The revised logic PPLAMBDA
A reference manual

Lawrence Paulson

March 1983
The Revised Logic PPLAMBDAP\textsuperscript{1}

A Reference Manual

Lawrence Paulson

Cambridge University

March 1983

Abstract

PPLAMBDAP 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 PPLAMBDAP formulas.

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

\footnote{Research supported by S.E.R.C. Grant number GR/B67766.}
The Logic PPLAMDBA

Table of Contents

1 Introduction .................................................. 1
2 Syntax .................................................. 2
3 Functions for Manipulating PPLAMDBA Objects ................. 3
  3.1 Abstract Syntax Primitives ................................ 3
  3.2 Derived Syntax Functions ............................... 4
  3.3 Functions Concerning Substitution ....................... 5
4 Axioms and Basic Lemmas .................................. 7
5 Predicates ............................................... 9
6 Predicate Calculus Rules .................................. 10
  6.1 Rules for quantifiers .................................. 10
  6.2 Rules for basic connectives ......................... 12
  6.3 Rules for derived connectives ..................... 13
7 Additional rules ......................................... 14
8 Fixed point induction .................................... 16
  8.1 Admissibility for short types .................... 16
  8.2 Stating type properties in PPLAMDBA .............. 18
9 Derived Inference Rules .................................. 19
  9.1 Predicate Calculus Rules ........................... 19
  9.2 Rules About Functions and the Partial Ordering .... 22
10 Differences from Edinburgh LCF ......................... 25
  10.1 Formula Identification .......................... 26
  10.2 The Definedness Function DEF .................... 26
  10.3 Data Structures .................................. 27
References ............................................. 28
1. Introduction

The proof assistant LCF is an interactive computer program that helps a user prove theorems and develop theories about computable functions, using a logic called PPLAMBDA. It can reason about non-terminating computations, arbitrary recursion schemes, and higher-order functions, by virtue of Scott’s theory of continuous partial orders (Stoy [1977]). PPLAMBDA uses standard natural deduction rules (Dummet [1977]).

The version known as Edinburgh LCF (Gordon, Milner, Wadsworth [1979]) has been used for many projects, for example, Cohn [1982, 1983]. Cambridge LCF (Paulson [1983]) is a descendant of Edinburgh LCF. Though based on the same principles, the new system is quite different from the old one. In particular, the logic PPLAMBDA has been revised to include disjunction, existential quantifiers, and predicates.

Some notes of caution: Cambridge LCF is still in a state of flux. The revised PPLAMBDA has been stable for only a few months. This report is largely self-contained, but you may wish to refer to Gordon et al. [1979] for background information. Please notify me of any major errors you dis-
cover, particularly in the section on fixed-point induction.

I would like to thank Mike Gordon for his many comments and corrections regarding this paper.

2. Syntax

In this paper, syntactic meta-variables obey the following conventions, possibly subscripted:

<table>
<thead>
<tr>
<th>name</th>
<th>PPLAMBDA construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>x,y,z</td>
<td>variables</td>
</tr>
<tr>
<td>t,u,v</td>
<td>terms</td>
</tr>
<tr>
<td>A,B,C</td>
<td>formulas</td>
</tr>
<tr>
<td>P,Q</td>
<td>predicate symbols</td>
</tr>
<tr>
<td>ty</td>
<td>types</td>
</tr>
</tbody>
</table>

**Standard types**

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>type containing only one element</td>
</tr>
<tr>
<td>tr</td>
<td>type of truth-values: TT, FF, WW</td>
</tr>
<tr>
<td>ty1 × ty2</td>
<td>Cartesian product of ty1 and ty2</td>
</tr>
<tr>
<td></td>
<td>— actually &quot;:(ty1,ty2)prod&quot;</td>
</tr>
<tr>
<td>ty1 → ty2</td>
<td>continuous functions from ty1 to ty2</td>
</tr>
<tr>
<td></td>
<td>— actually &quot;:(ty1,ty2)fun&quot;</td>
</tr>
</tbody>
</table>

**Terms**

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>constant, where c is a constant symbol</td>
</tr>
<tr>
<td>x</td>
<td>variable</td>
</tr>
<tr>
<td>\ x . t</td>
<td>lambda-abstraction over a term</td>
</tr>
<tr>
<td>t u</td>
<td>combination (application of function to argument)</td>
</tr>
<tr>
<td>p := t ; u</td>
<td>conditional expression — actually &quot;COND p t u&quot;</td>
</tr>
<tr>
<td>t . u</td>
<td>ordered pair — actually &quot;PAIR t u&quot;</td>
</tr>
</tbody>
</table>
3. Functions for Manipulating PPLAMBDA Objects

3.1. Abstract Syntax Primitives

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

Prefixes:

mk make an object (term, formula, type)
is test that an object has a given top-level constructor
dest take apart an object, yielding its top-level parts
### Suffixes for Terms

- `const` constant
- `var` variable
- `abs` abstraction
- `comb` combination
- `pair` ordered pair
- `cond` conditional expression

### Suffixes for Formulas

- `equiv` equivalence of terms
- `inequiv` inequivalence of terms
- `forall` universal quantifier
- `exists` existential quantifier
- `conj` conjunction
- `disj` disjunction
- `imp` implication
- `iff` if-and-only-if
- `pred` predicate

For example, there are three basic functions for manipulating universal quantifiers:

- `mk forall: (term θ form) -> form`
- `is forall: form -> bool`
- `dest forall: form -> (term θ form)`

### 3.2. Derived Syntax Functions

LCF provides syntax functions involving lists. Unless stated otherwise, n denotes any non-negative integer.
Constructors:

list_mk_abs  ["x1" ; ; ; "xn"], "t"  ----> "\ x1 ... xn . t"
list_mk_comb "t", ["u1" ; ; ; "un"]  ----> "t u1 ... un"
list_mk_conj ["A1" ; ; ; "An"]  ----> "A1 \ ... \ \ An",  n > 0
list_mk_disj ["A1" ; ; ; "An"]  ----> "A1 \ \ ... \ \ \ An",  n > 0
list_mk_imp ["A1" ; ; ; "An"], "B"  ----> "A1 == > ... == > An == > B"
list_mk_forall ["x1" ; ; ; "xn"], "A"  ----> "! x1 ... xn . A"
list_mk_exists ["x1" ; ; ; "xn"], "A"  ----> "? x1 ... xn . A"

Destructors:

strip_abs  "\ x1 ... xn . t"  ----> ["x1" ; ; ; "xn"], "t"
strip_comb  "t u1 ... un"  ----> "t", ["u1" ; ; ; "un"]
conjuncts  "A1 \ ... \ \ An"  ----> ["A1" ; ; ; "An"]

3.3. Functions Concerning Substitution

These functions are similar to those that Appendix 7 of Gordon et al. [1979] describes in detail. This summary is for the sake of completeness.

Choosing a variant of a variable

variant: (term list) -> term -> term

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

genvar: type -> term
Returning all variables in a PPLAMBDa object

\[\text{term\_vars: term } \rightarrow \text{ term list}\]
\[\text{form\_vars: form } \rightarrow \text{ term list}\]
\[\text{form\_list\_vars: (form list) } \rightarrow \text{ term list}\]

Returning the free variables in a PPLAMBDa object

\[\text{term\_frees: term } \rightarrow \text{ term list}\]
\[\text{form\_frees: form } \rightarrow \text{ term list}\]
\[\text{form\_list\_frees: (form list) } \rightarrow \text{ term list}\]

Returning the type variables in a PPLAMBDa object

\[\text{type\_tyvars: type } \rightarrow \text{ type list}\]
\[\text{term\_tyvars: term } \rightarrow \text{ type list}\]
\[\text{form\_tyvars: form } \rightarrow \text{ type list}\]
\[\text{form\_list\_tyvars: (form list) } \rightarrow \text{ type list}\]

Testing if two terms/formulas are alpha-convertible

\[\text{aconv\_term: term } \rightarrow \text{ term } \rightarrow \text{ bool}\]
\[\text{aconv\_form: form } \rightarrow \text{ form } \rightarrow \text{ bool}\]

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

\[\text{type\_in\_type: type } \rightarrow \text{ type } \rightarrow \text{ bool}\]
\[\text{type\_in\_term: type } \rightarrow \text{ term } \rightarrow \text{ bool}\]
\[\text{type\_in\_form: type } \rightarrow \text{ form } \rightarrow \text{ bool}\]
\[\text{term\_free\_in\_term: term } \rightarrow \text{ term } \rightarrow \text{ bool}\]
\[\text{term\_free\_in\_form: term } \rightarrow \text{ form } \rightarrow \text{ bool}\]
\[\text{form\_free\_in\_form: form } \rightarrow \text{ form } \rightarrow \text{ bool}\]

Substitution in a term/formula (at specified occurrence numbers)

\[\text{subst\_term: (term \# term list) } \rightarrow \text{ term } \rightarrow \text{ term}\]
\[\text{subst\_form: (term \# term list) } \rightarrow \text{ form } \rightarrow \text{ form}\]
\[\text{subst\_occ\_term: ((int list) list) } \rightarrow \text{ (term \# term list) } \rightarrow \text{ term } \rightarrow \text{ term}\]
\[\text{subst\_occ\_form: ((int list) list) } \rightarrow \text{ (term \# term list) } \rightarrow \text{ form } \rightarrow \text{ form}\]
The Logic PFLAMBD A

**Instantiation of types in a PFLAMBD A object**

inst_type: (type # type)list -> type -> type
inst_term: (term list) -> (type # type)list -> term -> term
inst_form: (term list) -> (type # type)list -> form -> form

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

4. Axioms and Basic Lemmas

The axioms of Scott theory (Igarashi [1972]) are bound to ML identifiers.

**Standard Tautology**

TRUTH

**Partial ordering**

LESS_REF L

LESS_ANTI_SYM

LESS_TRANS

**Monotonicity of function application**

MONO

**Extensionality of **

LESS_EXT

**Minimality of UU**

MINIMAL
Conditional expressions

\begin{align*}
\text{COND_CLAUSES} & \quad !x \ y. \ \text{UU} \Rightarrow x \land y \equiv \text{UU} \\
& \quad \text{TT} \Rightarrow x \land y \equiv x \\
& \quad \text{FF} \Rightarrow x \land y \equiv y
\end{align*}

Truth values

\begin{align*}
\text{TR_CASES} & \quad !p : \text{tr.} \ p \equiv \text{UU} \lor p \equiv \text{TT} \lor p \equiv \text{FF} \\
\text{TR_LESS_DISTINCT} & \quad \neg \text{TT} \ll \text{FF} \lor \neg \text{FF} \ll \text{TT} \lor \neg \text{TT} \ll \text{UU} \lor \neg \text{FF} \ll \text{UU}
\end{align*}

Ordered pairs

\begin{align*}
\text{MK_PAIR} & \quad !x. \ (\text{FST } x, \ \text{SND } x) \equiv x \\
\text{FST_PAIR} & \quad !x \ y. \ \text{FST}(x, y) \equiv x \\
\text{SND_PAIR} & \quad !x \ y. \ \text{SND}(x, y) \equiv y
\end{align*}

Fixed points

\begin{align*}
\text{FIX_EQ} & \quad !f. \ \text{FIX } f \equiv 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

\[
\text{BETA_CONV } "(\lambda x. u) v" \quad \text{returns} \quad \vdash (\lambda x. u) v \equiv u[v/x]
\]

LCF includes some basic lemmas that follow from the axioms.

Equality

\begin{align*}
\text{EQ_REFL} & \quad !x. \ x \equiv x \\
\text{EQ_SYM} & \quad !x \ y. \ x \equiv y \Rightarrow y \equiv x \\
\text{EQ_TRANS} & \quad !x \ y \ z. \ x \equiv y \lor y \equiv z \Rightarrow x \equiv z
\end{align*}
The Logic PPLAMBDA

Extensionality of ==

EQ_EXT  \[ \forall f, g. (\forall x. f x == g x) \Rightarrow f == g \]

Distinctness of the truth values

TR_EQ_DISTINCT

- TT == FF \land
- TT == UD \land
- FF == UD \land

The completely undefined function

MIN_COMB  \[ \forall x. U U x == U U \]
MIN_ABS  \[ \forall x. U U == U U \]

Validity of Eta-Conversion

ETA_EQ  \[ \forall f. (\forall x. f x == f) \]

5. Predicates

In Cambridge LCF, you can introduce predicate symbols. A predicate can be axiomatised abstractly, or as an abbreviation for a long formula. Examples:

\[ \text{STRCT} f \quad \iff \quad f U U == U U \]

\[ \text{TRANSITIVE} p \quad \iff \]
\[ \forall x y z. p x y == \text{TT} \land p y z == \text{TT} \Rightarrow p x z == \text{TT} \]

PPLAMBDA's type system allows these axioms to refer to the types of the operands of the predicates. There are many examples of predicates that require describe properties of types, not of values. You may adopt the
convention of writing \( UU \) as the operand when only its type is relevant.

\[
\text{FLAT (UU:* *) } \iff \\
\forall x1:* . \forall x2:* . x1 \neq x2 \implies UU = x1 \lor x1 = x2
\]

\[
\text{ISOMORPHIC (UU:* *, UU:* *) } \iff \\
\exists f . g . (\forall x:* . g(f x) = x) \land (\forall y:* . f(g y) = y)
\]

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

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 \([\text{square brackets}]\).

6.1. Rules for quantifiers

\underline{Forall introduction}

\begin{align*}
\text{GEN: term } & \to \text{ thm } \to \text{ thm} \\
& \quad \forall x \\
& \quad A(a) \\
& \quad \text{---------- where the variable "$a$" is not free in assumptions of premiss} \\
& \quad \forall x . A(x)
\end{align*}
The Logic PPLAMBEA

Foreach elimination

SPEC: term -> thm -> thm
    
    !x.A(x)  
    --------
    A(t)

Exists introduction

EXISTS: (form # term) -> thm -> thm
        
    A(t)  
    --------
    ?x.A(x)

You must tell the rule what its conclusion should look like, since it is rarely desirable to replace every t by x. For example, you can conclude two different results from the theorem |-"TT==TT":

EXISTS ("?x. x==TT", "TT") (|-"TT==TT") --> |-"?x. x==TT"

or

EXISTS ("?x. x==x", "TT") (|-"TT==TT") --> |-"?x. x==x"

Exists elimination

CHOOSE: (term # thm) -> thm -> thm

        a

    ?x.A(x)  [ A(a) ] B
    -----------------------------
    B

where the variable "a" is not free anywhere except in B's assumption A(a)
6.2. Rules for basic connectives

**Conjunction introduction**

CONJ: thm -> thm -> thm

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

**Conjunction elimination**

CONJUNCT1, CONJUNCT2: thm -> thm

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

**Disjunction introduction**

DISJ1: thm -> form -> thm
DISJ2: form -> thm -> thm

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

**Disjunction elimination**

DISJ_CASES: thm -> thm -> thm -> thm

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

**Implication introduction**

DISCH: form $\rightarrow$ thm $\rightarrow$ thm

\[ A \quad \frac{[ \quad A \quad ] \quad B \quad }{A \rightarrow B} \]

**Implication elimination**

MP: thm $\rightarrow$ thm $\rightarrow$ thm

\[ A \rightarrow B \quad A \quad \frac{B}{\frac{A}{\rightarrow}} \]

6.3. **Rules for derived connectives**

The formula \( A \leftrightarrow B \) 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

CONJ_IFF and IFF_CONJ map between the two formulas.

The formula \( \neg A \) denotes \( A \rightarrow \text{FALSTY}() \). The rules for negation are special
cases of the rules for implication, and are not provided separately. Any
inference rule that works on implications also works on negations.

**If-and-only-if introduction**

CONJ_IFF: thm $\rightarrow$ thm

\[ (A \rightarrow B) \land (B \rightarrow A) \quad \frac{}{A \leftrightarrow B} \]
If-and-only-if elimination

IFF_CONJ: thm -> thm

A <=>

B

----------------------
(A => B) \and (B => A)

Negation introduction

DISCH: form -> thm -> thm

A

[ A ] FALSITY()

---------------
\neg A

Negation elimination

MP: thm -> thm -> thm

\neg A

A

---------------
FALSITY()

7. Additional rules

Assumption

ASSUME: form -> thm

A

----------------------
[ A ] A
The Logic PPLAMBDA

Contradiction rule

CONTR: form -> thm -> thm
      A
      ------
      FALSITY()        A
                     ------
                        A

Classical contradiction rule

CCONTR: form -> thm -> thm
      A
      [  ~A ] FALSITY()  ------
                     A

Intuitionists (Dummet [1977]) can get rid of this rule by typing "let CCONTR=();;". However, PPLAMBDA does not seem suitable for constructive proof. The cases axiom TR_CASES allows dubious instances of the excluded middle. The theory of admissibility for disjunctions and short types, discussed below, seems to rely on classical reasoning.

Simultaneous Substitution

SUBST: (thm # term)list -> form -> thm -> thm
      xi   A(xi)
      ---------
      ti == ui  A(ti)
                     ------
                        A(ui)

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

\[ \text{INST}_{\text{TYPE}} \ (\text{type } \# \text{ type})\text{list} \rightarrow \text{thm} \rightarrow \text{thm} \]
\[ \quad \quad \quad \quad \quad \text{tyi} \quad \text{vtyi} \]

where the type variables vtyi do not occur in the assumptions

\[ A(vtyi) \]
\[ \rightarrow \]
\[ A(tyi) \]

Instantiation of Terms

\[ \text{INST} \ (\text{term } \# \text{ term})\text{list} \rightarrow \text{thm} \rightarrow \text{thm} \]
\[ \quad \quad \quad \quad \quad \text{ti} \quad \text{xi} \]

where the variables xi do not occur in the assumptions

\[ A(xi) \]
\[ \rightarrow \]
\[ A(ti) \]

8. Fixed point induction

Fixed-point induction on a variable x and formula \( A(x) \) is only sound if the formula \( A \) is "chain-complete" with respect to \( x \). For any ascending chain of values \( z_1, z_2, \ldots \), if \( A(z_i) \) is true for every \( z_i \), 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 PPLAMBDAB, with its implications, disjunctions, quantifiers, and user-definable predicates.

8.1. Admissibility for short types

Igarashi [1972] considered admissibility in a logic containing all these connectives, but his admissibility test can be considerably liberalised.
An important special case is that all structural inductions over flat types are admissible.

Definition: A short type is one with no infinite ascending chains.\(^2\)

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

Suppose that \(f\) is the limit (least upper bound) of an ascending chain \(f_0, f_1, \ldots\).

(1) Suppose that \(\lambda z. A(f \, z)\) holds for all \(i\).

Then the limit case \(\lambda z. A(f \, z)\) holds also, for consider any \(z'\). Since the type of "\(f \, z'\)" is short, the chain \((f_0 \, z'), (f_1 \, z'), \ldots\) reaches its limit at some finite \(i\).\(^3\) For this \(i\), "\(f \, z'\)" equals "\(f \, z'\)".

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

From this argument it appears that the admissibility test may be liberalised to allow any occurrence of the induction variable within some term of short type, with restrictions on what variables the term may contain. If

---

\(^2\) Gordon et al. [1979] call these "easy" types.

\(^3\) The intuitionistic validity of this inference is questionable, as is the justification of the admissibility rule for disjunctions. Both rely on the "pigeon-hole principle": if you partition an infinite set in two, one of the two sets must be infinite.
the term contains existentially quantified variables, the formula may not be chain-complete.

Example:

\(?z.f \ z \equiv \text{UU}, \text{ where } f \text{ maps every natural number to } "\text{TT}". \text{ Suppose that for all } i, f_i \text{ maps all numbers less than } i \text{ to } \text{TT}, \text{ the rest to } \text{UU}. \text{ Then } f \text{ is the limit of the } f_i, \text{ the formula holds for each } f_i, \text{ 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 \(t \equiv u\) or \(t = u\) are chain-complete, as discussed on page 77 of Gordon et al. [1979]. It treats \(t = u\) as the equivalent formula \(t \equiv\text{UU},\) which is chain-complete in \(t\) in both positive and negative positions.

8.2. Stating type properties in PPLAMDBA

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

\[ \vdash !x:ty. x = \text{c}_1 \lor \ldots \lor x = \text{c}_n \]

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

\[ \vdash !x_1 \ldots !x_n : ty. x_1 \equiv x_2 \land \ldots \land x_{(n-1)} \equiv x_n \Rightarrow\]

\(\text{UU} = x_1 \lor x_1 = x_2 \lor \ldots \lor x_{(n-1)} = x_n\)
The Logic PPLAMBDA

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

\[ !x_1 \ x_2. \ x_1 < x_2 \implies uu = x_1 \lor x_1 = x_2 \]

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

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

\[ \text{funi} \quad B_i \]

\[ B_1 \ldots B_n \quad A(uu) \quad !f_1 \ldots f_n. \ A(f_i) \implies A(\text{funi } f_i) \]

\[ \text{A(FIX funi)} \]

9. Derived Inference Rules

For your convenience, LCF provides inference rules that can be derived from the primitive rules of PPLAMBDA. A few of these are wired in for efficiency, but most derive their conclusions by proper \(^4\) inferences.

9.1. Predicate Calculus Rules

\(^4\) Intuitionists will be glad to hear that none use the classical contradiction rule, CCONTR.
Substitution (at specified occurrence numbers)

\text{SUBS}: \text{thm list} \rightarrow \text{thm} \\
\text{SUBS\_OCCS}: \text{(int list \# thm) list} \rightarrow \text{thm} \rightarrow \text{thm}

\begin{align*}
\text{ti} &= \text{ui} \\
A(\text{ti}) &\quad \text{------------} \\
A(\text{ui})
\end{align*}

Generalising a theorem over its free variables

\text{GEN\_ALL}: \text{thm} \rightarrow \text{thm}

\begin{align*}
A(x_i) &\quad \text{-------------} \\
!x_1 \ldots x_n. A(x_i)
\end{align*}

Discharging all hypotheses

\text{DISCH\_ALL}: \text{thm} \rightarrow \text{thm}

\begin{align*}
[A_1; \ldots; A_n] B &\quad \text{-------------} \\
A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B
\end{align*}

Iterated SPEC

\text{SPECL}: \text{term list} \rightarrow \text{thm} \rightarrow \text{thm}

\begin{align*}
\text{ti} &\quad \text{------------} \\
!x_1 \ldots x_n. A(x_i) &\quad \text{SPECL} [t_1; \ldots; t_n] \\
A(\text{ti})
\end{align*}

Un-discharging an assumption

\text{UNDISCH}: \text{thm} \rightarrow \text{thm}

\begin{align*}
A &\quad \Rightarrow \quad B \\
\text{-------------} \\
[A] B
\end{align*}
Undischarging all assumptions

UNDISCH_ALL : thm -> thm

A1 => ... => An => B

--------------------------------------------------
[A1; ...; An] B

Specialisation over outer universal quantifiers

SPEC_ALL : thm -> thm

!x1 ... xn. A[xi]

-----------------------------------------------
A[xi/xi]

where the xi' are not free in hyps of A

Using a theorem A to delete a hypothesis of B

PROVE_HYP : thm -> thm -> thm

A B

-----------------------------------------------
A [A] B

B

Conjoining a list of theorems

LIST_CONJ : (thm list) -> thm

Ai

A1 ... An

-----------------------------------------------
A1 \( \land \) ... \( \land \) An

where n>0

Splitting a theorem into its conjuncts

CONJUNCTS : thm -> (thm list)

A1 \( \land \) ... \( \land \) An

-----------------------------------------------
A1 ... An

where n>0
Iterated Modus Ponens

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

\[ \begin{align*}
\text{A1} & \ldots \text{A} \text{n} \\
\text{A1} \implies \ldots \implies \text{An} \implies \text{B} \\
\end{align*} \]

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

\[ \text{B} \]

Contrapositive of an implication

\[ \text{CONTRA FOS: thm} \rightarrow \text{thm} \]

\[ \begin{align*}
\text{A} & \implies \text{B} \\
\text{-------------------------------------------} \\
\neg \text{B} & \implies \neg \text{A} \\
\end{align*} \]

Converting disjunction to implication

\[ \text{DISJ IMP: thm} \rightarrow \text{thm} \]

\[ \begin{align*}
\text{A} \lor \text{B} \\
\text{-------------------------------------} \\
\neg \text{A} & \implies \text{B} \\
\end{align*} \]

\[ \text{DISJ CASES UNION: thm} \rightarrow \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} \]

\[ \begin{align*}
\text{A} \lor \text{B} & \quad [\text{A}] \text{C} & \quad [\text{B}] \text{D} \\
\text{-----------------------------------------------} \\
\text{C} \lor \text{D} & \\
\end{align*} \]

9.2. Rules About Functions and the Partial Ordering

These are mostly the same as in Gordon et al. [1979], sometimes with different spellings. I retain the convention that \(=\) stands for either of the relations \(=\) or \(<\), the same at each occurrence within a rule unless otherwise stated.
Reflexivity of equality

\text{REFL: term} \rightarrow \text{thm}

\begin{align*}
\text{"t"} & \quad \rightarrow \quad \vdash \text{"t} = \text{t"} \\
\end{align*}

Symmetry of equality

\text{SYM: thm} \rightarrow \text{thm}

\begin{align*}
t = u & \quad \rightarrow \quad u = t \\
\end{align*}

Analysis of equality

\text{ANAL: thm} \rightarrow \text{thm}

\begin{align*}
t = u & \quad \rightarrow \quad \vdash \neg (t < u \lor u < t) \\
\end{align*}

Synthesis of equality

\text{SYNTH: thm} \rightarrow \text{thm}

\begin{align*}
t < u \lor u < t & \quad \rightarrow \quad \vdash t = u \\
\end{align*}

Transitivity (infix operator)

\text{TRANS: thm} \rightarrow \text{thm} \rightarrow \text{thm}

\begin{align*}
t = \text{t} \quad \text{u} = \text{v} & \quad \rightarrow \quad \vdash \text{t} = \text{v} \\
\end{align*}

Extensionality

\text{EXT: thm} \rightarrow \text{thm}

\begin{align*}
\forall x. \text{u} x = \text{v} x & \quad \rightarrow \quad \vdash \text{u} = \text{v} \\
\end{align*}
Minimality of \text{UU}

\text{MIN: } \text{term} \rightarrow \text{thm}
\begin{array}{c}
\text{t} \\
\text{\text{"t" \rightarrow \text{\text{\text{"UU <= t"}}} }}
\end{array}

\text{LESSUU\_RULE: } \text{thm} \rightarrow \text{thm}
\begin{array}{c}
\text{t <=UU} \\
\hline \\
\text{t ==UU}
\end{array}

Construction of a combination

\text{LE\_MK\_COMB: } (\text{thm} \# \text{thm}) \rightarrow \text{thm}
\begin{array}{c}
\text{f <= g} \\
\text{t <= u} \\
\hline \\
\text{f t <= g u} \quad \text{<< unless both hypotheses use ==}
\end{array}

Application of a term to a theorem

\text{AP\_TERM: } \text{term} \rightarrow \text{thm} \rightarrow \text{thm}
\begin{array}{c}
\text{t} \\
\text{u <= v} \\
\hline \\
\text{t u <= t v}
\end{array}

Application of a theorem to a term

\text{AP\_THM: } \text{thm} \rightarrow \text{term} \rightarrow \text{thm}
\begin{array}{c}
\text{t} \\
\text{u <= v} \\
\hline \\
\text{u t <= v t}
\end{array}
The Logic PPLAMBDA

Construction of an abstraction

MK_ABS: thm -> thm

\!x. u \leq v

-------------
\\x.u \leq \\x.v

HALF_MK_ABS: thm -> thm

\!x. u \ x \leq t

-------------
\ u \leq \ \x.t

Alpha-conversion (renaming of bound variable)

ALPHA_CONV: term -> term -> thm

x (\y.t)

-------------
\y.t == \x. t[x/y]

10. Differences from Edinburgh LCF

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

However, the new PPLAMBDA is not just an extension of the old. Its syntax has changed to use \ instead of &, and => instead of IMP. The ML names and types of many of the inference rules have changed. There are other, more subtle differences.
10.1. Formula Identification

Edinburgh LCF forced every formula into a canonical form. For instance, you could not build the formulas "!x.TRUTH()" and "A=>TRUTH()". The constructor functions mk_forall and mk_imp automatically simplified these to TRUTH(). 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,

\[ \text{dest_conj (mk_conj (A, B))} \rightarrow (A, B) \]

10.2. The Definedness Function DEF

Edinburgh LCF provided a function DEF, satisfying

\[
\begin{align*}
\text{DEF } UU &= UU \\
\text{DEF } x &= TT \\
\end{align*}
\]

for any \( x \) except \( UU \)

The formula "DEF \( x = TT \)" asserts that \( x \) is defined. However, it is easier to write "\( x = UU \)". DEF is no longer provided, though you can easily axiomatise it yourself.

\[\text{5 Here I am using the notation of Cambridge LCF, though describing Edinburgh LCF.}\]
10.3. Data Structures

In Edinburgh LCF, data structures were axiomatised using sum, product, and lifted types. This was originally done manually, and later by Milner's structural induction package (Cohn and Milner [1982]).

In Cambridge LCF, data structures can be axiomatised using disjunction and existential quantifiers. A descendant of Milner's package introduces the axioms automatically. The sum and lifted types have been removed, along with the operators UP, DOWN, INL, INR, ISL, OUTL, OUTR (for sum types) and UP, DOWN (for lifted types). The structural induction package makes it easy to define such type operators.
References


