Technical Report

UCAM-CL-TR-21
ISSN 1476-2986

Number 21

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The correctness of a precedence parsing algorithm in LCF

A. Cohn

April 1982

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

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

© 1982 A. Cohn

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

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

ISSN 1476-2986

THE CORRECTNESS OF A PRECEDENCE PARSING ALGORITHM IN LCF

A. Cohn
University of Cambridge, Computer Laboratory,
Corn Exchange Street, Cambridge CB2 3QG, England

April 1982

Abstract. This paper describes the proof in the LCF system of a correctness property of a precedence parsing algorithm. The work is an extension of a simpler parser and proof by Cohn and Milner (Cohn & Milner 1982). Relevant aspects of the LCF system are presented as needed. In this paper, we emphasize (i) that although the current proof is much more complex than the earlier one, many of the same *metalanguage strategies* and *aids* developed for the first proof are used in this proof, and (ii) that (in both cases) a general strategy for doing some limited forward search is incorporated neatly into the overall goal-oriented proof framework

1. INTRODUCTION

In this paper we give an account of the proof in LCF of a correctness property of a precedence parsing algorithm for a small expression language. The correctness of the algorithm is relative to a particular 'unparsing' (parse tree flattening) function. The work is an extension of a parser (for a language fully disambiguated by parentheses) formulated and proved by R. Milner (Cohn & Milner 1982). Milner's work was based in turn on a parser proved in the Boyer-Moore system by P. Gloess (Gloess 1978). This paper is intended to be self-contained, although it follows naturally from Milner's earlier parser proof. Relevant aspects of the LCF system are explained as required, but readers unfamiliar with the system may wish to refer to the system manual (Gordon *et al.*1979).

The points we emphasize here are (i) that although the current proof is much longer and more complex than the earlier LCF parser proof (about 150 times as long in the number of actual formal inferences), the metalanguage strategies and aids used in the original problem form a large part of what is needed for the current one, and (ii) that (in both cases) a general strategy for doing some limited forward proof is incorporated neatly into the overall goal-oriented proof framework. More generally, we emphasize the success of the LCF system in the expression and solution of this rather complicated problem; in particular, (i) the natural way in which the parsing algorithm and its correctness property are expressed in LCF's extensible logic PPLAMBDA, and (ii) the power of LCF's metalanguage, ML, in expressing proof generation strategies and in extending the logic and implementing new rules of inference.

We describe the parser and unparser and the correctness property in section 2. A sketch of the informal proof is given in section 3. Section 4 describes the formalisation of the problem in PPLAMBDA, and the generation of the proof using ML strategies is presented in section 5.

2. THE PRECEDENCE PARSING ALGORITHM

2.1. The language and domains

Words (well-formed expressions), w, of our language are given by

$$w ::= \; I \;\mid\; LB \; w \; RB \;\mid\; u \; w \;\mid\; b \; w$$

where I is an identifier, u and b are unary and binary operators respectively, and LB and RB are constants standing for left and right brackets, respectively. Brackets are optional; operators $op, op_1, op_2, \ldots,$ have *precedences* which can be compared. We write

$$op_1 > op_2, \qquad op_1 = op_2$$

to indicate, in turn, that op_1 has greater or equal binding power than op_2. Precedences enable words to be interpreted unambiguously. (We assume that operators with equal precedence are identical, for simplicity.)

To state the problem clearly, we specify the following domains and abbreviations:

$I \in$ IDEN	*identifiers*
$u \in$ UNOP	*unary operators*
$b, b_1, b_2 \in$ BINOP	*binary operators*
$op \in$ OP = UNOP + BINOP	*operators*
BRAC = {LB,RB}	*brackets*
SYMB = IDEN + BRAC + UNOP + BINOP	*symbols*
WORD = SYMB LIST	*words*
$t, t_1, t_2 \in$ PTREE = IDEN + (UNOP × PTREE) + (BINOP × PTREE × PTREE)	*parse trees*

SYMB LIST is intended to be a recursively defined domain (we could also write IDEN LIST, integer LIST, etc.). We assume LISTs have the usual primitives: "." for the constructor, "@" for concatenation, and "nil" to denote the empty LIST.

PTREE is also a recursively defined domain. We let

```
mkTIP : IDEN -> PTREE
mkUN : UNOP -> PTREE -> PTREE
mkBIN : BINOP -> PTREE -> PTREE -> PTREE
```

be the functions which construct trees from their components.

Binary operators must be either left or right associative, so that we can parse expressions such as "I_1 b I_2 b I_3". We introduce predicates "left" and "right" on binary operators, to determine associativity.

2.2 The parser and unparser

The parser works by mapping states to states. A state consists in (i) the whole or remaining part of the input word (a list of symbols), (ii) a stack (or list) of operators and brackets to be used later, and (iii) a list of parse trees to be combined eventually into the final parse tree.

$$\text{ParserState} = \text{SYMB LIST} \times \text{OP}' \text{ LIST} \times \text{PTREE LIST}$$

where OP' = BRAC + OP. The function "parse" has type: ParserState -> ParserState.

The parser begins by examining the leading symbol of the input word. A typical ParserState is written (w,os,rs) for the input word, operator-bracket stack and result stack. The clauses defining the parser (the function "parse") are given below. (We are not very formal at this point; for example, "b.w" (where "b" has type BINOP and "w" has type SYMB LIST) is not really well-typed, but suggests coercing "b" to have type SYMB. In section 4 we introduce the necessary injection and projection functions.)

1. parse(nil, nil, rs) = (nil, nil, rs)
2. parse(nil, b.os, t_2.t_1.rs) =
   parse(nil, os, (mkBIN b t_1 t_2).rs)
3. parse(nil, u.os, t.rs) = parse(nil, os, (mkUN u t).rs)
4. parse(b.w, nil, rs) = parse(w, b.nil, rs)
5. parse(I.w, os, rs) = parse(w, os, (mkTIP I).rs)
6. parse(u.w, os, rs) = parse(w, u.os, rs)
7. *if* u > op *or* u = op *then*
   parse(op.w, u.os, t.rs) = parse(op.w, os, (mkUN u t).rs)
8. *if* b > op *then*
   parse(b.w, op.os, rs) = parse(w, b.op.os, rs)
9. *if* b_2 > b_1 *then*
   parse(b_1.w, b_2.os, t_2.t_1.rs) =
   parse(b_1.w, os, (mkBIN b_2 t_1 t_2).rs)
10. *if* b_2 = b_1 *then if* left b_1 *then*
    parse(b_1.w, b_2.os, t_2.t_1.rs) =
    parse (b_1.w, os, (mkBIN b_2 t_1 t_2).rs) *else*
    parse(b_1.w, b_2.os, rs) = parse(w, b_1.b_2.os, rs)
11. parse(b.w, LB.os, rs) = parse(w, b.LB.os, rs)
12. parse(LB.w, os, rs) = parse(w, LB.os, rs)
13. parse(RB.w, os, rs) = clear(w, os, rs)
14. clear(w, u.os, t.rs) = clear(w, os, (mkUN u t).rs)
15. clear(w, b.os, t_2.t_1.rs) =
    clear(w, os, (mkBIN b t_1 t_2).rs)
16. clear(w, LB.os, rs) = parse(w, os, rs)

The function "clear" has the same type as "parse"; the two are mutually recursive.

The workings of the parser are explained as follows:
2. and 3. If there is no more of the input word, the stacked-up tree fragments are simply combined, using the operator stored at the front of the operator list. 1. If there is no operator list, the parser terminates.
4. and 6. If the input word starts with a binary operator and the operator list is empty, *or* if the leading symbol of the input is a unary operator, the operator in question is stored on the operator stack.
5. If the leading symbol is an identifier, a corresponding one-tip tree is constructed and placed on the result stack, to be incorporated in the final parse tree later. 7 - 10. If the leading symbol of the input is an

operator, and the first element of the operator stack is too, then the precedences of the two operators are compared. If the stacked operator has the greater precedence (*or* if the operators are identical and left associative), then the one (or two) most recent subtree(s) are combined into a tree, with the top of the operator stack as its top node. Otherwise, the leading symbol of the input is placed on the operator stack for later use, and the analysis of the input continues. 11. and 12. If the input starts with a left bracket, the bracket is placed on the operator stack; if a left bracket is uncovered on the operator stack, the leading input symbol is simply stacked 'over' it. 13. If a right bracket is uncovered on the input word, the function "clear" is called. The clauses defining it are 14. - 16. The function "clear" keeps building up tree fragments until the corresponding left bracket is found on the operator stack; then parsing begins again.

These clauses are sufficient to unambiguously parse any well-formed word, and that is all that is required for what we prove.

The unparsing function flattens trees into words. Although unparse is really a relation rather than a function (since there is a whole class of flattening functions that would do) we arbitrarily choose the unparsing function which adds the *least* number of brackets to the word returned in order to be able to parse it again. (To show the correctness of the parser on *all* inputs, we would have to show the desired property of the parser for every unparsing function. We believe that the other proofs would be similar to the current one but easier, as the complexity of the proof arises mainly in those cases where precedence is not disambiguated by brackets. The problem, in terms of LCF, is that we do not know at present how to formulate a sentence in the logic which expresses correctness for *all* unparsing functions.)

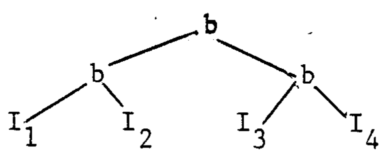
The function "unparse" takes as parameters (i) an operator relative to which it unparses, and (ii) an indication of whether it is unparsing a left, right or 'only' subtree. The precedence of the operator determines whether brackets are needed to 'protect' the word returned, and the side-indicator is needed to place brackets appropriately in cases where associativity is involved. We let the domain SIDE contain the side-indicators "L", "R" and "N" for left, right and 'neither' subtrees, respectively. The type of "unparse" is thus:

```
unparse  : OP -> SIDE -> PTREE -> SYMB LIST
```

It is defined by the three clauses below:

17. unparse op s (mkTIP I) = I.nil
18. unparse op s (mkUN u t) =
    if op > u *then* LB.u.(unparse u N t) @ RB.nil *else*
    u.(unparse u N t) @ nil
19. unparse op s (mkBIN b t_1 t_2) =
    let x = unparse b L t_1 *and* y = unparse b R t_2
    in if op > b *then* LB.x. @ b.y @ RB.nil *else*
    if op = b *and* ((left op *and* s = R) *or* (right op *and* s = L))
    then LB.x @ b.y @ RB.nil *else*
    if op = b *and* ((left op *and* s = L) *or* (right op *and* s = R))
    then x @ b.y @ nil *else* x @ b.y @ nil

That is: 17. A one-tip tree is flattened into a one-identifier word. 18. A unary tree is unparsed recursively into the top operator symbol followed by the unparsed subtree, with brackets around the *whole* resulting word if the precedence of the 'passed down' operator, op, is greater than the precedence of the top node, u, of the tree. The subtree is unparsed relative to that top node, and to the side-indicator "N". 19. Binary parse trees are treated in a similar way, the top node, b, appearing between the two unparsed subtrees. The left and right subtrees are parsed relative to the side-indicators "L" and "R", respectively. Brackets are placed around the whole resulting word if required (as in the unary case) to protect against the precedence of op (relative to b), *or* if there is a case of equal precedence of op and b, and the side-indicator is for a right subtree while the operator is left associative (or *vise versa*). This is to cope with parse trees such as



where b is left associative; we wish to unparse this tree into:

I_1.b.I_2.b.LB.I_3.b.I_4.RB @ nil

At the 'top level' we simply unparse words relative to some fixed unary operator, and to the side-indicator "N".

3. THE STATEMENT AND PROOF OF THE CORRECTNESS PROPERTY

In this section we describe the correctness property of our parsing algorithm, and we sketch the informal proof. (The reader should bear in mind that although the property is rather complicated to state, our main interest is not in the property itself, but in the *structure of the proof*.)

3.1. The statement of correctness

The formula expressing the correctness property of the parser relative to the unparser is something like this:

```
∀ t op s. parse((unparse op s t)@ nil, nil, nil) =
          (nil, nil, t.nil)
```

That is, if we unparse a tree "t" (relative to an operator "op" and a side-indicator "s") to get a word, and then parse that word in a state in which the operator and result stacks are empty, we get back a state comprising the empty input word, an empty stack of pending operators, and a result list containing exactly the original tree "t".

We actually have to prove something more general (though this is not quite it yet):

```
∀ t op s w os rs. parse((unparse op s t)@ w, os, rs) =
                  parse(w, os, t.rs)
```

That is, if "t" is unparsed, attached to an arbitrary word "w", and parsed in an arbitrary state, the original tree "t" is put on the result stack, and the word "w" is isolated.

However, we are dealing with *domains* (complete partial orders) in which there is always an undefined element "⊥". (This is used for representing non-terminating computations.) Our property is not true for *all* trees, as trees may be infinite, or contain undefined parts. We therefore introduce a predicate "WD [t]" of trees which characterises finite, well-defined trees. (This treatment follows (Cohn & Milner 1982) in which the same problem arises.) The properties we require of "WD" are as follows, where the predicate "DEF" determines whether an element of a domain is defined:

if DEF[t] does not hold *then* WD[t] does not hold

$\forall$ I. WD[mkTIP I]

$\forall$ u t. WD[mkUN u t] $\supset$ WD[t]

$\forall$ u t. WD[mkUN u t] $\supset$ DEF[u]

$\forall$ b t_1 t_2. WD[mkBIN b t_1 t_2] $\supset$ WD[t_1]

$\forall$ b t_1 t_2. WD[mkBIN b t_1 t_2] $\supset$ WD[t_2]

$\forall$ b t_1 t_2. WD[mkBIN b t_1 t_2] $\supset$ DEF[b]

In addition, we require "op" and "s" to be defined, for our property to be true. (Also, "L", "R" and "N" must be defined, and "op_1 > op_2" must be defined if an only if "op_1" and "op_2" are as well.) All of these conditions, however, are still not enough to make the conjecture true. It may still be the case that when "t" is unparsed and attached to "w", operators in "w" may take precedence and cause a different reparsing than intended. In the end, three rather elaborate relations between "op", "w", "os" and "s" are found to be sufficient. We introduce predicates "isunary" and "isbinary" to determine whether operators are unary or binary. We let the functions "hd", "tl" and "null", on lists respectively take the head and tail of a list, and determine whether a list is empty.

```
rel1(op,w,s)  iff either 1. null w
                         2. isbinary(hd w) and op > (hd w)
                         3. isbinary(hd w) and op = (hd w)
                            and either 1. s = N
                                       2. s = L
                                       3. left op
                         4. hd w = RB

rel2(op,os,s) iff either 1. null os
                         2. op > (hd os)
                         3. op = (hd os)
                            and either 1. s = N
                                       2. s = R
                                       3. right op
                         4. hd os = LB

rel3(op,s)    iff either 1. s = N and isunary op
                         2. (s = L or s = R) and isbinary op
```

If relation "rel1" holds of "(op,w, s)" it ensures that the leading symbol of w (unless w is empty) is a binary operator whose precedence is not greater than that of "op"; if it is equal in precedence we need the extra insurance that either "op" is left associative, or that we are not dealing with a right subtree of a binary tree. The relation "rel2", likewise, ensures that the leading symbol of the operator stack does not have stronger precedence than "op". The relation "rel3" is just a consistency condition; if we are dealing with a binary tree, "op" should be a binary operator, and not otherwise.

These relations are technical rather than deep, and are sufficient to make the conjecture true. We prove:

```
∀ t op s w os rs. WD[t] and DEF[op] and DEF[s] and
                  rel1(op,w,s) and rel2(op,os,s) and
                  rel3(op,s) ⊃
                  parse((unparse op s t)@ w, os, rs) =
                  parse(w, os, t.rs)
```

To get the desired result, we choose a fixed unary operator which is defined (called it "UO") and prove:

```
rel1(UO, nil, N)
rel2(UO, nil, N)
rel3(UO, N)
```

which implies that

```
∀ t. WD[t] ⊃ parse((unparse UO N t)@ nil, nil, nil) =
             (nil, nil, t.nil)
```

3.2. The informal proof

In this section we sketch the informal proof of the correctness property of the parser. An understanding of the structure of the proof motivates the metalanguage strategies which generate the formal proof in LCF.

The proof of the main theorem is by structural induction on parse trees. We appeal to the following rule of induction (in which hypotheses are written above the line and conclusion below):

$$\frac{\begin{array}{l} P[\bot] \\ \forall\ \mathrm{I}.\ P[\mathrm{mkTIP\ I}] \\ \forall\ \mathrm{u\ t}.\ P[\mathrm{t}] \supset P[\mathrm{mkUN\ u\ t}] \\ \forall\ \mathrm{b}\ \mathrm{t}_1\ \mathrm{t}_2.\ P[\mathrm{t}_1]\ \mathit{and}\ P[\mathrm{t}_2] \supset P[\mathrm{mkBIN\ b}\ \mathrm{t}_1\ \mathrm{t}_2] \end{array}}{\forall\ \mathrm{t}.\ P[\mathrm{t}]}$$

"*P*[t]" means that the property "*P*" holds of tree "t". "⊥" is the undefined tree. The rule states that if "*P*" holds in the basis cases (the Undefined and Tip Cases), and if "*P*" is preserved when trees are built up in the step cases (the Unary and Binary Cases), then "*P*" holds for all trees "t".

We note that we must prove our property for *all* values of the quantified variables so that the induction hypotheses can be instantiated at a variety of instances, and also that the whole implication must be proved by induction. (Thus, in order to invoke an induction hypothesis, its antecedent must be satisfied first.)

The proof has four main cases raised by the induction on "t". Several of these have subcases based on the possible values if the side-indicator "s", the ways in which the relations "rel1" and "rel2" may hold, the precedence of "op" relative to the top node of the parse tree in question, and the associativity of "op". Certain facts about lists, precedences and the propositional calculus are used without mention in the informal proof sketch (we formalise these in section 4).

<u>To prove</u>: ∀ t op s w os rs. WD[t] *and* DEF[op] *and* DEF[s] *and* rel1(op,w,s) *and* rel2(op,os,s) *and* rel3(op,s) ⊃ parse((unparse op s t)@ w, os, rs) = parse(w, os, t.rs)

<u>Undefined Case</u> This is vacuously true, since WD[⊥] doesn't hold

<u>Tip Case</u> <u>Assume</u>: WD[t], DEF[op], DEF[s], rel1(op,w,s), rel2(op,os,s), rel3(op,s)
<u>Prove</u>: parse((unparse op s (mkTIP I))@ w, os, rs) = parse(w, os, (mkTIP I).rs)

This follows by using parser/unparser clauses 17 and 5 simply as rewrite rules.

```
Unary Case  Assume: ∀ op s w os rs. WD[t] and DEF[op] and
                 {                    DEF[s] and rel1(op,w,s)
                 {                    and rel2(op,os,s) and
     Induction   {                    rel3(op,s) ⊃
     Hypothesis  {   parse((unparse op s t)@ w, os, rs) =
                 {   parse(w, os, t.rs)

            Assume also: WD[mkUN u t],   DEF[op],  DEF[s],
                         rel1(op,w,s),  rel2(op,os,s),
                         rel3(op,s)
            Prove:   parse((unparse op s (mkUN u t))@w, os, rs) =
                     parse(w, os, (mkUN u t).rs)
```

Cases

Three cases must be considered: where op > u, u = op and u > op. In the first case, we rewrite (using clauses 18, 12 and 6) to change the left-hand-side to:

```
parse((unparse u N t)@ RB.w, u.LB.os, rs)
```

We could now apply (instantiate) the induction hypothesis *if* we could satisfy its antecedent, namely:

```
WD[t] and DEF[u] and DEF[N] and
rel1(u, RB.w, N) and rel2(u, u.LB.os, N) and
rel3(u, N)
```

The conjuncts are argued separately. The second three are really lemmas with slightly complicated proofs of their own; we discuss these proofs later. WD[t] follows from the property of WD that

```
∀ t. WD[mkUN u t] ⊃ WD[t]
```

combined with the assumption that WD[mkUN u t]. We assumed that DEF[N]. We infer DEF[u] from the fact that op > u. Once the antecedent is thus established the appropriate instance of the conclusion follows, reducing the left-hand-side further, to

```
parse(RB.w, u.LB.os, t.rs)
```

and the rest follows, using clauses 13, 14 and 16 as rewrite rules.

In the other two cases, where u > op and u = op, a further case argument is made based on the four ways in which rel1(op,w,s) can hold. (These arguments may be factored out and also proved as lemmas.) One of the following must hold:

1. null w
2. isbinary(hd w) *and* op > (hd w)
3. isbinary(hd w) *and* op = (hd w) *and*
   (s = N *or* s = L *or* left op)
4. hd w = RB

In each of the four subcases we use the (implicitly assumed) transitivity of > =; the fourth subcase requires a use of the induction hypothesis. The four cases, for both u > op and u = op, use clauses 3, 7, 13 and 14 to rewrite.

Binary Case: This case is similar to the preceding one, so we give less detail.

Assume:

Induction Hypothesis: ∀ op s w os rs. WD[t_1] *and* DEF[op] *and* DEF[s] *and* rel1(op,w,os) *and* rel2(op,os,s) *and* **rel3(op,s)** ⊃ parse((unparse op s t_1)@w, os, rs) = parse(w,os,t_1.rs)

Induction Hypothesis: ∀ op s w os rs. WD[t_2] *and* DEF[op] *and* DEF[s] *and* rel1(op,w,os) *and* rel2(op,os,s) *and* rel3(op,s) ⊃ parse((unparse op s t_2)@w, os, rs) = parse(w,os,t_2.rs)

Assume also: WD[mkBIN b t_1 t_2], DEF[op], DEF[s], rel1(op,w,s), rel2(op,os,s), rel3(op,s)

Prove: parse((unparse op s (mkBIN b t_1 t_2))@ w, os, rs) = parse(w, os, (mkBIN b t_1 t_2).rs)

Again, there are three cases, depending on the relative precedence of b and op.

Case op > b This case requires an instantiation of each of the induction hypotheses. The arguments for satisfying the antecedents are as in the Unary Case. (Clauses 19, 12, 11, 13, 15 and 16 are used as rewrite rules.)

Case b > op This case is also easy; we do a case analysis on the four ways in which rel1(op,w,s) can hold, followed by a similar case analysis for rel2. (Clauses 19, 4, 8, 11, 2, 9, 13 and 15 are used.)

Case b = op This is the difficult case, as there is a conflict of precedences; the outcome of parsing depends on whether we are in a left or right subtree, and on whether the operator "b" is left or right associative. Thus we do a further case analysis on whether s is L, N or R; and on whether b is left or right associative.

Subcase s = N This case is *a priori* impossible since we have assumed that rel3 holds of op and s -- but b = op implies that b and op are identical, so op is also binary. This contradicts the assumption that rel3 holds.

<u>Subcases (left op *and* s = R), (right op *and* s = L)</u>

In these cases we simply rewrite (using clauses 19, 12, 11, 13, 15 and 16) with an instantiation of each of the induction hypotheses during the rewriting.

<u>Subcases (left op *and* s = L), (right op *and* s = R)</u>

These two cases are messier because brackets are not inserted to disambiguate; associativity makes it clear how to parse. The two cases require another case analysis based on the ways in which rel1(op,w,s) can hold, followed by a similar case analysis on rel2. The rewritings involve clauses 19, 4, 8, 11, 2, 9, 10, 13 and 15, and (in both cases) an instantiation of each induction hypothesis. Some of the cases are argued by contradiction. For example, where b = op and left op and s = L, if we consider the ways in which rel2(op,os,s) might hold, it cannot be that the third way applies -- where op = (hd os). If it did apply, we would have in addition that s = N or S = R or else right op, any of which contradict the assumptions of this particular case.

The proof, as indicated, depends on several lemmas which allow us to use the induction hypotheses (by satisfying their antecedents); these must be proved as well. A typical lemma states that if rel1 holds originally, it holds for one of the subsequent instances of the induction hypothesis:

<u>To prove</u>: ∀ op w s b. rel1(op,w,s) *and* b = op *and* right op
and s = R ⊃
rel1(b,w,R)

This lemma would be used in the Binary Case, in the Subcase in which b = op, and so on. There is a lemma for each instance required of each induction hypothesis, but all the proofs are similar. We sketch this one as an example. (There are about thirty in all.)

<u>Assume</u>: rel1(op,w,s), b = op, right op, s = R
<u>Prove</u>: rel1(b,w,R)

As usual, we perform a case analysis on the ways in which rel1 could hold. Where null w holds, rel1(b,w,R) holds in the same way. Where isbinary(hd w) *and* op > (hd w), we know that b and op must be identical, so isbinary b holds, and b > (hd w) by transitivity; thus rel1(b,w,R) also holds in the second way. In the third case, where isbinary(hd w) *and* op = (hd w) *and* (s = N *or* s = L *or* left op), we argue by contradiction: we have assumed s = R and that right op holds in this case. Thus rel1(op,w,s) cannot hold in the third way. Where (hd w) = RB, rel1(b,w,R) holds for that reason.

(About eight further lemmas can factor out the case arguments on rel1, rel2.) What is important to observe about the main proof and the proof of the lemma is that most of the steps are a matter of routinely rewriting or 'unfolding' according to the clauses which define "parse" and "unparse" (and facts about lists, precedences and the propositional calculus). Application of the induction hypotheses involves instantiation of their bound variables and satisfaction of their antecedents before the hypotheses can be used as rewrite rules. The proofs involve the usual steps of proving a universally quantified **statement** by proving the statement for arbitrary values; and of proving an implication by assuming the antecedent, proving the consequent, and later discharging the antecedent. There are several varieties of case arguments in the proof. A few of these lead to proofs by contradiction.

In the next two sections we go on to show how this problem can be stated formally in PPLAMBDA, and to show how the structure of the proof is reflected in ML strategies which generate the whole formal proof for us.

4. THE FORMALISATION

In this section we show how the parser and its statement of correctness are represented formally in LCF. This involves constructing a hierarchy of *theories* (extensions of the basic logic PPLAMBDA) to express the problem. The two main difficulties in the formalisation are (i) dealing neatly with the undefined cases which arise when all types correspond to domains, and (ii) expressing relations in PPLAMBDA.

LCF consists of the logic PPLAMBDA coupled with a general-purpose programming language, ML, through which logical objects are manipulated. The terms t, t_1 and t_2 of the logic, are given by

$$t ::= c \mid x \mid t_1\ t_2 \mid \backslash v.\ t \mid t_1, t_2$$

where c is a set of basic constants and x is a variable. A term can also be an application of one term to another, a lambda abstraction or an ordered pair. Basic constants include the truth values "UU", "TT" and "FF", for ⊥, true and false, respectively, the function "DEF" to test definedness, and several others. (The unusual notation, as for lambda expressions, is for the sake of machine printing.) Each term has a type which corresponds to a domain, such as the type "tr" of truth values. An expression "t:*" means that the object "t" has type "*", as in "TT: tr". (Type variables are *, **, etc.)

The formulae of PPLAMBDA are as in the predicate calculus. A formula w, w_1 or w_2 can be

$$w ::= \text{tautology} \mid t_1 == t_2 \mid t_1 << t_2 \mid w_1 \,\&\, w_2 \mid w_1 \text{ IMP } w_2 \mid\ !x.w$$

where x is a variable. That is, a formula can be one of several standard tautologies, an equivalence or inequivalence of terms (in the sense of the domain ordering), an implication or a universal quantification.

The logic may be extended by the use of metalanguage functions which add new types, constants and axioms to PPLAMBDA to form new logical theories. Theories can be built up hierarchically so that the types, etc., of one theory are accessible within descendent theories.

To express the parsing algorithm in PPLAMBDA we must be able to talk about parse *trees* and several kinds of *lists*. We choose to work with general theories of lists and trees -- useful in other proofs -- of which parse trees and our various lists are special instances. For types *, **, ***, we view (*,**,***)TREE and * LIST as ternary and unary type operators, respectively, which map triples of types, or types, into new types. For certain recursively defined types such as these, once we specify the 'shape' of the domain being defined, the construction of the corresponding PPLAMBDA theory is a standard matter. This process has been mechanised by R. Milner as an ML procedure; it is described in the appendix of (Cohn & Milner 1982). For example, the shape of the domain (*,**,***)TREE is:

```
* + (** × (*,**,***)TREE) + (*** × (*,**,***)TREE ×
                                   (*,**,***)TREE)
```

Given this, the ML procedure can declare new constants, such as

```
mkUN: ** -> (*,**,***)TREE -> (*,**,***)TREE
```

and the other various constructor functions for trees; and it can introduce new axioms defining these new constants in terms of primitive PPLAMBDA constants. In addition, the procedure can produce an ML function implementing the appropriate rule of structural induction for trees. The treatment of lists is similar. The ML procedure can define the constructor function "CONS: * -> * LIST -> * LIST", the constant "NIL: * LIST", and the list induction rule. (We can also add the constants "HD: * LIST -> *" and "TL: * LIST -> * LIST" and "NULL: * LIST -> tr" for hd, tl and null, and and "APP:* LIST -> * LIST -> * LIST" for append or concatenation.)

Next, we need some simple theories about propositional calculus, precedences and symbols.

The theory of propositional calculus required must include the constants "AND", "OR" and "NOT" which appear, for example, in the definitions of the relations rel1, rel2 and rel3. We build a theory containing the constants

```
OR: tr -> tr -> tr
AND: tr -> tr -> tr
NOT: tr -> tr
```

and axioms including

```
|- !p:tr. p OR TT == TT
|- !p:tr. p AND TT == p
|- NCT TT == FF
|- !p: tr. !q:tr. p AND q == TT IMP p == TT
|- !p: tr. !q:tr. p AND q == TT IMP q == TT
```

The symbol "|-" before a formula marks a theorem or axiom; this is discussed again later.

We next construct a theory of orderings to express the ordering of operator precedences. This theory is a descendent of the theory of propositional calculus, so we can use the constants and axioms of the latter. We introduce a new type "rank", and new constants:

```
=: rank -> rank -> tr
>: rank -> rank -> tr
```

Ranks are governed by a set of axioms which includes:

```
|- !r:rank. r > UU == UU
|- !r:rank. UU > r == UU
```

$|\!-\ !r_1{:}\mathrm{rank}.\ !r_2{:}\mathrm{rank}.\ r_1 = r_2 == TT\ IMP\ r_2 = r_1 == TT$

$|\!-\ !r_1{:}\mathrm{rank}.\ !r_2{:}\mathrm{rank}.\ !r_3{:}\mathrm{rank}.\ r_1 = r_2 == TT\ \&\ r_2 > r_3 == TT$
$\quad IMP\ r_1 > r_3 == TT$

$|\!-\ !r_1{:}\mathrm{rank}.\ !r_2{:}\mathrm{rank}.\ DEF\ r_1 == TT\ \&\ DEF\ r_2 == TT\ IMP$
$\quad (r_1 > r_2)\ OR\ (r_1 = r_2)\ OR\ (r_2 > r_1) == TT$

(We note that > and = are strict function, undefined on undefined arguments.)

We need also a domain of symbols, for the sort of symbols which make up words for our parser. The theory of symbols is a descendent of both list theory and ordering theory. It includes the new types "IDEN", "UNOP", "BINOP" and "BRAC" with the following type abbreviations:

```
OP = UNOP + BINOP
SYMB  = IDEN + BRAC + OP
OP'  = BRAC + OP
```

The new constants of the theory include

```
LB: BRAC
RB: BRAC
isRB: SYMB -> tr
left: BINOP -> tr
Prec: OP -> rank
BPrec: BINOP -> rank
UPrec: UNOP -> rank
```

where the function "isRB" determines whether a symbol is "RB"; "left" determines whether a binary operator is left associative; and the latter three functions return the precedence of an operator, a binary operator and a unary operator respectively. It is also convenient to have functions which do injections and projections for us:

```
PUTUW: UNOP -> SYMB LIST -> SYMB LIST
PUTUO: UNOP -> OP' LIST -> OP' LIST
PUTBO: BINOP -> OP' LIST -> OP' LIST
PUTBW: BINOP -> SYMB LIST -> SYMB LIST
PUTBRO: BRAC -> OP' LIST -> OP' LIST
```

These respectively place a unary operator on the symbol list, a unary operator on the operator list, a binary operator on the operator list, a binary operator onto a word, and a bracket on an operator list. We also add some more constants

```
GETOW: SYMB LIST -> OP
destBINOP: OP -> BINOP
mkBINOP: BINOP -> OP
mkUNOP: UNOP -> OP
OPisUNOP: OP -> tr
isBINOP: SYMB -> tr
```

which, in turn, remove a symbol from a word and consider it as an operator; consider an operator as a binary operator (if possible); the reverse; the same, for a unary operator; determine whether an operator is unary; and determine whether a symbol is binary. (These roughly correspond to "isunary" and "isbinary" in the informal presentation.)

All of these new constants (and more, which we need not mention here) are defined by axioms in terms of basic PPLAMBDA constants for injection and projection in sum domains.

Finally, we construct the theory of parsing, in which we define the parser and unparser, state the correctness property, and perform the proof. This theory is a descendent of the theory of symbols and trees, and hence indirectly of propositional calculus, orderings and lists. The hierarchy of theories can thus be drawn as follows:

$$\begin{tikzcd}
\text{Propositional Calculus} \arrow[d] & \text{Lists} \arrow[d] & \text{Trees} \arrow[dd] \\
\text{Orderings} \arrow[r] & \text{Symbols} \arrow[dr] & \\
 & & \text{Parsing}
\end{tikzcd}$$

The parser theory includes the type abbreviation

ParserState = SYMB LIST × OP' LIST × PTREE LIST

where "PTREE" abbreviates "(IDEN,UNOP,BINOP)TREE". The new constants of the theory include the following (in terms of a new type, "SIDE"):

```
WD: PTREE -> tr
L: SIDE
R: SIDE
N: SIDE
```

```
isleft: SIDE -> tr
isright: SIDE -> tr
isneither: SIDE -> tr
```

The first four constants are as in the informal presentation. The latter three are functions to determine whether a side-indicator is "L", "R" or "N", respectively. Simple axioms are given again for these constants.

The functions which comprise the parser are declared as new constants as well:

```
parse: ParserState -> ParserState
clear: ParserState -> ParserState
unparse: OP -> SIDE -> PTREE -> SYMB LIST
```

The clauses of the parser and unparser and unparser are easily stated now as new axioms, using the various new constants. For example, clauses 6 and 9 are:

```
6. !u:UNOP. !w:SYMB LIST. !os:OP' LIST. !rs:PTREE LIST.
   parse(PUTUW u w, os, rs) ==
   parse(w, PUTUO u os, rs)

9. !b2:BINOP. !w:SYMB LIST. !os:OP' LIST. !t2:PTREE. !t1:PTREE.
   !rs:PTREE LIST.
   BPrec b2 > Prec(GETOW w) == TT IMP
   parse(w, PUTBO b2 os, CONS t2(CONS t1 rs)) ==
   parse(w, os, CONS(mkBIN b2 t1 t2)rs)
```

The conditional, in the informal version of clause 9, becomes an implication in the above formal version. (The 'destructive' form, using "GETOW", turns out to be more convenient in the proof.) The rendition of the other clauses is similar.

The expression of the relations "rel1", "rel2" and "rel3" is fortunately easy, although PPLAMBDA does not admit relational constants. Because the relations are purely propositional, the *formula*

rel1(op,w,s) *iff* ... *or* ... *or* ... *or* ...

can be expressed as a disjunction of truth-valued PPLAMBDA *terms*. For example, the axiom defining "rel1" is:

```
|- !op:OP. !w: SYMB LIST. !s:SIDE. rell(op,w,s) ==
   (NULL w) OR
   (isBINOP(HD w) AND (Prec op) > (Prec(GETOW w))) OR
   (isBINOP(HD w) AND (Prec op) = (Prec(GETOW w)) AND
        ((isneither s) OR (isleft s) OR (left(destBINOP op)))) OR
   (isRB(HD w))
```

The other relations are treated similarly. We can then write "rell(op,w,s) == TT" where earlier we said "rell(op,w,s) holds", and then use the axiom to expand the expression only when necessary in the proof (*e.g.* in the case arguments based on the ways in which "rell(op,w,s)" can hold). We note that this treatment is not possible for relations in general.

To summarise, we have now constructed a hierarchy of theories in which new types, constants and axioms are added to PPLAMBDA to allow a natural expression of the parser and its properties. The theories of trees and lists are standard theories, and may be constructed automatically by an ML procedure by R. Milner. The proof of the correctness property is performed within the theory of the parser, from which the other theories are accessible. In the next section, we describe the generation of the formal proof.

5. THE PROOF IN LCF

5.1. Proof generation in LCF

In this section we describe the machine proof of the correctness property of our parser, and the proofs of the main lemmas. The relevant LCF concepts are explained concurrently.

The two parts of the LCF system, ML and PPLAMBDA, are connected by the *type* and *abstract type facilities* of ML. The logic PPLAMBDA is represented in the metalanguage by the ML types *term* and *form* for logical terms and formulae. A theorem (*thm*) is an abstract type in ML whose only accessible constructors are the rules of inference of PPLAMBDA. (This ensures that false theorems cannot be constructed.) Rules of inference are represented as ML procedures which return theorems as results. A theorem dependent on a set of assumptions, A, and with conclusion "w" is written "A |- w". Particular assumptions (or hypotheses) are occasionally represented as "." so that a theorem asserting "w" with two assumptions may be written ". . |- w". This notation is used where the assumptions are clear from the context.

LCF can accomodate both forward proof (successive application of rules of inference to build chains of theorems) and goal-oriented proof. The latter method consists in setting out a *goal* to be achieved and applying to it *tactics* to generate both subgoals and a means of mapping theorems achieving these subgoals to a theorem achieving the original goal. (This amounts to generating the intermediate chain of theorems.) Often, a mixture of forward and goal-oriented proof is successful; this section describes one way of mixing the two.

A goal is a composite object in ML. It includes, of course, the formula to be proved, such as the current one

```
!t. !op. !s. !w. !os. !rs. WD t == TT & DEF op == TT &
                           DEF s == TT & rel1(op,w,s) == TT &
                           rel2(op,os,s) == TT &
                           rel3(op,s) == TT IMP
                      parse(APP(unparse op s t)w, os, rs) ==
                      parse(w, os, CONS t rs)
```

or the formula for the lemma mentioned in section 3.2:

```
!op. !w. !s. !b. rel1(op,w,s) == TT & BPrec b = Prec op == TT &
                 NOT(left(destBINOP op)) == TT &
                 isright s == TT IMP
                 rel1(mkBINOP b, w, R) == TT
```

A goal also includes a list of formulae (the *assumption list*), representing the current assumptions at a point in the proof. For example, midway in proving the Unary Case of the main theorem, we happen to have a subgoal with the formula

```
parse(APP(unparse op s (mkUN u t) w), os, rs) ==
parse(w, os, CONS(mkUN u t) rs)
```

and with seven assumptions in its assumption list. These include the induction hypothesis and six more assumptions introduced in the course of the proof so far. (Lists in ML are written in the form "[e_1;...;e_n]".)

```
[!op s w os rs. WD t == TT & DEF op == TT & DEF s == TT &
                rel1(op,w,s) == TT & rel2(op,os,s) == TT &
                rel3(op,s) == TT IMP
                parse(APP(unparse op s t)w,os,rs) ==
                parse(w,os,CONS t rs);
 WD(mkUN u t) == TT;
```

```
DEF op == TT;
DEF s == TT;
rel1(op,w,s) == TT;
rel2(op,os,s) == TT;
rel3(op,s) == TT]
```

The subsequent case analysis (based on relative precedence) introduces one more assumption to this list, in each case.

The third component of a goal reflects the observation we have made about the informal proof: that most of the proof steps are left-to-right rewritings or unfoldings according to already proven theorems and axioms. For example, the parser clause 6, which we formalised in section 4, is used as a rewrite rule twice in the Unary Case of the proof:

```
!u w os rs. parse(PUTUW u w, os, rs) ==
            parse(w, PUTUO u os, rs)
```

The first use occurs in the case where op > u (*i.e.* Prec op > UPrec u == TT); it applies to a subgoal whose formula is:

```
parse(PUTUW u (APP(unparse u N t)(CONS RB w)),
      PUTBRO LB os, rs) ==
parse(w, os, CONS(mkUN u t)rs)
```

To apply clause 6, an instance of the left-hand-side of the clause is sought in the formula above. Here, the instance is the whole left-hand-side of the formula. The bound variable "u" is instantiated to the particular "u"; "w" to "APP(unparse u N t)(CONS RB w)"; "os" to "PUTBRO LB os"; and "rs" to the particular "rs". This completely instantiates clause 6, and allows us to rewrite the left-hand-side of our formula to be

```
parse(APP(unparse u N t)(CONS RB w), PUTUO u (PUTBRO LB os),
      rs) ==
parse(w, os, CONS(mkUN u t)rs)
```

and the proof is advanced a bit.

Facts such as clause 6 which are used in this way as *simplification rules* are included in the third part of a goal: the *simpset*. A simpset is an abstract type in ML containing representations of theorems intended to be used as rewrites (as illustrated).

Simplification rules (or simprules) may also be *conditional*. For example, clause 9, which we also formalised in section 4, is:

```
!b2 w os t1 t2 rs. BPrec b > Prec(GETOW w) == TT IMP
                   parse(w, PUTBO b2 os, CONS t2(CONS t1 rs)) ==
                   parse(w, os, CONS(mkBIN b2 t1 t2) rs)
```

The consequent of this fact can be used just as clause 6 to simplify a goal or subgoal *if* the antecedent, "BPrec > Prec(GETOW w) == TT", appropriately instantiated, can first be seen to be true. (It may be true either because it has already been assumed, or because it can itself be simplified to something obviously true. It must be the case that *all* of the instantiable variables of the antecedent, in this case "b" and "w", must occur in the left-hand-side of the consequent -- they do -- or else the matching will not meaningfully instantiate the antecedent.) Simprules of this form are called *conditional simprules*. In the case of clause 9, the antecedent will have been assumed by the time we wish to use the clause as a simprule. The induction hypotheses in the Unary and Binary Cases are other examples of useful conditional simprules.

To summarise, the ML type goal is defined as:

goal = form × simpset × form list

A goal with formula w, simpset ss and assumption list A is written as "(w,ss,A)", or occasionally as:

$$\begin{array}{|c|}\hline \mathrm{w} \\ \hline \mathrm{ss} \\ \hline \mathrm{A} \\ \hline \end{array}$$

Goal-oriented proofs are advanced by the application of *tactics* to goals. Tactics are ML procedures which given goals produce (i) lists of subgoals and (ii) justifications of the proof step made in moving from goal to subgoal:

tactic = goal -> (goal list × proof)

A proof is also an ML function, mapping the (respective) achievements of the subgoals (a list of theorems) to the achievement of the goal (a theorem):

```
proof: thm list -> thm
```

For example, one of the simple strategies used in the informal proof can be pictured as

```
(!x.w, ss, A)
==================
(w [x'/x], ss, A)          (x' not free in A)
```

meaning: to prove a fact about all x, try proving the fact for an arbitrary x' (not occurring free in A). The proof function returned when this tactic is applied to a goal appeals to the PPLAMBDA rule of inference GEN: term -> thm -> thm

```
A |- w
----------
A |- !x.w                  (x not free in A)
```

where x is a term (not occurring free in A). This means: from the upper theorem, deduce the lower one. Because it 'inverts' the inference rule GEN, we call the tactic GENTAC. GENTAC is a built-in tactic in LCF as it is so commonly used. Another simple tactic reflecting steps in the informal proof is

```
DISCHTAC: tactic
(w1 IMP w2, ss, A)
==================
 | w2               |
 | (. |- w1) + ss   |
 | w1.A             |
```

which reflects the strategy: to prove an implication, try proving the consequent having assumed the antecedent (and using the assumption as a simplification rule, too). A PPLAMBDA rule called DISCH is used to justify this step (hence the name of the tactic). The ". |- w1" indicates that the theorem depends on the one assumption. This tactic is not built in to LCF but is easily implemented as an ML procedure.

This last tactic generalises to

```
IMPCONJTAC: tactic
(w1 & w2 & ... & wn IMP w, ss, A)
=================================
| w                                  |
| ss + (. |- w1  + ... + . |- wn)    |
| [w1;wn;...;wn]@ A                  |
```

when we expect an antecedent which is a conjunction.

Simprules are engaged by a standard tactic called SIMPTAC which when applied to a goal (w,ss,A) uses all of the simprules in ss to rewrite the formula w as many times as possible until either no more simprules can be applied, *or* until a trivially easy subgoal is produced. Each rewriting step is justified in the proof function which SIMPTAC returns when applied. The consequents of conditional simprules are used when the appropriately instantiated antecedents can be reduced first to tautologies. (SIMPTAC can recognise certain trivially easy formulae and tautologies.) When a trivially easy subgoal is reached, and empty list of subgoals is returned.

5.2. <u>More complex tactics</u>

By combining small tactics such as GENTAC and DISCHTAC and by designing and implementing more sophisticated strategies, one is able to generate whole proofs, or large parts of proofs, by the application of tactics. Much of the interest of doing proofs in LCF lies in the search for useful, general strategies. Behind the scenes, one is assured that every inference step of the proof is being evaluated when the proof function is applied -- but to the extent that one's tactics are successful in reducing goals to trivial subgoals, one is not forced to be aware of the details of the proof. We go one to describe some of the more complex tactics needed in this proof (and useful in other proof attempts).

The proof calls for a structural induction tactic special to the recursively defined type TREE. In PPLAMBDA the only induction rule is Scott's rule of computation induction. However, for certain recursively defined domains, the appropriate rule of structural induction can be derived from computation induction. The derivation of such rules, like the construction of theories of these types, is a standard process, and is part of the ML procedure by R. Milner mentioned in section 4.

The tactic corresponding to the rule of tree induction is:

```
TINDUCTAC: tactic
(!t:(*,**,***)TREE . w[t], ss, A)
=====================================================
(w[UU], ss, A)
(!I. w[mkTIP I], ss, A)
(! u t. w[t] IMP w[mkUN u t], ss, A)
(! b t1 t2. w[t1] & w[t2] IMP w[mkBIN b t1 t2], ss, A)
```

When the proof function of this tactic is applied, the entire derivation of the rule of structural induction for TREEs from the rule of computation induction is performed. This is unavoidable as the structural induction rule cannot be expressed as a theorem in PPLAMBDA.

Very many of the facts and axioms needed in the proof are used without difficulty as simplification rules. These include many facts about lists, symbols, the propositional calculus, the three relations, the parser and unparser axioms, as well as several simple theorems about symbols. The next tactic is suggested by a number of lemmas and small facts needed in the proof which (for a variety of reasons) *cannot* be used directly as simprules (or conditional simprules). For example, to enable the induction hypotheses to be used as conditional simprules in the proof, we must, as we said, satisfy certain instances of their antecedents. As we saw in the informal proof, this requires an appeal to the axioms of well-definedeness of trees, for example:

```
|- ! u t. WD(mkUN u t) == TT IMP WD t == TT
```

This rule (and the two similar ones for binary trees) do not themselves make sense as conditional simprules -- if they did, a straightforward chain of conditional simplification could enable the induction hypotheses to be used. As remarked in section 5.1, these axioms do not make sense as simprules because their antecedents are not fully instantiated by an instantion of "WD t" (during a match to part of a subgoal). Since the use we wish to make of axioms of this form goes beyond simplification in the LCF sense, we must design a new tactic for using the axioms.

Other facts and axioms which present the same problem include the transitivity axiom for ranks

```
|- ! r1 r2 r3. r1 = r2 == TT & r2 > r3 == TT IMP r1 > r3 == TT
```

and the axioms about the constant "AND"

```
|- ! p q. p AND q == TT IMP p == TT
|- ! p q. p AND q == TT IMP q == TT
```

as well as several other axioms and simple lemmas such as the following (the second a minor one not mentioned earlier):

```
|- !op1 op2. Prec op1 = Prec op2 == TT IMP op1 == op2
|- ! b op. Prec(mkBINOP b) = Prec op == TT IMP
        DEF(destBINOP op) == DEF op
```

More importantly, the lemmas we discussed in section 3.2, another example of which is

```
|- !op w s b. rel1(op,w,s) == TT & BPrec b = Prec op == TT &
              NOT(left(destBINOP op)) == TT &
              isright s == TT IMP
              rel1(mkBINOP b, w, R) == TT
```

(and those lemmas which factor out the case arguments on rel1 and rel2) also share the property of being unsuitable as simplification rules.

A few more axioms are unsuitable as simprules for a different reason -- because, individually or collectively, they cause the LCF simplifier to loop. For example, a simple axiom of the theory of parsing (not mentioned earlier) is:

```
|- !s:SIDE. isleft s == TT IMP s == L
```

This is applied as a simprule by tring to replace an occurrence of a term matching "s" by "L" *if* "isleft s" can first be rewritten to "TT". To do *that*, the simplifier sets out to simplify "s", and so on *ad inf*.

We treat *all* of these facts and axioms in the same way. We first write a tactic, parameterised on a list of facts, to place the conclusions of the facts in the assumption list of a goal:

```
USELEMMASTAC[|-w1; |-w2; ... ; |-wn]
(w,ss,A)
_________________________________
(w, ss, [w1;w2;...;wn]@ A
```

The proof function of USELEMMASTAC simply discharges the extra assumptions and appeals to the PPLAMBDA rule of *Modus Ponens* (MP) to achieve the goal (w,ss,A).

Next we implement a tactic which searches for and 'resolves' certain pair of assumption in the assumption list, namely the new non-simplification-like formulae, and any instances of their antecedents which may also appear in the assumption list. For example,

```
! u t. WD(mkUN u t) == TT IMP WD t == TT
WD(mkUN u t) == TT
```

are a pair of formulae where the first is the problematic sort and the second an assumption arising in the Unary Case of the proof. The latter formula is matched to the antecedent of the former, so that the first is instantiated to

```
WD(mkUN u t) == TT IMP WD t == TT
```

for the particular values of "u" and "t" occurring in the latter. If we *assume* the second formula, and the correct instance of the first,

```
. |- WD(mkUN u t) == TT
. |- WD(mkUN u t) == TT IMP WD t == TT
```

and perform MP, we prove (in a forward way):

```
.. |- WD t == TT
```

This new theorem *can* be used as a simplification rule (and so helps to enable the induction hypothesis to be used as a conditional simprule). Here, a small chain of forward proof produces a useful simprule for the subsequent part of the goal-oriented proof. During the evaluation of the proof function, the two extra assumptions are discharged; then MP is used to dismiss the axiom about well-definedness.

To reflect this strategy we implement a tactic called RESTAC because it is a primitive version, in LCF terms, of classical resolution (Robinson 1979). RESTAC searches the list of assumptions of a goal for any pair of formulae which can be resolved in the manner described.

In general, RESTAC tries to resolve any pair of assumptions of the form (w, !x1...xn. w1 IMP w2) where w may be quantified but is not an implication. The tactic tries to match w to w1 to determine instantiations y_i for some (of all) of the x_i. It then assumes both w and the correct instance of the formula in question, evaluates MP to prove .. |- w2[y_i/x_i], and generalises again to those x_i which were not instantiated. (In the case illustrated above, there were no uninstantiated variables.) This new theorem's conclusion is placed in the assumption list of the subgoal returned (where it can participate in further resolutions) and the new theorem itself is placed in the simpset (if possible) to play a role in later simplifications. RESTAC fails if it cannot prove any new theorems from the current list of assumptions. Some subtlty is required in not adding to the simpset theorems which would *obviously* loop, but there is nothing heuristic about RESTAC's forward search.

RESTAC is useful in every one of the cases in this proof where certain non-simplification-like facts or axioms have to be used. This includes the use of the main lemmas, such as

```
|- !op w s b. rel1(op,w,s) == TT & BPrec b = Prec op == TT &
              NOT(left(destBINOP op)) == TT &
              isright s == TT IMP
              rel1(mkBINOP b, w, R) == TT
```

which can be resolved with the assumption "rel1(op,w,s)" to give

```
.. |- !b. BPrec b = Prec op == TT &
          NOT(left(destBINOP op)) == TT & isright s == TT IMP
          rel1(mkBINOP b, w, R) == TT
```

which *is* a useful conditional simprule. (Note that op and s are not instantiable variables because they occur in the assumption "rel1(op,w,s)".) This new theorem, as a simprule, also helps enable the induction hypothesis to be used. (The lemmas which factor out the case arguments on rel1 and rel2 also become useful conditional simprules via RESTAC.)

Another pair of useful, general tactics for the proof relates to our theory of propositional calculus. The first tactic we implement is a case analysis tactic (one of several we need). It is based on the PPLAMBDA rule of case analysis, which considers whether some truth-valued term is "TT", "FF" or "UU".

```
ORCASESTAC: thm -> tactic
A' |- p1 OR p2 OR ... OR pn == TT
(w,ss,A)
=====================================
| w                         |
| ss + (. |- pi == TT)      |
| (pi == TT).A              |
```

That is, suppose we know that *one* of the p_i is true; then the tactic produces n subgoals from the goal, assuming that each p_i in turn, is true. This tactic has the pleasant effect of concealing all matters to do with undefined cases, although a full proof in PPLAMBDA is carried out internally by the proof function, as always. The user also enjoys the effect of working in the propositional calculus, as is natural to the problem.

The second propositional calculus tactic we need, called ORRESTAC, is another simple resolution tactic. Like RESTAC, it examines the assumptions of a goal in an attempt to make some deductions which might be useful. This tactic looks for propositional formulae of the form "p1 OR p2 OR ... OR p_n == tv", where "tv" is a truth-valued constant, and $n \geq 2$. The tactic reduces (simplifies) such assumptions according to all of the axioms of the propositional calculus, the parser theory axioms such as "|- isleft L == TT" which are propositional, and any other current assumptions which are equivalences with a truth-valued right-hand-side. If the result of the simplification is either a contradiction, such as " |- FF == TT", or an equivalence with only one disjunct on the left-hand-side, then the tactic is considered to have been successful. Otherwise, it has not really advanced the proof, and is said to have *failed*. (The failure of tactics is implemented using the exception-handling facilities of ML.) If a contradiction is obtained, ORRESTAC returns an empty list of subgoals and a proof function which will achieve the original goal. If it can reduce the formula in question to an equivalence with one disjunct, that new result is assumed and used as a simprule. ORRESTAC uses, internally, a proof-by-contradiction tactic which is not difficult to implement in ML:

```
CONTRTAC: tactic
w, ss, [...; TT == FF; ...]        (or UU == FF, etc.)
===========================
[]
```

If an of the assumptions of the goal is a contradiction (in the three-valued logic), then one can prove the goal immediately, whatever it is. Thus, ORRESTAC sometimes completes the process of generating subgoals (like SIMPTAC), and sometimes just adds a new result to the simpset of the next subgoal.

The proof also requires a few more case analysis rules, like ORCASESTAC, which are all derived from the basic PPLAMBDA case analysis rule. We need tactics to do case analysis based on sidedness, associativity and relative precedence of two operators. We implement in ML:

```
SIDECASESTAC: tactic
w[s:SIDE], ss, A
=============================
| w                          |
| ss + (.|-isneither s == TT)|
| (isneither s == TT).A      |

| w                          |
| ss + (.|-isleft s == TT)   |
| (isleft s == TT).A         |

| w                          |
| ss + (.|-isright s == TT)  |
| (isright s == TT).A        |
```

This tactic is justified by the parser theory axiom that *one* of the assumptions must hold (if "s" can be shown to be defined).

```
ASSOCASESTAC: tactic
w[b:BINOP], ss, A
=====================================================================
| w                          |  | w                               |
| ss + (.|-left b == TT)     |  | ss + (.|-NOT(left b) == TT)     |
| (left b == TT).A           |  | (NOT(left b) == TT).A           |
```

This is based on the propositional calculus tactic that a proposition (if it is defined) or its negation must hold.

ORDERCASESTAC: tactic

w[op_1:OP, op_2:OP], ss, A

---

w
ss + (.|-Prec op1 > Prec op2 == TT)
(Prec op1 > Prec op2 == TT).A

w
ss + (.|-Prec op1 = Prec op2 == TT)
(Prec op1 = Prec op2 == TT).A

w
ss + (.|-Prec op2 > Prec op1 == TT)
(Prec op2 > Prec op1 == TT).A

The formula "w" may also contain "u:UNOP" or "b:BINOP". ORDERCASESTAC is is based on the order axiom that one of the three cases must hold, where "op_1" and "op_2" are defined.

In all of these case analysis tactics, the tactic fails when the appropriate objects cannot be shown to be defined -- but again, this reasoning is concealed from the user.

5.3. The proof in LCF

The tactics for the proof are all ready; it remains only to combine them. To do this we write (or use standard) *tacticals*. For example, for tactics T, T_1 and T_2, the standard sequencing tactical THEN combines T_1 and T_2 to produce a tactic (T_1 THEN T_2). This new tactic applies T_1 to a goal to obtain subgoals, and T_2 to each of the subgoals. The iterating tactic REPEAT is such that (REPEAT T) applies T to a goal to obtain subgoals, applies T to these, and so on, until T fails to apply (if ever). The tactic (T_1 ORELSE T_2) applies T_1 and only if that fails applies T_2. For brevity, we write

T_1

T_2

to suggest sequencing; T* for iteration and (T_1 ? T_2) for (T_1 ORELSE T_2). We let T+ mean (T THEN SIMPTAC), as this combination occurs frequently.

The proofs of the main theorem and the main lemmas are accomplished by (i) setting up goals whose simpsets contain the appropriate axioms and proved facts, (ii) building up compound strategies using tactics and tacticals, and (iii) applying the tactics to the goals, to generate empty lists of subgoals (eventually). The proof functions returned can then be applied to generate the desired theorems. In that process, all of the intermediate inference steps are *evaluated.*

We first discuss the tactic which solves the various lemmas (a typical one of which was described in section 3.2). We let "L" stand for the list of facts which are non-simplification-like (*e.g.* the axioms about the transitivity of "=" and ">"). The tactic which generates the proofs of all of the lemmas is:

```
USELEMMASTAC L
GENTAC*
IMPCONJTAC+
                . |- rel1(op,w,s) == TT
ORCASESTAC {
                . |- rel2(op,os,s) == TT
(RESTAC THEN (ORRESTAC ? SIMPTAC))*
```

We explain each line in turn:

- The first uses the facts in L by placing then in the assumption list, to be used later.
- The second strips off all of the quantifiers, proving the goal for arbitrary values of the variables.
- The third assumes the several antecedents of the implication being proved and returns the consequent as result (and then simplifies the subgoal).
- The fourth performs a cases analysis on the ways in which rel1 (or rel2, depending on which the lemmas is about) can hold.
- Finally, the tactic repeatedly (on each of the four subgoals resulting from the case analysis) resolves, so that elements of L meet with the various subgoals to produce new simprules; then resolves the propositional assumptions, such as the assumption that rel1 (or rel2) holds. This completes the proof in some cases, or may just add some new simprules. If ORRESTAC fails, SIMPTAC is engaged to use all of the new simprules. If there are still any unsolved subgoals, this whole line (beginning with a round of resolution) is again applied (to them).

This compound tactic is written in a general form so that the same tactic can be used for all of the lemma proofs. As some of these proofs are simpler than others, we sometimes arrive at an empty list of subgoals before the whole tactic is applied.

The simplification sets of all of the lemma goals contain some axioms (and simple facts) about symbols, some of the axioms from the parser theory, some axioms of propositional calculus, some axioms about ranks and the axioms defining the three relations. (The theorems and axioms in "L" have mostly been discussed in section 5.2.)

The LCF proof of the main result is not much more complicated than the lemma proofs. Again, some facts and axioms can be used as simprules without ado. These include the parser and unparser clauses, various facts about "CONS" and "APP" (from LIST theory), some axioms and simple theorems about symbols and the parser, some axioms of propositional calculus, and a few more. As before, some axioms and theorems are not suitable as simprules -- for example, none of the lemmas discussed are suitable. We call the class of unsuitable facts "L'". The generation of the machine proof begins with the application of the following tacic to the goal:

```
USELEMMASTAC L'
TINDUCTAC+
(GENTAC* THEN IMPCONJTAC)*
```

This tactic, in turn

- places the appropriate facts in the assumption list
- generates four subgoals, corresponding to the Undefined, the Tip, the Unary and the Binary Cases, and simplifies to solve the first two cases for us
- strips off quantifiers, to prove the goal for arbitrary values of the bound variables, and proves the consequent by first assuming the several antecedents (the whole line repeated if necessary)

At this point, two subgoals remain, the Unary and Binary Cases. A call of RESTAC is made in both cases to use the elements of L' by resolving them with the six current assumptions (the antecedents). This single round of resolution results in versions of the main lemmas which *are* useful as simprules (and the same for a few other elements of L' too). These new rules are placed in the simpset. Then, in both cases, we apply ORDER-CASESTAC to reflect the division into cases based on relative precedence (of "op" and "u", or "op" and "b" -- the tactic figures out which). We are left with three subgoals in each of the two cases.

In the Unary Case, all that is required to solve the three subgoals is a call to SIMPTAC to use the newly added simprules. The same is true of the two Binary Case subgoals in which there is unequal precednce (see the informal proof in section 3). For the remaining one subgoal, we have to work harder.

In that case, we next apply SIDECASESTAC to consider the three possible values for the side-indicator "s". For each of the three resulting subgoals we need another call of RESTAC to resolve the new case assumptions with the following element of L' (section 5.2), among others:

```
|- ! b op. Prec(mkBINOP b) = Prec op == TT IMP
           DEF(destBINOP op) == DEF op
```

The theorem deduced from this call of RESTAC is used later to enable the induction hypothesis to be applied.

The 'neither' subgoal is argued by contradiction, and a call of ORRESTAC solves it. The other two remaining subgoals are *again* subjected to case analysis -- a call of ASSOCASESTAC. SIMPTAC solves the final four subgoals, and the proof is completed.

The whole tactic which solves the goal is:

```
USELEMMASTAC L'
TINDUCTAC+
(GENTAC* THEN IMPCONJTAC)*
RESTAC
ORDERCASESTAC+
SIDECASESTAC
RESTAC
(ORRESTAC ? (ASSOCASESTAC+))*
```

The effect of the tactic on the goal is most clearly seen in the tree structure corresponding to the successive subgoals produced by the individual tactics:

```
                    USELEMMASTAC L'
                    TINDUCTAC+
                    (GENTAC* THEN IMPCONJTAC)*
                         |
      ----------------------------------------
      ▽                                      ▽
Unary Case                             Binary Case
RESTAC                                 RESTAC
ORDERCASESTAC                          ORDERCASESTAC
   ------------------------------         ------------------------------
   ▽          ▽          ▽                ▽          ▽           ▽
SIMPTAC    SIMPTAC    SIMPTAC          SIMPTAC    b=op Case   SIMPTAC
                                                      ▽
```

```
                              b=op Case
                              SIMPTAC
                              SIDECASESTAC
                                   |
              ------------------------------------------
              |                    |                    |
         s=N ∇Case            s=L ∇Case            s=R ∇Case
         RESTAC               SIMPTAC              RESTAC
         ORRESTAC             ASSOCASESTAC         ASSOCASESTAC
         SIMPTAC                   |                    |
                             -------------        -------------
                             |           |        |           |
                          ∇SIMPTAC   ∇SIMPTAC  ∇SIMPTAC   ∇SIMPTAC
```

This clearly reflects the steps of the informal proof, including the reasoning by contradiction (ORRESTAC) and the reasoning required to use the induction hypotheses (where RESTAC helps), as well as the many routine rewritings (accomplished by SIMPTAC) and the case analyses (ORCASESTAC, ORDERCASESTAC, SIDECASESTAC and ASSOCASESTAC). The component tactics, especially RESTAC and TINDUCTAC, are quite general, and useful in other proofs. Although the number of primitive inferences evaluated in the course of applying the proof function is very large, the tactic itself naturally reflects the proof structure. Aside from the case analyses, which are special to this problem, the tactic is not very much different from the tactic solving the simpler parser proof described in (Cohn & Milner 1982).

To prove the theorem really wanted, namely

```
!t. WD t == TT IMP
    parse(APP(unparse UO  N t) NIL, NIL, NIL) ==
    parse(NIL, NIL, CONS t NIL)
```

(as in section 3.1), we introduce the constant "UO ", include parser clause 1 in the simpset of the goal, and apply SIMPTAC.

6. CONCLUSIONS

In this paper we have described a precedence parsing algorithm, stated and informally proved a correctness property of the algorithm relative to an unparsing algorithm (the one inserting the least number of brackets), described the formalisation of the problem in the logic PPLAMBDA, and discussed the generation of the machine proof in LCF by the application of ML tactics.

To summarise, we show the tactics which solve the main lemmas and the theorem, below.

Main Lemmas
USELEMMASTAC L
GENTAC*
IMPCONJTAC+
ORCASESTAC{|-rel1(op,w,s) == TT
|-rel2(op,os,s) == TT
(RESTAC THEN (ORRESTAC ? SIMPTAC))*

Theorem
USELEMMASTAC L'
TINDUCTAC+
(GENTAC* THEN IMPCONJTAC)*
RESTAC
ORDERCASESTAC+
SIDECASESTAC
RESTAC
(ORRESTAC ? (ASSOCASESTAC+))*

The combination of USELEMMASTAC, IMPCONJTAC and RESTAC forms a pattern which can be thought of as a single conceptual step of using facts which are not handled by the standard apparatus of simplification. The proofs all depend on the 'resolution' tactic RESTAC to apply these facts. While most of the proof steps *are* accomplished by SIMPTAC, RESTAC supplements simplification by doing a small amount of forward search. What seems especially nice is the way RESTAC fits into the otherwise goal-oriented framework; it is just another tactic, with the end effect of adding to the set of simplification rules of a goal (and justifying that addition in its proof function). RESTAC meshes nicely with simplification for that reason, especially conditional simplification. In the problem described here, one round of resolution was in all cases enough to produce useful new simplification rules (often conditional ones). The burden of proof was then transferred back to the more efficient, direct and goal-oriented mechanism of simplification. The subgoaling style of proof was never interrupted. We feel that RESTAC is a primitive form of a potentially very useful and widely applicable tactic.

In this experiment, the logic PPLAMBDA was adequate for a very natural expression of the algorithm and its correctness property. Using ML functions for the purpose, new types, constants and axioms were introduced in an organised way, to form a structure of logical theories. A general ML procedure, due to R. Milner, to construct theories and induction rules and tactics for certain recursively defined data types was used to build general theories of lists and trees for the problem statement. This is an indication that general proof tools exist and can be implemented in ML.

The problem of being unable to express relational constants in PPLAMBDA was avoided here by the fortunate fact that the relations in question were purely propositional and could be expressed in terms of a simple theory of propositional calculus. (Had this not been the case, the rather cumbersome formulae spelling out the relations would have had to appear everywhere instead. We did manage to perform the proof this way as well, though.) This is a weak point of the current version of PPLAMBDA.

The undefined cases which often clutter up proofs in LCF are handled neatly in this proof. If we were to perform the individual tactics line-by-line as written so that all the intermediate subgoals could be seen, there would be no evidence of undefined cases, although they are all, of course, proved. Most are handled by simplification (*e.g.* the Undefined Case of the tree induction). Beyond that, the various derived case rules manage undefined cases internally (or else would have failed). The non-strict propositional calculus also helps; had "AND" and "OR" been defined to be strict (*e.g.* in terms of the basic PPLAMBDA conditional function) many more undefined cases would have arisen. ("AND" and "OR" are *axiomatised* rather than *defined*, though, at the cost of having to show the axioms consistent.)

In future work we would like to experiment with the resolution tactic in other settings, and to make it more efficient and sophisticated. As it is, no heuristics are used at all, and all possibly useful deductions of a certain sort are made. It would be interesting to try to import some of the ideas from classical resolution theory into this context.

REFERENCES

1. Avra Cohn and Robin Milner, "On using Edinburgh LCF to prove the correctness of a parsing algorithm", Edinburgh University Computer Science Dept. Technical Report, to appear, 1982
2. Paul Y. Gloess, "A proof of the correctness of a simple parser of expressions by the Boyer-Moore System", S.R.I. Technical Report N000 14-75-C-0816-SRI-7, August 1978
3. Michael Gordon, Robin Milner and Christopher Wadsworth, "Edinburgh LCF", Springer-Verlag, New York, 1979
4. J. A. Robinson, "Logic: Form and Function", Edinburgh University Press, Edinburgh, 1979

ACKNOWLEDGEMENTS

We thank Robin Milner for the idea of proving a precedence parser, and for doing the first parser proof; and Robin Milner and Mike Gordon for helpful criticisms of this work.