On Answer Substitutions in Logic Programming

Keehang Kwon
Dept. of Computer Engineering, DongA University
Busan 604-714, Korea
khkwon@dau.ac.kr

Abstract: Answer substitutions play a central role in logic programming. To support selective answer substitutions, we refine $\exists x$ in goals into two different versions: the noisy version $\exists^o x$ and the silent version $\exists x$. The main difference is that only the instantiation in $\exists^o x$ will be recorded in the answer substitutions. Similarly for $\forall x$.

keywords: Prolog, answer substitution, choice quantifiers.

1 Introduction

The notion of answer substitutions plays a key role in logic programming. Unfortunately, the notion of answer substitutions in traditional logic languages is unsatisfactory in the sense that every instantiation is visible to the user. For example, consider the program $\text{phone(tom, CS, 4450)}$ which represents that $\text{tom}$ is a CS major and his phone number is 4450. Now solving a query $\exists x \exists y \text{phone(tom, x, y)}$ is a success with the answer substitutions $x = \text{CS}, y = 4450$. What if we want to see only Tom’s phone number, but not his major? It is not possible to specify this in traditional logic programming.

To fix this problem, inspired by [4], we extend Prolog to include the following new formulas (called noisy universal/existential quantifiers):

- $\forall^o x D$ where $D$ is a definite clause.
- $\exists^o x G$ where $G$ is a goal formula in Prolog.

In the above, $\exists^o x G$ is identical to $\exists x G$ with the difference that, in the former, the instantiation $t$ for $x$ is visible and recorded in the answer substitution, but not in the latter. Thus the latter is a silent version of the former that chooses a term $t$ for $x$ silently. Similarly for $\forall^o x D$.

They are originally called choice quantifiers, denoted by $\sqcap, \sqcup$ in [4].
Thus it is $\exists^o x \ G$, rather than $\exists x \ G$, that adequately captures the notion of answer substitution. Furthermore, it is the notion of answer substitution instead of proof theory that really counts when dealing with logic programming.

Fortunately, in a simple setting such as Prolog, operational semantics for these new quantifiers can be easily obtained from $\forall$, $\exists$ by additionally recording each instantiation during execution. To be specific, we adopt the following operational semantics for $\exists$ and $\exists^o$.

- $pv(D, \exists x G, \text{nil})$ if $pv(D, [t/x]G, \_)$
- $pv(D, \exists^o x G, (x, t))$ if $pv(D, [t/x]G, \_)$

where the third argument is used to record the instantiation (if any) at the current proof step.

This paper also deals with anonymous variables. Anonymous variables in a clause $D$ can be interpreted as $\forall x D$, but not $\forall^o D$, while anonymous variables in a goal $G$ can be interpreted as $\exists x G$.

This paper proposes Prolog$^\theta$, an extension of Prolog with new quantifiers. The remainder of this paper is structured as follows. We describe Prolog$^\theta$ in the next sections. Section 4 concludes the paper.

## 2 The Language

The language is a version of Horn clauses with noisy quantifiers. It is described by $G$- and $D$-formulas given by the syntax rules below:

$$G ::= A \mid G \land G \mid \exists x G \mid \exists^o x G$$

$$D ::= A \mid G \supset A \mid \forall x D \mid \forall^o x D \mid D \land D$$

In the rules above, $A$ represents an atomic formula. A $D$-formula is called a Horn clause with noisy quantifiers.

We describe a proof procedure based on uniform proofs [7]. Note that proof search alternates between two phases: the goal-reduction phase and the

\footnote{In a more general setting where $\exists x \ D, \exists^o x \ D$ are allowed, there is a subtle yet critical difference between $\exists$ and $\exists^o$. That is, in $\exists^o x \ D$, $x$ must be instantiated to a ground term, while $x$ can be instantiated to an arbitrary term (a new constant, for example) in $\exists x$.}
backchaining phase. In the goal-reduction phase (denoted by $pv(D,G,\theta)$), the machine tries to solve a goal $G$ from a clause $D$ by simplifying $G$. If $G$ becomes an atom, the machine switches to the backchaining mode. In the backchaining mode (denoted by $bc(D_1,D,A,\theta)$), the machine tries to solve an atomic goal $A$ by first reducing a Horn clause $D_1$ to simpler forms and then backchaining on the resulting clause (via rule (1) and (2)).

A proof of $\langle D, G \rangle$ is a sequence of $bc(D_1,D,A,\theta_1), \ldots, pv(D_n,G_n,\theta_n)$, with $D = D_n$ and $G = G_n$. Each $pv(D_i,G_1,\theta_i)$ (or $bc(D_i',D_i,G_1,\theta_i)$) is called a proof step. Each $\theta_i$ records an instantiation (if any) performed at the $i$th proof step.

**Definition 1.** Let $G$ be a goal and let $D$ be a program. Then the notion of proving $\langle D, G \rangle$ with recording the answer substitution performed at the last proof step $\theta – pv(D,G,\theta)$ – is defined as follows:

1. $bc(A, D, A, nil)$. % This is a success.
2. $bc((G_0 \supset A), D, A, nil)$ if $pv(D,G_0,\_).$ % backchaining
3. $bc(D_1 \land D_2), D, A, nil)$ if $bc(D_1, D, A, \theta_1) \lor bc(D_2, D, A, \theta_2).$ % $\land$-L
4. $bc(\forall x D_1, D, A, nil)$ if $bc([t/x]D_1, D, A, \theta).$ % $\langle x, t \rangle$ is not recorded in the answer substitution.
5. $bc(\forall o x D_1, D, A, \langle x, t \rangle)$ if $bc([t/x]D_1, D, A, \_).$ % $\langle x, t \rangle$ is recorded in the answer substitution.
6. $pv(D, A, nil)$ if $bc(D, D, A, \_).$ % switch to backchaining mode
7. $pv(D, G_1 \land G_2, nil)$ if $pv(D,G_1,\_) \land pv(D,G_2,\_).$
8. $pv(D, \exists x G, nil)$ if $pv(D,[t/x]G,\_)$ % $\langle x, t \rangle$ is not included in the answer substitution.
9. $pv(D, \exists o x G, \langle x, t \rangle)$ if $pv(D,[t/x]G,\_)$ % $\langle x, t \rangle$ is included in the answer substitution.

These rules are straightforward to read.

Once a proof tree is built, execution is simple. It simply displays its proof steps with the corresponding answer substitution in a bottom-up manner.
3 Examples

As an example, consider solving a query $\exists x \exists^o y \text{phone}(\text{tom}, x, y)$ from the program $\text{phone}(\text{tom}, \text{CS}, 4450)$. Note that this query can be rewritten as $\exists^o y \text{phone}(\text{tom}, y)$ using an anonymous variable.

Below is its proof with answer substitution $\langle y, 4450 \rangle$.

1. $\text{bc}($phone$(\text{tom}, \text{CS}, 4450), \text{phone}(\text{tom}, \text{CS}, 4450), \text{phone}(\text{tom}, \text{CS}, 4450, \text{nil}))$
2. $\text{pv}($phone$(\text{tom}, \text{CS}, 4450), \text{phone}(\text{tom}, \text{CS}, 4450), \text{nil})$
3. $\text{pv}($phone$(\text{tom}, \text{CS}, 4450), \exists^o y \text{phone}(\text{tom}, \text{CS}, y), \langle y, 4450 \rangle)$
4. $\text{pv}($phone$(\text{tom}, \text{CS}, 4450), \exists^o y \text{phone}(\text{tom}, \text{CS}, y), \langle y, 4450 \rangle)$

4 Conclusion

In this paper, we have considered an extension to Prolog [1] with new noisy quantifiers. This extension makes Prolog programs more versatile. If we allow $\exists x, \exists^o x$ in $D$-formulas (and $\forall x, \forall^o x$ in $G$-formulas), some important features – information hiding [7] and user interaction [4], etc – can be achieved. However, implementation becomes way more complicated.

In the near future, we plan to investigate this possibility of including these features into logic programming.

References

[1] I. Bratko, “Prolog:programming for AI “, Addison Wesley, 2001 (3rd edition).

[2] J.Y. Girard, “Linear Logic”, Theoretical Computer Science, vol.50, pp.1–102, 1987.

[3] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, Information and Computation, vol.110, pp.327–365, 1994.

[4] G. Japaridze, “Introduction to computability logic”, Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.

[5] E. Komendantskaya and V. Komendantsky, “On uniform proof-theoretical operational semantics for logic programming”, In J.-Y. Beziau
and A. Costa-Leite, editors, Perspectives on Universal Logic, pages 379–394. Polimetica Publisher, 2007.

[6] D. Miller, “A logical analysis of modules in logic programming”, Journal of Logic Programming, vol.6, pp.79–108, 1989.

[7] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.