Techniques for Verifying Prolog Implementations

Keehang Kwon, Jang-Wu Jo and Sanghoon Kim
Department of Computer Engineering, DongA University
840 Hadan, Pusan 604-714, Korea
{ khkwon, jwjo, shkim }@daunet.donga.ac.kr

Abstract. This paper presents some techniques that reduce the complexity of the verification of Prolog implementations. Two key techniques are stepwise refinements and bisimulation. The method of stepwise refinements captures various optimization techniques employed in the implementation process. The correctness of each refinement is justified using a notion called bisimulation. Our definition of bisimulation makes use of the notion of *essential* states in a machine. The notion of essential states further reduces the complexity of the proof. We illustrate our method by presenting two equivalent machines: a nondeterministic interpreter for Prolog and its refinement that executes compiled codes.
Keywords: Prolog, compilers, bisimulation

1 Introduction

An implementation of a programming language typically employs various optimization techniques such as compilation, sharing, *etc.* Consequently, the resulting bytecode is significantly different from its original semantics and implementation verification is therefore essential to bridge this gap.

There have been previous works to proving implementation correctness related to Prolog [1, 2, 9]. However, most of these works have relied on an *ad hoc* method and do not provide much insight regarding a general strategy towards implementation verification. In this paper, we present a general framework that can be applied to verifying most implementations. Our approach is based on the notions of stepwise refinements and bisimulation. The method of stepwise refinements shows how a bytecode of a Prolog can be arrived at logically from its operational semantics. Thus, starting from a state transition machine that corresponds to an operational semantics of a language, we build a hierarchy of equivalent machines to obtain a machine that corresponds to a final bytecode. This method thus captures all the techniques employed in the implementation process. Furthermore, it provides a direct proof of the correctness of the bytecode provided each refinement step is verified.

The equivalence of the refined machine to the earlier one is shown by using a framework called bisimulation. Such a framework has been previously studied in computability theory [4]. We demonstrate here that this framework is well-suited to verifying Prolog implementations as well.

We illustrate our method by presenting two implementations of Prolog: an original interpreter for the language and its bytecode that are obtained by compiling the source language. While this example is of a simple nature, it nevertheless permits all the important aspects of implementation verification.

The rest of this paper is organized as follows. Section 2 summarizes the important notions pertaining to bisimulation and stepwise refinements. Section 3 presents how our approach can be applied to Prolog implementations. Section 4 is the conclusion of this paper.

2 The Framework

Our framework is based on the notions of stepwise refinements and bisimulation. The method of stepwise refinements is a useful tool for proving correctness of implementations. To achieve this task, one can build, by the method of successive refinements, a series of more concrete machines $\mathcal{M}_1, \ldots, \mathcal{M}_n$ where $\mathcal{M}_1$ is the original high-level specification and $\mathcal{M}_n$ is the target implementation. The equivalence of $\mathcal{M}_n$ with respect to $\mathcal{M}_1$ naturally follows from the equivalence of $\mathcal{M}_{i+1}$ with respect to $\mathcal{M}_i$, for $1 \leq i \leq n-1$.

Most equivalences in this context can be shown by the notion that $\mathcal{M}_{i+1}$ and $\mathcal{M}_i$ are bisimilar to each other. The notion of one machine simulating another has been previously studied (*e.g.*, see [4, 5, 8]) and is formulated as follows.

Definition 2.1. A state z in machine $\mathcal{M}_1$ *simulates* a state x in machine $\mathcal{M}_2$, written as $z \sim x$, if the following hold:

(a) If x is a final state, then z is a final state with identical observable outputs, and
(b) For any state y of $\mathcal{M}_2$ that can be reached by a single transition α from x, there is a state w of $\mathcal{M}_1$ that similates y and that can be reached by a finite (possibly empty) sequence of transitions β^* from z in $\mathcal{M}_1$.

z is said to be *bisimilar* to x, written as $z \approx x$, if z simulates x and x simulates z.

There are different notions of simulations. A simplest kind of this framework is called strong simulation in which the behaviours of two machines corresponds step for step. A more general framework is called weak simulation in which one step in a machine corresponds to multiple steps in another machine.

In applying simulation by subprogram to real-world examples, it is particularly useful to partition a terminating transition sequence into subsequences. This technique greatly reduces the complexity of the proof. To formulate this idea, we define the set of *essential* states of a machine that is a subset of states that are reachable from an initial state such that any transition sequence from an initial state to a final state in the machine can be decomposed into a sequence of transitions between essential states. Under this new interpretation, x, z, y, w in Definition 2.1 are restricted to essential states.

3 Prolog

The language based on Horn clause logic (*i.e.*, pure Prolog) can be described by two classes of formulas, called G- and C-formulas. They are given by the syntax rules below:

$$G ::= A \mid G \wedge G$$

$$C ::= A \mid G \supset A \mid \forall x C.$$

In the rules above, A represents an atomic formula. G-formulas will function as goals or queries and lists of C-formulas (*i.e.*, Horn clauses) will constitute programs. In the description that follows, we use the constant *nil*, the infix symbol :: as list constructors, the infix symbol @ as the operator which appends two lists. We often write $a_1 :: \ldots :: a_n :: nil$ as $[a_1, \ldots, a_n]$. We also use the notation $\mathcal{F}(F)$ to denote the set of free variables in a formula F, $\theta(F)$ to denote the application of a substitution θ to a formula F, and $\theta_1 \circ \theta_2$ to denote the the composition of θ_1 and θ_2, *i.e.*, $\theta_1 \circ \theta_2(x) = \theta_1(\theta_2(x))$. Further, we assume the existence of the set $\mathcal{W}$ of the form $\{w(0), w(1), \ldots, w(n), \ldots\}$ where each $w(i), i \geq 0$, denotes a free variable.

We also need the notion of replacing variables in a goal with free variables relative to a base value. This is made precise below.

Definition 3.1. Given a goal G, the function Φ_G is defined as follows: $\Phi_G(G, \kappa) = \chi(G)$ where $\chi = \{\langle x_i, w(\kappa+i-1)\rangle \mid x_i \in \mathcal{F}(G)\}$.

A nondeterministic interpreter for this language can be decribed by means of a transition system. The states of this transition system are given by a tuple of the form $\langle \mathcal{P}, \mathcal{G}, \mathcal{I}, \theta \rangle$. In the tuple, $\mathcal{P}$ is a list of clauses, $\mathcal{G}$ represents a list of goals, $\mathcal{I}$ represents an index to the set $\mathcal{W}$, and θ denotes a substitution. Transition rules in the system of interest are those given by the following definition.

Definition 3.2. Given a state $\langle \mathcal{P}_1, \mathcal{G}_1, \mathcal{I}_1, \theta_1 \rangle$, the state $\langle \mathcal{P}_2, \mathcal{G}_2, \mathcal{I}_2, \theta_2 \rangle$ can be obtained from it in one of the following ways:

(1) Suppose that $\mathcal{G}_1$ is $A :: \mathcal{G}'$ and that C is a clause $\forall x_1 \ldots \forall x_n A'$ in $\mathcal{P}_1$. Let $\rho = \{\langle x_1, w(\mathcal{I}_1)\rangle, \ldots, \langle x_n, w(\mathcal{I}_1+n-1)\rangle\}$ be a substitution. If $\theta_1(A)$ and $\theta_1 \circ \rho(A')$ are unifiable with a most general unifier σ, then the new state may be obtained by setting $\mathcal{G}_2$ to $\mathcal{G}'$, $\mathcal{I}_2$ to $\mathcal{I}_1 + n$ and θ_2 to $\sigma \circ \theta_1$.

(2) Suppose that $\mathcal{G}_1$ is $A :: \mathcal{G}'$, and that C is a clause $\forall x_1 \ldots \forall x_n (G \supset A')$ in $\mathcal{P}_1$. Let $\rho = \{\langle x_1, w(\mathcal{I}_1)\rangle, \ldots, \langle x_n, w(\mathcal{I}_1+n-1)\rangle\}$ be a substitution. If $\theta_1(A)$ and $\theta_1 \circ \rho(A')$ are unifiable with a most general unifier σ, then the new state may be obtained by setting $\mathcal{G}_2$ to $\rho(G) :: \mathcal{G}'$, $\mathcal{I}_2$ to $\mathcal{I}_1 + n$ and θ_2 to $\sigma \circ \theta_1$.

(3) If $\mathcal{G}_1$ is $(G_1 \wedge G_2) :: \mathcal{G}'$, then the new state may be obtained by setting $\mathcal{G}_2$ to $G_1 :: G_2 :: \mathcal{G}'$.

Given an initial goal G such that $\mathcal{F}(G) = \{x_1, \ldots, x_m\}$ and a program $\mathcal{P}$, an initial state is given by $\langle \mathcal{P}, \Phi_G(G, 0), m, \emptyset\rangle$. A final state is any state where the $\mathcal{G}$ component is an empty list. In this case, θ in a final state is referred to as the corresponding *answer substitution*.

The refinement of $\mathcal{M}_1$ we consider below uses environments to permit a delaying of substitutions. Given a a clause of the form $\forall x_1 \forall x_2 (p(x_2) \supset p(x_1))$, a substitution can be implicitly performed by simply recording a base value κ and by reading x_1 as $w(\kappa)$ and x_2 as $w(\kappa{+}1)$. Under this scheme, each state needs to maintain a list of environments for the clauses that have been used along the path from the initial state. For this purpose, a new component $\mathcal{E}$ is provided. When a clause is selected, a new environment (a natural number) for the clause will be added to the front of $\mathcal{E}$ and, when this clause is "solved", this environment will be removed from $\mathcal{E}$ list. An instruction *dealloc* will be included in the the goal sequence to remove an environment.

On top of that, we consider a simple translation of clauses and goals into low-level instructions that are intended to produce the same behavior. These instructions are straightforward. In particular, the translation of $\wedge$ will result in a sequence of instructions to be executed as such.

In order to treat the compilation process formally, we introduce the notion of compiled goals and compiled clauses. These will be a sequence of instructions whose meaning should be clear from the informal discussion above and will also be made precise by the transition rules we define in the next section.

Definition 3.3. Compiled goals, clauses and nonempty sequences of clauses are denoted by G and C respectively and are given by the following syntax rules:

$$G ::= [\mathit{call}(A)] \mid \mathit{call}(A) :: G$$
$$C ::= [\mathit{alloc}(N), \mathit{unify}(A)]@G@[\mathit{dealloc}] \mid [\mathit{alloc}(N), \mathit{unify}(A), \mathit{dealloc}]$$

where A represents atomic formulas.

The intended correspondence between clauses and goals and their compiled versions is made clear by defining compiling functions with respect to clauses and goals. These two functions are called $\mathit{compile}_C$ and $\mathit{compile}_G$ respectively and are defined below. We also extend $\mathit{compile}_G$ and $\mathit{compile}_C$ to functions $\mathit{compile}_{\mathcal{G}}$ and $\mathit{compile}_{\mathcal{P}}$ that takes a list of goals and clauses as input.

Definition 3.4.

(1) $\mathit{compile}_G(A) = [\mathit{call}(A)]$
(2) $\mathit{compile}_G(G_1 \wedge G_2) = \mathit{compile}_G(G_1)@\mathit{compile}_G(G_2)$
(3) Let $\mathcal{G}$ be $[G_1, \ldots, G_n]$. Then $\mathit{compile}_{\mathcal{G}}(\mathcal{G}) = \mathit{compile}_G(G_1)@\ldots@\mathit{compile}_G(G_n)$
(4) Let C be $\forall x_1 \ldots \forall x_n (G \supset A)$. Then $\mathit{compile}_C(C) = [\mathit{alloc}(n), \mathit{unify}(A)]@\mathit{compile}_G(G)@[\mathit{dealloc}]$
(5) Let C be $\forall x_1 \ldots \forall x_n A$. Then $\mathit{compile}_C(C) = [\mathit{alloc}(N), \mathit{unify}(A), \mathit{dealloc}]$
(6) Let D be $C_1 :: \ldots :: C_n :: \mathit{nil}$. Then $\mathit{compile}_{\mathcal{P}}(D) = [\mathit{compile}_C(C_1), \ldots, \mathit{compile}_C(C_n)]$

These various changes are manifest in the state transition machine called $\mathcal{M}_2$. The states of this transition system are given by a tuple of the form $\langle \mathcal{P}, \mathcal{G}, \mathcal{I}, \theta, \mathcal{E}, g\rangle$. Transition rules in the new machine are given by the following definition.

Definition 3.5. Given a state $\langle \mathcal{P}_1, \mathcal{G}_1, \mathcal{I}_1, \theta_1, \mathcal{E}_1, g_1\rangle$, the state $\langle \mathcal{P}_2, \mathcal{G}_2, \mathcal{I}_2, \theta_2, \mathcal{E}_2, g_2\rangle$ can be obtained from it in one of the following ways:

(1) Suppose that $\mathcal{G}_1$ is $call(A) :: \mathcal{G}'$ and that C is a compiled clause in $\mathcal{P}_1$. Let e be the first element of $\mathcal{E}_1$. Then the new state may be obtained by setting $\mathcal{G}_2$ to $C@\mathcal{G}'$, and $g = \theta_1(\Phi_G(A, e))$.
(2) Suppose that $\mathcal{G}_1$ is $alloc(n) :: \mathcal{G}'$. Then the new state may be obtained by setting $\mathcal{G}_2$ to $\mathcal{G}'$, $\mathcal{I}_2$ to $\mathcal{I}_1 + n$, and $\mathcal{E}_2$ to $\mathcal{I}_1 :: \mathcal{E}_1$.
(3) Suppose that $\mathcal{G}_1$ is $unify(A) :: \mathcal{G}'$. Let e be the first element of $\mathcal{E}_1$. If g and $\theta_1(\Phi_G(A, e))$ are unifiable with a most general unifier σ, then the new state may be obtained by setting $\mathcal{G}_2$ to $\mathcal{G}'$, and θ_2 to $\sigma \circ \theta_1$.
(4) If $\mathcal{G}_1$ is $dealloc :: \mathcal{G}'$ and $\mathcal{E}_1$ is $e :: \mathcal{E}'$, then the new state may be obtained by setting $\mathcal{G}_2$ to $\mathcal{G}'$, and $\mathcal{E}_2$ to $\mathcal{E}'$.

Let G be a goal such that $\mathcal{F}(G) = \{x_1, \ldots, x_m\}$ and $\mathcal{P}$ be a program. Then an initial state is given by $\langle compile_{\mathcal{P}}(\mathcal{P}), compile_G(G)@[dealloc], m, \emptyset, [0]\rangle$. A final state in the transition system is any state where the $\mathcal{G}$ component is an empty list. In this case, θ in a final state is referred to as the corresponding *answer substitution*.

We now show that $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent from the perspective of the computed substitution. To do this, we need to define the notion of correspondence between (control) states of $\mathcal{M}_2$ and $\mathcal{M}_1$. In doing so, we observe that an essential state in $\mathcal{M}_1$ is any state whose first goal is an atomic goal. Similarly, an essential state in $\mathcal{M}_2$ is any state whose first goal is the instruction *call*. After that, we can treat each subprogram as a single transition rule.

We make use of this observation in describing the correspondence. Now we need the notion of transforming goal sequences in $\mathcal{M}_2$ into a decoded form as given below.

Definition 3.6. Let $\mathcal{G}$ be a goal sequence and let $\mathcal{E}$ be an environment sequence associated with a state of $\mathcal{M}_2$. Then the function $\Phi_{\mathcal{G}}$ on $\mathcal{G}$ and $\mathcal{E}$ is defined as follows:

- $\Phi_{\mathcal{G}}(nil, nil) = nil$
- $\Phi_{\mathcal{G}}(dealloc :: \mathcal{G}, \kappa :: \mathcal{E}) = \Phi_{\mathcal{G}}(\mathcal{G}, \mathcal{E})$
- $\Phi_{\mathcal{G}}(G :: \mathcal{G}, \kappa :: \mathcal{E}) = \Phi_G(G, \kappa) :: \Phi_{\mathcal{G}}(\mathcal{G}, \kappa :: \mathcal{E})$

The notion of correspondence between essential states of $\mathcal{M}_2$ and $\mathcal{M}_1$ is given as follows.

Definition 3.7. An essential state $\langle \mathcal{P}_1, \mathcal{G}_1, \mathcal{I}_1, \theta_1\rangle$ in $\mathcal{M}_1$ *corresponds to* an essential state $\langle \mathcal{P}_2, \mathcal{G}_2, \mathcal{I}_2, \theta_2, \mathcal{E}_2, g_2\rangle$ in $\mathcal{M}_2$, if $compile_{\mathcal{P}}(\mathcal{P}_1) = \mathcal{P}_2$, $\mathcal{I}_1 = \mathcal{I}_2$, $\theta_1 = \theta_2$, and $compile_{\mathcal{G}}(\mathcal{G}_1) = \Phi_{\mathcal{G}}(\mathcal{G}_2, \mathcal{E}_2)$.

We note that the relation *corresponds to* is bijective and symmetric. The following lemma shows that the notion of correspondence is preserved between intermediate states and indeed is a bisimulation. Its proof follows from an observation that $\mathcal{M}_1$ and $\mathcal{M}_2$ simulate each other in lockstep under the decomposition of the transition sequence in $\mathcal{M}_2$ as discussed.

Lemma 1. *Let x be an essential state of $\mathcal{M}_1$ and let z be an essential state of $\mathcal{M}_2$ such that x corresponds to z. If x' and z' are next states that $\mathcal{M}_1$ and $\mathcal{M}_2$ transit to from x and z respectively, then x' corresponds to z'.*

Proof. We prove this lemma by an induction on the length of transitions between essential states.

Lemma 2. *Let I_1 be the initial state of $\mathcal{M}_1$ and let I_2 be the initial state of $\mathcal{M}_2$. Then I_1 is bisimilar to I_2.*

Proof. This lemma follows from the observation that the initial(final) state of $\mathcal{M}_1$ corresponds to the initial(final) state of $\mathcal{M}_2$ and from Lemma 1.

Theorem 1. *The machine $\mathcal{M}_1$ with program $\mathcal{P}$ and goal G is equivalent to the machine $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G.*

Proof. This theorem follows from the observation that the initial state of $\mathcal{M}_1$ is bisimilar to the initial state of $\mathcal{M}_2$.

4 Conclusion

There have been two verifications of the WAM [10] Prolog implementations. One of them is due to Russinoff [9], who employed the modified SLD refutation procedure which adopted "last call optimization" as an intermediate specification. He then used this intermediate specification to prove the equivalence of source and target specification. His work, while mathematically rigorous, is hard to follow due to the enormous complexity of the proof. The other verification is due to Börger and Rosenzweig [2] who, in contrast to Russinoff, employed stepwise refinements. Each refinement was followed by the verification that the resulting machine was equivalent to its predecessor, making the verification much easier to follow. Beierle and Börger in [1] also applied this framework to the verification of an implementation of an extension to Prolog with polymorphism and subtypes. They describe a state map which requires a transition in one system to mapped to a subprogram, *i.e.*, a *fixed* sequence of transitions, in another system. However, this does not appear to apply to a general case such as the example given in the previous section. Finally, a verification of a closured-based compilation method for embedded implications in logic programming appears in [5] using the framework proposed here.

Our proof method discussed in this paper can be applied in a natural way to implementing LogicWeb [7], Semantic Web [3] or other abstract machines. In particular, our interests lie in the abstract machine designed for dealing with a polymorphically typed version of Prolog [6].

References

1. Christopher Beierle and Egon Börger. Correctness proof for the WAM with types. In H. Kleine Büning E. Börger, G. Jöger and M. Richter, editors, *Computer Science Logic 91*, volume 626. Springer-Verlag, 1992. *Lecture Notes in Computer Science*.
2. Egon Börger and Dean Rosenzweig. The WAM — definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North–Holland, 1994.
3. J. Davies, D. Fensel, and F. V. Harmelen. *Towards the Semantics Web*. John Wiley and Sons, 2003.
4. R. W. Floyd and R. Beigel. *The Language of Machines: an Introduction to Computability and Formal Languages*. W. H. Freeman and Company, 1994.
5. Keehang Kwon. *Towards a Verified Abstract Machine for a Logic Programming Language with a Notion of Scope*. PhD thesis, Duke University, December 1994. Also available as Technical Report CS–1994–36 from Department of Computer Science, Duke University.
6. Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.
7. Seng Wai Lok and Andrew Davison. Logic Programming with the WWW. In *Proceedings of the 7th ACM conference on Hypertext*. ACM Press, 1996.
8. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
9. David Russinoff. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming*, 13:367–412, 1992.
10. D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.