Technical Report

UCAM-CL-TR-36
ISSN 1476-2986

Number 36

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The revised logic PPLAMBDA

A reference manual

Lawrence Paulson

March 1983

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

The Revised Logic PPLAMBDA[1]

A Reference Manual

Lawrence Paulson

Cambridge University

March 1983

Abstract

PPLAMBDA is the logic used in the Cambridge LCF proof assistant. It allows Natural Deduction proofs about computation, in Scott's theory of partial orderings. The logic's syntax, axioms, primitive inference rules, derived inference rules, and standard lemmas are described, as are the LCF functions for building and taking apart PPLAMBDA formulas.

PPLAMBDA's rule of fixed-point induction admits a wide class of inductions, particularly where flat or finite types are involved. The user can express and prove these type properties in PPLAMBDA. The induction rule accepts a list of theorems, stating type properties to consider when deciding whether to admit an induction.

---

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

Table of Contents

The Revised Logic PPLAMBDA

A Reference Manual

Lawrence Paulson

Cambridge University

March 1983

1. Introduction

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

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

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

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

2. Syntax

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

name	PPLAMBDA construct
x,y,z	variables
t,u,v	terms
A,B,C	formulas
P,Q	predicate symbols
ty	types

Standard types

void	type containing only one element
tr	type of truth-values: TT, FF, UU
ty1 # ty2	Cartesian product of ty1 and ty2 — actually ":(ty1,ty2)prod"
ty1 -> ty2	continuous functions from ty1 to ty2 — actually ":(ty1,ty2)fun"

Terms

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

Standard constants

```
UU:*                           bottom element for partial ordering
TT:tr                          truth-value "true"
FF:tr                          truth-value "false"
FIX:(* -> *) -> *              fixed-point operator
COND:(tr-> * -> * -> *)        function for making conditional expressions
PAIR:(* -> ** -> (*#**))       function to construct an ordered pair
FST:(* # **) -> *              selector for the first element of a pair
SND:(* # **) -> **             selector for the second element of a pair
(): void                       sole element of the type ":void"
```

Formulas

```
TRUTH()      tautology
FALSITY()    contradiction
t == u       equality of t and u -- actually "equiv(t,u)"
t << u       Scott partial ordering -- actually "inequiv(t,u)"
P t          where P is a predicate symbol

!x.A         universal quantifier
?x.A         existential quantifier
A /\ B       conjunction
A \/ B       disjunction
A ==> B      implication
A <=> B      if-and-only-if
~A           negation -- actually "A ==> FALSITY()"
```

3. Functions for Manipulating PPLAMBDA Objects

3.1. Abstract Syntax Primitives

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

Prefixes:

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

Suffixes for Terms

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

Suffixes for Formulas

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

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

```
mk_forall: (term # form) -> form
is_forall: form -> bool
dest_forall: form -> (term # form)
```

3.2. 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"]
disjuncts      "A1 \/ ... \/ An"   --->  ["A1";...;"An"]
strip_imp      "A1 ==> ... ==> An ==> B"   --->  ["A1";...;"An"], "B"
strip_forall   "!x1 ... xn. A"   --->  ["x1";...;"xn"], "A"
strip_exists   "?x1 ... xn. A"   --->  ["x1";...;"xn"], "A"
```

3.3. Functions Concerning Substitution

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

Choosing a variant of a variable

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

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

genvar: type -> term

Returning all variables in a PPLAMBDA object

```
term_vars: term -> term list
form_vars: form -> term list
forml_vars: (form list) -> term list
```

Returning the free variables in a PPLAMBDA object

```
term_frees: term -> term list
form_frees: form -> term list
forml_frees: (form list) -> term list
```

Returning the type variables in a PPLAMBDA object

```
type_tyvars: type -> type list
term_tyvars: term -> type list
form_tyvars: form -> type list
forml_tyvars: (form list) -> type list
```

Testing if two terms/formulas are alpha-convertible

```
aconv_term: term -> term -> bool
aconv_form: form -> form -> bool
```

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

```
type_in_type: type -> type -> bool
type_in_term: type -> term -> bool
type_in_form: type -> form -> bool

term_freein_term: term -> term -> bool
term_freein_form: term -> form -> bool

form_freein_form: form -> form -> bool
```

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

```
subst_term: (term # term)list -> term -> term
subst_form: (term # term)list -> form -> form

subst_occs_term: ((int list)list) -> (term#term)list -> term -> term
subst_occs_form: ((int list)list) -> (term#term)list -> form -> form
```

Instantiation of types in a PPLAMBDA object

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

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

4. Axioms and Basic Lemmas

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

Standard Tautology

```
TRUTH          TRUTH()
```

Partial ordering

```
LESS_REFL      !x. x<<x
LESS_ANTI_SYM  !x y. x<<y  /\  y<<x  ==>  x==y
LESS_TRANS     !x y z. x<<y  /\  y<<z  ==>  x<<z
```

Monotonicity of function application

```
MONO           !f g x y. f<<g  /\  x<<y  ==>  f x << g y
```

Extensionality of <<

```
LESS_EXT       !f g. (!x. f x << g x)  ==>  f<<g
```

Minimality of UU

```
MINIMAL        !x. UU<<x
```

Conditional expressions

```
COND_CLAUSES
                    !x y. UU => x | y == UU   /\
                          TT => x | y == x    /\
                          FF => x | y == y
```

Truth values

```
TR_CASES        !p:tr. p==UU  \/  p==TT  \/  p==FF

TR_LESS_DISTINCT
    ~ TT<<FF  /\  ~ FF<<TT  /\  ~ TT<<UU  /\  ~FF<<UU
```

Ordered pairs

```
MK_PAIR         !x. (FST x, SND x) == x

FST_PAIR        !x y. FST (x,y) == x

SND_PAIR        !x y. SND (x,y) == y
```

Fixed points

```
FIX_EQ          !f. FIX f == f (FIX f)
```

There is one axiom scheme: beta-conversion. If x is a variable, and u, v are terms, and u[v/x] denotes the substitution of v for x in u, then

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

LCF includes some basic lemmas that follow from the axioms.

Equality

```
EQ_REFL         !x. x==x

EQ_SYM          !x y. x==y  ==>  y==x

EQ_TRANS        !x y z. x==y  /\  y==z  ==>  x==z
```

Extensionality of ==

```
EQ_EXT          !f g. (!x. f x == g x)  ==>  f==g
```

Distinctness of the truth values

```
TR_EQ_DISTINCT
                ~ TT == FF  /\  ~ FF == TT  /\
                ~ TT == UU  /\  ~ UU == TT  /\
                ~ FF == UU  /\  ~ UU == FF
```

The completely undefined function

```
MIN_COMB        !x. UU x == UU

MIN_ABS         \x.UU == UU
```

Validity of Eta-Conversion

```
ETA_EQ          !f. \x.f x == f
```

5. Predicates

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

```
STRICT f   <=>  f UU == UU

TRANSITIVE p   <=>
   !x y z. p x y == TT  /\  p y z == TT  ==>  p x z == TT
```

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

```
FLAT (UU:*)  <=>
        !x1:*. !x2:*. x1<<x2  ==>  UU==x1  \/  x1==x2


ISOMORPHIC (UU:*, UU:**)   <=>
        ?f g. (!x:*. g(f x) == x) /\ (!y:**. f(g y) == y)
```

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

6. Predicate Calculus Rules

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

6.1. Rules for quantifiers

Forall introduction

```
GEN: term -> thm -> thm
      x

       A(a)
     ---------  where the variable "a" is not free in assumptions of premiss
      !x.A(x)
```

Forall elimination

```
SPEC: term -> thm -> thm
       t

         !x.A(x)
      ------------
          A(t)
```

Exists introduction

```
EXISTS: (form # term) -> thm -> thm
                 t

          A(t)
      -------------
         ?x.A(x)
```

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

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

or

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

Exists elimination

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

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

where the variable "a" is not free anywhere except in B's assumption A(a)

6.2. Rules for basic connectives

Conjunction introduction

CONJ: thm -> thm -> thm

```
     A      B
-----------------
      A /\ B
```

Conjunction elimination

CONJUNCT1, CONJUNCT2: thm -> thm

```
     A /\ B
---------------
  A          B
```

Disjunction introduction

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

```
      A                  B
-----------       -------------
  A \/ B              A \/ B
```

Disjunction elimination

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

```
     A \/ B      [ A ] C      [ B ] C
---------------------------------------
                  C
```

Implication introduction

```
DISCH: form -> thm -> thm
        A

     [ A ]  B
   -----------
     A ==> B
```

Implication elimination

```
MP: thm -> thm -> thm

     A ==> B       A
   -------------------
           B
```

6.3. Rules for derived connectives

The formula A <=> B is logically equivalent to (A ==> B) /\ (B ==> A), but LCF does not expand it as such, to avoid duplicating A and B. The rules CONJ_IFF and IFF_CONJ map between the two formulas.

The formula ~A denotes A ==> FALSITY(). The rules for negation are special cases of the rules for implication, and are not provided separately. Any inference rule that works on implications also works on negations.

If-and-only-if introduction

```
CONJ_IFF: thm -> thm

     (A ==> B)  /\   (B ==> A)
   -----------------------------
              A <=> B
```

If-and-only-if elimination

IFF_CONJ: thm -> thm

```
             A <=> B
   ----------------------------
      (A ==> B)  /\  (B ==> A)
```

Negation introduction

DISCH: form -> thm -> thm

```
       A

      [ A ]  FALSITY()
   ---------------------
          ~A
```

Negation elimination

MP: thm -> thm -> thm

```
       ~A        A
   -----------------
       FALSITY()
```

7. Additional rules

Assumption

ASSUME: form -> thm

```
        A

   --------------
      [ A ]  A
```

Contradiction rule

```
CONTR: form -> thm -> thm
        A

        FALSITY()
    ---------------
           A
```

Classical contradiction rule

```
CCONTR: form -> thm -> thm
         A

        [ ~A ]  FALSITY()
    ---------------------
              A
```

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

Simultaneous Substitution

```
SUBST: (thm # term)list -> form -> thm -> thm
                xi            A(xi)

        ti == ui     A(ti)
    -------------------------
             A(ui)
```

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

Instantiation of Types

```
INST_TYPE (type # type)list -> thm -> thm
            tyi    vtyi
```

where the type variables vtyi do not occur in the assumptions

```
        A(vtyi)
    -----------------
        A(tyi)
```

Instantiation of Terms

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

where the variables xi do not occur in the assumptions

```
        A(xi)
    -----------------
        A(ti)
```

8. Fixed point induction

Fixed-point induction on a variable x and formula A(x) is only sound if the formula A is "chain-complete" with respect to x. For any ascending chain of values z1, z2, ..., if A(zi) is true for every zi, then A(z) must hold for the least upper bound, z. In Scott's original logic, the only formulas are conjunctions of inequivalences, which are all chain-complete. Things are more complicated in PPLAMBDA, with its implications, disjunctions, quantifiers, and user-definable predicates.

8.1. Admissibility for short types

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

An important special case is that all <u>structural</u> inductions over flat types are admissible.

<u>Definition</u>: A <u>short</u> type is one with no infinite ascending chains.[2]

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

> Suppose that f is the limit (least upper bound) of an ascending chain f0, f1, ...

(1) Suppose that !z.A(fi z) holds for all i.

> Then the limit case !z.A(f z) holds also, for consider any z'. Since the type of "fi z'" is short, the chain (f0 z'), (f1 z'), ... reaches its limit at some finite i.[3] For this i, "fi z'" equals "f z'".
>
> Our assumption (1) implies that A(fi z') holds, so A(f z') holds too. Since we chose z' arbitrarily, we conclude !z.A(f z). Thus the induction is admissible.

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

---

[2] Gordon et al. [1979] call these "easy" types.

[3] The intuitionistic validity of this inference is questionable, as is the justification of the admissibility rule for disjunctions. Both rely on the "pigeon-hole principle": if you partition an infinite set in two, one of the two sets must be infinite.

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

Example:

> ?z.f z==UU, where f maps every natural number to "TT". Suppose that for all i, fi maps all numbers less than i to TT, the rest to UU. Then f is the limit of the fi, the formula holds for each fi, and the formula does not hold in the limit.

LCF allows induction whenever the above term contains only constants, free variables, and outermost universally quantified variables. The test ignores quantifiers over finite types, as these are essentially finite disjunctions or conjunctions. The test also notices the special cases where free occurences of t<<u or t==u are chain-complete, as discussed on page 77 of Gordon et al. [1979]. It treats t==UU as the equivalent formula t<<UU, which is chain-complete in t in both positive and negative positions.

8.2. Stating type properties in PPLAMBDA

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

```
|- !x:ty. x==c1  \/  ...  \/  x==cn
```

where the ci are constants, states that the type "ty" is finite. Any theorem

```
|- !x1 ... xn:ty. x1<<x2  /\  ...  /\  x(n-1)<<xn  ==>
     UU==x1  \/  x1==x2  \/  ...  \/  x(n-1)==xn
```

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

```
!x1 x2. x1<<x2  ==>  UU==x1  \/  x1==x2
```

To inform LCF of such properties when checking admissibility, the induction rule accepts a list of theorems, B1, ..., Bn. Each Bi should state the finiteness or shortness of a type.

Scott Fixed-Point Induction

```
INDUCT: (term list) -> (thm list) -> (thm # thm) -> thm
              funi           Bi

        B1 ... Bn     A(UU)     !f1 ... fn. A(fi)  ==>  A(funi fi)
    -----------------------------------------------------------------
                      A(FIX funi)
```

9. Derived Inference Rules

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

9.1. Predicate Calculus Rules

[4] Intuitionists will be glad to hear that none use the classical contradiction rule, CCONTR.

<u>Substitution</u> (at specified occurrence numbers)

```
SUBS: (thm list) -> thm -> thm
SUBS_OCCS: ((int list) # thm) list -> thm -> thm

     ti == ui        A(ti)
   ----------------------
           A(ui)
```

<u>Generalising a theorem over its free variables</u>

```
GEN_ALL: thm -> thm

           A(xi)
   ----------------------
       !x1...xn.A(xi)
```

<u>Discharging all hypotheses</u>

```
DISCH_ALL: thm -> thm

        [A1; ...; An] B
-------------------------------
    A1 ==> ... ==> An ==> B
```

<u>Iterated SPEC</u>

```
SPECL: (term list) -> thm -> thm
            ti

        !x1 ... xn. A(xi)
   --------------------------  SPECL [t1; ...; tn]
            A(ti)
```

<u>Un-discharging an assumption</u>

```
UNDISCH: thm -> thm

        A   ==>  B
   -----------------
         [A] B
```

<u>Undischarging all assumptions</u>

```
UNDISCH_ALL: thm -> thm

         A1 ==> ... ==> An ==> B
    -----------------------------------
             [A1; ...; An] B
```

<u>Specialisation over outer universal quantifiers</u>

```
SPEC_ALL: thm -> thm

        !x1 ... xn. A[xi]
    -------------------------     where the xi' are not free in hyps of A
            A[xi'/xi]
```

<u>Using a theorem A to delete a hypothesis of B</u>

```
PROVE_HYP: thm -> thm -> thm
            A      B

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

<u>Conjoining a list of theorems</u>

```
LIST_CONJ: (thm list) -> thm
               Ai

        A1 ... An
    --------------------          where n>0
     A1 /\ ... /\ An
```

<u>Splitting a theorem into its conjuncts</u>

```
CONJUNCTS: thm -> (thm list)

     A1 /\ ... /\ An
    --------------------          where n>0
        A1 ... An
```

Iterated Modus Ponens

```
LIST_MP: (thm list) -> thm -> thm
             Ai

      A1 ... An       A1 ==> ... ==> An ==> B
   ----------------------------------------
                      B
```

Contrapositive of an implication

```
CONTRAPOS: thm -> thm

       A ==> B
   --------------
      ~B ==> ~A
```

Converting disjunction to implication

```
DISJ_IMP: thm -> thm

       A \/ B
   -------------
      ~ A ==> B
```

```
DISJ_CASES_UNION: thm -> thm -> thm -> thm

       A \/ B       [A] C      [B] D
   -------------------------------------
                  C \/ D
```

9.2. Rules About Functions and the Partial Ordering

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

Reflexivity of equality

```
REFL: term -> thm
   "t"   --->    |-"t==t"
```

Symmetry of equality

```
SYM: thm -> thm

      t==u
   -----------
      u==t
```

Analysis of equality

```
ANAL: thm -> thm

             t == u
   ------------------------
      t << u  /\  u << t
```

Synthesis of equality

```
SYNTH: thm -> thm

      t << u  /\  u << t
   ------------------------
             t == u
```

Transitivity (infix operator)

```
TRANS: thm -> thm -> thm

     t =< u       u =< v        possibly different relations
   ------------------------
            t =< v              << unless both hypotheses use ==
```

Extensionality

```
EXT: thm -> thm

      !x. u x =< v x
   --------------------
          u =< v
```

Minimality of UU

```
MIN: term -> thm
       t

    "t"  --->  |- "UU << t"
```

```
LESS_UU_RULE: thm -> thm

       t<<UU
    -----------
       t==UU
```

Construction of a combination

```
LE_MK_COMB: (thm # thm) -> thm

        f =< g
        t =< u
   -------------
     f t =< g u        << unless both hypotheses use ==
```

Application of a term to a theorem

```
AP_TERM: term -> thm -> thm
           t

        u =< v
    -------------
     t u =< t v
```

Application of a theorem to a term

```
AP_THM: thm -> term -> thm
                 t

        u =< v
    -------------
     u t =< v t
```

Construction of an abstraction

```
MK_ABS: thm -> thm

      !x. u =< v
   ----------------
    \x.u =< \x.v
```

```
HALF_MK_ABS: thm -> thm

     !x. u x =< t
   ----------------
      u =< \x.t
```

Alpha-conversion (renaming of bound variable)

```
ALPHA_CONV: term -> term -> thm
             x      (\y.t)

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

10. Differences from Edinburgh LCF

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

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

10.1. Formula Identification

Edinburgh LCF forced every formula into a canonical form. For instance, you could not build the formulas "!x.TRUTH()" and "A==>TRUTH()". The constructor functions mk_forall and mk_imp automatically simplified these to TRUTH().[5] This "formula identification" caused unpredictable behavior in programs that manipulated formulas.

Cambridge LCF does not have formula identification. Instead, you can implement your own canonical forms in ML. The constructor and destructor functions are inverses of each other. For instance,

```
dest_conj (mk_conj (A, B))  --->  (A, B)
```

10.2. The Definedness Function DEF

Edinburgh LCF provided a function DEF, satisfying

```
DEF UU == UU
DEF x == TT                 for any x except UU
```

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

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

10.3. Data Structures

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

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

References

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

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

A. Cohn. "The Equivalence of Two Semantic Definitions: A Case Study in LCF." SIAM Journal of Computing, May 1983.

M Dummet. Elements of Intuitionism. Oxford University Press, 1977.

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

S. Igarashi. "Admissibility of Fixed-Point Induction in First Order Logic of Typed Theories," Memo AIM-168, Stanford University, 1972.

L. Paulson. "Recent Developments in LCF: Examples of Structural Induction." Technical Report No. 34, Computer Laboratory, University of Cambridge, 1983.

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