Completeness of Propositional Logic as a Program

Ryan Stansifer
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901
ryan@cs.fit.edu

March 2001

Abstract

The proof of completeness for propositional logic is a constructive one, so a computer program is suggested by the proof. We prove the completeness theorem for Łukasiewicz' axioms directly, and translate the proof into the functional languages SML and Haskell. In this paper we consider this proof as a program. The program produces enormous proof trees, but it is, we contend, as good a proof of completeness as the standard mathematical proofs. The real value of the exercise is the further evidence it provides that typed, functional languages can clearly express the complex abstractions of mathematics.

1 Introduction

We have written a program for finding proofs of tautologies in propositional logic and implemented it in SML [12] and in Haskell [9, 8]. We merely implement the steps of the constructive proof of the completeness for propositional logic. SML and Haskell are good languages to capture the intended functions primarily because of their recursive data structures. The programs themselves are proofs—clear proofs to those that can read a functional programming language. Viewed as a proof the programs may be clearer than the mathematical proofs that often fumble with expressing algorithmic content.

This paper serves as a guide to these SML and Haskell programs. The complete code is too long to include here in its entirety—it can be found elsewhere [16].

The programs find proofs in a particular axiom system. These proofs are nearly impossible to discover by the human logician. But the proof trees constructed by the algorithm are extremely large. We examine how large they tend to be as well as try various optimizations.

2 Propositional Logic

Propositional logic concerns reasoning about propositions P, Q, etc. Sentences or propositional formulas are built out of connectives for conjunctions, disjunction, negation, implication, etc. We will be content with using just negation and implication, as the others can be viewed as abbreviations or macros. This simplicity may be welcome to human logicians, but it may be of no advantage to computer programs—a single complex connective, a clause, say, may be more suitable for computers that can handle more details simultaneously. Nonetheless, it is these propositional formulas we model.

```
datatype prop = prop of string |
  impl of prop * prop |
  neg  of prop;
```

A truth assignment to the constituent propositions makes any propositional formula true or false. The value can be computed using the usual truth-table semantics for the connectives.

2.1 Tautology

A propositional formula is a tautology if it is true for all possible assignments to the propositions. This suggests a simple theorem prover that checks all combinations. Of course, it has exponential complexity. Moreover, the tautology checker does not actually build a proof in any particular system of axioms and inference rules. Figure 1 is a list of some tautologies. These are some of the ones used in the analysis later.

$(P \Rightarrow P)$	`impl(P,P)`
$((\neg \neg P) \Rightarrow P)$	`impl(neg (neg P),P)`
$(P \Rightarrow (\neg \neg P))$	`impl(P,neg (neg P))`
$(P \Rightarrow (Q \Rightarrow P))$	`impl(P,impl(Q,P))`
$(P \Rightarrow (Q \Rightarrow Q))$	`impl(P,impl(Q,Q))`
$((\neg P \Rightarrow P) \Rightarrow P)$	`impl(impl(neg P, P), P)`
$(P \Rightarrow (\neg P \Rightarrow Q))$	`impl(P, impl(neg P, Q))`
$(\neg P \Rightarrow (P \Rightarrow Q))$	`impl(neg P, impl(P, Q))`
$((\neg (P \Rightarrow P)) \Rightarrow Q)$	`impl(neg (impl(P,P)), Q)`
$(P \Rightarrow (\neg (P \Rightarrow \neg P)))$	`impl(P, neg (impl(P,neg P)))`
$((P \Rightarrow \neg P) \Rightarrow \neg P)$	`impl(impl(P,neg P), neg P)`
$((\neg (P \Rightarrow Q)) \Rightarrow P)$	`impl(neg (impl(P,Q)), P)`
$((\neg (P \Rightarrow Q)) \Rightarrow (\neg \neg P))$	`impl(neg (impl(P,Q)),neg(neg P))`
$((\neg (P \Rightarrow Q)) \Rightarrow \neg Q)$	`impl(neg (impl(P,Q)), neg Q)`
$((P \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q))$	`impl(impl(P,neg P), impl(P,Q))`
$((P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P))$	`impl(impl(P,Q), impl(neg Q,neg P))`
$((P \Rightarrow \neg Q) \Rightarrow (Q \Rightarrow \neg P))$	`impl(impl(P,neg Q), impl(Q,neg P))`
$((\neg P \Rightarrow \neg Q) \Rightarrow (Q \Rightarrow P))$	`impl(impl(neg P,neg Q), impl(Q,P))`
$((\neg P \Rightarrow \neg Q) \Rightarrow ((\neg P \Rightarrow Q) \Rightarrow P))$	`impl(impl(neg P, neg Q),impl(impl(neg P,Q),P))`
$((\neg (P \Rightarrow Q)) \Rightarrow (Q \Rightarrow R))$	`impl(neg (impl(P,Q)),impl(Q,R))`
$((\neg (P \Rightarrow Q)) \Rightarrow (\neg P \Rightarrow R))$	`impl(neg (impl(P,Q)),impl(neg P,R))`
$((P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R)))$	`impl(impl(P,Q), impl(impl(Q,R), impl(P,R)))`

Figure 1: List of tautologies

It is worth considering the tautology checker before continuing.

```
local
  fun check’ sg phi =
    value sg phi handle not_found p =>
      check’ (update sg p true) phi andalso
        check’ (update sg p false) phi;
in
  fun checker phi = check’ undef phi;
end;
```

The function `value` computes the value, true or false, for a propositional formula ϕ given a particular assignment of the propositions. The function `update` extends the assignment of the propositions to another proposition. The essence of the tautology checker is to check if both (true and false) extensions make the formula ϕ true. We will see that the completeness theorem takes the same form.

It will be observed that the tautology checker does not form all 2^n assignments first—it does not bother to determine the n propositions that occur in the formula ϕ. Rather it optimistically evaluates the truth value of ϕ and if the value cannot be determined because of the absence of a proposition in the assignment, it uses exception handling to retreat and try again after extending the assignment. This takes advantage of the fact that the truth value of implication is known, if the antecedent is false. The result is some reduction of the checking. We will see that the completeness function operates similarly.

2.2 Axiom Schemata

In this paper we are concerned with constructing proofs that formulas are tautologies. To do so, we build proof trees out of the venerable rule of inference *modus ponens* plus some axioms. Actually, not axioms but *families* of axioms of axiom schemata.

There are many systems of axioms to select from. Frege [4] originally proposed six axiom schemata.

Frege's Axiom System

Fr 1	$A \Rightarrow (B \Rightarrow A)$
Fr 2	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
Fr 8	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow (A \Rightarrow C))$
Fr 28	$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow A)$
Fr 31	$\neg \neg A \Rightarrow A$
Fr 41	$A \Rightarrow \neg \neg A$

No question of completeleness, consistency (they are), or independence (they aren't) was raised. Hilbert and Ackermann [7] showed that only three schemata were necessary.

Hilbert and Ackermann's Axiom System

HA 1	$A \Rightarrow (B \Rightarrow A)$
HA 2	$(\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A)$
HA 3	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

Though not the smallest known, Łukasiewicz [11] proposed a well-known collection that saves a few symbols.

Łukasiewicz' Axiom System

Łu 1	$(\neg A \Rightarrow A) \Rightarrow A$
Łu 2	$A \Rightarrow (\neg A \Rightarrow B)$
Łu 3	$(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))$

It is Łukasiewicz' axiom system that we choose to use here.

In SML a proof tree is a data structure with the axioms for leaves and the rule of inference *modus ponens* as the sole interior node. For the moment, we can use this data structure:

```
datatype proof =
  ax1 of prop      |     (*  Lu1  *)
  ax2 of prop*prop |     (*  Lu2  *)
  ax3 of prop*prop*prop |(*  Lu3  *)
  mp  of proof * proof;
```

Later, we will have cause to add a constructor for assumptions.

Proofs in Łukasiewicz' axiom system are quite tedious. We give an example proof in the next section after introducing some helpful lemmas.

3 Proof of Completeness

All instances of the axiom schemata are tautologies, as can easily be verified using truth tables. The important question is: can proofs for *all* tautologies be constructed starting from just these few axioms. They can, and this result is known as the completeness theorem for propositional logic. In his doctoral dissertation of 1920 Post [13] was the first to give a proof. He used the propositional subset of *Principia Mathematica*.

In the meantime, many proofs of completeness for Łukasiewicz' axioms have been given. Often these proofs are indirect, as in [14], relying on other formal systems. One [1] is very direct, but relies on a different notion of derivation. One proof is given in [3]. It is especially interesting, since it reveals how one might originally come up with such a proof. We give the most economical proof of completeness, giving rise, we hope, to the best program.

3.1 Two Initial Lemmas

The combination of modus ponens and Łukasiewicz' axiom 3 appear many times in the subsequent proofs. So we begin with two lemmas that incorporate this pattern called the Backward Propagation Lemma and the Transitivity Lemma by Dúinlang [3].

Lemma 1 (Backward Propagation) *For any propositional formula C, if* $\vdash A \Rightarrow B$*, then* $\vdash (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$,

Proof. Given a proof of $A \Rightarrow B$, an application of modus ponens to axiom 3 gives the desired result.

$$\dfrac{\dfrac{axiom 3}{(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))} \quad \dfrac{given}{A \Rightarrow B}}{(B \Rightarrow C) \Rightarrow (A \Rightarrow C)}$$

∎

Lemma 2 (Transitivity) *If* $\vdash A \Rightarrow B$ *and* $\vdash B \Rightarrow C$, *then* $\vdash A \Rightarrow C$.

Proof. Given a proof of $A \Rightarrow B$ and $B \Rightarrow C$

$$\dfrac{\dfrac{\dfrac{axiom3}{(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))} \quad \dfrac{given}{A \Rightarrow B}}{(B \Rightarrow C) \Rightarrow (A \Rightarrow C)} \quad \dfrac{given}{B \Rightarrow C}}{A \Rightarrow C}$$

∎

3.2 A Proof Of $A \Rightarrow A$

Before continuing with the proof of correctness, we digress to give an example of a proof of a tautology. The simplest proof of an interesting propositional formula is most probably the proof of $A \Rightarrow A$. It follows almost immediately from the Transitivity Lemma just proved. Curiously, it is the only theorem one seems to come up with when playing with Łukasiewicz' axioms. Apparently the proofs of all other interesting theorems are too obscure to discover by accident. The proof of $A \Rightarrow A$ is shown in Figure 2. Compare it to the SML tree expression of type `proof` that represents it:

```
mp (
  mp (
    ax3 (A,impl(neg A,A),A),
    ax2 (A,A)),
  ax1 A)
```

$$\dfrac{\dfrac{\dfrac{axiom3}{(A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A))} \quad \dfrac{axiom2}{A \Rightarrow (\neg A \Rightarrow A)}}{((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)} \quad \dfrac{axiom1}{(\neg A \Rightarrow A) \Rightarrow A}}{A \Rightarrow A}$$

Figure 2: A proof of $A \Rightarrow A$

It is these proof trees that the completeness function must discover. The proof trees soon get too cumbersome and wide to fit on the page, so if they must be displayed, we write them in a linear fashion.

1	$(\neg A \Rightarrow A) \Rightarrow A$	axiom 1
2	$A \Rightarrow (\neg A \Rightarrow A)$	axiom 2
3	$(A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A))$	
4	$((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)$	$\mathit{MP}\ 2,3$
5	$A \Rightarrow A$	$\mathit{MP}\ 1,4$

The SML expression using `let` nicely linearizes the proof expression. It corresponds in form (as well as in substance) closely to the linear proof above.

```
let
  val pr1 = ax1 A;    (* (~A=>A)=>A *)
  val pr2 = ax2 (A,A);(* A=>(~A=>A) *)
  val pr3 = ax3 (A,impl(neg A,A),A);
  val pr4 = mp (pr3,pr2);
in
  mp (pr1, pr4)
end
```

The proof that $A \Rightarrow A$ is derivable can be further simplified by taking advantage of the Transitivity Lemma (Lemma 2), as follows:

1	$(\neg A \Rightarrow A) \Rightarrow A$	axiom 1
2	$A \Rightarrow (\neg A \Rightarrow A)$	axiom 2
3	$A \Rightarrow A$	lemma 2 $(2,1)$

Also, the SML proof expression can take advantage of a lemma, a function `transitivity` that applies modus ponens to two proof expressions. The result is a function that creates a proof expression for $A \Rightarrow A$ given any A:

```
fun derived1 (A) =
   transitivity (ax2(A,A), ax1 A);
```

3.3 Another Proof

Another tautology with a short proof is $\neg (A \Rightarrow A) \Rightarrow B$.

1	$(\neg A \Rightarrow A) \Rightarrow A$	axiom 1
2	$A \Rightarrow (\neg A \Rightarrow A)$	axiom 2
3	$(A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A))$	axiom 3
4	$((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)$	$MP\ 3, 2$
5	$A \Rightarrow A$	$MP\ 1, 4$
6	$(A \Rightarrow A) \Rightarrow (\neg (A \Rightarrow A) \Rightarrow B)$	axiom 2
7	$\neg (A \Rightarrow A) \Rightarrow B$	$MP\ 6, 5$

3.4 Other Lemmas

A chain of lemmas is required for the proof of the deduction theorem. We list them here without proof—the proofs should be obvious from looking at the code. Lemma 7 is Hilbert's axiom 1.

Lemma 3 *If* $\vdash \neg B \Rightarrow \neg A$*, then* $\vdash (A \Rightarrow B)$.

Lemma 4 *If* $\vdash A$*, then* $\vdash (B \Rightarrow A)$.

Lemma 5 *If* $\vdash B$ *and* $\vdash A \Rightarrow (B \Rightarrow C)$*, then* $\vdash (A \Rightarrow C)$.

Lemma 6 *For any propositional formulas* A *and* B*,* $\vdash (\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$.

Proof. Suppose $BB = (\neg B \Rightarrow B)$.

1	$BB \Rightarrow B$	axiom 1 (B)
2	$((\neg A \Rightarrow B) \Rightarrow BB) \Rightarrow ((BB \Rightarrow B) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow B))$	axiom 3
3	$((\neg A \Rightarrow B) \Rightarrow BB) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow B)$	lemma 5 $(1, 2)$
4	$(\neg B \Rightarrow \neg A) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow (\neg B \Rightarrow B))$	axiom 2
5	$(\neg B \Rightarrow \neg A) \Rightarrow ((\neg A \Rightarrow B) \Rightarrow B)$	lemma 2 $(4, 3)$
6	$A \Rightarrow (\neg A \Rightarrow B)$	axiom 2
7	$((\neg A \Rightarrow B) \Rightarrow B) \Rightarrow (A \Rightarrow B)$	lemma 1 $(B, 6)$
8	$(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$	lemma 2 $(5, 7)$

∎

Lemma 7 *For any propositional formulas* A *and* B*,* $\vdash A \Rightarrow (B \Rightarrow A)$.

Lemma 8 *For any propositional formula A,* $\vdash \neg\neg A \Rightarrow A$.

Lemma 9 *For any propositional formula A,* $\vdash A \Rightarrow \neg\neg A$.

Lemma 10 *If* $\vdash A \Rightarrow B$ *and* $\vdash \neg A \Rightarrow B$*, then* $\vdash B$.

Lemma 11 *For any propositional formulas A and B,* $\vdash \neg A \Rightarrow (A \Rightarrow B)$.

Lemma 12 *If* $\vdash A \Rightarrow (A \Rightarrow B)$*, then* $\vdash A \Rightarrow B$.

Lemma 13 *If* $\vdash A \Rightarrow (B \Rightarrow C)$ *and* $\vdash A \Rightarrow B$*, then* $\vdash (A \Rightarrow C)$.

3.5 The Deduction Theorem

The first version of the deduction theorem appeared in Herbrand's thesis [5]. It is a considerable breakthrough in theorem construction technique. But it requires that the notion of a proof be enlarged to include proofs using assumptions, hence a surprising extra constructor `assume` in the `proof` data type. The deduction theorem shows how to eliminate a use of an assumption in a proof tree, and provides another, completely different, way to build assumption-free proofs. It hinges on Lemmas 4 and 13.

Theorem 1 (Deduction) *Given a proof of A (possibly assuming B), there is a proof of $B \Rightarrow A$ without any assumptions of B.*

Like the bracket abstraction algorithm for combinators [2, 15] considerable savings can be obtained by eliminating the assumption only when it is actually used. In that case Lemma 4 can use used and the recursive use of the deduction theorem is avoided. Otherwise the two recursive calls result in an exponential explosion in proof size.

3.6 The Completeness Theorem

Lemma 14 *If* $\vdash A$ *and* $\vdash \neg B$*, then* $\vdash \neg(A \Rightarrow B)$.

Proof. Using $\vdash A$, $\vdash \neg B$ and the assumption $A \Rightarrow B$, a proof of $\neg(A \Rightarrow B)$ can be found. Using the deduction theorem and Lemma 10 an assumption-free proof can be found. ∎

```
fun deduction a (assume b) =
      if a=b then derived1(a) else lemma_2 a (assume b)
 |  deduction a (mp (p1,p2,_)) =
      let
        fun f p = if occurs a p then deduction a p else lemma_2 a p;
      in
        lemma_11 (f p1, f p2)
      end
  |  deduction a p = lemma_2 a p
;
fun F sg (prop x) = if sg x then assume (prop x) else assume (neg(prop x))
 |  F sg (neg p) =
      if value sg p
        then modus_ponens (lemma_7 p, F sg p)   (*  |- p  ==>  |- ~~p  *)
        else F sg p                             (*  |- ~p              *)
 |  F sg (impl(p,q)) =
      if value sg p
        then if value sg q
          then lemma_2 p (F sg q)   (* |- q ==> |- p=>q  *)
          (*  |- p  && |- ~q  ==>  |- ~(p=>q)  *)
          else lemma_12 (F sg p, F sg q)
        else modus_ponens (lemma_9 (p,q), F sg p)(* |- ~p ==> |- p=>q *)
;
local
  fun elim v prt prf =
    lemma_8 (deduction (prop v) prt, deduction (neg (prop v)) prf);
  fun allp phi sg  nil     = F sg phi
   |  allp phi sg (v::vs)  =
        let
           val prt = allp phi (update sg v true) vs)
           val prf = allp phi (update sg v false) vs)
        in
           elim v prt prf
        end
in
  fun completeness phi = allp phi undef (propositions phi nil)
end;
```

Figure 3: The `completeness` function.

With this lemma we can write a function (called `F` in figure 3) which, given any assignment and any propositional formula, can construct a proof of the formula or its negation (depending on which is true in the assignment). The proof assumes a proof of each proposition occurring in the formula or its negation (depending on which is true in the assignment). With this proof-constructing function we are finally ready for the Completeness Theorem.

Theorem 2 (Completeness) *Given a propositional formula A that is a tautology, then there is a proof of A.*

Proof. Since A is a tautology, the function `F` will construct a proof of it assuming any combination of values for the propositions. It systematically uses the deduction theorem to get proofs of $P \Rightarrow A$ and $\neg P \Rightarrow A$, and then uses Lemma 10 to get a proof of A. After all propositions P have been eliminated, the proof is assumption free. ∎

A look at the code (shown partially in Figure 3) makes this clear. Many standard mathematical proofs with their unnatural induction arguments over natural numbers not only obscure the procedure, but also fail to be fully convincing.

4 Conclusions

No proof can be constructed by any of the SML and Haskell functions that does not really represent a proof in Łukasiewicz' axiom system. The type system insures the "soundness" of any proofs. To ensure this requires the hiding of the type constructor `mp` by the function `modus_ponens`.

```
local
  fun check (impl(p,q),r) =
        if p=r then q else raise error
   |  check (p,_) = raise error
in
  fun modus_ponens (p,q) =
    mp(p,q,check(proof_of p,proof_of q))
end;
```

This can be accomplished by the `abstype`/`with` construct in the SML language. For convenience we keep the formula for which modus ponens is a proof in the third argument of the `mp` constructor.

	F	back	trans	MP	size
$P \Rightarrow P$	4	145	317	842	1,673
$(\neg\neg P) \Rightarrow P$	6	173	379	1,006	1,999
$P \Rightarrow (\neg\neg P)$	6	173	379	1,006	1,999
$P \Rightarrow (Q \Rightarrow P)$	10	689	1,493	3,974	7,889
$P \Rightarrow (Q \Rightarrow Q)$	10	899	1,943	5,174	10,269
$(\neg P \Rightarrow P) \Rightarrow P$	7	742	1,603	4,270	8,473
$P \Rightarrow (\neg P \Rightarrow Q)$	12	458	1,003	2,663	5,291
$\neg P \Rightarrow (P \Rightarrow Q)$	12	458	1,003	2,663	5,291
$(\neg (P \Rightarrow P)) \Rightarrow Q$	16	572	1,253	3,325	6,607
$P \Rightarrow (\neg (P \Rightarrow \neg P))$	8	770	1,665	4,434	8,799
$(P \Rightarrow \neg P) \Rightarrow \neg P$	8	770	1,665	4,434	8,799
$(\neg (P \Rightarrow Q)) \Rightarrow P$	14	725	1,578	4,194	8,329
$(\neg (P \Rightarrow Q)) \Rightarrow (\neg\neg P)$	16	753	1,640	4,358	8,655
$(\neg (P \Rightarrow Q)) \Rightarrow \neg Q$	15	935	2,028	5,394	10,709
$(P \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$	16	1,652	3,575	9,519	18,891
$(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$	16	2,351	5,065	13,496	26,777
$(P \Rightarrow \neg Q) \Rightarrow (Q \Rightarrow \neg P)$	15	2,323	5,003	13,332	26,451
$(\neg P \Rightarrow \neg Q) \Rightarrow (Q \Rightarrow P)$	15	2,407	5,183	13,812	27,403
$(\neg (P \Rightarrow Q)) \Rightarrow (Q \Rightarrow R)$	30	1,982	4,301	11,439	22,711
$(\neg (P \Rightarrow Q)) \Rightarrow (\neg P \Rightarrow R)$	32	1,618	3,525	9,367	18,603
$(\neg P \Rightarrow \neg Q) \Rightarrow ((\neg P \Rightarrow Q) \Rightarrow P)$	18	3,732	8,28	21,399	42,451
$(P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$	34	10,070	21,642	57,694	114,447
$(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$	36	10,975	23,581	62,866	124,705

Figure 4: Indications of the size of the proofs found by the completeness theorem

On the other hand, the language does not insure that the completeness function, a function from propositional formulas `prop` to proofs `proof`, actually performs as advertised on all formulas. It could build a proof with assumptions or it could build a proof of some other propositional formula. The required property is

```
proof_of (completeness (phi)) = phi
```

for all formulas `phi`. It can easily be seen that each step of the program/proof builds a proof of the expected propositional formula.

Some optimizations are necessary to the completeness function to get it to work efficiently at all. Most importantly, the deduction must remove assumptions in a proof only when they are in fact used. To apply the transformation needlessly results in even larger proof trees. It is also possible to exploit partial assignments in the manner of the tautology checker mentioned earlier. This has a modest effect and is not shown in figure 3. The algorithm rarely tries to prove any instances of the three axioms. So, an optimization that checks for that situation has little effect.

The proof trees created by the completeness function are quite large. Figure 4 lists some measures of the work done by the function for a number of examples. The meaning of the columns is given here:

F Calls to the function `F`.

back Calls to the Backward Propagation Lemma (Lemma 1).

trans Calls to the Transitivity Lemma (Lemma 2).

MP Calls to the proof constructor modus ponens

size Number of times modus ponens is used in the final proof plus the number of axioms used.

The poor performance is obvious; the proof of $A \Rightarrow A$ given earlier has size 5. The completeness function finds a proof as promised, but it has size 1,673 (the first line of the table).

Notice that the deduction theorem tears down proof trees and builds them back up again. It is for this reason that the number of times modus ponens is used in the completeness function is greater than the number of times modus ponens appears in the final proof tree.

The Haskell code does not differ significantly from the SML code. In particular, the use of lazy evaluation does not appear to be any advantage in these functions. However, proof tactics, functions that discover proofs, could benefit. If a proof is known, it should be substituted before the expense of finding one using the completeness function.

One interesting programming note concerns exception handling. We have seen two different uses of exceptions in the SML snippets that appear here. One is for errors and one controls the execution of the opportunistic tautology checker. Pure functional languages such as Haskell have no exception handling since it introduces issues with the order of evaluation. This is not missed in the first case. But a tautology checker that takes advantage of the truth table of implication is harder to write without exception handling.

References

[1] Stanley N. Burris. *Logic for Mathematics and Computer Science*. Prentice Hall, Upper Saddle River, New Jersey, 1998.

[2] Haskell Brooks Curry and Robert Feys. *Combinatory Logic*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, 1958.

[3] Colm Ó Dúinlaing. Completeness of some axioms of Łukasiewicz's: An exercise in problem-solving. TCDMATH 97-05, Trinity College, Dublin, June 1997.

[4] Friedrich Ludwig Gottlob Frege. *Begriffsschrift, eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*. Halle, 1879. Translation appears in [17] pages 1–82.

[5] Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, Paris, France, 1930. Translation of Chapter 5 appears in [17] pages 525–581.

[6] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Verlag von Julius Springer, Berlin, second, revised edition, 1938.

[7] David Hilbert and Wilhelm Ackermann. *Principles of Mathematical Logic*. Chelsea, 1950. Edited with notes by Robert E. Luce. Translation of [6] by Lewis M. Hammond, George G. Leckie and F. Steinhardt.

[8] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), May 1992. A newer version is available on the WWW at `http://www.haskell.org/tutorial/`.

[9] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report of the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[10] Jan Łukasiewicz. *Jan Łukasiewicz, Selected Writings*. North-Holland, 1970. Edited by L. Borowski.

[11] Jan Łukasiewicz and Alfred Tarski. Untersuchungen über den Aussagenkalkül. *Comptes Rendus des Séances de la Societé des Sciences et des Lettres de Varsovie, Classe III*, 23:1–21, 1930. Reprinted and translated in [10].

[12] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, England, second edition, 1996.

[13] Emil Leon Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics 43*, pages 163–185, 1921. Reprinted in [17] pages 264–282.

[14] Steve Reeves and Michael Clarke. *Logic for Computer Science*. International computer science series. Addison-Wesley, Wokingham, England, 1990.

[15] Ryan Stansifer. *The Study of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[16] Ryan Stansifer. Completeness of propositional logic as a program (with code). Technical Report CS-2001-1, Department of Computer Sciences, Florida Institute of Technology, 2001. Available at `www.cs.fit.edu/~tr/2001`.

[17] Jan van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

A SML Code

```
datatype prop = prop of string | impl of prop * prop | neg of prop;

(* assigment of infinite number of propositions to their value. *)
type assignment = string -> bool;

(* value of a formula given an assignment *)
fun value sg (prop n)     = sg n
 |  value sg (impl (h,s)) = not (value sg h) orelse (value sg s)
 |  value sg (neg phi)    = not (value sg phi)
 ;

exception not_found of string;

fun undef n = raise not_found n;
fun update f x y z = if z=x then y else f z;

(* semantic tautology checker *)
local
  fun check' sg phi =
    value sg phi handle not_found p =>
      check' (update sg p true) phi andalso
        check' (update sg p false) phi;
in
  fun checker phi = check' undef phi;
end;

datatype proof =
   assume of prop |
   ax1 of prop    |        (* Lk1:    (~P => P) => P           *)
   ax2 of prop*prop   |    (* Lk2:    P => (~P=>Q)             *)
   ax3 of prop*prop*prop | (* Lk3:    P=>Q => ((Q=>R)=>(P=>R)) *)
   mp of proof*proof*prop;

fun axiom1 p       = impl(impl(neg p,p),p);
fun axiom2 (p,q)   = impl(p, impl(neg p,q));
fun axiom3 (p,q,r) = impl (impl(p,q), impl(impl(q,r), impl(p,r)));
```

```

fun proof_of (assume p)  = p
 |  proof_of (ax1 p)     = axiom1 p
 |  proof_of (ax2 (p,q)) = axiom2 (p,q)
 |  proof_of (ax3 (p,q,r)) = axiom3 (p,q,r)
 |  proof_of (mp (_,_,p))= p;

(*  is the formula "p" used in the a proof?    *)
fun occurs p (assume q) = p=q
 |  occurs p (mp (p1,p2,_)) = occurs p p1 orelse occurs p p2
 |  occurs p (_) = false
 ;

exception not_implication of prop;
exception not_hypothesis;

(*  The constructor of type "proof" should not be used, because it
    does not (and cannot) check its arguments to see if they are in
    the right form.
*)
local
  fun check (impl(p,q),r) = if p=r then q else raise not_hypothesis
   |  check (p,_) = raise not_implication p
in
  fun modus_ponens (p,q) = mp (p,q,check (proof_of p, proof_of q))
end;

(*  example proofs  *)
val P = prop"P"; val Q = prop"Q"; val R = prop"R"; val S = prop"S";
val pr1 = ax3 (impl(neg P,P), P, Q);
val pr2 = modus_ponens (pr1, ax1 (P)); (* (P=>Q) => ((~P=>P)=>Q) *)
val pr3 = modus_ponens (ax3(P,impl(neg P,P),P), ax2(P,P));


(* backward propagation; derived rule of inference;
   Given any proposition C,  |-A=>B ==> |- (B=>C) => (A=>C)
```

```
 *)
fun backward C (pr1) =
  let
    val impl(A,B) = proof_of (pr1);
  in
    modus_ponens (ax3(A,B,C), pr1)
  end;

(* transitivity; derived rule of inference;
   |-A=>B, |-B=>C ==> |-A=>C
 *)
fun transitivity (pr1, pr2) =
  let
    val impl(A,B) = proof_of (pr1);
    val impl(D,C) = proof_of (pr2);
    (*  ax3:    A=>B => ((B=>C)=>(A=>C))   *)
    val pr3 = modus_ponens (ax3(A,B,C), pr1);  (* (B=>C)=>(A=>C) *)
  in
    (*  If D<>B, then the next application of MP won't work!    *)
    modus_ponens (pr3, pr2)
  end;

fun derived1 (A) = transitivity (ax2(A,A), ax1 A);  (*  A=>A  *)

(*  Lemma 1.
    |- ~B=>~A  ==>  |- A=>B
*)
fun lemma_1 (pr1) =
  let
    val impl(neg B,neg A) = proof_of pr1;
    val pr2 = backward B pr1;                (* ~A=>B => ~B=>B *)
    val pr3 = transitivity (ax2(A,B), pr2);  (* A=>(~B=>B)     *)
  in
    transitivity (pr3, ax1 B)       (*  A=>B *)
  end;

(*  Lemma 2.    Requires lemma_1
    |- A  ==>  |- B=>A
*)
```

```
fun lemma_2 B pr1 =
  let
    val A = proof_of pr1;
    val pr2 = modus_ponens (ax2(A,neg B), pr1);  (* ~A=>~B  *)
  in
    lemma_1 pr2
  end;

(*  Lemma 3.    Requires lemma_2
    |- B,  |- A=>(B=>C)  ==>  |- A=>C
*)
fun lemma_3 (pr1,pr2) =
  let
    val impl(A,impl(B,C)) = proof_of pr2;
    val pr3 = lemma_2 (neg C) pr1;  (*  ~C => B  *)
    val pr4 = backward C pr3;   (* B=>C  => (~C=>C)  *)
    val pr5 = transitivity (pr4, ax1 C);  (* B=>C  => C*)
  in
    transitivity (pr2, pr5)
  end;

(* Lemma 4.    Requires lemma_3
   |- (~B => ~A) => (A=>B)
*)
fun lemma_4 (A,B) =
  let
    val pr1 = ax3 (impl(neg A,B), impl(neg B,B), B);
    val pr2 = lemma_3 (ax1 B, pr1);
    val pr3 = ax3 (neg B, neg A, B);
    val pr4 = transitivity (pr3, pr2);
    val pr5 = backward B (ax2(A,B));
  in
    transitivity (pr4, pr5)
  end;

(*  Lemma 5.    Requires lemma_4.
    |- A => (B=>A)
*)
fun lemma_5 (A,B) = transitivity (ax2 (A,neg B), lemma_4 (B,A));
```

```
(*  Lemma 6.    Requires lemma_5, lemma_4
    |- ~~A => A
*)
fun lemma_6 A =
  let
    val pr1 = lemma_5 (neg(neg A), neg A);(* ~~A=>(~A=>~~A) *)
    val pr2 = lemma_4 (neg A, A);         (* (~A=>~~A) => (~A=>~A)*)
    val pr3 = transitivity (pr1, pr2);    (*  ~~A => (~A => A) *)
  in
    transitivity (pr3, ax1 A)
  end;

(*  Lemma 7   |-  A => ~~A  *)
fun lemma_7 A = lemma_1 (lemma_6 (neg A));

(*  Lemma 8.    Requires lemma_4, lemma_7
    |-  A=>B,  |- ~A=>B  ==>  |- B
*)
fun lemma_8 (pr1,pr2) =
  let
    val impl(neg A,B) = proof_of pr2;
    val pr3 = transitivity (pr2, lemma_7 B);   (* ~A=>~~B  *)
    val pr4 = modus_ponens (lemma_4 (neg B, A), pr3); (* ~B=>A *)
    val pr5 = transitivity (pr4, pr1);  (* ~B => B *)
  in
    modus_ponens (ax1 B, pr5)  (*  B  *)
  end;

(*  Lemma 9.    Requires lemma_7
    |- ~A => (A=>B)
*)
fun lemma_9 (A,B) =
  let
    val pr1 = ax2 (neg A, B);  (* ~A => (~~A=>B)      *)
    val pr2 = lemma_7 A;       (* A => ~~A             *)
    val pr3 = backward B pr2;  (* (~~A=>B) => (A=>B)  *)
  in
```

```
    transitivity (pr1, pr3)
  end;


(*  Lemma 10.    Requires lemma_9
    |-  A => (A=>B)  ==>  |- A=>B
*)
fun lemma_10 (pr1) =
  let
    val impl(A,impl(_,B)) = proof_of pr1;
    val pr2 = lemma_9 (A,B);   (* ~A => (A=>B)  *)
  in
    lemma_8 (pr1, pr2)
  end;

(*  Lemma 11.    Requires lemma_10
    |- A=>(B=>C)   |- A=>B  ==>  |- A=>C
*)
fun lemma_11 (pr1, pr2) =
  let
    val impl(A,impl(B,C)) = proof_of (pr1)
    val pr3 = backward C pr2;            (* B=>C => A=>C  *)
    val pr5 = backward (impl(A,C)) pr1;  (* (B=>C)=>(A=>C)=>(A=>(A=>C))*)
    val pr6 = modus_ponens (pr5, pr3);   (* A=> (A=>C) *)
  in
    lemma_10 (pr6)
  end;

(*
   The deduction theorem
*)
fun deduction a (assume b) =
      if a=b
        then derived1 a   (* A=>A *)
        else lemma_2 a (assume b)
 |  deduction a (mp (p1,p2,_)) =
      let
        fun f p = if occurs a p then deduction a p else lemma_2 a p;
```

```
        (* deduction a p1 :  A=>(P=>Q)  *)
        (* deduction a p2 :  A=>P       *)
      in
        lemma_11 (f p1, f p2)
      end
 |  deduction a p = lemma_2 a p
 ;


(*  Lemma 12.    Requires the deduction theorem, lemma_8.
    |- A,  |- ~B  ==>  |- ~(A=>B)
*)
fun lemma_12 (pr1, pr2) =
  let
    val A = proof_of (pr1);
    val neg B = proof_of (pr2);
    val i = impl (A,B);
    val pr4 = modus_ponens (assume i, pr1);    (*  B  *)
    val pr5 = modus_ponens (ax2 (B, neg i), pr4);
    val pr6 = modus_ponens (pr5, pr2);     (*  ~(A=>B)  *)
    val pr7 = deduction i pr6;    (*  (A=>B) => ~(A=>B) *)
    val pr8 = derived1 (neg i);   (*  ~(A=>B) => ~(A=>B)  *)
  in
    lemma_8 (pr7, pr8)
  end;

fun assuming sg x =
  assume (if sg x then prop x else neg(prop x));

fun F sg (prop x) = assuming sg x
 |  F sg (neg p) =
      if value sg p
        then modus_ponens (lemma_7 p, F sg p) (* |-p=>~~p,|-p ==> |-~~p *)
        else F sg p                           (* |- ~p *)
 |  F sg (impl(p,q)) =
      if value sg p
        then if value sg q
          then lemma_2 p (F sg q)        (* |- q ==> |- p=>q *)
```

```
          else lemma_12 (F sg p, F sg q) (* |-p,|-~q  ==> |-~(p=>q) *)
        (* |-~p=>(p=>q), |-~p  ==>  |-p=>q    *)
        else modus_ponens (lemma_9 (p,q), F sg p)
 ;

local
  fun elim v prt prf =
    lemma_8 (deduction (prop v) prt, deduction (neg (prop v)) prf);

  fun allp sg phi =
    F sg phi handle not_found v =>
     let
       val prt = allp (update sg v true) phi  (* v, ...|- phi *)
       val prf = allp (update sg v false) phi (* ~v,...|- phi *)
     in
       elim v prt prf
     end;
in
  fun completeness phi = allp undef phi
end;
```

B Haskell Code

```
data Formula = Prop String | Neg Formula | Impl (Formula,Formula)
               deriving(Eq)

instance Show Formula where
  showsPrec p (Prop s) = shows s
  showsPrec p (Neg (Prop s)) = showChar '~' .  shows s
  showsPrec p (Neg phi) = showString "(~" .  shows phi .  showChar ')'
  showsPrec p (Impl (x,y)) = showChar '(' .  shows x .  showString " => " .  shows y .  showChar ')'


propositions (Prop s) l     = if elem s l then l else s:l
propositions (Neg p)  l     = propositions p l
```

```
propositions (Impl (p,q)) l = propositions q (propositions p l)

value sg (Prop s) = sg s
value sg (Neg phi) = not (value sg phi)
value sg (Impl (phi,psi)) = (not (value sg phi)) || (value sg psi)

undef _ = error "not found"
update f x y z = if z==x then y else f z

check phi = check’ phi undef (propositions phi [])
  where
    check’ phi sg []     = value sg phi
    check’ phi sg (v:vs) =
       check’ phi (update sg v True) vs && check’ phi (update sg v
       False) vs


data Proof = Assume Formula                |
             Ax1 Formula                   |
             Ax2 (Formula,Formula)         |
             Ax3 (Formula,Formula,Formula) |
             Mp (Proof,Proof,Formula)
             deriving(Eq)

instance Show Proof where
    showsPrec p x = shows (proof_of x)

proof_of (Assume p) = p
proof_of (Ax1 p) = axiom1 p
proof_of (Ax2 (p,q)) = axiom2 p q
proof_of (Ax3 (p,q,r)) = axiom3 p q r
proof_of (Mp (_,_,p)) = p

occurs p (Assume q)     = (p==q)
occurs p (Mp (p1,p2,_)) = occurs p p1 || occurs p p2
occurs p (_)            = False

axiom1 p = Impl (Impl (Neg p, p), p)
axiom2 p q = Impl (p, (Impl (Neg p, q)))
axiom3 p q r = Impl (Impl (p,q), Impl (Impl (q,r), (Impl (p,r))))
```

```
modus_ponens p q = Mp (p,q, check (proof_of p, proof_of q))
  where
    check (Impl(p,q),r) = if p==r then q else error "not hypothesis"
    check (p, _)        = error "not implication"


backward c pr1 = modus_ponens (Ax3 (a,b,c)) pr1
  where
    Impl (a,b) = proof_of pr1

transitivity (pr1, pr2) = modus_ponens (modus_ponens (Ax3 (a,b,c)) pr1) pr2
  where
    Impl (a,b) = proof_of pr1
    Impl (_,c) = proof_of pr2

derived1 a = transitivity (Ax2(a,a), Ax1 a)

lemma_1 pr1 = transitivity (pr3, Ax1 b)
  where
    pr3 = transitivity (Ax2 (a,b), pr2)
    pr2 = backward b pr1
    Impl (Neg b, Neg a) = proof_of pr1

lemma_2 b pr1 =lemma_1 pr2
  where
    pr2 = modus_ponens (Ax2(a,Neg b)) pr1  -- ~A=>~B
    a = proof_of pr1

lemma_3 (pr1,pr2) = transitivity (pr2, pr5)
  where
    pr5 = transitivity (pr4, Ax1 c)
    pr4 = backward c pr3
    pr3 = lemma_2 (Neg c) pr1
    Impl(a,Impl(b,c)) = proof_of pr2

lemma_4 (a,b) = transitivity (pr4,pr5)
  where
```

```
    pr5 = backward b (Ax2 (a,b))
    pr4 = transitivity (pr3, pr2)
    pr3 = Ax3 (Neg b, Neg a, b)
    pr2 = lemma_3 (Ax1 b, pr1)
    pr1 = Ax3 (Impl (Neg a,b), Impl (Neg b,b), b)


lemma_5 (a,b) = transitivity (Ax2 (a, Neg b), lemma_4 (b,a))

lemma_6 a = transitivity (pr3, Ax1 a)
  where
    pr3 = transitivity (pr1, pr2)
    pr2 = lemma_4 (Neg a, a)
    pr1 = lemma_5 (Neg (Neg a), Neg a)


lemma_7 a = lemma_1 (lemma_6 (Neg a))

lemma_8 (pr1, pr2) = modus_ponens (Ax1 b) pr5
  where
    pr5 = transitivity (pr4, pr1)
    pr4 = modus_ponens (lemma_4 (Neg b, a)) pr3  -- ~B=>A
    pr3 = transitivity (pr2, lemma_7 b)          -- ~A=>~~B
    Impl (Neg a, b) = proof_of pr2

lemma_9 (a,b) = transitivity (pr1, pr3)
  where
    pr3 = backward b pr2;  -- (~~A=>B) => (A=>B)
    pr2 = lemma_7 a;       -- A => ~~A
    pr1 = Ax2 (Neg a, b);  -- ~A => (~~A=>B)

lemma_10 (pr1) = lemma_8 (pr1, pr2)
  where
    pr2 = lemma_9 (a,b);
    Impl(a,Impl(_,b)) = proof_of pr1;


lemma_11 (pr1, pr2) = lemma_10 (pr6)
  where
    pr6 = modus_ponens pr5 pr3;      -- A => (A=>C)
```

```
    pr5 = backward (Impl (a,c)) pr1  -- (B=>C)=>(A=>C)=>(A=>(A=>C))
    pr3 = backward c pr2;            -- (B=>C) => (A=>C)
    Impl(a,Impl(_,c)) = proof_of pr1;

deduction a (Assume b)    = if a==b then derived1 a else lemma_2 a (Assume b)
deduction a (Mp (p1,p2,_))= lemma_11 (f p1, f p2)
  where
    f p = if occurs a p then deduction a p else lemma_2 a p;
deduction a p             = lemma_2 a p

lemma_12 (pr1, pr2) = lemma_8 (pr7,pr8)
  where
    pr8 = derived1 (Neg i);      -- ~(A=>B) => ~(A=>B)
    pr7 = deduction i pr6;       --  (A=>B) => ~(A=>B)
    pr6 = modus_ponens pr5 pr2;
    pr5 = modus_ponens (Ax2 (b, Neg i)) pr4;
    pr4 = modus_ponens (Assume i) pr1;
    i = Impl (a,b);
    Neg b = proof_of pr2;
    a = proof_of pr1;

assuming sg x = Assume (if sg x then Prop x else Neg (Prop x))

f sg (Prop x)    = assuming sg x
f sg (Neg p)     = if value sg p then modus_ponens (lemma_7 p) (f sg p) else f sg p
f sg (Impl (p,q))=
  if value sg p
    then if value sg q then lemma_2 p (f sg q) else lemma_12 (f sg p, f sg q)
    else modus_ponens (lemma_9 (p,q)) (f sg p)

elim v prt prf = lemma_8 (deduction (Prop v) prt, deduction (Neg (Prop v)) prf)

completeness phi = completeness' phi undef (propositions phi [])
  where
    completeness' phi sg []     = f sg phi
```

```
    completeness' phi sg (v:vs) =
       elim v
         (completeness' phi (update sg v True) vs)
         (completeness' phi (update sg v False) vs)

p = Prop "P"
x = Impl(p,p)
```