Proving Inductive Properties of PROLOG Programs in Second-Order Intuitionistic Logic

L. Thorne McCarty
Computer Science Department
Rutgers University
New Brunswick, NJ 08903, USA
mccarty@cs.rutgers.edu

Abstract

We use second-order intuitionistic logic for two purposes in this paper: first, to formulate a large class of circumscriptive queries about logic programs; and second, to formulate a class of induction schemata that can be used to answer these queries. Then, applying the techniques of intuitionistic logic programming, we develop an interpreter (written in PROLOG) that constructs inductive proofs and answers circumscriptive queries automatically. As an illustration, we prove the symmetry of ‘Reverse’, the associativity of ‘Plus’ and the commutativity of ‘Times’.

1 Introduction

Most of the research on automated induction has been situated squarely within the functional programming paradigm. The landmark study was by Boyer and Moore [1], who worked with a variant of LISP, and much of the subsequent work inspired by the Boyer/Moore Theorem-Prover has continued in the same tradition [2]. Recent work by Bundy, et al., uses a version of Martin-Löf’s constructive type theory [22] as an object language, and searches for inductive proofs within this framework. There have been many contributions from this research: The idea of “rippling” as a proof strategy has been extensively developed [4], and the search for induction schemata has been greatly facilitated by the use of abstract “proof plans” [3]. The study of inductive proofs has also been an important part of the work on Nuprl [7] and LCF [35].

By contrast, the research on automated induction within the logic programming paradigm has been less extensive. There is an early paper by Clark and Tarnlund [6] on the subject, but the pioneering research was by Kanamori and Seki [17] and Kanamori and Fujita [16]. Kanamori and Seki [17] proposed an extended model of PROLOG execution, and showed
how this could be used for program verification. A companion paper by Kanamori and Fujita [16] analyzed several techniques for the formulation of induction schemata within this extended model, and showed how two or more such schemata could be merged into one. These ideas have been further developed and refined in a series of papers by Fribourg [10, 11, 12]. Other contributions include the work of Hsiang and Srivas [15], Elkan and McAllester [8] and Lever [20]. However, compared with the extensive literature on functional programming, the literature on automated induction in logic programming is relatively sparse.

This is surprising, since logic programs would appear to have several advantages as an object language for inductive proofs. For example, many of the applications of “rippling” are motivated by the need to move constructors “up” out of deeply nested function applications, and the “flat” syntax of logic programs would obviate such operations. Also, given the simple and uniform nature of a “proof” in logic programming, the concept of a “proof plan” ought to be correspondingly simple and uniform. In this paper, we will validate these observations to some extent by formulating the problem of induction in a simple, but powerful, logical language [37]: second-order intuitionistic logic.

Actually, we use second-order intuitionistic logic in this paper in two ways. First, we define the problem we are trying to solve using circumscription [23, 24]. The circumscription axiom is usually interpreted as a sentence in second-order classical logic, but we interpret it instead as a sentence in second-order intuitionistic logic. For many applications, the classical and intuitionistic interpretations are equivalent, but the intuitionistic version provides additional flexibility in certain situations. For example, we show how to represent both “closed world” and “open world” assumptions circumspectively, and how to combine these assumptions with a uniform semantics for “negation-as-failure”. A brief discussion of the possible range of circumscriptive queries is presented in Section 2, but the reader is referred to [29, 27, 28] for a more detailed analysis.

The other use of second-order intuitionistic logic arises as part of our application of intuitionistic logic programming [13, 14, 25, 26, 32, 33]. Intuitionistic logic programming allows implications to appear as goals, and thus allows rules to have implications embedded in their antecedents. A strong version of intuitionistic logic programming, first discussed in [25, 26], allows rules to have embedded first-order universal quantifiers as well. In the present paper, we extend this language further to allow embedded second-order universal quantifiers. This particular fragment of intuitionistic logic has previously been used by Miller [34] to provide a lexical scoping mechanism for predicates.

Our application is different, but similar in some respects. We use embedded implications with embedded second-order universal quantifiers to represent a class of induction schemata for circumscriptive queries. We then use the standard nondeterministic search procedure for intuitionistic logic
programs — as described in [26], for example — to try to find a proof of a given query from a given induction schema. This approach has several advantages: First, the proof procedure can automatically propose inductive lemmas during the course of its search, by an analysis of the failed branches in the proof tree. Second, to prune out those inductive lemmas that cannot possibly succeed, the proof procedure attempts initially to construct a prototypical proof, i.e., a proof in the base case. If the prototypical proof fails, then the inductive proof of the lemma will also fail. But if the prototypical proof succeeds, then the proof procedure can automatically generate a new induction schema in the form of a second-order embedded implication, and the search can continue. These ideas are presented in Sections 3–4.

To demonstrate that this is a practical approach, we have written an interpreter for circumscriptive queries (in PROLOG) and tested it on the simple list examples and arithmetical examples in [1]. We describe our interpreter briefly in Section 5, and show how it finds inductive proofs for three standard examples: the symmetry of ‘Reverse’, the associativity of ‘Plus’ and the commutativity of ‘Times’.

2 Circumscriptive Queries

We analyze the role of circumscription [23, 24] in this section by means of two simple examples. The theoretical foundations of this work have been presented elsewhere [29, 27, 28].

Example 2.1: Consider the usual definition of ‘Naive Reverse’:

\[
\text{Append}([], l, l) \Leftarrow \quad (1)
\]
\[
\text{Append}(k | l, m, [k | n]) \Leftarrow \text{Append}(l, m, n) \quad (2)
\]
\[
\text{Reverse}([], []) \Leftarrow \quad (3)
\]
\[
\text{Reverse}(q | r, p) \Leftarrow \text{Reverse}(r, s) \land \text{Append}(s, [q], p) \quad (4)
\]

Intuitively, ‘Reverse’ should be a symmetric relation, and thus the following universally quantified implication should be true:

\[
(\forall x, y)[\text{Reverse}(y, x) \Leftarrow \text{Reverse}(x, y)]. \quad (5)
\]

However, (5) does not follow from (1)–(4), nor does it follow from Clark’s Predicate Completion [5] applied to (1)–(4). □

Instead, we ask whether (5) follows from the circumscription of ‘Append’ and ‘Reverse’ in (1)–(4). In this case, the answer is: Yes.

Thus, circumscription is a way of stating the “closed world” assumption with respect to the predicates ‘Append’ and ‘Reverse’. However, we can also use circumscription to guarantee that certain predicates are instead subject to an “open world” assumption.

Example 2.2: Suppose the following definition is part of the “blocks world”:

\[
\text{Above}(x, y) \Leftarrow \text{On}(x, y) \quad (6)
\]
Above\((x, y) \iff On(x, z) \land Above(z, y)\) \hfill (7)

We know that there may be one or more ‘On’ relationships true in the world, but we don’t know which ones, i.e., we have adopted an “open world” assumption with respect to the predicate ‘On’. However, suppose we have also been told that ‘Above(a,b)’ is true according to the definition in (6)–(7).

Is there something on ‘b’? Intuitively, the answer should be: Yes. □

To formalize this problem, let us first adopt the following auxiliary definition:

Covered\((y) \iff On(w, y)\) \hfill (8)

We now circumscribe the predicate ‘Above’ in (6)–(7) — but not the predicate ‘On’ — and ask whether the following implication:

\[
\text{Covered}(b) \iff \text{Above}(a, b)
\] \hfill (9)

is entailed by the circumscription axiom. (Note that we are using the auxiliary definition here simply to ask a hypothetical question with an existential conclusion, and, by using a similar device, we could obviously ask a hypothetical question whose conclusion consisted of any positive existential formula.) The answer to this query is: Yes.

We call this the circumscription query problem. Although we could certainly use classical logic in these examples, as is customary [21], we prefer to formalize circumscription in intuitionistic logic and to analyze its properties using Kripke models [19, 9]. Let \(\mathcal{R}\) be a finite set of definite Horn clauses, and let \(P = <P_1, P_2, \ldots, P_k>\) be a tuple consisting of the “defined predicates” that appear on the left-hand sides of the sentences in \(\mathcal{R}\). Let \(\mathcal{R}(P)\) denote the conjunction of the sentences in \(\mathcal{R}\), with the predicate symbols in \(P\) treated as free parameters, and let \(\mathcal{R}(X)\) be the same as \(\mathcal{R}(P)\) but with the predicate constants \(<P_1, P_2, \ldots, P_k>\) replaced by predicate variables \(<X_1, X_2, \ldots, X_k>\).

**Definition 2.3:** The circumscription axiom is the following sentence in second-order intuitionistic logic:

\[
\mathcal{R}(P) \land (\forall X)[(\forall x)[X_i(x) \Rightarrow P_i(x)]] \Rightarrow (\forall x)[P_i(x) \Rightarrow X_i(x)]
\]

We denote this expression by \(\text{Circ}(\mathcal{R}(P); P)\), and we refer to it as “the circumscription of \(P\) in \(\mathcal{R}(P)\).” The circumscription axiom has the same intuitive meaning here that it has in classical logic. It states that the extensions of the predicates in \(P\) are as small as possible, given the constraint that \(\mathcal{R}(P)\) must be true. Since the logic is intuitionistic, however, the axiom minimizes extensions at every state of every Kripke structure that satisfies \(\mathcal{R}\).

One reason for using intuitionistic logic in Definition 2.3 rather than classical logic is that the circumscripive query problem then has a natural generalization beyond the class of Horn clauses. The queries in the preceding
examples, (5) and (9), are themselves implications, and we ought to be able to embed such implications into other definitional rules. One particularly interesting case occurs when the embedded implications are actually embedded negations.

Example 2.4: Suppose ‘Above’ and ‘Covered’ are defined as in Example 2.2. Assume that ‘¬’ denotes negation, and define:

\[ \text{Clear}(y) \iff \neg \text{Covered}(y). \] (10)

There are now two cases to consider. First, if we adopt the “closed world” assumption with respect to the predicate ‘On’, then we would expect ‘¬’ to behave like negation-as-failure, and we should be able to conclude ‘Clear(b)’ from the absence of information about ‘On(w,b)’. On the other hand, if we adopt the “open world” assumption with respect to the predicate ‘On’, then we ought not to conclude ‘Clear(b)’ in such a situation. However, we should still be able to conclude:

\[ (\forall x)[\bot \iff \text{Above}(x,b) \land \text{Clear}(b)] \] (11)

In other words: If ‘b’ is clear, then there is nothing above it. □

It turns out that we can handle both of these situations if we treat ‘¬’ as intuitionistic negation, writing ‘\bot \iff P’ for ‘\neg P’, and apply an appropriate form of circumscription. Thus (10) becomes:

\[ \text{Clear}(y) \iff [\bot \iff \text{Covered}(y)], \] (12)

and we would circumscribe the predicates ‘Above’ in (6)–(7), ‘Covered’ in (8) and ‘Clear’ in (12). But what about the occurrence of the predicate ‘Covered’ in (12)? It turns out that we do not want to circumscribe the occurrence of a predicate in this position, i.e., we want to adopt a form of partial intuitionistic circumscription. Under this interpretation, we can achieve the desired results in Example 2.4 simply by including ‘On’ in the circumscription, thereby adopting the “closed world” assumption — or excluding it, thereby adopting the “open world” assumption — as we wish.

The circumscription of embedded implications and embedded negations is discussed in greater detail in [27], and the suggested interpretation of negation-as-failure using partial intuitionistic circumscription is discussed in [31]. The idea of using intuitionistic negation in logic programming was first suggested in [14]. A full discussion of this form of negation is beyond the scope of the present paper, but we will show how to prove query (11) in Section 3.

3 Inductive Proofs

Let us write the circumscriptive query problem as follows: Circ(\mathcal{R}(P);P) \models \psi. We assume that \psi is a universally quantified implication with a conjunction of atomic formulae in its antecedent and a single atomic formula in
its conclusion, i.e., a first-order Horn clause. But the circumscriptive query problem itself is second-order. How can we solve it?

First, if \( R \) is a set of nonrecursive Horn clauses, then the solution is the same in intuitionistic logic as it is in classical logic [36, 21]. Let \( \text{Comp}(R) \) denote Clark’s Predicate Completion [5]. We have the following result:

**Theorem 3.1:** Let \( R \) be a set of nonrecursive Horn clauses. Then 
\[
\text{Circ}(R(P); P)
\]
is equivalent to \( \text{Comp}(R) \). \( \square \)

Second, for recursive Horn clauses, we need an inductive proof procedure. We restrict our attention, initially, to the following class of recursive Horn clauses:

**Definition 3.2:** \( R \) is a linear recursive definition of the predicate \( A \) if it consists of:

1. A Horn clause with ‘\( A(x) \)’ on the left-hand side and a conjunction of nonrecursive predicates on the right-hand side, and
2. A Horn clause that is linear recursive in \( A \).

\( \square \)

Notice that Examples 2.1 and 2.2 in Section 2 involve only linear recursive definitions. Given such a definition, we adopt the following notation:

- Let ‘\( A(x) \Rightarrow A^0(x) \)’ be the rule obtained from (1) by applying Clark’s Predicate Completion. We say that ‘\( A(x) \Rightarrow A^0(x) \)’ is the prototypical definition of \( A(x) \).
- Let ‘\( X(x) \Rightarrow \Delta X(x) \)’ be the rule obtained from (2) by applying Clark’s Predicate Completion and then replacing the predicate constant \( A \) with the predicate variable \( X \). The implication ‘\( X(x) \Rightarrow \Delta X(x) \)’ then implicitly defines the transformation associated with \( A(x) \).

Intuitively, we think of \( \Delta \) as an operator that takes a relation \( X \) as input and produces the relation \( \Delta X \) as output. We now use these concepts to construct an induction schema for circumscriptive queries.

For example, rules (6)–(7) in Section 2 constitute a linear recursive definition of the predicate ‘Above’, in which

\[
\text{Above}(x, y) \Rightarrow \text{On}(x, y)
\]
is the prototypical definition, and

\[
X(x, y) \Rightarrow (\exists z)[\text{On}(x, z) \land X(z, y)]
\]
defines the transformation. The right-hand side of (13) would be written as ‘\( \text{Above}^0(x, y) \)’, and the right-hand side of (14) would be written as ‘\( \Delta X(x, y) \)’. To see how this is related to induction, imagine that the second-order variable \( X \) initially has the value ‘\( \text{Above}^0 \)’, which is simply the relation ‘On’. Then \( \Delta X(x, y) \) has the value ‘\((\exists z)[\text{On}(x, z) \land \text{On}(z, y)]\)’. We
could then reset the value of $X$ to this new relation, compute a new value for $\Delta X(x, y)$, and so on.

Now let $\Phi(A)$ be any Horn clause in which the predicate constant $A$ appears on the right-hand side. For example:

$$
\Phi(A) \equiv (\forall x) \left[ P(x) \leftarrow A(x) \land \bigwedge_{i=1}^{l} B_i(x) \right]
$$

We treat $\Phi(A)$ as a schema that depends on $A$, so that we are free to substitute $A^0$, $\Delta X$ and $X$ as we wish. Note that the predicates $B_i$ can be either recursive or nonrecursive.

**Definition 3.3:** The induction schema for $\Phi(A)$ is the following sentence in second-order intuitionistic logic:

$$
\Phi(A) \leftarrow \Phi(A^0) \land (\forall X)[\Phi(\Delta X) \leftarrow \Phi(X)].
$$

The interesting point about this induction schema is that it takes the form of an embedded implication with an embedded second-order universal quantifier. Second-order intuitionistic logic has no complete proof procedure, of course, but it turns out that a set of second-order sentences in this form does have a complete proof procedure [28, 34]. The procedure is similar to the first-order proof procedure for universally quantified implications discussed in [26]. It is based on a system of tableaux, each of which has both a database and an initial goal. To prove a first-order universally quantified implication: (1) we replace the universally quantified variables, $y_1, y_2, \ldots$, with a special set of constants, $!y_1, !y_2, \ldots$; (2) we add the antecedent of the implication to the database of a new auxiliary tableau; and (3) we use the consequent of the implication to start a new SLD proof in this tableau. The second-order case works in the same way. To prove the second conjunct on the right-hand side of Definition 3.3, we replace the predicate variable ‘$X$’ with a new predicate constant ‘!$X$’, we assert $\Phi(!X)$ into the database of a new auxiliary tableau, and we try to prove $\Phi(\Delta !X)$ in this tableau. If this proof succeeds, then we have proven the goal: $(\forall X)[\Phi(\Delta X) \leftarrow \Phi(X)]$.

Let us see how this works for Example 2.2. A successful proof of query (9) is shown in Figure 1. Since the query is an implication without quantifiers, we construct an initial tableau, $T_0$, with ‘Above(a,b)’ in its data base, and with ‘Covered(b)’ as its goal. We then try to prove this goal using the prototypical definition in (13). Figure 1 shows the result, which is a successful prototypical proof in the tableau $T_0$. Our task now is to “strengthen” this proof into a proof of the original query by an appropriate use of the induction schema in Definition 3.3. As a first step, we need to generalize the proof in $T_0$ from a proof that works for the constant ‘a’ to a proof that works for the variable ‘$x$’. (See [18] for the analysis of a similar problem in “explanation-based generalization”.) It is easy to see that this generalization is successful.
We now have a proof of the following universally quantified implication:

\( (\forall x)[\text{Covered}(b) \iff \text{On}(x, b)] \). (15)

Let us call this implication \( \Phi(\text{Above}^0) \). Then \( \Phi(\text{Above}) \) is the following universally quantified implication:

\( (\forall x)[\text{Covered}(b) \iff \text{Above}(x, b)] \). (16)

If we can prove (16), we will also have a proof of our original query (9). Therefore, using the induction schema in Definition 3.3, we try to prove \( (\forall X)[\Phi(\Delta X) \iff \Phi(X)] \). This goal is an implication with a second-order universal quantifier, so we create a new tableau, \( T_1 \), we add \( \Phi(!X) \) to the data base, and we try to prove \( \Phi(\Delta!X) \) in \( T_1 \).

Let us write out each of these schemata in detail. \( \Phi(!X) \) is the following implication:

\( (\forall x) [\text{Covered}(b) \iff !X(x, b)] \), (17)

and \( \Phi(\Delta!X) \) is equivalent to the following implication:

\( (\forall x, z) [\text{Covered}(b) \iff \text{On}(x, z) \land !X(z, b)] \). (18)

To prove (18), we instantiate ‘\( x \)’ and ‘\( z \)’ to the special constants ‘\( !x_2 \)’ and ‘\( !z_2 \)’, we add the right-hand side of (18) to the data base of \( T_1 \), and we try to prove the left-hand side of (18). The successful proof is shown in Figure 1. The main point to note is that the proof makes essential use of \( \Phi(!X) \), the hypothesis of the induction schema, since the only way it can succeed is by unifying \( !X(x_3, b) \) with \( !X(!z_2, b) \).
A slight modification of this proof yields a proof of query (11) from Example 2.4. Here we would initially add ‘Above(!x₁, b)’ and ‘Clear(b)’ to the database of $T₀$, and we would attempt to prove ‘⊥’. Since the circumscription of ‘Clear’ in (12) is equivalent to predicate completion, ‘Clear(b)’ implies ‘⊥ ⇐ Covered(b)’. Thus the goal ‘⊥’ in $T₀$ can be reduced to the goal ‘Covered(b)’, and the proof continues exactly as in Figure 1.

A justification of this proof procedure is given in the following two theorems, which are proven in [28]. In the statement of these theorems, $P$ is a tuple consisting of the recursively defined predicates in $R$, which is assumed to include only linear recursive definitions, $P(P)$ denotes the set of prototypical definitions of the predicates in $P$ given by Definition 3.2, and $S(P)$ denotes the set of all induction schemata for the predicates in $P$ that can be constructed using Definition 3.3.

**Theorem 3.4:** $\text{Comp}(R) \cup P(P) \models \psi \iff \text{Circ}(R(P); P) \models \psi$

**Theorem 3.5:** $\text{Comp}(R) \cup S(P) \models \psi \Rightarrow \text{Circ}(R(P); P) \models \psi$

Intuitively, Theorem 3.4 tells us that prototypical proofs are complete but not necessarily sound, while Theorem 3.5 tells us that inductive proofs are sound but not necessarily complete. Together, these theorems sanction the strategy illustrated in Example 2.2: Try to find a prototypical proof first, and then use this proof to suggest a suitable induction schema.

### 4 Example: Naive Reverse

We now analyze Example 2.1. Is ‘Naive Reverse’ a symmetric relation? Specifically, does query (5) follow from the circumscription of ‘Append’ and ‘Reverse’ in (1)–(4)? Obviously, this example is more complex than Example 2.2. In fact, Example 2.2 happens to belong to a decidable class of circumscriptive query problems [29], but Example 2.1 does not. Nevertheless, Example 2.1 can be solved by our inductive proof procedure.

The first part of the proof is shown in Figure 2. We initially construct a tableau, $T₀$, with ‘Reverse(!x₁,!y₁)’ in its data base and ‘Reverse(!y₁,!x₁)’ as its goal. The prototypical definition of ‘Reverse’ is given by Definition 3.2, as before, but its use in the tableau proof is slightly more complicated here. Applying Clark’s Predicate Completion to rule (3) alone, we have:

$$\text{Reverse}(x, y) \Rightarrow x = \emptyset \land y = \emptyset.$$  \hspace{1cm} (19)

Thus ‘$\text{Reverse}^0(!x₁,!y₁)$’ is the assertion that ‘$!x₁ = \emptyset$’ and ‘$!y₁ = \emptyset$’, and when these values are substituted throughout the tableau $T₀$ the goal succeeds immediately, as indicated in Figure 2. We thus have a proof of the following universally quantified implication:

$$\forall x, y |\text{Reverse}(y, x) \Leftrightarrow \text{Reverse}^0(x, y).$$  \hspace{1cm} (20)

Let us call this implication $\Phi₁(\text{Reverse}^0)$. Then the implication in (5), our ultimate goal, is $\Phi₁(\text{Reverse})$.  

The prototypical proof in Figure 2 has suggested an induction schema, and we now compute the expression $(\forall R)[\Phi_1(\Delta R) \iff \Phi_1(R)]$ where $R$ is a predicate variable. We can immediately write:

$$\Phi_1(R) \equiv (\forall x, y)[\text{Reverse}(y, x) \iff R(x, y)].$$ (21)

Also, by Definition 3.2, the transformation associated with ‘Reverse’ is given by:

$$R(x, y) \Rightarrow (\exists q, r, s)$$
$$R(r, s) \land \text{Append}(s, [q], y) \land x = [q \mid r],$$ (22)

and we can therefore write:

$$\Phi_1(\Delta R) \equiv (\forall x, y, q, r, s)$$
$$\text{Reverse}(y, x) \iff R(r, s) \land \text{Append}(s, [q], y) \land x = [q \mid r].$$ (23)

Tableau $T_1$ in Figure 2 shows our attempt to prove the right-hand side of this induction schema. We add $\Phi_1(!R)$ to the data base and we try to prove $\Phi_1(\Delta!R)$. Notice that the equality ‘$x = [q \mid r]$’ in (23) can be eliminated when we attempt this proof.

However, as Figure 2 indicates, this proof does not succeed immediately. Instead, we are able to reduce the goal in tableau $T_1$ to another universally quantified implication:

$$(\forall y, q, r, s)$$
$$[\text{Reverse}(y, [q \mid r]) \iff \text{Reverse}(s, r) \land \text{Append}(s, [q], y)].$$ (24)

We now attempt, in Figure 3, to prove (24). The strategy here is exactly the same: Find a proof using the prototypical definitions $P(P)$, and then try to
“strengthen” this proof into a proof from $S(P)$. The prototypical definition of ‘Append’ is:

$$\text{Append}(x, y, z) \Rightarrow y = z \land x = [].$$ (25)

Thus, to assert ‘$\text{Reverse}^0(|s_2, r_2|)$’ and ‘$\text{Append}^0(|s_2, [q_2], y_2|)$’ is to assert ‘$|s_2 = !r_2 = []$’ and ‘$|y_2 = [q_2]|$’. When these values are substituted throughout the tableau $T_1$, as shown in Figure 3, the goal succeeds. We thus have a proof of the following universally quantified implication:

$$\forall y, q, r, s \quad [\text{Reverse}(y, [q | r]) \iff \text{Reverse}^0(s, r) \land \text{Append}^0(s, [q], y)].$$ (26)

Our task now is to strengthen the proof of (26) into a proof of (24).

Since there are two recursive predicates on the right-hand side of (24), we can expect the construction of an induction schema here to be more complicated than it was in our prior examples. However, it turns out that we can transform the relations ‘Reverse’ and ‘Append’ conjunctively in this case. Suppose we define:

$$\Phi_2(R \land A) \equiv \forall y, q, r, s \quad [\text{Reverse}(y, [q | r]) \iff R(s, r) \land A(s, [q], y)],$$ (27)

where $R$ and $A$ are predicate variables and ‘$R \land A$’ is their conjunction. By Definition 3.2, the transformation for ‘Append’ is:
Figure 4: "Naive Reverse," second inductive proof.

\[ A(x, y, z) \Rightarrow (\exists k, l, n) \]
\[ A(l, y, n) \land x = [k \mid l] \land z = [k \mid n], \]
and combining this with the transformation for ‘Reverse’, we have:
\[ R(s, r) \land A(s, [q], y) \Rightarrow (\exists k, l, n, z) \]
\[ R(l, z) \land \text{Append}(z, [k], r) \land A(l, [q], n) \land \]
\[ s = [k \mid l] \land y = [k \mid n]. \]
Substituting the right-hand side of (29) into the schema \( \Phi_2 \), we have:
\[ \Phi_2(\Delta R \land A) \equiv (\forall y, q, r, k, l, n, z) \]
\[ \text{Reverse}(y, [q \mid r]) \leftarrow \]
\[ R(l, z) \land \text{Append}(z, [k], r) \land A(l, [q], n) \land y = [k \mid n]. \]

Figure 4 now shows that the proof using this induction schema is successful.

How do we know that this use of a conjunctive transformation is correct? Our analysis here is similar to the analysis in [16] and [10]. There are two conditions that have to be checked:

1: When we apply the prototypical definitions of ‘Reverse’ and ‘Append’ to the conjuncts ‘Reverse(s, r)’ and ‘Append(s, [q], y)’, are the instantiations of the variables consistent? In this case, they are, since:
\[ \text{Reverse}^0(s, r) = \text{Reverse}([], []), \]
5 A PROLOG INTERPRETER

\[ \text{Append}^0(s, [q], y) = \text{Append}([], [q], [q]). \]

However, if the conjunction happened to be:

\[ \text{Reverse}(y, r) \land \text{Append}(s, [q], y) \]

then the prototypical instantiations would clash.

2: When we apply the transformations associated with ‘Reverse’ and ‘Append’, are the shared variables transformed identically? In this case, they are, as shown by the equality ‘\( s = [k \mid l] \)’ in (29). However, suppose the conjunction happened to be:

\[ \text{Reverse}(r, s) \land \text{Append}(s, [q], y). \]

In this case, although the prototypical instantiations would be consistent, the transformations of the shared variable \( s \) would clash, and a conjunctive transformation would not be allowed.

Since these two conditions are satisfied in our example, it is easy to see that the conjunction of the relations ‘Reverse\(^0\)(s, r)’ and ‘Append\(^0\)(s, [q], y)’ transformed conjunctively, is equivalent to the single conjunctive relation ‘Reverse(s, r) \land Append(s, [q], y)’.

We have thus shown, by Theorem 3.5, that (5) is entailed by the circumscription of ‘Append’ and ‘Reverse’ in rules (1)–(4).

5 A PROLOG Interpreter

The inductive proof procedure presented in Sections 3–4 is relatively easy to implement, and we analyze the performance of a PROLOG implementation in this section. Although there is a substantial amount of nondeterminism in the choice of inductive lemmas and the selection of appropriate induction schemata, there are also many opportunities to prune unsuccessful branches early. In particular, the simple check to see if a lemma will succeed in the base case turns out to be a powerful pruning technique.

One additional proof method is needed, however. Recall that the formulation of Theorems 3.4 and 3.5 used \( \text{Comp}(\mathcal{R}) \) as well as \( \mathcal{P}(\mathcal{P}) \) and \( \mathcal{S}(\mathcal{P}) \). This is a necessary ingredient in the proof procedure, as the following example indicates.

Example 5.1: Suppose the data type \texttt{num} is defined by:

\begin{align*}
\text{num}(0). \\
\text{num}(\text{succ}(N)) & : \text{num}(N).
\end{align*}

We would expect an interpreter for circumscriptive queries to treat the following query as true:

\[ \text{\texttt{\mid ?- forall([X], (num(X) :- num(s(X))))).}} \]
In fact, our interpreter responds: yes to this query, but it gets the correct answer not by induction but by predicate completion. Since num(s(X)) cannot be equivalent to num(0), it must be equivalent to num(s(N)) and thus num(X) must be true. In practice, it is often necessary to reduce circumscriptive queries in this way, and our interpreter always tries to apply predicate completion to a query before it tries to construct an inductive proof.

It turns out that ‘Naive Reverse’ is a very simple problem for our interpreter. The inductive lemma proposed in tableau $T_1$ is the only possibility at this point, and the proof procedure immediately considers the joint transformation of ‘Reverse’ and ‘Append’. The successful proof in tableau $T_2$ is then constructed without backtracking. The following examples are more complex.

**Example 5.2:** Let num be defined as in Example 5.1 and define pluss as follows:

\[
\text{pluss}(0, I, I) :- \text{num}(I).
\]
\[
\text{pluss}(s(I), J, s(K)) :- \text{pluss}(I, J, K).
\]

We would like to show that pluss is associative:

\[
| ?- \forall([X, Y, Z, V, W], (\text{pluss}(Y, Z, V_1), \text{pluss}(X, V_1, W) :- \\
\text{pluss}(X, Y, V), \text{pluss}(V, Z, W))).
\]

By convention, we interpret $V_1$ to be an existential variable with narrow scope. Although this query does not fit within our definition of $\psi$, i.e., it is not a Horn clause, it can easily be accommodated within our framework by the use of an auxiliary definition, as in Example 2.2.

The interpreter first considers the application of the prototypical definition of pluss to $[\text{pluss}(X, Y, V), \text{pluss}(V, Z, W)]$ jointly, but rejects this because it fails condition 1 at the end of Section 4. It then considers the prototypical definition applied to $[\text{pluss}(X, Y, V)]$ alone. This leads to the following problem:

\[
\forall([A, B, C], ([\text{pluss}(A, B, _6830), \text{pluss}(0, _6830, C)] :- \\
\text{num}(A), \text{pluss}(A, B, C))).
\]

(The symbols A, B, C, etc., are Quintus PROLOG “frozen” variables.) This query does not succeed immediately, since it is not possible to prove num(C) from the original definitions. The interpreter thus tries to prove by induction:

\[
\forall([A, B, C], ([\text{num}(C)] :- [\text{num}(A), \text{pluss}(A, B, C)]))
\]

This proof is successful, although we omit the details. The interpreter has now succeeded in constructing a prototypical proof for the initial query.

Next, the interpreter tries to prove the right-hand side of the induction schema suggested by the prototypical proof. This generates the following expression for $\Phi(\{X\})$:

\[
[\text{pluss}(14254, 14164, A), \text{pluss}(14253, A, 14165)] :- \\
[\text{xxx('!3', -14253, -14254, 14163), pluss(-14163, -14164, -14165)]
\]

and the following expression for $\Phi(\Delta X)$:
forall([A,B,C,D,E,F],
   ([pluss(C,E,_6859),pluss(s(B),_6859,F)] :-
    [xxx('!3',B,C,D),pluss(s(D),E,F)]))

(The frozen variable A in Φ(!X) is a skolem constant.) The next step is to
unify the ‘xxx’ predicates in these two expressions, and thus reduce Φ(!X)
to the following simpler form:

[pluss(C,_14164,A),pluss(B,A,_14165)] :-
[pluss(D,_14164,_14165)]

The interpreter then applies predicate completion to the right-hand side of
Φ(Δ!X), to reduce the term s(D) to D, and unifies Φ(!X) with Φ(Δ!X) again
to generate the following problem:

forall([A,B,C,D,E,F],
   ([pluss(C,E,_66743),pluss(s(B),_66743,s(F))] :-
    [pluss(C,E,A),pluss(B,A,F)]))

But this query succeeds immediately using the original definitions, and the
interpreter responds: yes.

Example 5.3: Let num and pluss be defined as in Examples 5.1 and 5.2,
and define timess as follows:

```
times(0,I,0) :- num(I).
times(s(I),J,K) :- times(I,J,M), pluss(J,M,K).
```

We would like to show that timess is commutative:

```
| ?- forall([X,Y,Z], (times(Y,X,Z) :- times(X,Y,Z))).
```

The proof for this query is longer than the proof in Example 5.2, and we
will analyze it using a coarser mesh.

Since there is only one choice for the application of a prototypical defi-
nition here, and since the prototypical proof for this definition succeeds,
the interpreter quickly attempts to prove the right-hand side of the correspond-
ing induction schema. This leads to the following inductive lemma:

forall([A,B,C,D,E],([pluss(C,E,A),timess(B,C,D)] :-
    [pluss(C,E,D),timess(B,C,E)]))

However, this lemma is rejected immediately because it fails in the base case.
(Note that there is no way to prove num(A)).
The interpreter then considers the alternative, applying the prototypical definition of $\text{times}$ to $\text{times}(B,A,C)$ alone, and the new prototypical proof succeeds. The corresponding induction schema eventually leads to the following lemma:

$$\forall[A,B,C,D,E], $$ $$\begin{array}{l}
(\text{pluss}(A,_{67553},E),\text{pluss}(C,B,_{67553}) :- \\
\text{pluss}(A,B,D),\text{pluss}(C,D,E))
\end{array}$$

This lemma is similar to Example 5.2, but more complex. To prove it by hand, we would need one application of associativity and two applications of commutativity. However, our interpreter does not have access to the commutative and associative lemmas for $\text{pluss}$, and it must solve the problem directly. Omitting the failed branches (of which there are several), the interpreter eventually reduces the problem to the following lemma:

$$\forall[A,B,C,D,E], $$ $$\begin{array}{l}
(\text{pluss}(D,_{100440},s(C)),\text{pluss}(s(B),E,_{100440}) :- \\
\text{pluss}(D,A,C),\text{pluss}(B,E,A))
\end{array}$$

This lemma is much simpler, but still not provable from the original definitions. However, after another inductive step, the interpreter arrives at the following problem:

$$\text{pluss}(s(C),_{108471},s(s(D))),\text{pluss}(s(E),F,_{108471}) :- $$ $$\text{pluss}(C,A,s(D)),\text{pluss}(s(E),F,A)$$

Since this implication is directly provable, the interpreter now responds: yes.

The interpreter described here has been optimized to prove properties of PROLOG programs. (For example, it assumes that all the predicates in the domain are circumscribed.) It has been tested on the simple list examples and arithmetical examples from Boyer and Moore [1], translated into Horn clauses, and it performs surprisingly well. A slightly different version of the interpreter has been tested on several common-sense reasoning examples, such as the “Stack of Christmas Blocks” problem in [29] and the “Mermaids Get the Sack” problem in [30]. A future paper will discuss the interpreter and its search strategy in greater detail, and compare its performance to the performance of several other inductive theorem-provers in the literature.

6 Conclusion

Previous work by Kanamori and Seki [17] and Kanamori and Fujita [16], extended and refined by Fribourg [10, 11, 12], has demonstrated the advantages of the logic programming paradigm for the automatic construction of inductive proofs. In this paper, we have formulated the induction problem for logic programs in second-order intuitionistic logic, and further substantiated the power of this approach. As an illustration, we have implemented our proof procedure in a PROLOG program and shown that it is able to
solve a number of traditional inductive problems, such as the symmetry of ‘Reverse’, the associativity of ‘Plus’ and the commutativity of ‘Times’.

There are several questions left open for future research:

1. Although we presented an interpretation of negation-as-failure in Section 2, we did not present a serious example of an inductive proof that uses this concept. However, our basic techniques should also be capable of proving properties of PROLOG programs with negation. Are any additional proof methods needed to handle these cases?

2. Although many standard inductive problems can be formulated using linear recursive definitions, this class of Horn clauses is too restrictive for some purposes. It is straightforward to extend our approach to mutually recursive definitions, while retaining the linearity assumption. Can we also handle general nonlinear recursion? We expect to find guidance here in the literature on deductive databases, where similar issues have been addressed.

3. One of the main applications of inductive proofs is in the area of program validation and synthesis, a topic that has been investigated extensively within the logic programming commmunity. Can our techniques for constructing induction schemata also be applied in this area?

We will pursue these questions in subsequent papers.

References


REFERENCES


REFERENCES


