Contents

Letter from the editors
Cambridge Reports:
    D.C.J. Matthews:
    D.C.J. Matthews:
    J. Fairbairn:
    Lawrence Paulson:
    Lawrence Paulson:
    Lawrence Paulson:
    C. Kitchen and B. Linch:
Addenda to the Mailing List
Mailing Changes

Poly Report
Introduction to Poly
Ponder and its Type System
Recent Developments in LCF:
    Examples of Structural Induction
Rewriting in Cambridge LCF
The Revised Logic PPLambda:
    A Reference Manual
ML under Eunice
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
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
overheads. 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 overheads. 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. INTRODUCTION

Poly was designed to provide a programming system with the same flexibility as a dynamically typed language but without the run-time overheads. 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

constr'(name) is equivalent to

constr( <name> ) (name)

and

<identifier> is equivalent to

<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 digits> | <symbol>

<letter> ::= a|b|c|...|z|A|B|C|...|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/or in comments. The following words have special meaning and cannot be used as ordinary identifiers.
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.
2.2 Procedures

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 preceding 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

```plaintext
proc (tt : type end ; z : t); 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

```plaintext
(type specification) ::= type ((identifier));
```

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) : 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 (l) zero: l; succ: proc(l); end` would match a context with specification `type (l) succ: proc(l); end` but not the other way round. The type specification of a context can be regarded as a filler 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 replaced by `t`'content' (if `t` has such an attribute, i.e. the 'content' procedure of the type is called to return the current value).
3. STATEMENTS AND EXPRESSIONS

\[
\text{<expression> ::= <if expression>}
\]
\[
\quad | \quad <\text{while expression}>
\]
\[
\quad | \quad <\text{infix expression}>
\]
\[
\quad | \quad <\text{raise expression}>
\]

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

\[
\text{<declaration> ::= let <identifier>:= \{ <specification>\} \}
\]
\[
\quad | \quad \text{letrec <identifier>:= \{ <specification>\} \}
\quad | \quad <\text{expression}>
\]

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.

\text{let} and \text{letrec} differ in that \text{letrec} declares the identifier before the expression, making it available inside it, while \text{let} declares the identifiers afterwards. \text{letrec} 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

\[
\text{<if expression> ::= if <expression> then <expression>}
\]
\[
\quad | \quad \text{else <expression>}
\]
\[
\quad | \quad \text{if <expression> then <expression>}
\]

The if expression causes an expression to be executed depending on the
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 then is executed. If the guard is "false" the expression following the else 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 else. An else-part is thus paired with the nearest unpaired then.

### 3.3 Compound Expression

<compound expression> ::= begin
  <expression block>
end

<expression block> ::= {<declaration> | <expression>}

A 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 catch 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> <infix expression>

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 7. 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 (1) : proc (1; 1) 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

\langle name \rangle :: \langle identifier \rangle \mid \langle name \rangle \& \langle identifier \rangle

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 'type' has specification

\begin{verbatim}
type (a)  
x: y: proc(a);  
z: type (x)  
p: proc (z)  
end

then

type $x$ atype $y$ atype $z$ atype $p$
\end{verbatim}

are all valid names.

3.7 Manifests

\langle manifest \rangle :: \langle number \rangle
\mid \langle single-quoted sequence \rangle
\mid \langle double-quoted sequence \rangle

\langle number \rangle :: \langle digit \rangle \langle alphanumeric \rangle

\langle single-quoted sequence \rangle :: \langle any char \rangle*

\langle double-quoted sequence \rangle :: \langle any char \rangle*

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 0x6783 "x" "hello" "am"

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

\begin{verbatim}
convert: proc(string)integer raises conversionerror
\end{verbatim}

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 convertn, convertn or convertn from that type.

3.8 Procedure Constructor

\langle procedure constructor \rangle ::
\langle proc (operator mode)\rangle
\mid \langle proc \rangle [\langle argument list \rangle]
\mid \langle specification \rangle \langle raises (exception list) \rangle
\langle compound expression \rangle*

The procedure constructor states 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

\langle type constructor \rangle ::
\langle type \rangle [\langle identifier \rangle]
\mid \langle declaration \rangle
\mid \langle extends \langle expression \rangle \rangle
\mid \langle declaration \rangle

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

\begin{verbatim}
let newint = type (int) extends integer;
let cube = proc(int):int;
end;
\end{verbatim}

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

\[ \text{union type} ::= \text{union } \{ \text{identifier}\} : \text{specification}\] 

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, \( \text{in}_x \) creates a union from a value of
specification \( T \) and \( \text{proj}_x \) extracts a value of specification \( T \) from
the union. \( \text{in}_x \) is a predicate which is true only if the union was created
with \( \text{in}_x \). \( \text{proj}_x \) is only valid if \( \text{in}_x \) is true, otherwise "projectionerror"
will be raised.

For instance the specification of the union created by \( \text{union}(x; y; z) \)
is

\begin{verbatim}
  type (U)
  in_x, in_y : proc (U) t boolean;
  in_y : proc (T) U;
  proj_x : proc (U) S raises projectionerror;
  proj_y : proc (U) T raises projectionerror
end
\end{verbatim}

Two different identifiers listed with the same specification (e.g.
\( \text{union}(x, y, T) \)) create different variants, so \( \text{proj}_y \) is not allowed on a
union created with \( \text{in}_x \).

3.11 Record Type

\[ \text{record type} ::= \text{record } \{ \text{identifier}\} : \text{specification}\]

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 "\( \text{constr} \)"
which makes a record out of values with the field specifications. For
instance the record created by \( \text{record}(x; y; z) \) has specification

\begin{verbatim}
  type (U)
  x : proc(U) S;
  y : proc(U) T;
  constr : proc(S; T) U
end
\end{verbatim}

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

\[ \text{raise expression} ::= \text{raise } \text{identifier} \]

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.

3.12 While Statement

\[ \text{while expression} ::= \text{while } \text{expression} \text{ do } \text{expression} \]

The while expression \( \epsilon \) uses the expression after the \text{do} to be executed
repeatedly until the \text{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.
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 ivederror;
  succ, pred, neg, abs: proc (i) i raises rangeerror;
  >, >=, req, <, < : proc (i; i) boolean;
  print: proc (i)
end
```

`Integer range` is 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.
A.4 New

Specification
proc [base: type end] (initialval: base)
  type
    assign: proc (base);
    content: proc (base)
  end
  "new" is a procedure which creates and initializes 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.

A.5 Vector

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

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 initialized to the value 'initial' when the array is constructed. The size of the vector must be at least 1 otherwise rangeerror will be raised.

A.6 Char

Specification
proc [c]
  print: proc (c);    
  succ, pred: proc (c); raises rangeerror;
  =, ≮ : proc (c); boolean;
  convert: proc (string); c
end

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

A.7 String

Specification
proc [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, convert, and convert, 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 's' and compared using '≈' and '≠'.

5. REFERENCES

TR79-371 Department of Computer Science, Cornell University, 1979

TR 79-389 Department of Computer Science, Cornell University, 1979

Lecture Notes in Computer Science No. 114, Springer-Verlag, 1981
Technical Report No 29

INTRODUCTION TO POLY

by

D. C. J. Matthews
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., /usr/doc/poly at Cambridge). The compiler will then reply with a prompt (>). To exit from Poly at any time type ctrl-D (end-of-test) 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 bracketing 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.

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 bracketing 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 to 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 message == 
  # proc(m : string)
  # begin
  #   print("The message is ");
  #   print(m)
  # end;
This can be called by typing

> message("Hello");
The message is: Hello
or by typing

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

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.
1. 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'.

\[ ? "message"; \]
\[ message : PROC(string) \]

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

\[ message(message); \]

The message is :Hello

will work. Likewise the call

\[ message("Hi"); \]

The message is :Hi

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

\[ message(message); \]

Error - specifications have different forms

will fail because 'message' has the wrong specification. Incidentally, 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.

2. 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.

\[ \text{print}(42); \]

\[ 42 \]

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

\[ 4*3*2; \]

\[ 16 \]

prints 16 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 (the != operator), greater-than (the > operator), and less-than (the < operator).

\[ \text{let two := 2;} \]

\[ 1 := two; \]

false

\[ 2 := two; \]

true

\[ 3 < 8; \]

true

\[ 8 > 5; \]

false

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

\[ \text{let testtwo := two > 1;} \]

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.

\[ \text{if two = 2} \]

\[ \# \text{then print("It is two")} \]

\[ \# \text{else print("It isn't two")}; \]

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.

\[ \text{let iszero :=} \]

\[ \# \text{proc(integer)} \]

\[ \# \text{(if 1 = 0 then Print("It is zero")} \]

\[ \# \text{else Print("It isn't zero")}; \]

which could then be called to test a value.

\[ \text{iszero(4)}; \]

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
Introduction to Poly

> 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")
>    # y 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

> sq(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(n: integer)integer
>    #   (if n = 1 then 1
>    #    else n*fact(n-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 x = 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*);
>   # end;
> x := prod(x);
> end;

prints the square of all the numbers from 10 down to 1. The body of the loop (the statement after the 'while' word do) is executed repeatedly while the
Introduction to Poly

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 (* : integer)integer (*x*);

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

> sq 3;

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;

define : 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 $.

> integerprint(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. $ selects 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;

define : PROC(integer; integer)rec

END

"construct" 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 rec = rec$construct(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 'rec'.

> rec$a(rec);

3

> rec$b(rec);

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 area =

# 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(x: area)integer

# begin

# let pp = area$p(x);

# pp(area$x(x));

# 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
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 = uniq_inj_int(3);

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

> let stringunion = uniq_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.

> uniq_int(intunion); true

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

> uniq_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'.

> uniqproj_int(intunion); 3

> uniqproj_str(stringunion);

The string

print the original values. It is possible to write

> uniqproj_str(intunion);

Exception projector raised

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

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.

> proc();
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)
# begin
# if x.dlo = 9999
# then dconst( succ(x.dhi), 0)
# else if x.dhi < 0 & (x.dlo = 0)
# then dconst( succ(x.dhi), neg(9999))
# else dconst( x.dhi, succ( x.dlo))
# end;
# let pred :=
# proc(x:d)
# begin
# if x.dlo = neg(9999)
# then dconst( pred(x.dhi), 0)
# else if (x.dhi > 0) & (x.dlo = 0)
# then dconst( pred(x.dhi), neg(9999))
# else dconst( x.dhi, pred( x.dlo))
# end;

Introduction to Poly

# let print :=
# proc(x:dp)
# begin
# if x.dhi < 0
# then begin
# print(x.dhi);
# if abs(x.dlo) < 10
# then print("0000")
# else if abs(x.dlo) < 100
# then print("00")
# else if abs(x.dlo) < 1000
# then print("0")
# print(x.dlo))
# end
# else pr'.c(x.dlo(z))
# end;
# let zero := dconst(0,0);
# let iszero :=
# proc(x:dp) boolean
# ((x.dhi = 0) & (x.dlo(z) = 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 'snuo' 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 is brackets or 'begin' and 'end' can have a 'catch phrase' as the last item. A
Introduction to Poly

catch phrase in 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.

```haskell
>let proccatches = 
  # proc(exop: string) (print(exop)); 
  begin 
  # procraisess(true); 
  # catch proccatches 
  # end; 
  default

'proccatches' has been declared as a procedure which takes a argument of type string. The exception is raised by 'procraisess' and, since it is not caught in that procedure it propagates back to the point at which 'procraisess' 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 
  # procraisess(false); 
  # catch proccatches 
  # 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.

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

```haskell
>let intpair = 
  # type 
  # intfirst = 1; 
  # intsecond = 2 
  # end;

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

1 and 2. 'intpairfirst' and 'intpairssecond' 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.

```haskell
>let x = new();
>x = assign(x, assign(x, 1));
>x = content(

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.

```haskell
>let sr = 
  # begin 
  # let var1 = new(0); 
  # type 
  # let content = var1.##content; 
  # let assign = 
  # proc(t: integer) 
  #   (if (t > 0) (t > 10) 
  #       then .else rangeerror 
  #       .else var1.assign(t)) 
  # end;

'var1' is an integer variable which is initially set to 0. 'assign' checks the value before assigning it to 'var1', 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.

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

```
17. Specifications in Declarations

The double-precision type declared above has one drawback. The specification contains the 'hi', 'lo' and 'const' 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.

```poly
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)
begin
  print("");
  print(x);
  print("\n")
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 'as'. However it must be possible to convert the
specification of the value to the explicit specification (the 'context').

```poly
let avar : new();
let boonst : integer == avar;
declares 'avar' to be an integer variable and 'boonst' to be an integer
constant. In the latter case the specification is necessary, otherwise
'boonst' 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
begin
  $== integer$;
declares 'plus' as a procedure which is the same as the 'as' attribute of
integer except that it is not an infix operator.

`plus(3,4);`
Introduction to Poly

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(stype: type end)
  # type (t)
  # into: proc(stype)t;
  # outof: proc(t)stype
  # end
  # begin
  # type (t) extends stype;
  # 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 'stype' 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$into(5);
  copyint$downof(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 an 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(stype:
    # type (t)
    # succ: proc(t)t raises rangeerror
    # end;
    # val: stype)
    # (stype$succ(stype$succ(val)));

The specification of 'val' is that it must be a constant, and its type is 'stype'. However 'stype' 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);

Similarly

> add2(char, 'A');

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. Implicit 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 unnecessary, 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'
Introduction to Poly

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 precede 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();
```

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 'prin'.

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

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(m: type (a) + : proc infix (a,s) raises rangeerror
# end;
# (1, j: m), m
# (1 + j);
would be used by writing
> add(integer, 1, 2);
```

However, by writing

```
> let +
# proc infix [m: type(a)
# s, t: proc infix (s,s) raises rangeerror
# end]
# (1, j: m,m, m)
# raises rangeerror
# == add;
```

's' 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 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 OHYFE 3p1459
```

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
'converts' 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
'converts' which converts single-quoted sequences into values of the type
'char'. These definitions can be overridden by preceding the literal by
the name of a type and a $ sign. For instance
Introduction to Poly

> let int == integer;
> let one == int!;

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

\[
\text{let list} ==
\begin{array}{c}
\text{proc}(\text{base}: \text{type end})
\end{array}
\]

\[
\begin{array}{c}
\text{type (\text{list})}
\end{array}
\]

\[
\text{car} : \text{proc}(\text{list})\text{base raises nil_list;}
\]

\[
\text{cdr} : \text{proc}(\text{list})\text{list raises nil_list;}
\]

\[
\text{cons} : \text{proc}(\text{base}; \text{list})\text{list;}
\]

\[
\text{nil} : \text{list;}
\]

\[
\text{null} : \text{proc}(\text{list})\text{boolean}
\]

\[
\text{end}
\]

\[
\begin{array}{c}
\text{let node} == \text{record}(\text{or base}; \text{od list;})
\end{array}
\]

\[
\begin{array}{c}
\text{extend union(\text{nl void; mnl node;})}
\end{array}
\]

\[
\text{let cons} ==
\begin{array}{c}
\text{proc}(\text{bb base; ll list list)(list nil node; mnode extended mnode of mlist;})
\end{array}
\]

\[
\text{let car} ==
\begin{array}{c}
\text{proc}(\text{ll list base}(\text{node node; mnode extended mnode of mlist;})
\end{array}
\]

\[
\begin{array}{c}
\text{catch proc(string base raises nil_list)}
\end{array}
\]

\[
\text{end;}
\]

\[
\text{let cdr} ==
\begin{array}{c}
\text{proc}(\text{ll list list)(list nil node; mnode extended mnode of mlist;})
\end{array}
\]

\[
\begin{array}{c}
\text{catch proc(string list raises nil_list)}
\end{array}
\]

\[
\text{end;}
\]

\[
\text{let nil} == \text{list nil node; mnode extended mnode of mlist;}
\]

\[
\text{let null} == \text{list nil node; mnode extended mnode of mlist;}
\]

\[
\text{end;}
\]

'\text{void}' is a standard type which has only one value (\text{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
definite respectively, and 'null' tests for the value 'nil'. 'car' and 'cdr' both trap
the exception which would be raised if a projection error occurred 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\text{cons}(1, ilist\text{cons}(2, ilist\text{cons}(3, ilist\text{nil}(nil))));

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

> let cons ==
\begin{array}{c}
\text{proc}(\text{base base end;})
\end{array}

\[
\begin{array}{c}
\text{list type (1) cons: proc(base; ll end)
\end{array}
\]

\[
\begin{array}{c}
\text{(bb base; ll list)(list)(ilist nil(bb; ll))
\end{array}
\]

> let il == cons(1, cons(2, cons(3, ilist\text{nil})))

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

> letrec append ==
\begin{array}{c}
\text{proc}(\text{base base end;})
\end{array}

\[
\begin{array}{c}
\text{list type (1)
\end{array}
\]

\[
\begin{array}{c}
\text{car: proc(1 base raises nil_list;)
\end{array}
\]

\[
\begin{array}{c}
\text{cdr: proc(1 list raises nil_list;)
\end{array}
\]

\[
\begin{array}{c}
\text{cons: proc(base; ll)
\end{array}
\]

\[
\begin{array}{c}
\text{null: proc(1 boolean end)
\end{array}
\]

\[
\begin{array}{c}
\text{(first second list list)
\end{array}
\]

\[
\begin{array}{c}
\text{if first then second
\end{array}
\]

\[
\begin{array}{c}
\text{else cons(car(first); append(cdr(first); second))
\end{array}
\]

> letrec reverse ==
\begin{array}{c}
\text{proc}(\text{base base end;})
\end{array}

\[
\begin{array}{c}
\text{list type (1)
\end{array}
\]

\[
\begin{array}{c}
\text{car: proc(1 base raises nil_list;)
\end{array}
\]

\[
\begin{array}{c}
\text{cdr: proc(1 list raises nil_list;)
\end{array}
\]

\[
\begin{array}{c}
\text{cons: proc(base; ll)
\end{array}
\]

\[
\begin{array}{c}
\text{nil: 1;
\end{array}
\]

\[
\begin{array}{c}
\text{null: proc(1 boolean end)
\end{array}
\]

\[
\begin{array}{c}
\text{nil: list list)
\end{array}
\]

\[
\begin{array}{c}
\text{(if null(1) then list nil)
\end{array}
\]

\[
\begin{array}{c}
\text{else append(reverse(cdr(1)); cons(car(1); list nil))
\end{array}
\]

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

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

The list created above can now be printed.

```haskell
pr([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.
Technical Report No 31

PONDER AND ITS TYPE SYSTEM

by

J. Fairbairn
can be constructed from other primitives.

For example, unions and pairs are not built in, but
primitive is built in. The smallest, possibly number of
concatenation, the smallest, possibly number of
factors, of the whole of the

The main objection of this note is to correct the
point.

Abstract

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

$$\text{let } x = (\lambda y. z) \text{ in } (\lambda z. x)$$

may trivially be changed to

$$\text{let } x = \lambda z. x$$

so that we can see what is happening. So in these circumstances the proper thing to do is to change the names that the last 'let' is actually meant to be bound to the first one.

$$\text{let } x = \lambda y. z$$

In this case, we must try

$$\text{let } x = \lambda z. x$$

to simplify the inside of

$$\text{let } x = (\lambda y. z) \text{ in } (\lambda z. x)$$

which is what you get from

$$\text{let } x = \lambda z. x$$

and then

$$((\lambda z. x) x)$$

becomes

$$(\lambda z. x) x$$

in that

$$x = (\lambda y. z)$$
example

This ∑ is a mathematical symbol.

In algebra, the symbol ∑ represents the sum of a series of numbers.

\[ \sum_{i=1}^{n} a_i = a_1 + a_2 + \ldots + a_n \]

In this expression, \( a_1, a_2, \ldots, a_n \) are the terms of the series, and \( n \) is the number of terms.

The general form is:

\[ \sum_{i=m}^{n} f(i) \]

which indicates the sum of the function \( f(i) \) from \( i = m \) to \( i = n \).

The value of \( m \) and \( n \) depends on the specific problem or context.

In some cases, the symbol \( \sum \) can be replaced by an integral sign \( \int \) to represent a continuous sum.

The symbol also appears in various mathematical contexts, such as in calculus, where it is used to represent limits and integrals.

In computer science, the symbol is used to denote repeated operations, such as loops in a program.

For example:

\[ \sum_{i=0}^{n} i^2 \]

represents the sum of the squares of the first \( n + 1 \) natural numbers.

The result is the sum of the squares of the integers from 0 to \( n \) (inclusive).

\[ \sum_{i=0}^{n} i^2 = 0^2 + 1^2 + 2^2 + \ldots + n^2 \]

This is a common mathematical operation in various fields, including physics, engineering, and computer science.

In programming, the symbol is used to denote loops, such as:

```
for (i = 0; i < n; i++) { // loop from 0 to n-1
    // perform operations
}
```

This is a simple example of a loop that iterates from 0 to \( n-1 \), performing some operations on each iteration.

The symbol is also used in other contexts, such as in set theory, where it can represent the union of sets.

For example:

\[ \bigcup_{i=1}^{n} A_i \]

represents the union of the sets \( A_1, A_2, \ldots, A_n \).

The resulting set contains all the elements that belong to any of the sets \( A_1, A_2, \ldots, A_n \).

In conclusion, the symbol is a versatile and powerful tool in mathematics and computer science, allowing for the concise representation of complex operations and concepts.
Chapter 5: Lambda Functions

Section 5: Lambda

Definition: lambda

A lambda function is a function of the lambda calculus.

Example: 

\[ \lambda x . x^2 \]

This expression represents a function that takes a single argument and returns its square.

For example, the expression \( \lambda x . x^2 \) can be evaluated as follows:

\[ (\lambda x . x^2)(3) = 3^2 = 9 \]

To transform a lambda function into another lambda function:

\[ \lambda \alpha . \lambda x . \alpha(x) \]

This expression represents a function that takes an argument \( \alpha \) and applies it to an argument \( x \).

(Home expression) expression

Let name = expression

Let declaration may be transformed into lambda expression.

Example:

\[ \lambda x . 2x + 1 \]

This expression represents a function that takes an argument \( x \) and returns \( 2x + 1 \).

In summary:

- A lambda function is a function of the lambda calculus.
- Lambda functions can be evaluated by performing function application.
- Lambda functions can be transformed into other lambda functions.
- Lambda functions can be used to create more complex functions.
The following facts define the relation:

• \( (r, x, y) \) is a relation if and only if \( r \) is a relation on the set \( X \times Y \).

The following notation is used in quantifiers or bound variables:

\( \exists x \in A \) means "for all \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"

\( \exists x \in A \) means "there exists \( x \) in \( A \)"

\( \forall x \in A \) means "for all \( x \) in \( A \)"
Section of Real Type

with the same properties. Any object is acceptable for

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.

Please fill in the parameters, we have

0.0

0.0

and its extension at all. It will return an object

since it is empty.
Proposition

\[ G \subseteq A \]

**Case I:** 

\[ G \subseteq A \]

By structural induction on \( G \)

Proof:

\[ \text{Then } z \in A \]

If \( z \in A \)

Apply Lemma 2

Corollary

Section 2: Properties

\[ A \subseteq C(\tilde{t}, \tilde{t}) \subseteq A \]

Proposition 2

**Example.**

When quantifiers appear in a type variable in the argument to \( A \).

Proposition 1

**Rule.**

The following properties are straightforward consequences of the
Here is one version of the type of functional data structures:

```
9.2.15

function `apply` (f: Applicative, x: x) = foldr (flip f) x id

function `apply` (f: Applicative, x: x) = foldr (flip f) x id

```

These functions are similar to the ones in the previous example, but they are designed to work with Applicative functors. The `apply` function takes an `Applicative` and an input `x`, and returns the result of applying `f` to `x` using the `foldr` function. The `foldr` function is defined as `foldr (flip f) x id`, where `f` is the function to apply, `x` is the input, and `id` is the identity function.

The code for computing the union is as follows:

```
```

This code is similar to the previous example, but it is written in a different style. The `union` function takes two sets and returns their union. The `setUnion` function is defined as `let union = setUnion = foldr (flip union) id`, where `union` is the function to apply, and `id` is the identity function. The `foldr` function is used to fold the `union` function over the first set, starting with the second set as the initial value.

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

```
```

This code is similar to the previous example, but it is written in a different style. The `union` function takes two sets and returns their union. The `setUnion` function is defined as `let union = setUnion = foldr (flip union) id`, where `union` is the function to apply, and `id` is the identity function. The `foldr` function is used to fold the `union` function over the first set, starting with the second set as the initial value.
Section 3: General Terms

Table of Contents:

1. General Overview
   1.1 Online Form Completion Rules
   1.2 Electronic Signature Requirements
   1.3 Confidentiality Agreement

2. Privacy Policy

3. Terms and Conditions

4. Warranty

5. Indemnification

6. Limitation of Liability

7. Dispute Resolution


9. Governing Law and Jurisdiction

10. Miscellaneous

---

This document contains important information about the online form completion process. Please read and agree to the terms and conditions before proceeding. Any unauthorized use of the forms or information contained herein is prohibited.
(10.5) Parentheses and the Typewriter

(10.6) Parentheses and the Typewriter

(10.7) Parentheses and the Typewriter

(10.8) Parentheses and the Typewriter

(10.9) Parentheses and the Typewriter

(10.10) Parentheses and the Typewriter

(10.11) Parentheses and the Typewriter

(10.12) Parentheses and the Typewriter

(10.13) Parentheses and the Typewriter

(10.14) Parentheses and the Typewriter

(10.15) Parentheses and the Typewriter

(10.16) Parentheses and the Typewriter

(10.17) Parentheses and the Typewriter

(10.18) Parentheses and the Typewriter

(10.19) Parentheses and the Typewriter

(10.20) Parentheses and the Typewriter
1. A prefix operator.

2. Postfix operators.

3. Some prefix operators.

4. The expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

5. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

6. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

7. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

8. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

9. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.

10. In the expression "let x = 5; let y = 10; let z = x + y;" evaluates to 15.
import

program unit
(production)

END.

section 1: demographical

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

For documentation [Volume 1],

Section 15: Projections

...
In the case of a motion variable, the \texttt{function} representation of the motion variable is

\begin{verbatim}
function representation;
\end{verbatim}
UNIVERSITY of CAMBRIDGE
COMPUTER LABORATORY

Technical Report No 34

RECENT DEVELOPMENTS IN LCF:
EXAMPLES OF STRUCTURAL
INDUCTION

by

Larry Paulson
1. Introduction

January 1990

University of Cambridge

By Larry Paulson

Examples of structural induction
Recent developments in LFP

2. Essential background

The theorems from grammar and mathematics are:

...
The task that proceeds the verbatim theorem is called "TaC1" repeatedly on the goal and its subgoals.

REPEAT TaC1, if it fails then exit TaC1S

The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.

Theorem: The basic ones are: TaC1, TaC2, and TaC3. They are, together, and repeatedly applied. The basic ones are: TaC1S, TaC2S, and TaC3S. They are, together, and repeatedly applied.
Two common error subroutines in the accumulator.

Example:

```
[acons] 1
[acons] 2
[acons] 3
```

Each case:

- Call to subroutine.
- Error handling routine.
- Execution of main program.

The error handling routine is designed to handle unexpected errors and provide a graceful fallback in case of failure. It interacts with the main program to ensure that the system remains stable and information is recorded for debugging purposes.

Diagram of error handling:

```
[acons] 1
[acons] 2
[acons] 3
```

```
[acons] 4
[acons] 5
[acons] 6
```

```
[acons] 7
[acons] 8
[acons] 9
```

With the above framework, it can change our approach to a different strategy. A thorough analysis using analog forms on 1L and 2L parts is necessary.
Recent developments in LTR 1

Recent developments in LTR 10

2. Memory and addressing. Production of the publication.

1979.

3. Future trends. The state-of-the-art of computer languages.

1969.


1981.


Recent developments in LISP

The Lisp source code for CSG2 is now available for downloading. Upon checking the documentation, I would like to thank Robert Huber for providing...
Technical Report No 35

REWWRITING IN CAMBRIDGE LCF\textsuperscript{1}

by

Lawrence Paulson
Rewriting in Cambridge LCF

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 PFLAHBDI 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 \( \vdash t = u \), 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

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

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.
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 PFLAMBDA 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, PFLAMBDA. 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 |-It A stands for an infinity of theorems, one for each x. In a proof, you are likely to need some of these instances.
Rewriting in Cambridge LCF

PLAMBDA Terms

PLAMBDA Formulas

rewriting a pair of lists, describing how to instantiate the pattern’s variables and type variables to make it alpha-equivalent 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

\text{PART\_MATCH}(\text{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.

\text{PART\_MATCH} makes it easy to define inference rules that involve matching. For instance, PLAMBDA includes an axiom stating that the bottom element is smaller than any other element:

\text{MINIMAL:} \quad \vdash I x. u < x

Since axioms are sometimes inconvenient to use, LCF provides an inference rule \text{MIN} that maps any term \( t \) to the theorem \( \vdash u < t \). This rule can be expressed using \text{PART\_MATCH} and the destructor function rhs:

\text{MIN:} \quad \text{term match} \quad \text{pattern object}
The function PART_PFMATCH 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.\footnote{The composite function (fst o dest_impl) takes the first part of an implication, which is the antecedent.}

```ocaml
let MATCH_MP Impth = in 
  \th. HP (match (concl th) \h); 
```

Calling MATCH_MP \(\{:-\text{fl}... \text{xn}, A \equiv B^*\} \{:-A^*\}\), where \(A^*\) is an instance of \(A\), returns the corresponding instance of \(B\); \(\equiv\). One of my proofs (Paulson [1983]) involves a total function \(\text{VARS}_\text{OF}\) and a theory of strict lists. The function \(\text{MAP}\), 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 \(\text{MAP} \text{VARS}_\text{OF}\) is total (Figure 4).

### 3. Term Conversions

Let a term conversion be any function of type "term->term" that maps a term \(t\) to a theorem \(\vdash t\equiv u\). 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 \(\vdash t\equiv u\). 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.
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 $u[x := v]$ denotes the substitution of $v$ for $x$ in $u$, then

$$\text{BETA_CONV } "(\backslash x.u)v" \rightarrow "(\backslash x.u)v := u[x := v]"$$

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...\x_n \ t \equiv \ u")

Figure 4. Examples of MATCH_MP

Figure 5. Examples using the conversion BETA_CONV

is a conversion that takes any instance of $t$, such as $t'$, and instantiates the variables $x_1$ ... $x_n$, to return the theorem $!:t' = u'$. As Figure 5 shows, REWRITE_CONV fails on all terms that do not match $t$. It is implemented using the matching function PART_MATCH:

let REWRITE_CONV = PART_MATCH (fst o dest_eqtv) ;;

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 (RESEQUIC) provides this notion of alternation. For functions $f$ and $g$, and argument $t$,

```
# some FLAMBDA axioms, used also in later figures

#let [COND_UU; COND_TT; COND_FF; # M N ABS; # M N PAIR; FST PAIR; SND PAIR] # = rewrites;

COND_UU = \lambda (UU \Rightarrow x \cdot y) \Rightarrow UU : thm
COND_TT = \lambda (TT \Rightarrow x \cdot y) \Rightarrow x* : thm
COND_FF = \lambda (FF \Rightarrow x \cdot y) \Rightarrow y* : thm
M N ABS = \lambda (\lambda x. UU) \Rightarrow UU* : thm
M N PAIR = \lambda (FST, SND x \cdot x) \Rightarrow x* : thm
FST PAIR = \lambda (FST x, y) \Rightarrow x* : thm
SND PAIR = \lambda (SND x, y) \Rightarrow y* : thm

#let COND_TT_CONV = REWRITE_CONV COND_TT;;
COND_TT_CONV = \lambda t : conv

#COND_TT_CONV cond1;;
\lambda (\lambda x. UU \Rightarrow x \cdot y) \Rightarrow UU* : thm

#COND_TT_CONV cond2;;
\lambda (\lambda y. FF \Rightarrow x \cdot y) \Rightarrow x* : thm

#COND_TT_CONV cond3;;
evaluation failed term_match

#let FST_CONV = REWRITE_CONV FST PAIR;;
FST_CONV = \lambda t : conv

# FST_CONV "FST (TT, FF)";;
\lambda (FST (TT, FF) \Rightarrow 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 \land 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

It is interesting to compare these with tacticals. The tactic ALL_TAC,\(^5\) which passes on its goal unchanged, is the identity for THENC. The tactic

NO_TAC, which fails on all goals, is the identity for ORELSEC. The tactical

ORELSEC is implemented exactly like ORELSEC. However, the tactical THENC is entirely different from THENC.

\(^5\) called ISTAC in Edinburgh LCF.
Figures 7 and 8 show examples of the conversions defined in Figure 4 (and the corresponding definitions). Each example shows how the conversion rules are applied to a term, along with the corresponding theorem proving steps. The examples illustrate the power of the conversion rules in simplifying complex expressions and proving theorems.

For combining a list of conversions, it is convenient to use FIRST_CONV, defined using Ilist and NO_CONV, where

FIRST_CONV [conv1; ...; convn] = conv1 ORELSSEC ... ORELSSEC 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.
These conversions and operators cannot traverse terms recursively; they can only rewrite top-level terms. LCF provides functions for converting sub-terms: \textsc{comb} \textsc{comb} handles combinations, while \textsc{abs} \textsc{comb} handles abstractions. They fail on terms that do not have the corresponding form.

The conversion (\textsc{comb} \textsc{conv} \texttt{"f t"}) derives

\begin{align*}
\text{\[f\texttt{sg}\]} & \quad \text{by \textsc{conv}} \\
\text{\[t\texttt{su}\]} & \quad \text{by \textsc{conv}} \\
\text{\[\texttt{f t} := g \texttt{u}\]} & \quad \text{by substitution}
\end{align*}

The conversion (\textsc{abs} \textsc{conv} \texttt{\texttt{"x.t"}}) derives

\begin{align*}
\text{\[t := v\]} & \quad \text{by \textsc{conv}} \\
\text{\[\texttt{\texttt{"x.t}} := \texttt{\texttt{"x.v\}}\]} & \quad \text{by extensionality, possibly renaming x}
\end{align*}

Let us combine \textsc{comb} \textsc{conv} and \textsc{abs} \textsc{conv} into a conversion that rewrites a term's top-level sub-terms. A term can be a constant, variable, combination, or abstraction.

\begin{verbatim}
let \textsc{sub} \textsc{conv} = 
(\textsc{comb} \textsc{conv} \texttt{conv}) \textsc{obrelsec} 
(\textsc{abs} \textsc{conv} \texttt{conv}) \textsc{obrelsec} 
\texttt{all \textsc{conv}};
\end{verbatim}

Now it is simple to write a conversion \textsc{depth} \textsc{conv} that recursively rewrites all sub-terms of a term:

\begin{verbatim}
letrec \textsc{depth} \textsc{conv} \textsc{t} = 
(\textsc{sub} \textsc{conv} (\textsc{depth} \textsc{conv} \texttt{conv}) \textsc{thenc} (\textsc{repeatc} \textsc{conv})) 
\texttt{t};
\end{verbatim}

\textsc{depth} \textsc{conv} may return a term that is not in simplest form. A more sophisticated conversion (Figure 9) reevaluates the term every time it is rewritten.

\begin{verbatim}
letrec \textsc{redepth} \textsc{conv} \textsc{t} = 
(\textsc{sub} \textsc{conv} (\textsc{redepth} \textsc{conv} \textsc{conv}) \textsc{thenc} 
((\textsc{conv} \textsc{thenc} (\textsc{redepth} \textsc{conv} \textsc{conv})) \textsc{obrelsec} \texttt{all \textsc{conv}})) 
\texttt{t};
\end{verbatim}

\textsc{depth} \textsc{conv} and \textsc{redepth} \textsc{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 \texttt{null(cons x 1)} to \texttt{ff} without simplifying \texttt{x} or \texttt{1}, which could be quicker.

\begin{verbatim}
# let \textsc{many} \textsc{conv} = 
\textsc{try} all the top-level conversions!!
# \textsc{first} \textsc{conv} (map \textsc{rewrite} \textsc{conv} \textsc{rewrites}) \textsc{obrelsec}
# \texttt{beta} \texttt{conv};
\texttt{many} \texttt{conv} = \texttt{- : \textsc{conv}}

# let \texttt{d} \texttt{conv} x \texttt{depth} \textsc{conv} \texttt{many} \textsc{conv};
\texttt{d} \textsc{conv} = \texttt{- : \textsc{conv}}

#\texttt{d} \texttt{conv} ab\textsc{ss}; \texttt{\$} does both beta-conversions!!
\texttt{\[\neg(x,(\lambda u.\texttt{ab})\texttt{ff})\texttt{tt} \rightarrow \texttt{tt},\texttt{ff}\]} : \texttt{thm}

#\texttt{d} \texttt{conv} cond\textsc{ss}; \texttt{\$} simplifies the condition, then uses it!!
\texttt{\[\neg(\textsc{fst}(\texttt{tt},\texttt{ff}) \rightarrow x \land y) \rightarrow x\]} : \texttt{thm}

#\texttt{d} \texttt{conv} ab\textsc{ss}; \texttt{\$} result could be simplified further!!
\texttt{\[\neg(\texttt{fun}(\texttt{fun}(\texttt{tt},\texttt{ff}) \rightarrow x \land y)\texttt{ff}) \rightarrow x \land y\]} : \texttt{thm}

# let \texttt{rd} \texttt{conv} = \texttt{redepth} \textsc{conv} \texttt{many} \texttt{conv};
\texttt{rd} \textsc{conv} = \texttt{- : \textsc{conv}}

#\texttt{rd} \texttt{conv} ab\textsc{ss};
\texttt{\[\neg(\texttt{fun}(\texttt{fun}(\texttt{tt},\texttt{ff}) \rightarrow x \land y)\texttt{ff}) \rightarrow x\]} : \texttt{thm}
\end{verbatim}

Figure 9. Examples of \textsc{depth} \textsc{conv} and \textsc{redepth} \textsc{conv}
4. Formula Conversions

The same ideas apply to the rewriting of formulas. Let a **formula conversion** be any function of type \(\text{form} \rightarrow \text{thm}\), mapping formulas \(A\) to theorems \(\vdash A \iff B\). 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. Analogies of Term Conversions

The formula conversion \(\text{REWRITE}\_\text{FCOV}\) allows the rewriting of a formula according to a theorem

\[ \vdash \text{"all...an. } A \iff B\]

Such theorems are called **formula rewrites**. The conversion is implemented like \(\text{REWRITE}\_\text{CONV}\), using the instantiation function \(\text{PART}\_\text{MATCH}\):

\[
\text{let } \text{REWRITE}\_\text{FCOV} = \text{PART}\_\text{MATCH} (\text{fst} \circ \text{dest}\_\text{iff});
\]

This is useful for expanding out the definition of a predicate, such as

\[ \vdash \text{rel}\_\text{TRANSITIVE}\_\text{rel} \iff \]
\[ \forall x y z. \text{rel } x y \land \text{rel } y z \iff \text{rel } x z \iff \]

Rewriting in Cambridge LCF

LCF provides the identity conversions \(\text{NO}\_\text{FCOV}\), which always fails, and \(\text{ALL}\_\text{FCOV}\), which maps \(A\) to \(\vdash \text{"}\neg \neg A \iff A\text{"}\). For sequencing, the conversion \(\text{FCOV}_1\) \(\text{THENFCOV}_2\) is defined in terms of Modus Ponens. The operators \(\text{ORELSEFC}, \text{REPEATFC},\) and \(\text{FIRST}\_\text{FCOV}\) are implemented like their term counterparts.

If \(P\) is a predicate, then \(\text{REWRITE}\_\text{FCOV} \vdash P(t)\) is a formula conversion that derives

\[ \vdash \text{"\neg \neg P(t) \iff P(u)\"} \]

by \(\text{conv}\)

\[ \vdash \text{"P(u) \iff P(t)\"} \]

by substitution and \(\text{substitution}\)

and returns

\[ \vdash \text{"P(t) \iff P(u)\"} \]

Similarly, \(\text{PFLAMBD\text{\_}\text{DA}}\)'s inference rules allow us to implement conversion operators for the quantifiers and logical connectives. The conversion \(\text{SUB}\_\text{FCOV}\) applies a conversion to all top-level terms and formulas of a formula:

\[
\text{let } \text{SUB}\_\text{FCOV} \text{ FCov conv } = \]
\[
(\text{CONJ FCOV conv FCov}) \text{ ORELSEFC}
(\text{DISJ FCOV conv FCov}) \text{ ORELSEFC}
(\text{IMP FCOV conv FCov}) \text{ ORELSEFC}
(\text{EWALL FCOV conv FCov}) \text{ ORELSEFC}
(\text{EXISTS FCOV conv FCov}) \text{ ORELSEFC}
(\text{PRED FCOV conv});
\]

For mapping a conversion over all sub-formulas of a formula, the conversions \(\text{DEPTH}\_\text{FCOV}\) and \(\text{REDDEPTH}\_\text{FCOV}\) are defined like their term analogs.

Figure 30 offers more examples.
letrec DEPTH FCONV conv fconv w =  
(SUB FCONV conv (DEPTH FCONV conv fconv)) THENFC  
(REPEAT FC fconv)) w ;;

letrec REDDEPTH FCONV conv fconv w =  
(SUB FCONV conv (REDDEPTH FCONV conv fconv)) THENFC  
(fconv THENFC (REDDEPTH FCONV conv fconv)) (ELSEFC ALL FCONV)) w ;;

4.2. Eliminating Propositional Tautologies

LCF includes conversions that recognise propositional tautologies. For instance, TAU FC FCONV can derive

\[ \text{TRUTH}() \land A \leftrightarrow A \]
\[ A \land \text{TRUTH}() \leftrightarrow A \]
\[ \text{FALSE}() \land A \leftrightarrow \text{FALSE}() \]
\[ A \land \text{FALSE}() \leftrightarrow \text{FALSE}() \]

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

\[ \text{FORALL TRUTH} \quad \vdash \quad (x,y).\text{TRUTH}() \leftrightarrow \text{TRUTH}() \]
\[ \text{FORALL FALSE} \quad \vdash \quad (x,y).\text{FALSE}() \leftrightarrow \text{FALSE}() \]

Using these, the "forall" tautology conversion can simplify \( (x,\text{TRUTH}() \leftrightarrow \text{TRUTH}() \) and \( (x,\text{FALSE}() \leftrightarrow \text{FALSE}() \). Its ML code is

\[ \text{APPROD FCONV} \quad \text{APROD FCONV} \quad \text{APROD FCONV} \quad \text{APROD FCONV} \quad \text{APROD FCONV} \quad \text{APROD FCONV} \quad \vdash \]
\[ \quad \vdash \quad (x,y).\text{TRUTH}() \leftrightarrow \text{TRUTH}() \]
\[ \quad \vdash \quad (x,y).\text{FALSE}() \leftrightarrow \text{FALSE}() \]

\[ \text{LETREC FCONV conv fconv w =} \]
\[ (\text{SUB FCONV conv (\text{REDDEPTH FCONV conv fconv})} \quad \text{THENFC}) \]
\[ (\text{REPEAT FC fconv}) \quad \text{w ;;}) \]

\[ \text{LETREC FC FCONV conv fconv w =} \]
\[ (\text{SUB FCONV conv (\text{REDDEPTH FCONV conv fconv})} \quad \text{THENFC}) \]
\[ (\text{fconv THENFC (\text{REDDEPTH FCONV conv fconv})}) \quad (\text{ELSEFC ALL FCONV})) \]
\[ \text{w ;;) \]

\[ \text{Figure 10. Examples of formula conversions} \]

\[ \text{LET TAU FC FCONV} \quad \text{REWRITE FCONV FORALL TRUTH} \]
\[ (\text{ELSEFC}) \]
\[ \text{REWRITE FCONV FORALL FALSE} ;; \]

The family of conversions is modular. To improve the tautology test, just write a better version of FORALL_TAU FC FCONV. Perhaps it should simplify
LCP provides the conversion $\text{BASIC\_TAUT\_FCNV}$, which tries all the tautology tests in turn, failing if none apply.

Let $\text{BASIC\_TAUT\_FCNV} =$

\[
\begin{align*}
\text{FIRST\_FCNV}\{ & \text{TAUT\_CONJ\_FCNV;} \\
& \text{TAUT\_DISJ\_FCNV;} \\
& \text{TAUT\_IMP\_FCNV;} \\
& \text{TAUT\_IFF\_FCNV;} \\
& \text{TAUT\_FOR\_ALL\_FCNV;} \\
& \text{TAUT\_EXISTS\_FCNV;} \\
& \text{TAUT\_PRED\_FCNV} \}; 
\end{align*}
\]

There are many ways of building simplifiers from these conversions. The standard one, $\text{BASIC\_FCNV}$, uses $\text{REDEPTH\_CONV}$ to simplify terms, $\text{DEPTH\_FCNV}$ to simplify formulas, and $\text{BASIC\_TAUT\_FCNV}$ to find tautologies in the resulting formulas. This is a pragmatic choice. The $\text{REDEPTH\_CONV}$ is somewhat thorough; terms generally need them but formulas do not. Figure 11 shows it solving the tautologies that $\text{DEPTH\_FCNV}$ missed.

Let $\text{BASIC\_FCNV} = \text{FCNV (REDEPTH\_CONV \ conv \ FCNV (REDEPTH\_CONV \ conv \ FCNV (REDEPTH\_CONV \ conv \ FCNV \ \text{BASIC\_TAUT\_FCNV})};$

5. Anatomy of a Rewriting Tactic

In studying all these functions, let us remember their original purpose: to help us prove theorems. LCP provides a tactic, $\text{ASM\_REWRITE\_TAG}$, 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-

Figure 11. Examples of $\text{BASIC\_FCNV}$ solving tautologies

The above rewriting conversions accept rewriting theorems of the form $\forall x. \ \text{t} \Rightarrow \text{u}$ or $\forall x. \ \text{t} \Rightarrow \text{a} \Rightarrow \text{b}$. 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.
In general, much implicative rewrites have the form:

$$A_1 \Rightarrow (\ldots \Rightarrow (A_n \Rightarrow t :: u) \ldots)$$

$$A_1 \Rightarrow (\ldots \Rightarrow (A_n \Rightarrow B(s)C) \ldots)$$

(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 A_1' to A_n', then, by Modus Ponens, it can return the theorem \( \Gamma :: t' :: u' \). 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_CONV tac \( \Gamma :: \ldots \Rightarrow (A \Rightarrow t :: u) \ldots \)

IMP_F_CONV tac \( \Gamma :: \ldots \Rightarrow (A \Rightarrow B(s)C) \ldots \)

The standard tactic ASM_REWRITE_TAC uses IMP_CONV and IMP_F_CONV, so these conversions perform more checking than REWRITE_CONV and REWRITE_F_CONV. 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, \( x \times y = y \times x \). 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 PROLOG implicants 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. 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: \( l \equiv uu \).

LCF provides a first-order theory of lists, with NIL, a strict CONS, and an inlined operator APP to append lists. The theory includes theorems asserting that these constants create defined lists:

\[
\begin{align*}
1 &:: \text{NIL} :: \text{UU} \\
1 &:: \text{CONS} (\text{NIL} :: \text{UU}) \\
\text{APP} (\text{CONS} (\text{CONS} \text{UU}) :: \text{UU})
\end{align*}
\]

In a particular proof, you may have assumptions that some terms \( t_1, \ldots, t_n \), and some lists \( l_1, \ldots, l_m \), 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 \( \Gamma :: A \Rightarrow B \), instantiates the theorem to \( \Gamma :: A' \Rightarrow B' \), and calls itself recursively to prove the antecedent \( A' \), proving

---

6 However, let me mention that it, too, uses PART_MATCH — 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 t2 == UU)
- (CONS t1 (CONS t2 (CONS t3 NIL))) == UU]

5.3. Canonical Forms

A predicate logic such as PFLAMBDA allows many different ways of saying the same thing. For instance, \((A \land B) \Rightarrow C\) is logically equivalent to \(A \Rightarrow (B \Rightarrow C)\), though IMP_REM_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. 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 \(A \Rightarrow B\), then it calls itself to convert \(A \Rightarrow B\) to \(A \Rightarrow B\); \(A \Rightarrow B\), then discharges \(A\) to return \(A \Rightarrow B\); \(A \Rightarrow B\). Otherwise it calls itself recursively using the rules:

\[
\begin{align*}
A \land B & \rightarrow A, B \\
(A \land B) \Rightarrow C & \rightarrow A \Rightarrow (B \Rightarrow C) \\
\text{\(t\),} A & \Rightarrow B \rightarrow A[t/x] \Rightarrow B \quad \text{(a variant of \(t\) not used elsewhere)} \\
1. \Rightarrow B & \rightarrow B
\end{align*}
\]

7 Edinburgh LCF provided a standard canonical form. For example, if you attempted to construct the formula "\(A \land \text{TRUTH}\)" or "\(\text{TRUTH} \Rightarrow A\)" 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

\[- (A1 \Rightarrow \ldots \Rightarrow (A_n \Rightarrow B) \ldots )\]

Such implications are fine for IMP_REM_CONV and IMP_SEARCH_FCONV, but the formula conversion IMP_REM_CONV expects \(B\) to have the form \(C \Rightarrow D\), which does not appear often in practice. The inference rule FCONV_CANON alters the consequent \(B\) using the rules:

\[
\begin{align*}
P(x) & \Rightarrow P(x) \Rightarrow \text{TRUTH()} \\
P(x) & \Rightarrow P(x) \Rightarrow \text{FALSEST()} \\
C \Rightarrow B & \Rightarrow \text{unchanged}
\end{align*}
\]

Thus FCONV_CANON proves logical equivalences that rewrite predicates to TRUTH() and negated predicates to FALSEST(). It passes on any formula.

---

Figure 12. IMP_CANON putting theorems into canonical form
rewrites it encounters.

5.3. 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 \textsc{fcnv} uses \textsc{fcnv} to convert a goal \textit{A} to a new one \textit{B}, leaving its assumptions unchanged. If the goal \textit{B} is just \textsc{truth()}, then \textsc{fcnv} has achieved the goal \textit{A}, and returns an empty sub-goal list.

5.2. The Rewriting Tactic

The tactic \textsc{rewrite} requires all the above pieces. It puts its input, a list of theorems, into canonical form using \textsc{imp_canon} and \textsc{fcnv_cannot}. It handles implicative rewrites using the conversions \textsc{imp_rew_conv} and \textsc{imp_rew_fconv}. It combines these conversions, along with \textsc{beta_conv}, using \textsc{first_conv}, \textsc{first_fconv}, and \textsc{basic_fconv}. It solves antecedents of implicative rewrites by backwards chaining, using the tactic \textsc{imp_search}. To solve trivial sub-goals that arise during backwards chaining, it adds the reflexivity axiom \textsc{eq_refl} \textit{(x:x=x)} to the list of input theorems.

The tactic \textsc{asm_rewrite} calls \textsc{rewrite}, adding the goal's assumptions to the input list of theorems. Figure 11 shows the definitions of both tactics.

6. Examples of Solving Goals by Rewriting

To see \textsc{asm_rewrite} in use, consider a proof of mine (Paulson [1983]). It uses a theory of combinator expressions and concerns infixed functions \textsc{occs} and \textsc{occs_eq}. The relation \textit{t1 occs t2} matches \textit{t2} for an occurrence of \textit{t1}, returning \textsc{tt} if it finds one. The relation \textsc{occs_eq} is the reflexive closure of \textsc{occs}. Figure 14 shows their definitions in \textsc{lcp}.

We will see how \textsc{asm_rewrite} helps to prove that the relation \textsc{occs} is transitive:

\begin{align*}
\texttt{ta, tb, tc \Rightarrow occs ta \Rightarrow occs tb \Rightarrow occs tc} \\
\texttt{ta, tb, tc \Rightarrow occs tc \Rightarrow occs ta \Rightarrow occs tb}
\end{align*}

Inducting on the variable \textit{"ta"} yields four sub-goals (Figure 15). Compare these with the axiom defining \textsc{occs}; three of them contradict the antecedent, \texttt{tb occs tc \Rightarrow tt}. Using the axioms \textsc{occs_clauses} and \textsc{occs_eq},
let OCCS_EQ =
new_axiom ('OCCS_EQ'),
"t1 t2. t OCCS_EQ t2 == (t1 t2) OR (t OCCS EQ t2)*";

let OCCS_CLAUSES =
negation axioms ('OCCS_CLAUSES'),
"t: t OCCS UU == UU" /
(1c. "e:UU =>" t OCCS (CONST c) == FF) /
(1v: "v:UU =>" t OCCS (VAR v) == FF) /
(1i1 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 ASL_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 "e:UU", and rewrites
"t OCCS (CONST c)" to FF. Then it rewrites the antecedent FF to FAUL-
SITY(). Similarly, it rewrites the consequent to FALSEITY(), yielding
the analogous goal FALSEITY() => FALSEITY(). ASL_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 ASL_REWRITE_TAC.

7. Conclusions
Rewriting in Cambridge LCF

I can imagine you thinking, "This isn't very elegant, but it must be hopelessly inefficient." In fact, conversions are efficient enough to help perform complex proofs. ACM_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 minimize 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 MATCH, REWRITE_CONV, TAC_CONV_CONV, IMP_CANNON, carry out small, well-defined tasks. They have simple specifications and implementations. Together they can express the rewriting tactic, ACM_REWRITE_TAC, in only a dozen lines.

Its modularity makes ACM_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 ACM_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 stack, 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_MATCH 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


Technical Report No. 36

THE REVISED LOGIC PPLAMBDAA

A REFERENCE MANUAL

by

Lawrence Paulson
Abstract

PFLAMBDA 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 PFLAMBDA formulas.

PFLAMBDA'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 PFLAMBDA. The induction rule accepts a list of theorems, stating type properties to consider when deciding whether to admit an induction.

---------------------------------------------------------------------

1. Introduction .............................................................................. 1
2. Syntax ...................................................................................... 2
3. Functions for Manipulating PFLAMBDA Objects ......................... 3
4.1 Abstract Syntax Primitives ...................................................... 3
4.2 Derived Syntax Functions ...................................................... 4
4.3 Functions Concerning Substitution .......................................... 5
5. Axioms and Basic Lemmas ....................................................... 7
6. Predicates ................................................................................. 9
7. Predicate Calculus Rules ......................................................... 10
8.1 Rules for quantifiers .............................................................. 10
8.2 Rules for basic connectives .................................................... 12
8.3 Rules for derived connectives ................................................ 13
9. Additional rules ......................................................................... 14
10. Fixed point induction ............................................................. 16
11. Admissibility for short types .................................................... 16
12. Stating type properties in PFLAMBDA ..................................... 18
13. Derived Inference Rules ......................................................... 19
14. Predicates Calculus Rules ....................................................... 19
15. Rules About Functions and the Partial Ordering ....................... 22
16. Differences from Edinburgh LCF ............................................ 25
17. Formula Identification ............................................................ 26
18.7 The Definitional Function LCF ............................................. 26
19.1 Data Structures ..................................................................... 27
20. References ............................................................................... 28

1 Research supported by S.I.R.C. Grant number GB/167/66.
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 PFLAHODA. It can reason about non-terminating computations, arbitrary recursion schemes, and higher-order functions, by virtue of Scott's theory of continuous partial orders (Stry [1977]). PFLAHODA 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, Coq [1985, 1981]. Cambridge LCF (Paulson [1981]) 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 PFLAHODA 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 PFLAHODA 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 dis-
sensitive, particularly to the presence of 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 notation follows the following conventions, possibly subtleties:

- **Syntax**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>term</code></td>
<td><code>HLLGDBA_constr</code></td>
</tr>
</tbody>
</table>

- **Variables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A</code></td>
<td><code>term</code></td>
</tr>
<tr>
<td><code>P</code></td>
<td><code>term</code></td>
</tr>
</tbody>
</table>

- **Predicates**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>truth-value</code></td>
<td><code>bottom</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>truth-value</code></td>
</tr>
</tbody>
</table>

- **Operands**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix</code></td>
<td><code>HLLGDBA_constr</code></td>
</tr>
<tr>
<td><code>combine</code></td>
<td><code>HLLGDBA_constr</code></td>
</tr>
</tbody>
</table>

**Standard types**

- `set` type containing only one element
- `prop` type of truth-values: `true`, `false`
- `ty` type of a lambda-term: `ty` and `ty`:
  - actually `"(ty, ty)_pred"`
- `tyb` type of lambda-terms: `ty` and `ty`:
  - actually `"(ty, ty, ty)_pred"`

**Terms**

- `constant` where `c` is a constant symbol
- `variable` `A`
- `migration` `migration` over a term
- `eval` evaluation (application of function to argument)
- `condexp` conditional expression `true"(ty, ty)_pred"`
- `pair` `HLLGDBA_constr` actually `"PAIR"`

**Formulas**

- `TRUE` logicality
- `FALSE` contradiction
- `\alpha = \beta` equality of `\alpha` and `\beta` actually `"equiv(\alpha, \beta)"
- `\alpha < \beta` with partial ordering -- actually `"ineq(\alpha, \beta)"
- `P(\alpha)` where `P` is a predicate symbol
- `\forall \alpha A` universal quantifier
- `\exists \alpha A` existential quantifier
- `A \land B` conjunction
- `A \lor B` disjunction
- `A \Rightarrow B` implication
- `\alpha \leftrightarrow \beta` if-and-only-if

**Preliminary**

1. Functions for Manipulating PFLMMBA Objects

1.1. Abstract Syntax Primitives

LCF provides functions to construct, test the form of, and take apart PFLMMBA terms, formulas, and types. These use standard naming conventions.

**Proto-typ**

- `mk` make an object (term, formula, type)
- `as` "test that an object has a given top-level constructor
dst `take apart an object, yielding its top-level part"
The logic HELASMA

Constructors:

structor

list.mk.abs \[x_1;.:.;x_n], n \to \{x_1;.:.;x_n, t\}
list.mk.emb \[t], \{x_1;.:.;x_n\} \to \{x_1;.:.;x_n, t\}
list.mk.eqj \{A_1;.:.;A_n\} \to \{A_1 \wedge \ldots \wedge A_n\, \forall \theta\}
list.mk.eqj \{A_1;.:.;A_n\} \to \{A_1 \vee \ldots \vee A_n\, \forall \theta\}
list.mk.imp \{A_1;.:.;A_n\}, \{t\} \to \{A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B\}
list.mk.fforall \{x_1;.:.;x_n\}, \{\theta\} \to \{x_1;.:.;x_n, \theta\}

list.mk.exists \{x_1;.:.;x_n\}, \{\theta\} \to \{x_1;.:.;x_n, \theta\}

Destructors:

structor

strip.abs \{x_1;.:.;x_n, t\} \to \{x_1;.:.;x_n\}, t
strip.emb \{t\}, \{x_1;.:.;x_n\} \to \{t\}, \{x_1;.:.;x_n\}
conjectures \{A_1 \wedge \ldots \wedge A_n\} \to \{A_1;.:.;A_n\}
disjuncts \{A_1 \vee \ldots \vee A_n\} \to \{A_1;.:.;A_n\}
strip.imp \{A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B\} \to \{A_1;.:.;A_n\}, B
strip.fforall \{x_1;.:.;x_n, \theta\} \to \{x_1;.:.;x_n\}, \theta
strip.exists \{x_1;.:.;x_n, \theta\} \to \{x_1;.:.;x_n\}, \theta

5.1. 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.

Checking a variant of a variable

variant : (term list) \to term \to term

Generating a new variable (distinct from any already in use)

generate : type \to term
Returning all variables in a PFLAMDA object

\[
\begin{align*}
term\_vars & : \text{term} \rightarrow \text{term list} \\
term\_vars & : \text{form} \rightarrow \text{form list} \\
form\_vars & : (\text{form list}) \rightarrow \text{form list}
\end{align*}
\]

Returning the free variables in a PFLAMDA object

\[
\begin{align*}
term\_frees & : \text{term} \rightarrow \text{term list} \\
term\_frees & : \text{form} \rightarrow \text{form list} \\
form\_frees & : (\text{form list}) \rightarrow \text{form list}
\end{align*}
\]

Checking if two terms/formulas are alpha-convertible

\[
\begin{align*}
\alpha\text{con}_\text{term} & : \text{term} \rightarrow \text{term} \rightarrow \text{bool} \\
\alpha\text{con}_\text{form} & : \text{form} \rightarrow \text{form} \rightarrow \text{bool}
\end{align*}
\]

Checking if one type/term/formula occurs (free) in another

\[
\begin{align*}
type\text{\_in}\_\text{type} & : \text{type} \rightarrow \text{type} \rightarrow \text{bool} \\
type\text{\_in}\_\text{term} & : \text{type} \rightarrow \text{term} \rightarrow \text{bool} \\
type\text{\_in}\_\text{form} & : \text{type} \rightarrow \text{form} \rightarrow \text{bool} \\
term\text{\_free}\_\text{in}\_\text{term} & : \text{term} \rightarrow \text{term} \rightarrow \text{bool} \\
term\text{\_free}\_\text{in}\_\text{form} & : \text{term} \rightarrow \text{form} \rightarrow \text{bool} \\
form\text{\_free}\_\text{in}\_\text{form} & : \text{form} \rightarrow \text{form} \rightarrow \text{bool}
\end{align*}
\]

Instantiation of types in a PFLAMDA object

\[
\begin{align*}
\text{inst\_type} & : (\text{type} \rightarrow \text{type}) \rightarrow \text{type} \\
\text{inst\_term} & : (\text{term list}) \rightarrow (\text{type} \rightarrow \text{type}) \rightarrow \text{term} \\
\text{inst\_form} & : (\text{form list}) \rightarrow (\text{type} \rightarrow \text{type}) \rightarrow \text{form}
\end{align*}
\]

May prime variables, avoiding those given in the (term list) arguments.

8. Axioms and Basic Lemmas

The axioms of Scott theory (Igarashi [1992]) are bound to ML identifiers.

\[
\begin{align*}
\text{Standard Tautology} \\
\text{TRUTH} & \quad \text{TRUTH}()
\end{align*}
\]

Partial ordering

\[
\begin{align*}
\text{LESS\_REFL} & \quad \lambda x. x\ll x \\
\text{LESS\_ANTI\_SYM} & \quad \lambda x y. x\ll y \land y\ll x \Rightarrow x \ll y \\
\text{LESS\_TRANS} & \quad \lambda x y z. x\ll y \land y\ll z \Rightarrow x\ll z
\end{align*}
\]

Monotonicity of function application

\[
\begin{align*}
\text{MONO} & \quad \text{if } g \times y, f\ll g \land x\ll y \Rightarrow f\times x \ll f\times g, y
\end{align*}
\]

Extensionality of \ll

\[
\begin{align*}
\text{LESS\_EXT} & \quad \text{if } g, (\times x) f\times x \ll g, x
\end{align*}
\]

Minimality of \ll

\[
\begin{align*}
\text{MINIMAL} & \quad \lambda x. \text{ll} x
\end{align*}
\]
Conditional expressions

\textbf{CONDCLAUSES}

\begin{align*}
& \text{ if } x, \text{ u } \Rightarrow x \land y = \text{ uu } \\
& \text{ if } \text{ tt } \Rightarrow x \land y = x \\
& \text{ if } \text{ ff } \Rightarrow x \land y = y
\end{align*}

Truth values

\textbf{TR_CASES}

\begin{align*}
& \text{ if } \text{ tr } \Rightarrow p = \text{ uu } \lor p = \text{ tt } \lor p = \text{ ff }
\end{align*}

\textbf{TR_LESS DISTINCT}

\begin{align*}
& \text{ tt } < \text{ ff } \land \text{ ff } < \text{ tt } \\
& \text{ tt } < \text{ uu } \land \text{ uu } < \text{ tt } \\
& \text{ ff } < \text{ uu } \land \text{ uu } < \text{ ff }
\end{align*}

Ordered pairs

\textbf{MK_PAIR}

\begin{align*}
& \text{ if } (x, \text{ fst } x, \text{ snd } x) = x
\end{align*}

\textbf{FST_PAIR}

\begin{align*}
& \text{ if } y, \text{ fst } (x, y) = x
\end{align*}

\textbf{SND_PAIR}

\begin{align*}
& \text{ if } y, \text{ snd } (x, y) = y
\end{align*}

Fixed points

\textbf{FIX_EQ}

\begin{align*}
& \text{ if } \text{ FIX } f = f (\text{ FIX } f)
\end{align*}

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

\textbf{BETA_CONV} \( \text{ "f}(\x . \text{u})\text{v} \) returns \( \text{ f}(\text{x}.\text{u})\text{v} = \text{ u}[\text{v}/\text{x}] \)

\( \text{ LCF } \) includes some basic lemmas that follow from the axioms.

Equality

\textbf{EQ_DEF}

\begin{align*}
& \text{ if } \text{ x } = \text{ x } \\
& \text{ if } y, y = y \\
& \text{ if } x, y = z \lor y = z \Rightarrow x = z
\end{align*}

\textbf{Exploring}

\textbf{The Logic PFLHDA}

Extensionality of \( = \)

\begin{align*}
& \text{ EQ_EXT} \\
& \text{ if } g, (\text{ f, f } x = \text{ g } x) \Rightarrow \text{ f } = \text{ g }
\end{align*}

Distinctness of the truth values

\begin{align*}
& \text{ TR_EQ DISTINCT} \\
& \text{ tt } = \text{ ff } \land \text{ ff } = \text{ tt } \\
& \text{ tt } = \text{ uu } \land \text{ uu } = \text{ tt } \\
& \text{ ff } = \text{ uu } \land \text{ uu } = \text{ ff }
\end{align*}

The completely undefined function

\begin{align*}
& \text{ MIN_COMB} \\
& \text{ if } x, u = \text{ uu }
\end{align*}

\begin{align*}
& \text{ MIN_ABB} \\
& \text{ \x , u } = \text{ uu }
\end{align*}

Validity of Eta-Conversion

\begin{align*}
& \text{ ETA_EQ} \\
& \text{ if } \text{ \x , f x } = \text{ f }
\end{align*}

\section{Predicates}

In \( \text{ Cambridge LCF } \), you can introduce predicate symbols. A predicate can be axiomatized abstractly, or as an abbreviation for a long formula. Examples:

\begin{align*}
& \text{ STRICT f } \quad \text{ if } \quad \text{ f uu } = \text{ uu }
\end{align*}

\begin{align*}
& \text{ TRANSITIVE } p \quad \text{ if } \quad \text{ p x y = tt } \land \text{ p y z = tt } \Rightarrow \text{ p x z = tt }
\end{align*}

\text{ PFLHDA'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 \( \mathtt{U} \) as the operand when only its type is relevant.

\[
\text{FLAT (ww*)} \iff \\
1x:* , 1x:* , x1<x2 \implies \mathtt{U}U = x1 \lor x1 = x2
\]

\[
\text{ISOMORPHIC (ww*, ww**)} \iff \\
？g. (1x:* , g(x) = x) \land (1y:* , f(y) = y)
\]

All predicates have exactly one argument, which may be a tuple of values or the empty value {} (read "empty"). In particular, we must write "\text{TRUTH}()" and "\text{FALSE}()".

6. Predicate Calculus Rules

These are conventional natural deduction rules (Dummet [1977]). In the notation below, assumptions of a premise are only mentioned if they will be discharged in that inference. The assumptions of the conclusion include all other assumptions of the premises. Explicit assumptions are written inside [ square brackets ].

6.1. Rules for quantifiers

For all introduction

\[
\text{Forall introduction} \\
\text{GEN: term } \to \text{thm } \to \text{thm } \\
\begin{align*}
\forall x . A(x) & \quad \text{where the variable "a" is not free in assumptions of premise} \\
\end{align*}
\]

Existential introduction

\[
\text{Exists introduction} \\
\begin{align*}
\exists \overline{x} . \overline{\text{term}} & \to \text{thm } \to \text{thm } \\
A(t) & \quad \text{t} \\
\end{align*}
\]

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 \( \text{=TT=T} \):

\[
\begin{align*}
\text{exists (}^{\forall x} , x=x=T^n \text{, } "T" \text{)} (\text{=TT=T}^n) & \quad \implies \quad \text{=}^{\forall x} , x=x=T^n \\
\text{or} & \\
\text{exists (}^{\exists x} , x=x^n \text{, } "T" \text{)} (\text{=TT=T}^n) & \quad \implies \quad \text{=}^{\exists x} , x=x^n \\
\end{align*}
\]

Existential elimination

\[
\text{Exists elimination} \\
\text{CHOOSE: (term } \& \text{ thm) } \to \text{thm } \to \text{thm } \\
\begin{align*}
\exists x . A(x) \quad [ A(a) ] B \\
\hline
B & \quad \text{where the variable "a" is not free anywhere except in B's assumption A(a)}
\end{align*}
\]
6.2. Rules for basic connectives

Conjunction introduction

CONJ: thm \rightarrow thm \rightarrow thm

\[ \begin{array}{c}
A \\
B \\
\hline
A \land B
\end{array} \]

Conjunction elimination

CONJUNCT1, CONJUNCT2: thm \rightarrow thm

\[ \begin{array}{c}
A \land B \\
\hline
A
\end{array} \]

Disjunction introduction

DISJ1: thm \rightarrow form \rightarrow thm
DISJ2: form \rightarrow thm \rightarrow thm

\[ \begin{array}{c}
A \\
B \\
\hline
A \lor B
\end{array} \]

Disjunction elimination

DISJ_CASES: thm \rightarrow thm \rightarrow thm \rightarrow thm

\[ \begin{array}{c}
A \lor B \\
[ A ] C \\
[ B ] C \\
\hline
C
\end{array} \]

6.3. Rules for derived connectives

Implication introduction

\[ \text{DISCH: form \rightarrow thm \rightarrow thm} \]

\[ \begin{array}{c}
A \\
[ A ] B \\
\hline
A \rightarrow B
\end{array} \]

Implication elimination

\[ \text{MP: thm \rightarrow thm \rightarrow thm} \]

\[ \begin{array}{c}
A \rightarrow B \\
A \\
\hline
B
\end{array} \]

The formula \( A \rightarrow B \) is logically equivalent to \( (A \rightarrow B) \land (B \rightarrow A) \). But LCF does not expand it as such, to avoid duplicating \( A \) and \( B \). The rules \text{CONJ\_IFF} and \text{IFF\_CONJ} map between the two formulas.

The formula \( \neg \neg A \) denotes \( A \equiv \text{FALSE} \). 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

\[ \text{CONJ\_IFF: thm \rightarrow thm} \]

\[ \begin{array}{c}
(A \rightarrow B) \land (B \rightarrow A) \\
\hline
A \leftrightarrow B
\end{array} \]
If and-only-if elimination

\[
\text{IFF\_CONJ; form} \rightarrow \text{thm} \\
A \leftrightarrow B \\
\text{------------} \\
(A \leftrightarrow B) \land (B \leftrightarrow A)
\]

Negation introduction

\[
\text{DISCH; form} \rightarrow \text{thm} \rightarrow \text{thm} \\
A \\
\text{---------} \\
\text{FALSEITY()}
\]

Negation elimination

\[
\text{MP; thm} \rightarrow \text{thm} \rightarrow \text{thm} \\
\neg A \\
\text{---------} \\
\text{FALSEITY()}
\]

7. Additional rules

Assumption

\[
\text{ASSUME; form} \rightarrow \text{thm} \\
A \\
\text{---------} \\
\text{[ A ] A}
\]

Contradiction rule

\[
\text{CONTR; form} \rightarrow \text{thm} \rightarrow \text{thm} \\
A \\
\text{FALSEITY()} \\
\text{---------} \\
A
\]

Classical contradiction rule

\[
\text{CONTR; form} \rightarrow \text{thm} \rightarrow \text{thm} \\
A \\
\text{---------} \\
\text{FALSEITY()}
\]

Intuitionists (Dummett [1977]) can get rid of this rule by typing "let CONTR:();". However, PFLAMBDIA does not seem suitable for constructive proof. The cases axiom \text{TH\_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

\[
\text{SUBST; (thm \# term)\_list} \rightarrow \text{form} \rightarrow \text{thm} \rightarrow \text{thm} \\
\text{x1 A(x1)} \\
\text{---------} \\
\text{ti \equiv ti A(ti)} \\
\text{------------} \\
\text{A(ti)}
\]

The formula \text{A(x1)} serves as a template to control the substitution; the variables \text{x1} mark the places where substitution should occur.
Instantiation of Types

INST_TYPE (type # type) list -> thm -> thm
  tyl vtyl

where the type variables vtyl do not occur in the assumptions

\[ A(vtyl) \]

\[ \text{------------------------} \]

\[ A(tyl) \]

Instantiation of Terms

INST (term # term) list -> thm -> thm
  ti xi

where the variables xi do not occur in the assumptions

\[ A(xi) \]

\[ \text{------------------------} \]

\[ A(ti) \]

3. 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 \( z_1, z_2, \ldots \), if \( A(z) \) is true for every \( z \), 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 PL-LAMBDA, with its implications, disjunctions, quantifiers, and user-definable predicates.

3.1. Admissibility for short types

Igarashi [1972] considered admissibility in a logic containing all these connectives, but his admissibility test can be considerably liberalized.

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.

Suppose we wish to prove \( \forall x. y. A(x) \) by structural induction, where the type "ty" is short. This requires computation induction on a variable \( f \) and formula \( \forall x. y. A(f, z) \). This formula is chain-complete in \( f \):

Suppose that \( f \) is the limit (least upper bound) of an ascending chain

\[ f_0, f_1, \ldots \]

Then the limit case \( \forall x. A(z) \) holds also, for consider any \( z' \). Since the type of \( f' \) is short, the chain \( f_0, z', f_1, z', \ldots \) reaches its limit at some finite \( l \). For this \( l \), \( \forall f(z) \) equals \( f(z') \).

Our assumption (1) implies that \( A(z') \) holds, so \( A(f(z')) \) holds too.
Since we chose \( z' \) arbitrarily, we conclude \( \forall x. A(f, z) \). Thus the induction is admissible.

From this argument it appears that the admissibility test may be liberalized 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 intuistic validity of this inference is questionable, as is the justification of the admissibility rule for disjunctions. Both rely on the "principle of excluded middle": if you partition an infinite set in two, one of the two sets must be infinite.
The Logic PFLAMBDA

The term contains existentially quantified variables, the formula may not be chain-complete.

Example:

?x.f x=xU, where f maps every natural number to \( \text{TT} \). Suppose that for all \( i \), \( f \) maps all numbers less than \( i \) to \( \text{TT} \), the rest to \( \text{U} \).

Then \( f \) is the limit of the \( f_i \), the formula holds for each \( f_i \), 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 occurrences of \( \text{tt} \) or \( \text{xxxx} \) are chain-complete, as discussed on page 77 of Gordon et al. [1979]. It treats \( \text{tt} \) as the equivalent formula \( \text{tt}+\text{tt} \), which is chain-complete in \( t \) in both positive and negative positions.

8.2. Stating type properties in PFLAMBDA

LCF recognizes certain theorems that state that a type is finite or short. Any theorem

\[ \vdash \text{ty} \cdot x=x_1 \lor \ldots \lor x=x_m \]

where the \( x_i \) are constants, states that the type \( \text{"ty"} \) is finite. Any theorem

\[ \vdash \text{ty} \cdot x=\text{ty} \cdot x=x_2 \land \ldots \land x=x_{(n+1)} \Rightarrow \]

\[ U=x_1 \lor x=x_2 \lor \ldots \lor x=x_{(n+1)} \Rightarrow \]

The Logic PFLAMBDA

states that the type \( \text{"ty"} \) is short. When \( n=2 \) this is the familiar flatness property:

\[ \vdash \text{ty} \cdot x=x_2 \Rightarrow U=x_1 \lor x=x_2 \]

To inform LCF of such properties when checking admissibility, the induction rule accepts a list of theorems, \( B_1, \ldots, B_n \). Each \( B_i \) should state the finiteness or shortness of a type.

Scott Fixed-Point Induction

\[ \text{INDUCT: } \text{term list} \rightarrow \text{thm list} \rightarrow \text{thm list} \rightarrow \text{thm } \]

\[ B_1 \cdots B_n \quad A(\text{U}) \quad \text{if } \cdots \text{ fn. } A(f) \Rightarrow A(f \text{ fun } f) \]

\[ \begin{array}{c}
\hline
A(\text{FIX fun })
\end{array} \]

9. Derived Inference Rules

For your convenience, LCF provides inference rules that can be derived from the primitive rules of PFLAMBDA. A few of these are wired in for efficiency, but most derive their conclusions by proper inferences.

9.1. Predicate Calculus Rules

Intuitionists will be glad to hear that now use the classical contradiction rule, \text{CONTRA},
**Substitution (at specified occurrence numbers)**

\[
\text{SUBS: } \text{(thm list)} \rightarrow \text{thm} \rightarrow \text{thm} \\
\text{SUBS OCCS: } \text{((int list) \# \text{thm}) list} \rightarrow \text{thm} \rightarrow \text{thm} \\
\text{ti \# \#} \quad \text{A(t)} \\
\hline
\text{A(t)}
\]

**Generalising a theorem over its free variables**

\[
\text{GEN ALL: } \text{thm} \rightarrow \text{thm} \\
\text{A(x)} \\
\hline
\text{\(x\),...\(x\),A(x)}
\]

**Discharging all hypotheses**

\[
\text{DISCH ALL: } \text{thm} \rightarrow \text{thm} \\
\text{[[A;...;A]} \quad \text{B} \\
\hline
\text{A1 \#...\#An \#B}
\]

**Iterated SPEC**

\[
\text{SPEC: } \text{(term list)} \rightarrow \text{thm} \rightarrow \text{thm} \\
\text{ti} \\
\text{\(t\),...\(t\),A(t)} \\
\hline
\text{SPEC [t;...;tn] A(t)}
\]

**Undischarging an assumption**

\[
\text{UNDISCH: } \text{thm} \rightarrow \text{thm} \\
\text{A \#B} \\
\hline
\text{[A]B}
\]

**Undischarging all assumptions**

\[
\text{UNDISCH ALL: } \text{thm} \rightarrow \text{thm} \\
\text{A1 \#...\#An \#B} \\
\hline
\text{[A1;...;An]B}
\]

**Specialisation over outer universal quantifiers**

\[
\text{SPEC ALL: } \text{thm} \rightarrow \text{thm} \\
\text{\(t1\),...\(tn\),A[x]} \\
\hline
\text{\([A[x]/x]\)\text{ where the } x\text{ are not free in hyps of } A}
\]

**Using a theorem A to delete a hypothesis of B**

\[
\text{PROVE HYP: } \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} \\
\text{A} \quad \text{B} \\
\hline
\text{A} \quad \text{[A]B} \\
\hline
\text{B}
\]

**Conjoining a list of theorems**

\[
\text{LIST CONJ: } \text{(thm list)} \rightarrow \text{thm} \\
\text{Ai} \\
\hline
\text{A1 \#...\#An} \\
\text{A1 \#...\#An} \quad \text{where } n>0
\]

**Splitting a theorem into its conjuncts**

\[
\text{CONJUNCTS: } \text{thm} \rightarrow \text{(thm list)} \\
\text{A1 \#...\#An} \\
\hline
\text{A1 \#...\#An} \quad \text{where } n>0
\]
Iterated Modus Ponens

\text{LIST}_{\text{MP}}: \text{thm} \longmapsto \text{thm} \rightarrow \text{thm}

\text{A1} ... \text{An} \rightarrow \text{B}

\text{Contrapositive of an implicat}

\text{CONTRA}: \text{thm} \rightarrow \text{thm}

\text{A} \rightarrow \text{B}

\text{B} \rightarrow \text{B}

\text{Converting disjunction to implication}

\text{DISJ}_{\text{IMP}}: \text{thm} \rightarrow \text{thm}

\text{A} \lor \text{B}

\text{A} \rightarrow \text{B}

\text{DISJ}_{\text{CASES}}: \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} \rightarrow \text{thm}

\text{A} \lor \text{B} \lor \text{C} \lor \text{D}

\text{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 \text{<}< stands for either of the relations \text{==} or \text{<<}, the same at each occurrence within a rule unless otherwise stated.
Minimality of \( \ll \)

\[ \text{MUN: term } \rightarrow \text{ thm} \]

\[ t \ll \quad \rightarrow \quad \text{ "MU} \ll t" \]

LESS_UU_RULE: thm \( \rightarrow \) thm

\[ \ll t u \quad \rightarrow \quad \ll u \ll t \]

Construction of a combination

LE_HK_COMB: (thm \$ thm) \( \rightarrow \) thm

\[ f \ll g \]
\[ t \ll u \]
\[ f \ll g \]
\[ u \ll t \]
\[ f \ll g \ll t \]

<< unless both hypotheses use \( \ll \)

Application of a Term to a theorem

AP_TERM: term \( \rightarrow \) thm \( \rightarrow \) thm

\[ u \ll v \]
\[ t \ll u \ll t v \]

Application of a theorem to a term

AP_THM: thm \( \rightarrow \) term \( \rightarrow \) thm

\[ u \ll v \]
\[ t \ll u \ll v t \]

Construction of an abstraction

MK_ABS: thm \( \rightarrow \) thm

\[ \ll x. u \ll v \]
\[ \ll x. u \ll \ll x. w \]

HALF_MK_ABS: thm \( \rightarrow \) thm

\[ \ll x. u \ll v \ll t \]
\[ \ll x. u \ll \ll x. t \]

Alpha-conversion (renaming of bound variable)

ALPHA_CONV: term \( \rightarrow \) term \( \rightarrow \) thm

\[ \ll (y.t) \]
\[ \ll (x.y) \]
\[ \ll (x.y.t) \ll x. (y.t) \]

10. Differences from Edinburgh LCF

The obvious differences are that PLONDON in Cambridge LCF provides the existential quantifier, the disjunction, negation, and if-and-only-if symbols, and predicate symbols. It includes the standard contradiction PAL-SIT(1), instead of expressing contradiction through formulas such as "TT\(=\)FF" or "IL\(=\)IL".

However, the new PLONDON is not just an extension of the old. Its syntax has changed to use \( \wedge \) instead of \( \times \), and \( \Rightarrow \) instead of \( \Rightarrow \). 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->TRUE()". The constructor functions mk_fforall and mk_imp automatically simplified these to TRUE(). 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 (U) = U
DEF x = TT

for any x except U

The formula "DEF x = TT" asserts that x is defined. However, it is easier to write "x=x=U". DEF is no longer provided, though you can easily generalize it yourself.

---

5 Here I am using the notation of Cambridge LCF, though describing Edinburgh LCF.
References


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
Impplode
ImpplodeAscii
InChar
InInt
InString
IntToTimeString
NewStream
OutChar
OutInt
OutString
Pair
PutStream
Ref
StringToInt
Sub
Tabulate
Update
ValidTP

These all occur in the file "mlout.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 "unix" under einuc, the image has problems getting started.

```
MAIN = mlglob.h mlmain.h mlstor.h mlscan.h mlpars.h mlanal.h mlcomp.h mlasm.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 mlasm.h mlmain.h mlrout.h mlasm.p
DEBG = mlglob.h mldebg.h mlmain.h mlpars.h mlanal.h mlcomp.h mlasm.h mlrout.h mldebg.p
STOK = 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 mlasm.o mlstor.o mlserv.o mlrout.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
mlasm.o: $(ASSM); pc -w -c -o mlasm.p
mldebgl.o: $(DEBG); pc -w -c -o mldebgl.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  
University College London  
Gower Street  
London WC1E 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 Schultz  
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