A trusted implementation of SLD-resolution

Konstantine Arkoudas

MIT Computer Science and AI Lab
arkoudas@csail.mit.edu

Abstract. Prolog’s implementation of SLD-resolution furnishes an efficient theorem-proving technique for the Horn-clause subset of first-order logic, and makes for a powerful addition to any automatic or semi-automatic verification system. However, due to the complexity of SLD-resolution, a naive incorporation of a Prolog engine into such a system would inordinately increase the overall trusted base. In this paper we show how to integrate this procedure in a disciplined, trusted manner, by making the Prolog engine justify its results with very simple natural deduction reasoning. In effect, instead of taking SLD-resolution as a primitive inference rule, we express it as a derived inference rule in terms of much simpler rules such as conditional elimination.

This reduction is an example of a general methodology for building highly reliable software systems called certified computation, whereby a program not only produces a result r for a given input x but also proves that r is correct for x. Such a proof can be viewed as a certificate for the result r, and can significantly enhance the latter’s credibility: if we trust the axioms and inference rules used in the proof, we can trust the result. We present a complete implementation of a certifying Prolog interpreter that relies only on three exceptionally simple inference rules: conditional elimination, universal specialization, and conjunction introduction.

1 Introduction

Prolog’s implementation [9, 20, 22] of the SLD-resolution technique of logic programming [15, 14, 16, 19, 2] provides an efficient theorem-proving tool for the Horn-clause subset of first-order logic, and makes for a powerful addition to any verification system. But Prolog’s execution model is too complex to be taken as a trusted primitive. A naive incorporation of a Prolog engine would greatly increase the system’s trusted base: we would need to trust the engine’s implementation of unification, backtracking, substitution data structures and operations, and a good deal of other non-trivial code. Here we show how to implement SLD-resolution in a trusted manner, by making the Prolog engine justify its results using very simple natural deduction reasoning. In effect, instead of taking SLD-resolution as a primitive inference rule, we show how to express it as a derived inference rule in terms of much simpler rules.

It is an interesting question to ask exactly how simpler those rules can be. Just as in foundational PCC one asks “what is the minimum possible size of the components that must be trusted” [1], we ask what is the minimum and
simplest set of inference rules that must be trusted in order for one to believe the results of SLD-resolution. We provide a precise answer to this question by implementing SLD-resolution as a tactic that proves its results using only three exceptionally simple inference rules:

**Conditional elimination:** From $P \Rightarrow Q$ and $P$ infer $Q$.

**Conjunction introduction:** From $P$ and $Q$ infer $P \land Q$.

**Universal specialization:** From $(\forall x) P$ infer $P[t/x]$.

Our work is thus in the vein of recent research on foundational PCC [1] and certified computation [5] that seeks to reduce the trusted base of a software system to a minimum in order to increase the credibility of the results produced by the system. Specifically, in certified computation an algorithm is expressed as a proof-search procedure that takes an input $x$ and not only produces a result $r$ but also proves a theorem stating that $r$ is a correct output for $x$. We refer to the completed proof as a *certificate* for the result $r$. The key idea is that if we trust the axioms and inference rules used in the certificate, we can trust the result. The certificate itself can be output as a side effect of the derivation and then checked independently by a separate proof checker. In this case, the certificates generated by our implementation of SLD-resolution use only the three inference rules mentioned above and can therefore be verified by extremely simple proof checkers. Accordingly, we have reduced our trusted base from (the implementation of) a large collection of sophisticated control and data structures to (the implementation of) a very rudimentary proof checker.

Our implementation is given in Athena, a denotational proof language (DPL [3]) for first-order logic with sorts and polymorphism. Athena is an interactive theorem proving system that includes a higher-order functional language in the style of Scheme and ML. It uses a Fitch-style block-structured natural deduction system that makes it easier to read and write proofs and proof-search algorithms, and offers a high degree of programmability through the use of *methods*, which are akin to the tactics and tacticals of HOL [4] and Isabelle [14]. Athena has a formal semantics based on the abstraction of *assumption bases* that offers a strong soundness guarantee; it features a flexible polymorphic sort system with built-in support for structural induction; and it is seamlessly integrated with high-performance first-order ATPs such as Vampire [23] and Spass [25], as well as with model builders such as Paradox [8]. Athena’s assumption-base semantics and powerful programming constructs such as first-order proof continuations, non-linear pattern matching, built-in deductive backtracking mechanisms, and the combination of updateable cells, lexical scoping and higher-order methods and functions allow the user to express very complex proof algorithms as trusted derived inference rules in a succinct and fluid style (e.g., see [4]). Our entire implementation of the certifying version of SLD-resolution takes about one page of code. The code is available online at <www.cog.uci.edu/~kostas/dpl/dlvs/athena/prolog>.

Even though our implementation is given in Athena, we first present a high-level sketch of it in pseudocode. Therefore, with some suitable adjustments, this
A trusted implementation of SLD-resolution

The technique could be implemented in any programmable theorem-proving system that has a rigorous notion of proof and proof validation. In fact our technique could be used not only by a theorem prover, but by any Prolog implementation (interpreter or compiler) that needs to be exceptionally reliable.

In what follows, Horn clauses will be called rules (or sometimes axioms), and will be represented by Athena propositions of the form

\[(\forall x_1) \cdots (\forall x_n) [P_1(x_1, \ldots, x_n) \land \cdots \land P_k(x_1, \ldots, x_n) \Rightarrow R(x_1, \ldots, x_n)]\]  

for \(n \geq 0, k \geq 1\), where \(R(x_1, \ldots, x_n)\) and each \(P_i(x_1, \ldots, x_n)\) are atomic propositions that may have free occurrences of \(x_1, \ldots, x_n\); no variables outside \(x_1, \ldots, x_n\) may occur in these atoms. We will refer to the atoms

\[P_1(x_1, \ldots, x_n), \ldots, P_k(x_1, \ldots, x_n)\]  

as the antecedents of the rule, and to the atom \(R(x_1, \ldots, x_n)\) as its conclusion. The conjunction of the antecedents, \(P_1(x_1, \ldots, x_n) \land \cdots \land P_k(x_1, \ldots, x_n)\), will be called the body of the rule. By a fact we will mean a rule whose antecedent is the sole atom true. A query is simply an atomic proposition. For any rule \(\mathcal{R}\) of the form (1), we define \(\text{Ant}(\mathcal{R})\) as the list of its antecedents (given in the order of (2)). It will be convenient to stipulate that \(\text{Ant}(\mathcal{R}) = []\) whenever \(\mathcal{R}\) is a fact. The symbol \(\oplus\) will denote the binary operation of list concatenation.

We can now give a partial formulation the problem as follows (a more precise problem statement will be given in Section 3): define a binary tactic \texttt{solve} that takes a list of rules \([\mathcal{R}_1, \ldots, \mathcal{R}_n]\) and a list of queries \([Q_1, \ldots, Q_m]\) and produces a theorem of the form \(Q'_1 \land \cdots \land Q'_m\) that is a ground instance of the proposition \(Q_1 \land \cdots \land Q_m\); in other words, such that

\[\theta(Q_1 \land \cdots \land Q_m) = Q'_1 \land \cdots \land Q'_m\]

for some substitution \(\theta\) that maps the free variables of \(Q_1, \ldots, Q_m\) to ground terms. (Of course \texttt{solve} may also fail or get into an infinite loop.) Thus the second argument to \texttt{solve} represents our queries and the first argument represents the "logic program"—a list of rules \([\mathcal{R}_1, \ldots, \mathcal{R}_n]\) of the form described above. These rules should be in the assumption base at the time when \texttt{solve} is invoked. The order in which the rules appear in the given list is important. The rules should be given in the order in which they would be listed in a Prolog program. This might affect termination. Further, \texttt{solve} should not use any axioms other than the given rules and no primitive methods other than simple introduction and elimination rules for the propositional connectives and quantifiers.

2 SLD trees

By a computation rule we will understand any unary computable function \(C\) that takes an arbitrary non-empty list of queries \(L\) and produces a triple \(\langle L_1, G, L_2 \rangle\) consisting of a query list \(L_1\), a single query \(G\), and another query list \(L_2\) such
that \( L_1 \oplus [G] \oplus L_2 = L \). Hence, the triple \((L_1, G, L_2)\) represents a decomposition of \( L \) into a prefix \( L_1 \), a selected goal \( G \), and a suffix \( L_2 \). Every deterministic interpreter of logic programs must fix a computation rule. Prolog makes a particularly simple choice: the selected goal is always the first element of the given query list. Therefore, using our definitions, for any given \( L = [G_1, \ldots, G_k] \), \( k > 0 \), Prolog’s computation rule returns the triple \( ([], G_1, [G_2, \ldots, G_k]) \).

For the remainder of this section fix a set of function symbols \( \mathcal{F} \) and a disjoint set of relation symbols \( \mathcal{R} \), with each \( f \in \mathcal{F} \) and \( p \in \mathcal{R} \) having a unique non-negative arity, and a countably infinite set of variables \( \mathcal{V} \), disjoint from \( \mathcal{F} \) and \( \mathcal{R} \). We will assume that \( \mathcal{V} \) is totally ordered by some computable binary relation \( < \), and we will use \( v \) as a metavariable ranging over \( \mathcal{V} \). The Herbrand universe \( \text{Terms}(\mathcal{F}, \mathcal{V}) \) of terms built over \( \mathcal{F} \) and \( \mathcal{V} \) is defined as usual. A substitution \( \theta \) is defined as any function from \( \mathcal{V} \) to \( \text{Terms}(\mathcal{F}, \mathcal{V}) \) that is the identity almost everywhere, except for some finite subset of \( \mathcal{V} \) known as the “support” of \( \theta \). A substitution with support \( \{x_1, \ldots, x_n\} \) that maps each \( x_i \) to \( t_i \) is often written as \( \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \). Every substitution \( \theta : \mathcal{V} \to \text{Terms}(\mathcal{F}, \mathcal{V}) \) has a unique homomorphic extension \( \overline{\theta} : \text{Terms}(\mathcal{F}, \mathcal{V}) \to \text{Terms}(\mathcal{F}, \mathcal{V}) \). For a list of terms \( L = [t_1, \ldots, t_n] \), we define \( \overline{\theta}(L) = [\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n)] \). A binary composition operation \( \circ \) on substitutions is defined as

\[
\sigma \circ \theta = \lambda v \in \mathcal{V}. \overline{\sigma}(\overline{\theta}(v)).
\]

It is well-known that the set of all substitutions forms a monoid under \( \circ \) \([10, 7, 24]\).

We say that \( n \) terms \( t_1, \ldots, t_n \) are unifiable if there is a substitution \( \theta \) such that \( \overline{\theta}(t_1) = \cdots = \overline{\theta}(t_n) \). There are efficient unification algorithms \([17, 21, 18, 13]\) that take any finite number of terms and produce the most general possible substitution that unifies them, if the terms are unifiable at all. The substitutions returned by such algorithms are called mgu’s (“most general unifiers”) and have a number of nice properties such as idempotence.

With variables, terms, and relation symbols at our disposal, propositions are defined as usual—we have atoms, propositional combinations, and quantifications. The application of a substitution \( \theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \) to a proposition \( P \), which we will denote simply as \( \theta P \), is also defined as usual: every free occurrence of \( x_i \) within \( P \) is replaced by \( t_i \).\footnote{In general we might first have to rename \( P \) via alphabetic conversion to avoid variable capture, but this will not be an issue here.} Finally, let \( V' \) be a finite subset of \( \mathcal{V} \) and let \( P \) be a proposition of the form \( (\forall x_1) \cdots (\forall x_k) Q \), \( k \geq 0 \), where \( Q \) contains no variables outside \( x_1, \ldots, x_k \). Setting \( \theta = \{x_1 \mapsto z_1, \ldots, x_k \mapsto z_k\} \), where \( z_i \) is the least variable (according to the ordering \( < \)) outside the set \( \{x_1, \ldots, x_k\} \cup V' \cup \{z_1, \ldots, z_{i-1}\} \), for \( i = 1, \ldots, k \), we define the \( V' \)-instance of \( P \) as the proposition

\[
(\forall z_1) \cdots (\forall z_k) \theta Q.
\]

Thus the \( V' \)-instance of \( P \) is simply a (uniquely defined) “freshly renamed” copy of \( P \), with the fresh variables being taken outside \( V' \) and of course also outside the variables that occur in \( P \).
Furthermore, fix a computation rule $C$ and let a logic program $P$ be given as a list of $n > 0$ rules $R_1, \ldots, R_n$, let $V_P$ be the set of all and only those variables that occur in some rule $R_i$, and consider a list of goals $L$. We define the SLD-tree of $L$, denoted $SLDT(L)$, as explained below. The nodes of the tree will be lists of goals, with $L$ at the root. Every edge in the tree will be decorated with a pair of the form $(\bar{R}_i, \theta)$, comprising an instance $\bar{R}_i$ of $R_i$, for some $i \in \{1, \ldots, n\}$, and a substitution $\theta$. For any node $M$ in the tree, we will write $\text{Var}(M)$ for the set of all and only those variables that occur in the unique path leading from the root to $M$. More precisely, $SLDT(L)$ is defined as follows:

1. The root of $SLDT(L)$ is $L$.
2. Suppose that a node of $SLDT(L)$ is a list of goals $M$. If $M$ is empty, then the node is a leaf—there are no children. Otherwise, let $C(M) = (M_1, G, M_2)$, and suppose that there is an $i \in \{1, \ldots, n\}$ such that the selected goal $G$ unifies with the conclusion of $\bar{R}_i$ under some mgu $\theta$, where $\bar{R}_i$ is the $(V_P \cup \text{Var}(M))$-instance of $R_i$. Then $M$ has a child $M' = \overline{\overline{M}_1 \oplus \text{Ant}(\bar{R}_i) \oplus M_2}$, joined to $M$ by an edge labeled with the pair $(\bar{R}_i, \theta)$. We refer to $\bar{R}_i$ and $\theta$ as the resolving rule and substitution of $M'$; $i$ is called the index of $M'$.

We will assume that the children of every node are totally ordered in accordance with their indices: a node $L_1$ precedes one of its siblings $L_2$ iff the index of $L_1$ is strictly smaller than that of $L_2$. Since an index $i$ indicates that the child was obtained by resolving the selected goal of its parent list with an instance of the $i^{th}$ rule in the program, this ordering entails that a depth-first traversal of $SLDT(L)$ examines the given rules from top to bottom. In tandem with a computation rule that always selects the first goal in a list of queries, this means that a depth-first search of $SLDT(L)$ gives us the operational semantics of Prolog.

There are two kinds of finite branches in a SLD-tree: success branches, ending in leaves that contain the empty list of goals; and failure branches, ending in leaves that contain non-empty goal lists whose selected goal cannot be resolved with any rule. There may be infinite branches as well.

3 Proof search

Given a list of queries $L$, a Prolog interpreter performs a depth-first search of $SLDT(L)$. If the search does not diverge, there are two possibilities: if there is a success branch, the result is the composition of all the resolving substitutions found along the edges of the leftmost success branch, with the composition proceeding from top to bottom; if there is no success branch, then failure is reported. Of course the search might diverge if there is an infinite path to the left of the leftmost success branch.

A precise formulation of the problem can now be given thus: define a binary method $\text{solve}$ that takes a logic program $P$ (represented as a list of Athena Horn

\[2\text{ Where a variable is understood to occur in a path if it occurs either in a node (a list of goals) on the path, or in one of the objects attached to an edge along the path.} \]
clauses in the manner described earlier) and a list of goals $L = [G_1, \ldots, G_k]$ (represented as a list of Athena atoms, as discussed earlier), and behaves as follows. If a depth-first search of $SLDT(L)$ diverges before discovering any success branches, solve should also diverge. Otherwise:

- if there is a success branch, then the theorem produced by solve should be $\theta(G_1 \land \cdots \land G_k)$, where $\theta$ is the composition of all substitutions found along the leftmost success branch; else
- if there are no success branches, solve should fail.

An $SLD$-derivation of a goal list $L$ from $\mathcal{P}$ is defined essentially as a success branch of $SLDT(L)$: a sequence of $n > 1$ triples

$$\langle L_1, P_1, \theta_1 \rangle, \ldots, \langle L_i, P_i, \theta_i \rangle, \ldots, \langle L_n, P_n, \theta_n \rangle$$

such that $L_1 = L$; for each $j = 1, \ldots, n-1$, $L_j$ is a child of $L_j$ in the SLD-tree of $L$, joined to its parent by an edge decorated with the resolving rule $P_j$ and substitution $\theta_j$; and the child of $L_n$ in $SLDT(L)$ is the empty list, with the edge leading from $L_n$ to $[]$ decorated with $P_n$ and $\theta_n$.

The term “derivation” is somewhat of a misnomer in the above context, as an SLD-derivation is not what one commonly understands as a “derivation” or “deduction”. For one thing, the result of an SLD-derivation is not a meaningful proposition but rather the empty list of goals $[]$. Moreover, even if we understand the triples of (3) as propositions of sorts, the SLD-resolution “inference rule” that takes us from each triple to the next is too coarse. The point of a deduction is to spell out why a conclusion follows from certain premises in a piecemeal manner that uses small steps that are easy to follow and verify—steps taken by simple rules such as modus ponens or universal specialization. Although some calculi must by necessity have more involved primitive inference rules, the guiding principle is always to make those rules as simple as possible. Apart from conceptual clarity, this has practical ramifications with respect to trust.

In the case of Prolog computation, we can achieve this by traversing the leftmost success branch of the SLD-tree from the bottom up and putting more and more intermediate conclusions together using simple inference rules such as conditional elimination, until we reach the top, at which point we deduce the appropriate ground instance of the conjunction of all the given goals. Ultimately, our proof will use nothing but conditional elimination, universal instantiation, and conjunction introduction. Therefore, if we trust the implementation of these inference rules, we can rest assured that the produced theorem is correct. This is a quite remarkable trust reduction, given the complexity of the proof search.

Our certificates will be quite similar to what Hodges calls “simple proofs” in his monograph on Horn clauses [11]. These certificates are intimately related to the least fixed point semantics of logic programs [16]. As Hodges points out, they do not only describe the stages in which the various atoms are “collected up”, but “also the process by which each of them [each atom] is collected”. Hodges defines his “calculus of simple proofs” from scratch and so he then needs to do extra work to prove its soundness. By contrast, we simply put to use the natural
deduction framework provided by Athena; our certificates are simply Athena proofs that use only the aforementioned primitive methods. Accordingly, we do not have to prove soundness separately.

4 Proof construction

Suppose that our initial list of queries is \( L = [G_1, \ldots, G_k] \), and that a depth-first search on the SLD-tree of this list eventually discovers a success branch that yields a substitution \( \theta \) as the final result. Since SLD-resolution is sound, this means that the conjunction \( \theta(G_1 \land \cdots \land G_k) \) follows logically from the given Horn clauses—provided our implementation is correct.

How can we prove that the result \( \theta(G_1 \land \cdots \land G_k) \) follows from the program’s axioms using simple natural deduction reasoning only? The answer becomes straightforward once we study the structure of SLD-trees. We will traverse the success branch backwards, visiting every edge along the way from the leaf to the root. Writing \([\text{US}], [\text{MP}], \) and \([\land\text{-I}]\) as abbreviations for universal specialization, conditional elimination (modus ponens), and conjunction introduction, here is what we do for each such edge connecting a parent \( M \) to a child \( N \):

1. Invariant: every atom in the list \( \theta(N) \) has already been proven.
2. Let \( R = (\forall v_1) \cdots (\forall v_m)P \), \( m \geq 0 \), be the resolving rule attached to the current edge. Perform \( m \) successive applications of \([\text{US}]\) on \( R \), instantiating \( v_1 \) with \( \theta(v_1), \ldots, v_m \) with \( \theta(v_m) \). The resulting theorem will be either of the form \( \text{true} \Rightarrow B \) or of the form

\[
A_1 \land \cdots \land A_l \Rightarrow B. \tag{4}
\]

In the first case, apply \([\text{MP}]\) on \( \text{true} \Rightarrow B \) and \( \text{true} \) to obtain \( B \). In the second case we claim that, inductively, the atoms \( A_1, \ldots, A_l \) have already been proven previously. This follows from the foregoing invariant because, by the way SLD-trees are constructed, the atoms \( A_1, \ldots, A_l \) are members of \( \theta(N) \). Accordingly, use \([\land\text{-I}]\) to obtain the conjunction \( A_1 \land \cdots \land A_l \), and then use \([\text{MP}]\) on (4) and the said conjunction to obtain \( B \).

3. Invariant: every atom in the list \( \theta(M) \) has now been proven.
4. Continue with the next edge, if there is one.

The second invariant follows from the first, because, with the exception of \( B \), every goal \( H_i \) in \( \theta(M) \) is also in \( \theta(N) \), so the first invariant guarantees that \( H_i \) has already been proven. Hence, once we also establish \( B \) via modus ponens, we will have proven every member of \( \theta(M) \). Graphically:

\[
\begin{align*}
\theta(M) &= [B, H_1, \ldots, H_p] \\
\theta(N) &= [A_1, \ldots, A_l, H_1, \ldots, H_p]
\end{align*}
\]

Specialized resolving rule: \( A_1 \land \cdots \land A_l \Rightarrow B \)
To start things off, the top invariant is vacuously true at the beginning of the algorithm. Upon conclusion, the bottom invariant guarantees that every atom in $\theta(L) = [\theta G_1, \ldots, \theta G_k]$ has been deduced, so at that point we can simply use conjunction introduction to infer the desired

$$\theta G_1 \land \cdots \land \theta G_k = \theta (G_1 \land \cdots \land G_k).$$

As an example, suppose that our first-order language comprises two unary function symbols $f$ and $g$, one constant symbol $a$, and three unary relation symbols $p$, $q$, and $r$. Moreover, suppose that our set of variables is $\mathcal{V} = \{x, y, z, x_1, x_2, x_3, x_4, \ldots\}$, ordered as listed. Now consider the following logic program:

1. $\text{true} \Rightarrow p(f(a))$
2. $(\forall x) [p(x) \Rightarrow q(x)]$
3. $(\forall x) [p(x) \land q(x) \Rightarrow r(x)]$
4. $(\forall x) [r(f(x)) \Rightarrow r(g(x))]$
Fig. 2. A conventional proof tree deriving the atom \( r(g(a)) \).

Assuming Prolog’s computation rule, the SLD-tree of the query \( r(g(z)) \) is shown in Figure 1. The leftmost branch is a failure branch, while the second branch is a success branch, resulting in the theorem \( r(g(a)) \), obtained by applying the computed substitution to the original goal \( r(g(z)) \). Using the preceding algorithm, we can prove this result as shown in Figure 2. The deduction of Figure 2 is depicted in classic “proof tree” style, where a leaf represents a premise and an interior node represents an intermediate lemma, obtained through the application of some \( n \)-ary inference rule to the \( n \) children of the node. The proposition at the root represents the conclusion of the entire deduction. Note that the leaves of the tree in Figure 2 consist only of program axioms, while the only inference rules used at interior nodes are \([\text{US}]\), \([\text{MP}]\), and \([\land\text{-I}]\).3

5 Athena implementation

We will now use the algorithm of the preceding section to implement the Athena method \texttt{solve} that was specified earlier. Although the algorithm can be implemented in any language so long as it eventually produces a certificate that can be understood by some proof checker with a rigorous notion of proof, the use of Athena carries a number of significant advantages.

---

3 To save space, the axiom \( \text{true} \Rightarrow p(f(a)) \) was written simply as \( p(f(a)) \).
First, the implementation effort is greatly facilitated by the assumption-base semantics of Athena, in tandem with high-level features such as sophisticated pattern matching, deduction backtracking via the try mechanism, first-class proof continuations, anonymous methods, and the combination of state with lexical scoping and higher-order methods and functions. We also leverage Athena's built-in support for terms and propositions, including substitutions and unification. As a result, the entire implementation takes less than one page of code. Moreover, if the application requires it, it is possible to decouple the construction of the certificate from its validation without writing any additional code. Athena can be instructed to produce a certificate as a side effect of evaluating any method call. That certificate could then be manipulated as necessary, e.g., shipped off to some other computer over the network and checked by a remote Athena interpreter. Because the certificate is so simple, that Athena interpreter need not understand the full language; it only needs to understand introduction and elimination rules and proof composition. Such an interpreter can be implemented in a couple of pages of code and thus constitutes a minimal-trusted-base proof checker.

The Athena implementation appears in its entirety in Figure 3. The reader who is unfamiliar with the details of the language can consult the appendix for a brief explanation of the most pertinent parts. (A more thorough presentation of Athena's syntax and semantics can be found in Teodoro Arvizó's thesis “A virtual machine for a type-ω denotational proof language” [6].) The main method in Figure 3 is solve, as specified earlier, aided by two auxiliary methods try-matches and resolve, and two functions match-conclusion-of and get-matches.

The function match-conclusion-of tries to match an atomic goal G with the conclusion of some rule (“axiom”) P. First a freshly renamed version P' of P is obtained, and then the conclusion of P' is unified with the goal G. If the unification successfully produces some mgu θ, then the pair [P', θ] is returned. Thus the result of match-conclusion-of comprises the two pieces of data that appear on an SLD-tree edge: a resolving rule and a resolving substitution. We will refer to the pair [P', θ] as a match. If the unification fails, then match-conclusion-of returns false. The function get-matches returns a list of all the matches between an atomic goal and a list of rules, as determined by match-conclusion-of. This list could be empty if the goal does not match the conclusion of any of the rules, which would cause the prover to fail. If the rule list constitutes a logic program, then the left-to-right processing ensures that we examine the rules in the given order.

The method try-matches takes a list of matches of the form [[P_1, θ_1], ..., [P_n, θ_n]] and a binary method process-match, which we will refer to as a match handler, and sequentially applies process-match to P_i and θ_i, for i = 1, ..., n, until such an application succeeds or until there are no more matches, in which case a failure occurs.

The core of solve is the internal method prove, which performs the analysis (search) of the SLD-tree. This can be thought of as the method that we invoke at
(define (match-conclusion-of axiom goal)
  (let ((remnamed-axiom (rename axiom)))
    (match remnamed-axiom
      ((forall (some-list vars) (if _ concl))
       (match (unify goal concl)
         (((some-sub theta) [renamed-axiom theta])
           (_ false)))
       (_ false))))
  (define (get-matches goal axioms)
    (letrec ((search (function axioms results))
      (match axioms
        (if (res (match-conclusion-of axiom goal))
          (match res
            (false (search more results))
            (_ (search more (add res results))))))))
  (define (try-matches matches process-match)
    (match matches
      ((list-of [axiom unifying-sub] rest-matches)
       (try (process-match axiom unifying-sub)
         (try-matches rest-matches process-match)))))
  (define (resolve P usub goals prove theta M)
    (match P
      (forall (some-list vars) (if (and (some-list ants)) _))
      (let ((new-goals (usub (join ants goals))))
        (prove new-goals (compose-sub usub theta)
          (method (sub) (begin [imp (compose-list P [sub vars]) (conj-intro (sub ants))] [(M sub)]))))))
  (define (solve axioms queries)
    (letrec ((prove (method (goals theta M))
      (match goals
        (if (IM theta))
        ((list-of goal more-goals)
         (try-matches (get-matches goal axioms)
           (method (P usub)
             (prove queries {} (method theta) (conj-intro (theta queries)))))))))
      (prove queries {} (method theta) (conj-intro (theta queries))))))

Fig. 3. Athena implementation of a certifying Prolog interpreter.

A given node $M$ of the SLD-tree in order to expand the search one level deeper. It takes three arguments:

1. the goal list of $M$;
2. the current substitution $\theta$ (this is the composition of all the substitutions above $M$, i.e., all the substitutions that can be found on the path leading from the root of the SLD-tree to $M$); and
3. a proof continuation, represented as a unary method $M$, that will be used for the synthetic task of putting the proof together. This continuation is increasingly stacked as we move down the tree, and is finally unwound once we reach a successful leaf node (i.e., an empty goal list), at which point it is invoked with the final substitution as its argument.
If the first argument given to prove is the empty goal list, then we have reached
the end of a success branch and we simply invoke the proof continuation with the
current substitution, which, at this point, is the final computed substitution—the
composition of all the substitutions along the path from the root to the present leaf. Otherwise we select the first goal goal from the list (in accordance
with Prolog’s computation rule), and we call try-matches on (a) all the matches
between goal and the various program rules, and (b) a match handler, expressed
as an anonymous method, that takes a resolving rule P and a resolving substitution usub and calls resolve on P, usub, and the rest of the goals more-goals, as
well as the method prove itself, the current substitution theta, and the current
continuation M. The method resolve will then recursively call prove with:

1. a new goal list, obtained by prepending the antecedents of the resolving rule
   P to more-goals and applying the resolving substitution usub to the resulting
   list;
2. a new current substitution, obtained by composing the resolving substitution
   usub with theta, and
3. a new anonymous proof continuation, obtained by stacking the necessary
   applications of mp, uspec-list, etc. (as described in Section 4) to M. Observe
   the form of the new continuation:

   \[
   \text{method (sub) (dbegin (mp (uspec-list P (sub vars))
      (conj-intro (sub ants)))
      (M sub))}
   \]

   The key point here is that the new continuation invokes M in the tail position
   of the dbegin, which ensures that the assumption base is threaded linearly—
   so that the conclusion of the modus ponens will be available throughout the
   evaluation of (M sub).

For instance, suppose that we issue the following declarations, definitions, and
assertions:

(domain Thing)

(define p q r (→ (Thing) Boolean))

(define f g (→ (Thing) Thing))

(define a Thing)

(define axiom-1 (if true (p (f a))))
(define axiom-2 (forall ?x (if (p ?x) (q ?x))))
(define axiom-3 (forall ?x (if (and (p ?x) (q ?x)) (r ?x))))
(define axiom-4 (forall ?x (if (r (f ?x)) (r (g ?x)))))

(define axioms [axiom-1 axiom-2 axiom-3 axiom-4])

(assert axioms)

Then the method call (!solve axioms [(r ?Z)]) will derive the theorem (r (g a)).
6 Extensions and improvements

Obtaining the final substitution

Clients of \texttt{solve} might wish to be given access to the final substitution $\theta$ that is computed internally during a successful call to \texttt{solve}. Although a client could easily obtain the values of $\theta$ for the query variables by matching the theorem produced by \texttt{solve} against the conjunction of the queries, it would be easier and more efficient if \texttt{solve} itself somehow passed $\theta$ out, so that anyone who was interested in it could immediately get it. But methods can only produce propositions, so \texttt{solve} cannot directly return a substitution along with its theorem. We can easily get around this through a judicious use of side effects. We can supply \texttt{solve} with a third argument, a cell, which can then be assigned to contain the final substitution:

\begin{verbatim}
(define (solve axioms queries sub-cell)
  (dletrec ((prove (method (goals theta M))
    (dmatch goals
      ([] (\theta theta))
      (list-of goal more-goals)
        (!try-matches (get-matches goal axioms)
          (method (P usub)
            (!resolve P usub more-goals
              prove theta M)))))))
    (!prove queries {} (method (theta)
      (dbegin (set! sub-cell theta)
        (!conj-intro (theta queries))))))

The rest of the code is unaffected by this change.

Multiple solutions

Prolog engines are capable of discovering not just one success branch but all of them, by performing a potentially exhaustive depth-first search of the SLD-tree. In practice this is usually implemented as follows: first, the substitution determined by the leftmost success branch is displayed, and then every time the user presses the “enter” key the substitution determined by the next success branch is computed and displayed, until the entire tree has been scanned. This is a powerful feature of logic programming, and it is natural to ask whether we can simulate it in our framework.

It turns out we can implement this quite succinctly owing to the powerful combination of higher-order methods, lexical scoping, and state. We will supply \texttt{solve} with an extra argument, a cell that will be assigned to contain a method of zero arguments (a “proof thunk”). After \texttt{solve} returns, every time we invoke the contents of that cell we will obtain the theorem determined by the next success branch, until there are no more solutions. As a very simple sample use of this new version of \texttt{solve}, consider a program consisting of the three facts
(p a), (p b), and (p c), listed in that order. The query (p ?X) should give rise to three theorems, obtained by consecutively instantiating ?X with a, b, and c. Supposing that tc is a cell, the initial call ((solve axioms (p ?X)) tc) will derive the theorem (p a), and also, as a side effect, put an appropriate thunk in tc. Then invoking thunk with the method call (!!ref tc)) will produce the theorem (p b). (Because a phrase such as (!!ref tc)) is, syntactically speaking, a deduction, the soundness theorem of Athena guarantees that its result will be a logical consequence of the assumption base.) One more invocation (!!ref tc)) will derive (p c), and any additional invocations after that point will fail.

The thunk cell supplied to solve will be passed on to try-matches, which will be responsible for modifying it appropriately:

(define (solve axioms queries thunk-cell)
  (dletrec ((prove (method (goals theta X))
    (dmatch goals
      (if (!= theta))
        (list-of goal more-goals)
        (list-of goal get-matches goal axioms)
        (method (P msub)
          (!resolve P msub more-goals
            prove theta X)
          thunk-cell)))))
  (!!!prove queries {} (method (theta) (!!conj-intro (theta queries))))))

All try-matches now has to do is form a thunk containing its own recursive call on the remaining matches and put that thunk into the given cell. The second arm of the try now becomes a simple invocation of the thunk:

(define (try-matches matches process-match thunk-cell)
  (dmatch matches
    ((list-of [axiom unifying-sub] rest)
      (dlet ((thunk (method ()
        (try-matches rest process-match thunk-cell)))))
      (_ (set! thunk-cell thunk))
      (try (!process-match axiom unifying-sub)
        (!!thunk))))))

Observe that the same thunk-cell is passed as an argument to the recursive call to try-matches in the body of the thunk. This is crucial in order for the cell to be appropriately updated every time its contents are called. The rest of the code is unaffected.

7 Conclusions

We have shown how to define Prolog’s implementation of SLD-resolution in terms of very simple inference rules. More precisely, we have extended the usual execution model of SLD-resolution to make it justify its operation by producing
formal proofs of correctness using only one introduction and two elimination rules for natural deduction. Such proofs are called certificates, and can be verified by exceptionally simple proof checkers. This work will be useful in any situation that demands an implementation of SLD-resolution that relies on a minimal trusted base. It can be regarded as an application of certified computation [5], a general methodology for producing highly reliable software.

This technique is different from conventional program verification. Program verification is static and total: we prove once and for all that an algorithm will always give correct results, no matter what input we supply to it. In the case of SLD-resolution, for instance, this has been done in the Calculus of Constructions (Coq) [12]. Certified computation, by contrast, is dynamic and partial: the correctness proof is performed dynamically, and pertains only to a particular result obtained for a particular input; no claims are made about other inputs and outputs. Thus certified computation provides a weaker guarantee than program verification. However, it is still a very valuable guarantee—if and when we get a result, we can be assured that it is a correct result. This is particularly important in theorem-proving systems. Moreover, total verification is usually vastly more difficult than certified computation. For instance, in the aforementioned correctness proof of SLD-resolution in Coq, more than 600 lemmas had to be formulated and proved. By contrast, our certifying implementation fits in one page of code. Another important difference is that static proofs such as the aforementioned one in Coq usually verify an abstract model of a software system, not actual code; whereas in certified computation the produced theorem refers to the actual result obtained in real time.

A complete implementation of a certifying Prolog interpreter was given in Athena and discussed in detail. A high-level algorithm was also presented, which should enable the implementation of this technique in a language other than Athena. We demonstrated that Athena’s sharp syntactic distinction between computations and deductions and its very high-level semantics allow for concise and fluid definitions of quite complex inference methods. We also considered several extensions and improvements, and showed that our implementation can readily accommodate them.

References


Appendix A

Below we present brief descriptions of some Athena primitives that are necessary to understand the code given in Figure 3.

Substitutions: Substitutions are primitive values in the computational universe of Athena. The empty substitution is denoted by {}. A substitution can be applied to a term as if it were a function. For instance, the expression
\((\{\}) \texttt{ ?foo}\) applies the empty substitution to the variable \texttt{ ?foo}, and will return the variable unchanged. The same notation can be used to apply a substitution \(\theta\) to a list of terms \([t_1, \ldots, t_n]\), producing the list \(\left[\theta(t_1), \ldots, \theta(t_n)\right]\) (provided that the application of \(\theta\) to each \(t_i\) does not result in any ill-sorted terms). For instance, \((\{\}) \texttt{ [true ?x]}\) would result in the list \texttt{ [true ?x]}. The predefined function \texttt{ unify} takes two terms \(t_1\) and \(t_2\) and produces an idempotent mgu for them, if \(t_1\) and \(t_2\) are unifiable, or the term \texttt{ false} otherwise; while \(\texttt{(compose-subs } \sigma \texttt{ } \theta)\) produces the substitution \(\sigma \circ \theta\).

**Pattern matching:** A pattern of the form

\[
(\forall v_1) \cdots (\forall v_n) Q
\]

is matched by any proposition of the form

\[
(\forall v_1) \cdots (\forall v_n) P
\]

and results in the bindings \(\texttt{vars} \mapsto [v_1, \ldots, v_n]\) and \(\texttt{body} \mapsto Q\). The pattern will in fact be matched by any proposition \(P\), even if it is not a universal quantification. In that case we simply have \(n = 0\), and we thus obtain the bindings \(\texttt{vars} \mapsto [\]\) and \(\texttt{body} \mapsto P\). A pattern of the form

\[
(\texttt{and (some-list conjuncts)})
\]

will be matched by any proposition of the form \(P_1 \land \cdots \land P_n\), resulting in the binding

\[
\texttt{conjuncts} \mapsto [P_1, \ldots, P_n].
\]

For instance, if \(Q\) is the proposition \(\texttt{(and Q_1 Q_2 Q_3)}\) (for some propositions \(Q_1\), \(Q_2\), and \(Q_3\)), then \(Q\) will match the pattern \(\texttt{(and (some-list props)}}\) under \(\texttt{props} \mapsto [Q_1, Q_2, Q_3]\). Pattern 5 will actually be matched by any proposition \(P\), even one that is not a conjunction; in that case we simply have \(n = 1\) and \(\texttt{conjuncts} \mapsto [P]\). The pattern \(\_\) is a “wildcard” matching any value. Accordingly, the pattern

\[
(\forall v_1) \cdots (\forall v_n) (P_1 \land \cdots \land P_m \Rightarrow P)
\]

for \(n \geq 0\), \(m \geq 1\), resulting in the bindings \(\texttt{vars} \mapsto [v_1, \ldots, v_n]\), \(\texttt{ants} \mapsto [P_1, \ldots, P_m]\).

**Built-in methods:** \texttt{sp} is Athena’s primitive method for modus ponens. A method call of the form \(\texttt{(mp P \Rightarrow Q P)}\) produces \(Q\) whenever the propositions \(P \Rightarrow Q\) and \(P\) are in the assumption base. By convention, the proposition \texttt{true} is a member of every assumption base. The unary primitive method \texttt{conj-intro} takes a list of propositions \([P_1, \ldots, P_k]\) as its argument and produces the conjunction \(\texttt{(and P_1 \cdots P_k)}\) whenever each \(P_i\) is in the assumption base. Finally, universal specialization is performed by the binary primitive method
uspec. The application (uspec (∀v) P t) yields the proposition obtained from P by replacing each free occurrence of v by t, provided that the generalization (∀v) P is in the assumption base, and that the most general sort of v in P is compatible (unifiable) with the sort of t. The predefined binary method uspec-list performs a sequence of consecutive universal specializations. It is not a primitive method, as it is readily defined in terms of uspec as follows:

\[
\text{(define (uspec-list P terms)}
\text{ (dmatch terms}
\text{ ([] (!claim P)))
\text{ ((list-of t rest) (!uspec-list (uspec P t) rest)))))}
\]

Because it is such a commonly used method, uspec-list is offered at the top Athena level.

The try construct allows for proof search via backtracking: (try D₁ D₂) first attempts to evaluate D₁ in the current assumption base β. If D₁ successfully yields a result P, then P becomes the result of the entire try. But if D₁ fails then the result of the try becomes the result of evaluating D₂ in β. The form (try D₁ D₂ D₃) is defined as (try D₁ (try D₂ D₃)), and likewise for (try D₁ ⋯ Dₙ) and arbitrary n > 0. Finally, we describe the top-level functions that are used in the code. The unary function rename produces an alphabetic variant of any given proposition P, obtained by replacing the quantified variables of P by uniquely fresh variables. E.g., (rename (forall ?x (= ?x ?x))) might yield (forall ?v47 (= ?v47 ?v47)). The binary function add is a “consing” function: it prepends a given element x in front of a list l; the function join performs list concatenation; and rev is a list reversal function.