: form -> term list
forml_vars: (form list) -> term list
```

Returning the free variables in a PPLAMBDA object

```
term_frees: term -> term list
form_frees: form -> term list
forml_frees: (form list) -> term list
```

Returning the type variables in a PPLAMBDA object

```
type_tyvars: type -> type list
term_tyvars: term -> type list
form_tyvars: form -> type list
forml_tyvars: (form list) -> type list
```

Testing if two terms/formulas are alpha-convertible

```
aconv_term: term -> term -> bool
aconv_form: form -> form -> bool
```

Testing if one type/term/formula occurs (free) in another

```
type_in_type: type -> type -> bool
type_in_term: type -> term -> bool
type_in_form: type -> form -> bool

term_freein_term: term -> term -> bool
term_freein_form: term -> form -> bool

form_freein_form: form -> form -> bool
```

Substitution in a term/formula (at specified occurrence numbers)

```
subst_term: (term # term)list -> term -> term
subst_form: (term # term)list -> form -> form

subst_occs_term: ((int list)list) -> (term#term)list -> term -> term
subst_occs_form: ((int list)list) -> (term#term)list -> form -> form
```


Instantiation of types in a PPLAMBDA object

```
inst_type: (type # type)list -> type -> type
inst_term: (term list) -> (type # type)list -> term -> term
inst_form: (term list) -> (type # type)list -> form -> form
```

May prime variables, avoiding those given in the (term list) arguments.

4. Axioms and Basic Lemmas

The axioms of Scott theory (Igarashi [1972]) are bound to ML identifiers.

Standard Tautology

```
TRUTH          TRUTH()
```

Partial ordering

```
LESS_REFL      !x. x<<x

LESS_ANTI_SYM  !x y. x<<y  /\  y<<x  ==>  x==y

LESS_TRANS     !x y z. x<<y  /\  y<<z  ==>  x<<z
```

Monotonicity of function application

```
MONO           !f g x y. f<<g  /\  x<<y  ==>  f x << g y
```

Extensionality of <<

```
LESS_EXT       !f g. (!x. f x << g x)  ==>  f<<g
```

Minimality of UU

```
MINIMAL        !x. UU<<x
```

Conditional expressions

```
COND_CLAUSES
                !x y. UU => x | y == UU  /\
                      TT => x | y == x   /\
                      FF => x | y == y
```

Truth values

```
TR_CASES        !p:tr. p==UU  \/  p==TT  \/  p==FF

TR_LESS_DISTINCT
  ~ TT<<FF  /\  ~ FF<<TT  /\  ~ TT<<UU  /\  ~FF<<UU
```

Ordered pairs

```
MK_PAIR         !x. (FST x, SND x) == x

FST_PAIR        !x y. FST (x,y) == x

SND_PAIR        !x y. SND (x,y) == y
```

Fixed points

```
FIX_EQ          !f. FIX f == f (FIX f)
```

There is one axiom scheme: beta-conversion. If x is a variable, and u, v are terms, and u[v/x] denotes the substitution of v for x in u, then

```
BETA_CONV "(\x.u)v"   returns   |-(\x.u)v == u[v/x]
```

LCF includes some basic lemmas that follow from the axioms.

Equality

```
EQ_REFL         !x. x==x

EQ_SYM          !x y. x==y  ==>  y==x

EQ_TRANS        !x y z. x==y  /\  y==z  ==>  x==z
```


Extensionality of ==

```
EQ_EXT          !f g. (!x. f x == g x)  ==>  f==g
```

Distinctness of the truth values

```
TR_EQ_DISTINCT
                ~ TT == FF  /\  ~ FF == TT  /\
                ~ TT == UU  /\  ~ UU == TT  /\
                ~ FF == UU  /\  ~ UU == FF
```

The completely undefined function

```
MIN_COMB        !x. UU x == UU

MIN_ABS         \x.UU == UU
```

Validity of Eta-Conversion

```
ETA_EQ          !f. \x.f x == f
```

5. Predicates

In Cambridge LCF, you can introduce predicate symbols. A predicate can be axiomatised abstractly, or as an abbreviation for a long formula. Examples:

```
STRICT f    <=>   f UU == UU


TRANSITIVE p    <=>
   !x y z. p x y == TT  /\  p y z == TT  ==>  p x z == TT
```

PPLAMBDA's type system allows these axioms to refer to the types of the operands of the predicates. There are many examples of predicates that require describe properties of types, not of values. You may adopt the convention of writing UU as the operand when only its type is relevant.

```
FLAT (UU:*)  <=>
        !x1:*. !x2:*. x1<<x2  ==>  UU==x1  \/  x1==x2


ISOMORPHIC (UU:*, UU:**)   <=>
        ?f g. (!x:*. g(f x) == x) /\ (!y:**. f(g y) == y)
```

All predicates have exactly one argument, which may be a tuple of values or the empty value () (read "empty"). In particular, we must write "TRUTH()" and "FALSITY()".

6. Predicate Calculus Rules

These are conventional natural deduction rules (Dummet [1977]). In the notation below, assumptions of a premiss are only mentioned if they will be discharged in that inference. The assumptions of the conclusion include all other assumptions of the premisses. Explicit assumptions are written inside [square brackets].

6.1. Rules for quantifiers

Forall introduction

```
GEN: term -> thm -> thm
       x

      A(a)
    ---------  where the variable "a" is not free in assumptions of premiss
     !x.A(x)
```


Forall elimination

```
SPEC: term -> thm -> thm
        t

         !x.A(x)
      ------------
          A(t)
```

Exists introduction

```
EXISTS: (form # term) -> thm -> thm
                  t

         A(t)
      -------------
        ?x.A(x)
```

You must tell the rule what its conclusion should look like, since it is rarely desirable to replace every t by x. For example, you can conclude two different results from the theorem |-"TT==TT":

```
EXISTS ("?x. x==TT", "TT")  (|-"TT==TT")   --->  |-"?x. x==TT"

or

EXISTS ("?x. x==x", "TT")  (|-"TT==TT")   --->  |-"?x. x==x"
```

Exists elimination

```
CHOOSE: (term # thm) -> thm -> thm
           a

         ?x.A(x)      [ A(a) ] B
      ------------------------------
                   B
```

where the variable "a" is not free anywhere except in B's assumption A(a)

6.2. Rules for basic connectives

Conjunction introduction

```
CONJ: thm -> thm -> thm

      A        B
  ----------------
       A /\ B
```

Conjunction elimination

```
CONJUNCT1, CONJUNCT2: thm -> thm

      A /\ B
  --------------
    A         B
```

Disjunction introduction

```
DISJ1: thm -> form -> thm
DISJ2: form -> thm -> thm

      A               B
  -----------    -------------
    A \/ B         A \/ B
```

Disjunction elimination

```
DISJ_CASES: thm -> thm -> thm -> thm

      A \/ B      [ A ] C      [ B ] C
  ----------------------------------------
                     C
```


Implication introduction

```
DISCH: form -> thm -> thm
        A

      [ A ]  B
  --------------
      A ==> B
```

Implication elimination

```
MP: thm -> thm -> thm

      A ==> B        A
  -------------------------
            B
```

6.3. Rules for derived connectives

The formula A <=> B is logically equivalent to (A ==> B) /\ (B ==> A), but LCF does not expand it as such, to avoid duplicating A and B. The rules CONJ_IFF and IFF_CONJ map between the two formulas.

The formula ~A denotes A ==> FALSITY(). The rules for negation are special cases of the rules for implication, and are not provided separately. Any inference rule that works on implications also works on negations.

If-and-only-if introduction

```
CONJ_IFF: thm -> thm

      (A ==> B)  /\   (B ==> A)
  -------------------------------
              A <=> B
```

If-and-only-if elimination

```
IFF_CONJ: thm -> thm

            A <=> B
    ---------------------------
     (A ==> B)  /\  (B ==> A)
```

Negation introduction

```
DISCH: form -> thm -> thm
         A

       [ A ]  FALSITY()
    ---------------------
           ~A
```

Negation elimination

```
MP: thm -> thm -> thm

        ~A      A
    -----------------
       FALSITY()
```

7. Additional rules

Assumption

```
ASSUME: form -> thm
          A

    -------------
      [ A ]  A
```


Contradiction rule

```
CONTR: form -> thm -> thm
         A

       FALSITY()
    --------------
          A
```

Classical contradiction rule

```
CCONTR: form -> thm -> thm
          A

       [ ~A ]  FALSITY()
    ---------------------
            A
```

Intuitionists (Dummet [1977]) can get rid of this rule by typing "let CCONTR=();;". However, PPLAMBDA does not seem suitable for constructive proof. The cases axiom TR_CASES allows dubious instances of the excluded middle. The theory of admissibility for disjunctions and short types, discussed below, seems to rely on classical reasoning.

Simultaneous Substitution

```
SUBST: (thm # term)list -> form -> thm -> thm
                 xi          A(xi)

       ti == ui     A(ti)
    -------------------------
            A(ui)
```

The formula A(xi) serves as a template to control the subtitution; the variables xi mark the places where substitution should occur.

Instantiation of Types

```
INST_TYPE (type # type)list -> thm -> thm
            tyi    vtyi
```

where the type variables vtyi do not occur in the assumptions

```
        A(vtyi)
   -----------------
        A(tyi)
```

Instantiation of Terms

```
INST (term # term)list -> thm -> thm
       ti      xi
```

where the variables xi do not occur in the assumptions

```
        A(xi)
   -----------------
        A(ti)
```

8. Fixed point induction

Fixed-point induction on a variable x and formula A(x) is only sound if the formula A is "chain-complete" with respect to x. For any ascending chain of values z1, z2, ..., if A(zi) is true for every zi, then A(z) must hold for the least upper bound, z. In Scott's original logic, the only formulas are conjunctions of inequivalences, which are all chain-complete. Things are more complicated in PPLAMBDA, with its implications, disjunctions, quantifiers, and user-definable predicates.

8.1. Admissibility for short types

Igarashi [1972] considered admissibility in a logic containing all these connectives, but his admissibility test can be considerably liberalised.

An important special case is that all structural inductions over flat types are admissible.

Definition: A short type is one with no infinite ascending chains.[2]

Suppose we wish to prove !x:ty.A(x) by structural induction, where the type "ty" is short. This requires computation induction on a variable f and formula !z:ty.A(f z). This formula is chain-complete in f:

> Suppose that f is the limit (least upper bound) of an ascending chain f0, f1, ...

(1) Suppose that !z.A(fi z) holds for all i.

> Then the limit case !z.A(f z) holds also, for consider any z'. Since the type of "fi z'" is short, the chain (f0 z'), (f1 z'), ... reaches its limit at some finite i.[3] For this i, "fi z'" equals "f z'".
>
> Our assumption (1) implies that A(fi z') holds, so A(f z') holds too. Since we chose z' arbitrarily, we conclude !z.A(f z). Thus the induction is admissible.

From this argument it appears that the admissibility test may be liberalised to allow any occurrence of the induction variable within some term of short type, with restrictions on what variables the term may contain. If

---

[2] Gordon et al. [1979] call these "easy" types.

[3] The intuitionistic validity of this inference is questionable, as is the justification of the admissibility rule for disjunctions. Both rely on the "pigeon-hole principle": if you partition an infinite set in two, one of the two sets must be infinite.

the term contains existentially quantified variables, the formula may not be chain-complete.

Example:

?z.f z==UU, where f maps every natural number to "TT". Suppose that for all i, fi maps all numbers less than i to TT, the rest to UU. Then f is the limit of the fi, the formula holds for each fi, and the formula does not hold in the limit.

LCF allows induction whenever the above term contains only constants, free variables, and outermost universally quantified variables. The test ignores quantifiers over finite types, as these are essentially finite disjunctions or conjunctions. The test also notices the special cases where free occurences of t<<u or t==u are chain-complete, as discussed on page 77 of Gordon et al. [1979]. It treats t==UU as the equivalent formula t<<UU, which is chain-complete in t in both positive and negative positions.

8.2. Stating type properties in PPLAMBDA

LCF recognises certain theorems that state that a type is finite or short. Any theorem

```
|- !x:ty. x==c1  \/  ...  \/  x==cn
```

where the ci are constants, states that the type "ty" is finite. Any theorem

```
|- !x1 ... xn:ty. x1<<x2  /\  ...  /\  x(n-1)<<xn  ==>
      UU==x1  \/  x1==x2  \/  ...  \/  x(n-1)==xn
```


states that the type "ty" is short. When n=2 this is the familiar flatness property:

```
!x1 x2. x1<<x2  ==>  UU==x1  \/  x1==x2
```

To inform LCF of such properties when checking admissibility, the induction rule accepts a list of theorems, B1, ..., Bn. Each Bi should state the finiteness or shortness of a type.

Scott Fixed-Point Induction

```
INDUCT: (term list) -> (thm list) -> (thm # thm) -> thm
             funi             Bi

        B1 ... Bn      A(UU)      !f1 ... fn. A(fi)  ==>  A(funi fi)
   ------------------------------------------------------------------
                          A(FIX funi)
```

9. Derived Inference Rules

For your convenience, LCF provides inference rules that can be derived from the primitive rules of PPLAMBDA. A few of these are wired in for efficiency, but most derive their conclusions by proper[4] inferences.

9.1. Predicate Calculus Rules

---

[4] Intuitionists will be glad to hear that none use the classical contradiction rule, CCONTR.

Substitution (at specified occurrence numbers)

```
SUBS: (thm list) -> thm -> thm
SUBS_OCCS: ((int list) # thm) list -> thm -> thm

   ti == ui        A(ti)
  ----------------------
          A(ui)
```

Generalising a theorem over its free variables

```
GEN_ALL: thm -> thm

           A(xi)
  ----------------------
      !x1...xn.A(xi)
```

Discharging all hypotheses

```
DISCH_ALL: thm -> thm

      [A1; ...; An] B
-------------------------------
   A1 ==> ... ==> An ==> B
```

Iterated SPEC

```
SPECL: (term list) -> thm -> thm
            ti

      !x1 ... xn. A(xi)
  -------------------------  SPECL [t1; ...; tn]
           A(ti)
```

Un-discharging an assumption

```
UNDISCH: thm -> thm

       A  ==>  B
  ----------------
        [A] B
```


Undischarging all assumptions

```
UNDISCH_ALL: thm -> thm

      A1 ==> ... ==> An ==> B
  ------------------------------
         [A1; ...; An] B
```

Specialisation over outer universal quantifiers

```
SPEC_ALL: thm -> thm

      !x1 ... xn. A[xi]
  ------------------------   where the xi' are not free in hyps of A
         A[xi'/xi]
```

Using a theorem A to delete a hypothesis of B

```
PROVE_HYP: thm -> thm -> thm
             A      B

      A        [A] B
  ---------------------
           B
```

Conjoining a list of theorems

```
LIST_CONJ: (thm list) -> thm
                Ai

       A1 ... An
  -------------------          where n>0
    A1 /\ ... /\ An
```

Splitting a theorem into its conjuncts

```
CONJUNCTS: thm -> (thm list)

      A1 /\ ... /\ An
  -------------------          where n>0
        A1 ... An
```

Iterated Modus Ponens

```
LIST_MP: (thm list) -> thm -> thm
                 Ai

   A1 ... An       A1 ==> ... ==> An ==> B
  -------------------------------------------
                    B
```

Contrapositive of an implication

```
CONTRAPOS: thm -> thm

    A ==> B
  -------------
   ~B ==> ~A
```

Converting disjunction to implication

```
DISJ_IMP: thm -> thm

    A \/ B
  ------------
   ~ A ==> B
```

```
DISJ_CASES_UNION: thm -> thm -> thm -> thm

     A \/ B       [A] C      [B] D
  -----------------------------------
                C \/ D
```

9.2. Rules About Functions and the Partial Ordering

These are mostly the same as in Gordon et al. [1979], sometimes with different spellings. I retain the convention that =< stands for either of the relations == or <<, the same at each occurence within a rule unless otherwise stated.

Reflexivity of equality

```
REFL: term -> thm
    "t"   --->    |-"t==t"
```

Symmetry of equality

```
SYM: thm -> thm

     t==u
  ----------
     u==t
```

Analysis of equality

```
ANAL: thm -> thm

            t == u
  ------------------------
    t << u  /\  u << t
```

Synthesis of equality

```
SYNTH: thm -> thm

    t << u  /\  u << t
  ------------------------
          t == u
```

Transitivity (infix operator)

```
TRANS: thm -> thm -> thm

    t =< u      u =< v       possibly different relations
  ------------------------
          t =< v             << unless both hypotheses use ==
```

Extensionality

```
EXT: thm -> thm

     !x. u x =< v x
  --------------------
        u =< v
```

Minimality of UU

```
MIN: term -> thm
       t

   "t"  --->  |- "UU << t"
```

```
LESS_UU_RULE: thm -> thm

      t<<UU
   -----------
      t==UU
```

Construction of a combination

```
LE_MK_COMB: (thm # thm) -> thm

        f =< g
        t =< u
   --------------
     f t =< g u         << unless both hypotheses use ==
```

Application of a term to a theorem

```
AP_TERM: term -> thm -> thm
          t

       u =< v
   --------------
     t u =< t v
```

Application of a theorem to a term

```
AP_THM: thm -> term -> thm
                 t

       u =< v
   --------------
     u t =< v t
```

Construction of an abstraction

```
MK_ABS: thm -> thm

      !x. u =< v
   ----------------
     \x.u =< \x.v
```

```
HALF_MK_ABS: thm -> thm

      !x. u x =< t
   ----------------
       u =< \x.t
```

Alpha-conversion (renaming of bound variable)

```
ALPHA_CONV: term -> term -> thm
             x      (\y.t)

   --------------------------
     \y.t == \x. t[x/y]
```

10. Differences from Edinburgh LCF

The obvious differences are that PPLAMBDA in Cambridge LCF provides the existential quantifier, the disjunction, negation, and if-and-only-if symbols, and predicate symbols. It includes the standard contradiction FALSITY(), instead of expressing contradiction through formulas such as "TT==FF" or "FF<<UU".

However, the new PPLAMBDA is not just an extension of the old. Its syntax has changed to use /\ instead of &, and ==> instead of IMP. The ML names and types of many of the inference rules have changed. There are other, more subtle differences.

10.1. Formula Identification

Edinburgh LCF forced every formula into a canonical form. For instance, you could not build the formulas "!x.TRUTH()" and "A==>TRUTH()". The constructor functions mk_forall and mk_imp automatically simplified these to TRUTH().[5] This "formula identification" caused unpredictable behavior in programs that manipulated formulas.

Cambridge LCF does not have formula identification. Instead, you can implement your own canonical forms in ML. The constructor and destructor functions are inverses of each other. For instance,

```
dest_conj (mk_conj (A, B))  --->  (A, B)
```

10.2. The Definedness Function DEF

Edinburgh LCF provided a function DEF, satisfying

```
DEF UU == UU
DEF x == TT            for any x except UU
```

The formula "DEF x == TT" asserts that x is defined. However, it is easier to write "~ x==UU". DEF is no longer provided, though you can easily axiomatise it yourself.

[5] Here I am using the notation of Cambridge LCF, though describing Edinburgh LCF.

10.3. Data Structures

In Edinburgh LCF, data structures were axiomatised using sum, product, and lifted types. This was originally done manually, and later by Milner's structural induction package (Cohn and Milner [1982]).

In Cambridge LCF, data structures can be axiomatised using disjunction and existential quantifiers. A descendant of Milner's package introduces the axioms automatically. The sum and lifted types have been removed, along with the operators UP, DOWN, INL, INR, ISL, OUTL, OUTR (for sum types) and UP, DOWN (for lifted types). The structural induction package makes it easy to define such type operators.

References

A. Cohn, R. Milner. "On using Edinburgh LCF to prove the correctness of a parsing algorithm." Technical Report CSR-113-82, University of Edinburgh, 1982.

A. Cohn. "The correctness of a predecence parsing algorithm in LCF." Technical Report No. 21, University of Cambridge, 1982.

A. Cohn. "The Equivalence of Two Semantic Definitions: A Case Study in LCF." SIAM Journal of Computing, May 1983.

M Dummet. Elements of Intuitionism. Oxford University Press, 1977.

M. Gordon, R. Milner, C. Wadsworth. Edinburgh LCF. Springer-Verlag, 1979.

S. Igarashi. "Admissibility of Fixed-Point Induction in First Order Logic of Typed Theories," Memo AIM-168, Stanford University, 1972.

L. Paulson. "Recent Developments in LCF: Examples of Structural Induction." Technical Report No. 34, Computer Laboratory, University of Cambridge, 1983.

J. Stoy. Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.

ML UNDER EUNICE

C. Kitchen & B. Lynch
2 December 1982
Trinity College Dublin

It is possible to link the UNIX version of ml under EUNICE. The purpose of this note is to facilitate this process for other ml users or possible users who are running EUNICE rather than true UNIX.

There are only two changes to make. One is to modify all the occurrences of all symbols in the following list:

```
___Array
___Closure
___Collect
___Composition
___Cons
___ConsRecord
___ConsVariant
___CopyStream
___Define
___DisArray
___DumClosure
___EmptyStream
___Explode
___ExplodeAscii
___FailWith
___GetStream
___Identity
___Implode
___ImplodeAscii
___InChar
___InInt
___InString
___IntToString
___NewStream
___OutChar
___OutInt
___OutString
___Pair
___PutStream
___Ref
___StringToInt
___Sub
___Tabulate
___Update
___ValidTP
```

These all occur in the file "mlrout.s". Treble underscores all need to become single underscores.

The second change is to modify the make file. I give a copy of our modified make file below; the major changes are to replace the "ld" with a "pc", to replace "as" with "unixas", and to replace "cc" with a "cc -s", "unixas" combination to ensure that "cc" does not produce a VMS object file. (The optimisation flags are optional.) If you use a "unixld" under eunice, the image has problems getting started.

```
MAIN  =     mlglob.h  mlmain.h  mlstor.h  mlscan.h  mlpars.h
            mlanal.h  mlcomp.h  mlassm.h  mldebg.h  mlrout.h
            mlfile.h  mlmain.p

SCAN =      mlglob.h  mlscan.h  mlmain.h  mlrout.h
            mlscan.p

PARS =      mlglob.h   mlpars.h  mlmain.h  mldebg.h  mlscan.h
            mlfile.h mlpars.p

ANAL =      mlglob.h  mlanal.h  mlmain.h  mlpars.h  mlscan.h
            mlanal.p

COMP =      mlglob.h  mlscan.h  mlcomp.h  mlmain.h  mlanal.h
            mlcomp.p

ASSM =      mlglob.h  mlassm.h  mlmain.h  mlrout.h
            mlassm.p

DEBG =      mlglob.h  mldebg.h  mlmain.h  mlpars.h  mlanal.h
            mlcomp.h   mlassm.h mlrout.h
            mldebg.p

STOR =      mlglob.h  mlstor.h  mlmain.h  mlrout.h
            mlstor.p

SERV =      mlglob.h  mlserv.h  mlmain.h  mldebg.h mlrout.h
            mlalloc.h mlserv.p

ROUT =      mlrout.s

FILE =      mlfile.c

MLSYS =     mlmain.o  mlpars.o  mlanal.o  mlcomp.o  mlscan.o
            mldebg.o  mlassm.o  mlstor.o  mlserv.o  mlrout.o
            mlfile.o


mlsys:      $(MLSYS);  pc -o mlsys -s  $(MLSYS)

mlmain.o:   $(MAIN);   pc -w -c -O   mlmain.p

mlscan.o:   $(SCAN);   pc -w -c -O   mlscan.p

mlpars.o:   $(PARS);   pc -w -c -O   mlpars.p
```

```
mlanal.o:  $(ANAL);   pc -w -c -O   mlanal.p

mlcomp.o:  $(COMP);   pc -w -c -O   mlcomp.p

mlassm.o:  $(ASSM);   pc -w -c -O   mlassm.p

mldebg.o:  $(DEBG);   pc -w -c -O   mldebg.p

mlstor.o:  $(STOR);   pc -w -c -O   mlstor.p

mlrout.o:  $(ROUT);   unixas -o mlrout.o    mlrout.s

mlserv.o:  $(SERV);   pc -w -c -O   mlserv.p

mlfile.o:$(FILE); cc -w -S -O mlfile.c
                        unixas -o mlfile.o mlfile.s
                        rm mlfile.s
```

We are a bit worried that this conversion may have introduced a couple of minor bugs; however, we have not found them yet.

Since the correspondence between versions of ml running and default operating systems is not obvious, may I suggest keeping the default operating system of each site in the sites list.

Addenda to the Mailing List

Roland Backhouse
University of Essex
Department of Computer Science
Wivenhoe Park
Colchester CO4 3SQ
England

Joseph Goguen
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
USA

Allen Matsumoto
Raytheon Co.
Submarine Signal Division
Mail Stop 330
P. O. Box 360
Portsmouth, RI 02871
USA

Brian Monahan
Standard Telecommunication
Laboratories Limited
London Road
Harlow, Essex, CM17 9NA
England

Alberto Pettorossi
IASI-CNR
via Buonarroti 12
00185 Roma
Italy

Simon L Peyton-Jones
Dept of Computer Science
Univerity College London
Gower Street
London WCIE 6BT
England
Tel 01 387 7050

Charles Rattray
Dept of Computing Science
University of Stirling
Stirling FK9 4LA
Scotland

Nabuo Saito
Dept. of Mathematics
Faculty of Science and Engineering
Keio University
3-14-1, Hiyoshi, Kohoku
Yokohama 223
Japan

Jon Schultis
Univ of Colorado
Dept of Computer Science
Boulder, Colorado
USA

Scott Smolka
Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
USA

Satish Thatte
Dept. of Computer and
Communication Sciences
The University of Michigan
221 Angell Hall
Ann Arbor, MI 48109
USA

James Weiner
Univ of New Hampshire
Computer Science Dept
Kingsberry Hall
Durham, NH 03824
USA

Mailing Changes

Old

Colleen Kitchen
Trinity College Dublin
201 Pearse Street
Dublin 2
Ireland

Gordon Plotkin
Room NE43-837
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA

New

Hans-Jurgen Kugler
Trinity College Dublin
201 Pearse Street
Dublin 2
Ireland

Albert Meyer
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA