A First-Order Interpreter for Knowledge-based Golog with Sensing based on Exact Progression and Limited Reasoning

Yi Fan  Minghui Cai  Naiqi Li  Yongmei Liu
Department of Computer Science
Sun Yat-sen University
Guangzhou 510006, China
ymliu@mail.sysu.edu.cn

Abstract

While founded on the situation calculus, current implementations of Golog are mainly based on the closed-world assumption or its dynamic versions or the domain closure assumption. Also, they are almost exclusively based on regression. In this paper, we propose a first-order interpreter for knowledge-based Golog with sensing based on exact progression and limited reasoning. We assume infinitely many unique names and handle first-order disjunctive information in the form of the so-called proper \( + \) KBs. Our implementation is based on the progression and limited reasoning algorithms for proper \( + \) KBs proposed by Liu, Lakemeyer and Levesque. To improve efficiency, we implement the two algorithms by grounding via a trick based on the unique name assumption. The interpreter is online but the programmer can use two operators to specify offline execution for parts of programs. The search operator returns a conditional plan, while the planning operator is used when local closed-world information is available and calls a modern planner to generate a sequence of actions.

Introduction

When it comes to high-level robotic control, the idea of high-level program execution as embodied by the Golog language provides a useful alternative to planning. However, current implementations of Golog offer limited first-order capabilities. For example, implementation of Golog is based on the closed-world assumption (CWA), and that of IndiGolog (De Giacomo, Levesque, and Sardà 2001) is based on a just-in-time assumption, which reduces to a dynamic CWA. The interpreter for knowledge-based Golog proposed by Reiter (2001b) is based on the domain closure assumption (DCA) and reduces first-order reasoning to propositional one. But in many real-world applications, CWA or even DCA are inappropriate. In particular, there is a need to deal with disjunctive information and the domain of individuals might be incompletely known. Such information can be represented as a proper \( + \) KB, which is equivalent to a possibly infinite set of ground clauses. Liu, Lakemeyer and Levesque (2004) proposed a logic of limited belief called the subjective logic \( SL \) and showed that \( SL \)-based reasoning with proper \( + \) KBs is decidable. Recently, Claßen and Lakemeyer (2009) proposed an \( SL \)-based Golog interpreter.

An essential component of any Golog interpreter is a query evaluation module, which solves the projection problem, that is, to decide if a formula holds after a sequence of actions have been performed. Two methods to solve the projection problem are regression and progression. An advantage of progression compared to regression is that after a KB has been progressed, many queries about the resulting state can be processed without any extra overhead. Moreover, when the action sequence becomes very long, regression simply becomes unmanageable. However, current implementations of Golog are almost exclusively based on regression. This might be due to the negative result that progression is not always first-order definable (Lin and Reiter 1997). However, Liu and Lakemeyer (2009) showed that for a restricted class of the so-called local-effect actions and proper \( + \) KBs, progression is not only first-order definable but also efficiently computable.

Golog interpreters can be put into three categories: online, offline, and a combination of the two. In the presence of sensing, Reiter’s interpreter for knowledge-based Golog is online, while the one for sGolog (Lakemeyer 1999) is offline, and generates conditional plans. IndiGolog combines online execution with offline execution of parts of programs, specified by the programmer with a search operator. To improve efficiency of Golog interpreters, there has been work on exploiting state-of-the-art planners. Baier et al. (2007) developed an approach for compiling procedural domain control knowledge written in a Golog-like program into a planning instance.

In this paper, we propose a first-order interpreter for knowledge-based Golog with sensing based on exact progression and limited reasoning. Hence we call our version \( LBGolog \) (Limited-Belief-based Golog). We handle first-order incomplete information in the form of proper \( + \) KBs which assume infinitely many unique names. Our implementation is based on the aforementioned progression and limited reasoning algorithms for proper \( + \) KBs by Liu, Lakemeyer and Levesque. To improve efficiency, we implement the two algorithms by grounding via a trick based on the unique name assumption. The interpreter is online but the programmer can use two operators to specify offline execu-
tion for parts of programs. The search operator returns a conditional plan, while the planning operator is used when locally complete information is available and calls a modern planner to generate a sequence of actions. We have experimented our interpreter with a number of domains, and the results show the feasibility and efficiency of our approach.

**Background work**

In this section, we introduce the background work of this paper, i.e., proper\(^+\) KBs, situation calculus, progression, subjective logic, and closed-world assumption on knowledge.

We assume a first-order language \(L\) with equality, a countably infinite set of constants, which are intended to be unique names, and no other function symbols. A literal is an atom or its negation, and a clause is a set of literals. We let \(E\) denote the union of the axioms of equality and the infinite set \(\{d \neq d' \mid d \text{ and } d' \text{ are distinct constants}\}\). Let \(\Gamma\) and \(\Gamma'\) be two sets of sentences. We write \(\Gamma \models \Gamma'\) to mean \(E \cup \Gamma\) classically entails \(\Gamma'\), and we write \(\Gamma \iff \Gamma'\) to mean \(\Gamma \models \Gamma'\) and vice versa. Let \(\phi\) be a formula, and let \(\mu\) and \(\mu'\) be two terms/formulas. We denote by \(\phi(\mu/\mu')\) the result of replacing every occurrence of \(\mu\) in \(\phi\) with \(\mu'\). We let \(\phi_\exists^d\) denote \(\phi\) with all free occurrences of variable \(x\) replaced by constant \(d\).

Intuitively, a proper\(^+\) KB is equivalent to a possibly infinite set of ground clauses. To formally define it, we let \(e\) range over ewffs, i.e., quantifier-free formulas whose only predicate is equality, and we let \(\forall \phi\) denote the universal closure of \(\phi\). We let \(\theta\) range over substitutions of all variables by constants, and write \(\phi(\theta)\) as the result of applying \(\theta\) to \(\phi\).

**Definition 1** Let \(e\) be an ewff and \(c\) a clause. A formula of the form \(\forall (e \supset c)\) is called a \(\forall\)-clause. A KB is proper\(^+\) if it is a finite non-empty set of \(\forall\)-clauses. Given a proper\(^+\) KB \(\Sigma\), \(\text{gnd}(\Sigma)\) is defined as \(\{e\theta \mid \forall (e \supset c) \in \Sigma \text{ and } \models_e e\theta\}\).

A proper\(^+\) KB is suitable for representing first-order disjunctive information for a possibly infinite domain. Also, it is easy to see that any propositional KB can be expressed as a proper\(^+\) KB. A proper\(^+\) KB describing a blocks world appears in the “Experiments” section.

**Situation calculus (Reiter 2001a)**

We will not go over the language \(L_{sc}\) here except to note the following components: action functions including those changing the world and binary sensing actions which do not change the world but inform the agent whether some condition holds in the current world; a constant \(S_0\) denoting the initial situation; a function \(do(a,s)\) denoting the successor situation to \(s\) resulting from performing action \(a\); a finite number of relational fluents, i.e., predicates taking a situation term as their last argument. Often, we need to restrict our attention to formulas that refer to a particular situation \(\tau\), and we call such formulas uniform in \(\tau\). We ignore functional fluents in this paper.

A particular domain of application is specified by a basic action theory (BAT) of the following form:

\[
D = \Sigma \cup D_{ap} \cup D_{ss} \cup D_{sf} \cup D_{una} \cup D_{S_0},
\]

where

1. \(\Sigma\) is the set of the foundational axioms for situations.
2. \(D_{ap}\) is a set of action preconditional axioms, one for each action, of the form \(\text{Poss}(A(\bar{x}), s) \equiv \Pi_A(\bar{x}, s)\), where \(\Pi_A(\bar{x}, s)\) is uniform in \(s\).
3. \(D_{ss}\) is a set of successor state axioms (SSAs), one for each fluent, of the form \(F(\bar{x}, do(a, s)) \equiv \Phi_F(\bar{x}, a, s)\), where \(\Phi_F(\bar{x}, a, s)\) is uniform in \(s\).
4. \(D_{sf}\) is a set of sensed fluent axioms, one for each sensing action, of the form \(SF(A(\bar{x}), s) \equiv \Psi_A(\bar{x}, s)\), where \(\Psi_A(\bar{x}, s)\) is uniform in \(s\).
5. \(D_{una}\) is the set of unique names axioms for actions.
6. \(D_{S_0}\), the initial KB, is a set of sentences uniform in \(S_0\).

We use \(D\) to represent a BAT throughout this paper.

**Progression**

Lin and Reiter (1997) formalized the notion of progression. Let \(\alpha\) be a ground action and let \(S_0\), \(S_t\) represent \(do(\alpha, S_0)\).

**Definition 2** Let \(M\) and \(M'\) be structures with the same domains for sorts action and object. We write \(M \sim S_{s_0} M'\) if the following two conditions hold: (1) \(M\) and \(M'\) interpret all situation-independent predicate and function symbols identically. (2) \(M\) and \(M'\) agree on all fluents at \(S_{s_0}\): For every relational fluent \(F\), and every variable assignment \(\sigma\), \(M, \sigma \models F(\bar{x}, S_0) \iff M', \sigma \models F(\bar{x}, S_0)\).

**Definition 3** Let \(D_{S_0}\) be a set of sentences uniform in \(S_{s_0}\).

\(D_{S_0}\) is a progression of the initial KB \(D_{S_0}\) wrt \(\alpha\) if for any structure \(M, M\) is a model of \(D_{S_0}\) iff \(M\) is a model of \(D\) such that \(M \sim S_{s_0} M'\).

Lin and Reiter showed that progression is not first-order definable in general. Recently, Liu and Lakemeyer (2009) showed that for local-effect actions, progression is always first-order definable and computable. Actions in many dynamic domains have only local effects in the sense that if an action \(A(\bar{e})\) changes the truth value of an atom \(F(\bar{d}, s)\), then \(\bar{d}\) is contained in \(\bar{e}\). This contrasts with actions having non-local effects such as moving a briefcase, which will also move all the objects inside the briefcase without having mentioned them. We skip the formal definition here. In fact, any action with bounded effects, which means that its execution will only change the truth values of a bounded number of fluent atoms, can be put into a local-effect action by including more action arguments. Hence the local-effect restriction is a reasonable one, but it may cause some inconvenience in domain encoding.

The proof of (Liu and Lakemeyer 2009) is a very simple one via the concept of forgetting (Lin and Reiter 1994).

**Definition 4** Let \(p\) be a ground atom. Let \(M_1\) and \(M_2\) be two structures. We write \(M_1 \sim_p M_2\) if \(M_1\) and \(M_2\) agree on everything except possibly on the interpretation of \(p\).

**Definition 5** Let \(T\) be a theory, and \(p\) a ground atom. A theory \(T'\) is a result of forgetting \(p\) in \(T\), if for any structure \(M, M \models T'\) iff there is a model \(M'\) of \(T\) s.t. \(M \sim_p M'\).

Clearly, if both \(T'\) and \(T''\) are results of forgetting \(p\) in \(T\), then they are logically equivalent. We use forget\((T, p)\) to denote the result of forgetting \(p\) in \(T\). Lin and Reiter showed that for any finite theory \(T\) and atom \(p\), forgetting \(p\) in \(T\) is
first-order definable and can be obtained from $T$ and $p$ by simple syntactic manipulations.

**Theorem 1** Let $\mathcal{D}$ be local-effect and $\alpha = A(\vec{c})$ a ground action. We use $\Omega(s)$ to denote the set of $F(\vec{a}, s)$ where $F$ is a fluent, and $\vec{a}$ is contained in $\vec{c}$. Then the following is a progression of $\mathcal{D}_{S_0}$ wrt $\alpha$:

\[
\text{forget}(\mathcal{D}_{S_0} \cup \mathcal{D}_{\varphi}[\Omega], \Omega(S_0)/(S_0/S_{\alpha})),
\]

where $\mathcal{D}_{\varphi}[\Omega]$ is the instantiation of $\mathcal{D}_{\varphi}$ wrt $\Omega$.

We now extend the notion of progression to accommodate sensing actions. For simplicity, for each ground sensing action $\alpha$, we introduce two auxiliary actions $\alpha_T$ and $\alpha_F$, which represent $\alpha$ with sensing results true and false, respectively. We use the notation $(\neg)\phi$ to denote $\phi$ if the sensing result is true, and $\neg\phi$ otherwise.

**Definition 6** Let $\alpha$ be a ground sensing action and $\mu \in \{T, F\}$. Let $\mathcal{D}_{S_0}$ be a set of sentences uniform in $\alpha$. $\mathcal{D}_{S_0}$ is a progression of $\mathcal{D}_{S_0}$ wrt $\alpha_\mu$ if for any structure $M$, $M$ is a model of $\mathcal{D}_{S_0}$ iff there is a model $M'$ of $\mathcal{D} \cup \{(-)S\alpha(S_0)\}$ s.t. $M \models S_{\alpha_\mu}$.

It is easy to prove the following:

**Theorem 2** For a ground sensing action $\alpha = A(\vec{c})$, $\{D_{S_0} \cup \{(-)\Psi_A(\vec{c}, S_0)\}\}(S_0/S_\alpha)$ is a progression of $\mathcal{D}_{S_0}$ wrt $\alpha_\mu$.

We say that $D_\varphi$ is quantifier-free if for each action function $\alpha(\vec{x})$, $\Psi_A$ is quantifier-free. We say that $D_{S_\varphi}$ is essentially quantifier-free if for each action function $\alpha(\vec{x})$ and each fluent $F$, by using $D_{\alpha \varphi} \Psi_F(\vec{x}, \alpha(\vec{x}), s)$, the resulting formula can be simplified to a quantifier-free formula. The following theorem follows from Theorem 5.13 of (Liu and Lakemeyer 2009).

**Definition 7** A well-formed basic action theory is a local-effect one such that $D_{S_\varphi}$ is essentially quantifier-free, $D_\varphi$ is quantifier-free, and $D_{S_\varphi}$ is proper.$^+$. 

**Theorem 3** Suppose that $\mathcal{D}$ is well-formed. Then progression of $\mathcal{D}_{S_0}$ wrt any ground action is definable as a proper$^+$ KB and can be efficiently computed.

We use $\text{prog}(\mathcal{D}_{S_0}, \alpha)$ to denote a proper$^+$ KB which is a progression of $\mathcal{D}_{S_0}$ wrt $\alpha$. It is straightforward to generalize the notation to $\text{prog}(\mathcal{D}_{S_0}, \sigma)$, where $\sigma$ is a ground situation. We also use $D_\sigma$ to denote $\text{prog}(\mathcal{D}_{S_0}, \sigma)$.

**The Subjective Logic $\mathcal{SL}$**

With the goal of specifying a reasoning service for first-order KBs with disjunctive information in the form of proper$^+$ KBs, (Liu, Lakemeyer, and Levesque 2004), later referred to by (LLL04), proposed a logic of limited belief called the subjective logic $\mathcal{SL}$. Reasoning based on $\mathcal{SL}$ is logically sound and sometimes complete. Given disjunctive information, it performs unit propagation, but only does case analysis in a limited way.

The language $\mathcal{SL}$ is a first-order logic with equality whose atomic formulas are belief atoms of form $B_k(\phi)$ where $\phi \in \mathcal{L}$ and $B_k$ is a modal operator for any $k \geq 0$. $B_k(\phi)$ is read as "$\phi$ is a belief at level $k$". We let $\mathcal{SL}_k$ denote the set of $\mathcal{SL}$-formulas whose only modal operators are $B_j$ for $j \leq k$. We call formulas of $\mathcal{L}$ objective, and formulas of $\mathcal{SL}$ subjective.

Let $s$ be a set of ground clauses. The notation $\text{UP}(s)$ is used to denote the closure of $s$ under unit propagation, that is, the least set $s'$ satisfying: 1. $s \subseteq s'$; and 2. if a literal $\rho \in s'$ and $\{\neg \psi \}$ $\in \mathcal{E}$, then $\psi \in s'$. Let $\phi \in \mathcal{L}$. The notation $(B_k \phi \bot)$, called belief reduction, is defined as follows:

1. $(B_k \phi \bot) = B_k(\phi)$, where $c$ is a clause;
2. $(B_k \epsilon \bot) = \epsilon$, where $c$ is an equality literal;
3. $(B_k \neg \phi \bot) = B_k \neg \phi$;
4. $(B_k (\phi \lor \psi) \bot) = (B_k \phi \lor B_k \psi)$, where $\phi$ or $\psi$ is not a clause; and $(B_k \neg (\phi \lor \psi) \bot) = (B_k \neg \phi \land B_k \neg \psi)$;
5. $(B_k \exists x \phi \bot) = \exists x B_k \phi$; and $(B_k \neg \exists x \phi \bot) = \forall x B_k \neg \phi$.

A setup is a set of non-empty ground clauses. For any setup $s$ and $\phi \in \mathcal{SL}$, $s \models \phi$ is defined inductively as follows:

1. $s \models (d = d')$ iff $d$ and $d'$ are the same constant;
2. $s \models \neg \phi$ iff $s \not\models \phi$;
3. $s \models \phi \lor \psi$ iff $s \models \phi$ or $s \models \psi$;
4. $s \models \exists x \phi$ iff for some constant $d$, $s \models \phi_d$;
5. $s \models B_k \phi$ iff one of the following holds:
   a. subsume: $k = 0$, $\phi$ is a clause $c$, and there is $c' \in \text{UP}(s)$ s.t. $c' \subseteq c$;
   b. reduce: $\phi$ is not a clause and $s \models (B_k \phi) \bot$;
   c. split: $k > 0$ and there is $c \in s$ s.t. for all $\rho \in c$, $s \models \{\rho\} \models B_k \neg \phi$.

A set $\Gamma$ of sentences entails a sentence $\phi$, written $\Gamma \models \phi$, if every setup $s$ satisfying every sentence of $\Gamma$ also satisfies $\phi$.

Reiter (2001b) introduced the closed-world assumption on knowledge that a given $K$ of axioms about what an agent knows captures everything that the agent knows; any knowledge sentences not following logically from $K$ are taken to be false. This assumption relieves the axiomatizer from having to figure out the relevant lack of knowledge axioms when given what the agent does know. Let $\Sigma$ be a proper$^+$ KB. We adapt Reiter’s idea to $\mathcal{SL}$ as follows:

**Definition 8** bcl$(B_0 \Sigma) \overset{\text{def}}{=} B_0 \Sigma \cup \neg B_0 \phi \mid B_0 \Sigma \not\models B_k \phi$.

By an important result from (LLL04) that $B_0 \Sigma \models B_k \phi$ iff $\text{gnd}(\Sigma) \models B_k \phi$, we have bcl$(B_0 \Sigma) = B_0 \Sigma \cup \neg B_0 \phi \mid \text{gnd}(\Sigma) \not\models B_k \phi$. Thus it is easy to prove

**Theorem 4** Let $\Sigma$ be a proper$^+$ KB. Then

1. $\text{bcl}(B_0 \Sigma)$ is satisfiable.
2. For any $\phi \in \mathcal{SL}$, $\text{bcl}(B_0 \Sigma) \models \phi$ or $\text{bcl}(B_0 \Sigma) \not\models \phi$.
3. For any $\phi \in \mathcal{SL}$, $\text{gnd}(\Sigma) \models \phi$ iff $\text{bcl}(B_0 \Sigma) \models \phi$.

Finally, we relate reasoning about subjective formulas to reasoning about objective formulas. Let $\phi \in \mathcal{SL}$. We define its objective formula, denoted by $\varphi_o$, as the formula obtained from $\phi$ by replacing each belief atom $B_k \phi$ with $\phi$. A proper$^+$ KB $\Sigma$ is proper if $\text{gnd}(\Sigma)$ is a consistent set of ground literals. It is easy to show that when a proper KB $\Sigma$ is complete, $\text{gnd}(\Sigma) \models \varphi$ iff $\Sigma \models \varphi_o$. Thus

**Theorem 5** Let $\Sigma$ be a complete proper KB. Then $\text{bcl}(B_0 \Sigma) \models \varphi$ iff $\Sigma \models \varphi_o$. 

LBGolog: syntax and semantics

The following are the programming constructs of LBGolog. A difference with normal Golog is that all tests \( \phi \) are SC0 formulas. For readability, we also write \( B_0(\psi) \) as \( \text{Knows}(\psi) \).

1. \( \alpha \): primitive action
2. \( \phi \): test action
3. \( (\delta_1 ; \delta_2) \): sequence
4. \( (\begin{array}{c} \phi \\rightarrow \delta \end{array}) \): nondeterministic choice of actions
5. \( (\pi \cdot x . \phi \rightarrow \delta) \): guarded nondet. choice of arguments
6. \( \delta^* \): nondeterministic iteration
7. \( \text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endif} \): conditional
8. \( \phi \; \text{do } \delta \text{ endwhile} \): while loop
9. \( P(x) \; \text{endProc} \): procedure definition
10. \( P(\bar{c}) \): procedure call
11. \( \Sigma \delta \): search operator
12. \( \Upsilon(\tau, \delta) \): planning operator

The first 10 constructs, called the basic constructs, are the same as those of Golog except that here we have guarded nondeterministic choice of arguments \( \pi \cdot x . \phi \rightarrow \delta \), where any variable of \( x \) must appear in \( \phi \), and it is executed by nondeterministically picking \( x \) such that \( \phi(x) \) holds and then performing \( \delta(x) \). We call a program basic if it uses only basic constructs. As in IndiGolog, there is a search operator \( \Sigma \delta \), where \( \delta \) is a basic program, which specifies that lookahead should be performed over \( \delta \) to ensure that nondeterministic choices are resolved in a way that guarantees its successful completion. We allow \( \delta \) in \( \Sigma \delta \) to use sensing actions, so the search operator returns a conditional plan, where branchings are conditioned on the results of sensing actions.

In addition, there is a planning operator \( \Upsilon(\tau, \delta) \), where \( \tau \) is a type predicate with a finite domain, that is, the initial KB \( D_S_0 \) contains a sentence of the form \( \forall x . \tau(x) \equiv x = d_1 \lor \ldots \lor x = d_n \), where \( d_1, \ldots, d_n \) are constants. Any use of \( \Upsilon(\tau, \delta) \) must satisfy the following restrictions: every constant appearing in \( \delta \) is of type \( \tau \), \( \delta \) is a basic program without sensing actions, and \( \delta \) does not support procedural calls other than the simple case that \( \delta \) is a procedural call itself. \( \Upsilon(\tau, \delta) \) is executed by calling a state-of-the-art planner to generate a sequence of actions constituting a legal execution of program \( \delta \) where objects are restricted to elements of type \( \tau \). Thus we require that when executing \( \Upsilon(\tau, \delta) \), the agent should have complete knowledge regarding the execution of \( \delta \) restricted to \( \tau \). We will formalize this requirement when we present the implementation of the planning operator.

From now on, we restrict our attention to well-formed BATs. Following (Claßen and Lakemeyer 2009), the formal semantics we present here is an adaptation of the single-step transition semantics of (De Giacomo, Lespérance, and Levesque 2000). A central concept is that of a configuration, denoted as a pair \( (\delta, \sigma) \), where \( \delta \) is a program (that remains to be executed) and \( \sigma \) a situation (of actions that have been performed). A configuration can be final, i.e., the run can successfully terminate in that situation, or it can make certain transitions to other configurations. Our semantics is based on progress, SL-based limited reasoning, and closed-world assumption on knowledge: when we evaluate a test \( \phi \) wrt a configuration \( (\delta, \sigma) \), we check if \( \text{bcl}(B_0 D_\sigma) \models \phi[\sigma] \), where \( D_\sigma \) denotes \( \text{proq}(D_{S_0}, \sigma) \). Note that \( \phi \) is a situation-suppressed formula, and \( \phi[\sigma] \) denotes the formula obtained from \( \phi \) by taking \( \sigma \) as the situation arguments of all fluents.

For lack of space, we leave out semantics of procedures. Conditional and loops are defined as abbreviations:

- \( \text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endif} \) defines \( \phi \equiv [\phi?; \delta_1] \cup [\neg \phi?; \delta_2] \),
- \( \phi \; \text{do } \delta \text{ endwhile} \) defines \( \phi \equiv [\phi?; \delta^*; \neg \phi?] \).

We first give the semantics for basic constructs. The set of final configurations wrt \( D \), denoted \( F_D \) (in the sequel, we often omit the \( D \) subscript), is inductively defined as follows:

1. \( (nil, \sigma) \in F_D \).
2. \( (\phi?, \sigma) \in F_D \) if \( \text{bcl}(B_0 D_\sigma) \models \phi[\sigma] \).
3. \( (\delta_1; \delta_2, \sigma) \in F_D \) if \( (\delta_1, \sigma) \in F_D \) and \( (\delta_2, \sigma) \in F_D \).
4. \( (\delta_1 | \delta_2, \sigma) \in F_D \) if \( (\delta_1, \sigma) \in F_D \) or \( (\delta_2, \sigma) \in F_D \).
5. \( (\pi \cdot x . \phi \rightarrow \delta, \sigma) \in F_D \) if there exist constants \( \bar{c} \) such that \( \text{bcl}(B_0 D_\sigma) \models \phi(\bar{c})[\sigma] \) and \( (\delta(\bar{c}), \sigma) \in F_D \).
6. \( (\delta^*, \sigma) \in F_D \).

The transition relation between configurations wrt \( D \), denoted \( \rightarrow_D \), is inductively defined as follows:

1. \( (\alpha, \sigma) \rightarrow (nil, \text{do}(\alpha, \sigma)) \) if \( \alpha = A(\bar{c}) \) is an ordinary primitive action and \( \text{bcl}(B_0 D_\sigma) \models B_0 \Pi_A(\bar{c}, \sigma) \).
2. \( (\alpha, \sigma) \rightarrow (nil, \text{do}(\alpha, \sigma)) \) if \( \alpha = A(\bar{c}) \) is a sensing action with result \( \mu \), and \( \text{bcl}(B_0 D_\sigma) \models B_0 \Pi_A(\bar{c}, \sigma) \).
3. \( (\delta_1; \delta_2, \sigma) \rightarrow (\gamma; \delta_2, \sigma') \) if \( (\delta_1, \sigma) \in F_D \) and \( (\delta_2, \sigma') \in F_D \).
4. \( (\delta_1 | \delta_2, \sigma) \rightarrow (\gamma, \sigma') \) if \( (\delta_1, \sigma) \in F_D \) and \( (\delta_2, \sigma') \in F_D \).
5. \( (\pi \cdot x . \phi \rightarrow \delta, \sigma) \rightarrow (\gamma, \sigma') \) if \( (\delta_1, \sigma) \rightarrow (\gamma, \sigma') \) or \( (\delta_2, \sigma) \rightarrow (\gamma, \sigma') \).
6. \( (\pi \cdot x . \phi \rightarrow \delta, \sigma) \rightarrow (\gamma, \sigma') \) if there exist constants \( \bar{c} \) s.t. \( \text{bcl}(B_0 D_\sigma) \models \phi(\bar{c})[\sigma] \) and \( (\delta(\bar{c}), \sigma) \rightarrow (\gamma, \sigma') \).

To define the semantics of search and planning operators, we define a relation \( C \): intuitively, \( (\delta, \sigma, \rho) \in C \) means an offline execution of program \( \delta \) in situation \( \sigma \) results in conditional plan \( \rho \). To define \( C \), we introduce an auxiliary relation \( E \): intuitively, \( (\rho, \delta, \sigma, \rho') \in E \) means in situation \( \sigma \), executing conditional plan \( \rho \) and then program \( \delta \) leads to conditional plan \( \rho' \). One might wonder why we need \( \rho, \delta, \sigma, \rho' \in E \) instead of simply \( \rho; \delta, \sigma, \rho' \in C \). The reason is that we would like to define \( \rho, \delta, \sigma, \rho' \in E \) by induction on \( \rho \). Formally, \( C \) is defined as follows:

1. \( (nil, \sigma, nil) \in C \).
2. \( (\alpha, \sigma, \alpha) \in C \) if \( \alpha = A(\bar{c}) \) and \( \text{bcl}(B_0 D_\sigma) \models B_0 \Pi_A(\bar{c}, \sigma) \).
3. \( (\phi?, \sigma, nil) \in C \) if \( \text{bcl}(B_0 D_\sigma) \models \phi[\sigma] \).
4. \( (\delta_1; \delta_2, \sigma, \rho) \in C \) if there exists \( \rho' \) such that \( (\delta_1, \sigma, \rho') \in C \) and \( (\rho', \delta_2, \sigma, \rho) \in E \).
5. \( (\delta_1 | \delta_2, \sigma, \rho) \in C \) if \( (\delta_1, \sigma, \rho) \in C \) or \( (\delta_2, \sigma, \rho) \in C \).
6. \( (\pi \cdot x . \phi \rightarrow \delta, \sigma, \rho) \in C \) if there exist constants \( \bar{c} \) such that \( \text{bcl}(B_0 D_\sigma) \models \phi(\bar{c})[\sigma] \) and \( (\delta(\bar{c}), \sigma) \in C \).
7. \( (\delta^*, \sigma, nil) \in C \); and \( (\delta^*, \sigma, \rho) \in C \) if there exists \( \rho' \) such that \( (\delta, \sigma, \rho') \in C \) and \( (\rho', \delta^*, \sigma) \in E \).

The formal definition of \( E \) is as follows:

1. \( (nil, \delta, \sigma, \rho') \in E \) if \( (\delta, \sigma, \rho') \in C \).
2. \((\alpha; \rho, \delta, \sigma, \alpha; \rho') \in E\) if \(\alpha\) is an ordinary primitive action and \((\rho, \delta, \text{do}(\alpha, \sigma), \rho') \in E\).

3. \((\alpha; \rho, \delta, \sigma, \rho') \in E\) if \(\alpha\) is a sensing action and there exist \(p_1 \) and \(p_2\) such that \(\langle \rho, \delta, \text{do}(\alpha; \sigma), p_1 \rangle \in E\) and \(\langle \rho, \delta, \text{do}(\alpha; \sigma), p_2 \rangle \in E\) and \(\rho'\) is: \(\alpha\) if \(\text{Knows}(\Psi_A(\mathbf{c}))\) then \(p_1\) else \(p_2\) endif.

4. \((\phi \text{ then } p_1 \text{ else } p_2 \text{ endif}, \delta, \sigma, \rho) \in E\) if the following holds: if \(\mathbf{c} = (\mathbf{B}_0D_\sigma) = \phi[\sigma]\), then \(\langle p_1, \delta, \sigma, \rho \rangle \in E\), otherwise \(\langle p_2, \delta, \sigma, \rho \rangle \in E\).

To define the semantics of \(\Upsilon(\tau, \delta)\), we define the restriction of \(\delta\) to \(\tau\), denoted by \(\delta_\tau\). Intuitively, \(\delta_\tau\) is \(\delta\) where objects are restricted to elements of type \(\tau\). Formally, \(\delta_\tau\) is obtained from \(\delta\) as follows: replace each formula of the form \(\forall x \phi\) with \(\forall x.\text{Knows}(\tau) \supset \phi\), and \(\exists x.\text{Knows}(\tau) \land \phi \rightarrow \delta\).

We can now expand the definition of \(\Upsilon\) as follows:

8. \(\langle \Sigma, \delta, \sigma \rangle \rightarrow (\rho, \sigma) \in C\).

9. \(\langle \Upsilon(\tau, \delta), \sigma \rangle \rightarrow (\rho, \sigma) \text{ if } (\tau, \delta, \sigma, \rho) \in C\).

Finally, an online execution of an \(LBGolog\) program \(\delta_0\) starting from a situation \(\sigma_0\) is a sequence of configurations \((\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)\), s.t. for \(i < n\), \((\delta_i, \sigma_i) \rightarrow (\delta_{i+1}, \sigma_{i+1})\). The execution is successful if \((\delta_n, \sigma_n) \in F\).

We now illustrate programming in \(LBGolog\) with the Wumpus world. We assume there is only one piece of gold. When writing the control program, we take a cautious strategy and ensure that the agent keeps alive. Below is the main program, where \(n_1\) is a constant for coordinate 1, and \(\text{coor}(x)\) is the coordinate type predicate. The agent first senses the environment. If she knows that the gold is at location \((1, 1)\), she grabs the gold, otherwise she explores the dungeon. Then she climbs out, or she moves to location \((1, 1)\) by use of the planning operator, and climbs out.

\[
\begin{align*}
\text{proc main} & \quad \text{sense_stench; sense_breeze; sense_gold;}
\quad \text{if Knows}(\text{gold}(n_1, n_1)) \text{ then grab}
\quad \text{else explore endif}
\quad (\text{climb} \mid \Upsilon(\text{coor}, \text{moveLoc}(n_1, n_1)); \text{climb})
\end{align*}
\]

endProc

The following procedure moves to location \((X, Y)\) by traversing only visited locations. Here \(\text{agt}(x, y)\) means that the agent is at location \((x, y)\).

\[
\begin{align*}
\text{proc moveLoc}(X, Y) & \quad [\pi x_0, y_0, x_1, y_1. \text{Knows}(\text{agt}(x_0, y_0) \land \text{explored}(x_1, y_1)) \\
& \rightarrow \text{move}(x_0, y_0, x_1, y_1)]^*; \\
\pi x_2, y_2. \text{Knows}(\text{agt}(x_2, y_2)) & \rightarrow \text{move}(x_2, y_2, X, Y)
\end{align*}
\]

endProc

The procedure below explores the dungeon. Here \(\text{wp}(x, y)\) means that the wumpus is at location \((x, y)\), and \(\text{pit}(x, y)\) means that there is a pit at location \((x, y)\). While the agent knows she has not got the gold, she picks an unvisited safe location, moves there, and senses the environment. If she knows the gold is at her location, she grabs the gold. Otherwise, if she knows that the wumpus is alive and she knows the location of the wumpus, she shoots the wumpus. We omit the procedure for shooting the wumpus.

\[
\begin{align*}
\text{proc explore} & \quad \text{while Knows}(\neg \text{getGold}) \land \\
& \exists x, y. \text{Knows}(\neg \text{wp}(x, y) \lor \neg \text{wpAlive} \land \neg \text{pit}(x, y)) \land \\
& \neg \text{Knows}(\text{explored}(x, y)) \text{ do}
\quad \pi x, y. \text{Knows}(\neg \text{wp}(x, y) \lor \neg \text{wpAlive} \land \neg \text{pit}(x, y)) \land \\
& \neg \text{Knows}(\text{explored}(x, y)) \rightarrow
\quad \Upsilon(\text{coor}, \text{moveLoc}(x, y)); \\
& \text{sense_stench; sense_breeze; sense_gold;}
\quad \text{if Knows}(\exists x_0, y_0. \text{agt}(x_0, y_0) \land \text{gold}(x_0, y_0)) \text{ then grab else}
\quad \text{if } \exists x_1, y_1. \text{Knows}(\text{wpAlive} \land \text{wp}(x_1, y_1)) \text{ then shootWumpus endif endif endWhile}
\end{align*}
\]

Implementing progression and query evaluation by grounding

To implement \(LBGolog\), we need to implement progression and evaluation of an \(SC\) formula against the closure of \(B_0D_\sigma\), where \(D_\sigma\) is the current KB. Initially, we implemented the progression and query evaluation algorithms by Liu, Lakemeyer and Levesque. However, the implementations were not efficient. So we decided to implement them by grounding. But we have infinitely many unique names. The trick is to use an appropriate number of them as representatives of those not mentioned by the KB. The general picture is this. We first ground the initial KB, perform unit propagation on it. When an action is performed, if it mentions new constants, we extend the current ground KB with these constants, then progress the ground KB, and do unit propagation on it. Whenever we need to evaluate a query, we use the current ground KB to answer the query. Proofs for results in this section are straightforward, and hence omitted.

Grounding

We define the width of a proper \(KB\) \(\Sigma\) as the maximum number of distinct variables in a \(\forall\)-clause of \(\Sigma\). Let \(j\) be the width of \(\Sigma\). For simplicity, we assume that there are a set \(U\) of \(j\) reserved constants \(u_1, \ldots, u_j\); they do not appear in the initial KB and will not be mentioned by any action. We call constants not in \(U\) normal constants. For a set \(\Gamma\) of formulas, we use \(H(\Gamma)\) to denote the set of normal constants appearing in \(\Gamma\), and let \(H^+(\Gamma)\) represent \(H(\Gamma)\) extended with a normal constant not appearing in \(\Gamma\).

Definition 9 Let \(\Sigma\) be a proper \(KB\) with width \(j\). Let \(N\) be a set of normal constants containing those appearing in \(\Sigma\). We define \(\text{prop}(\Sigma, N)\) as the set of those clauses of \(\text{gnd}(\Sigma)\) which uses only constants from \(N\) or \(U\). We use \(\text{upprop}(\Sigma, N)\) to represent \(UP(\text{prop}(\Sigma, N))\).

Example 1 Consider a simple proper \(KB\) \(\Sigma = \{\forall x. \neg \text{on}(x, A), \forall x, y, x \neq y \supset \neg \text{on}(x, y) \lor \neg \text{clear}(y)\}\). Then \(\Sigma\) has width 2, and \(U = \{u_1, u_2\}\). Let \(N = \{A\}. \) So \(\text{prop}(\Sigma, N) = \{\neg \text{on}(A, A), \neg \text{on}(u_1, A), \neg \text{on}(u_2, A), \neg \text{on}(A, u_1) \lor \neg \text{clear}(u_1), \neg \text{on}(A, u_2) \lor \neg \text{clear}(u_2), \neg \text{on}(u_1, A) \lor \neg \text{clear}(A), \neg \text{on}(u_1, u_2) \lor \neg \text{clear}(u_2), \neg \text{on}(u_2, A) \lor \neg \text{clear}(A), \neg \text{on}(u_2, u_1) \lor \neg \text{clear}(u_1)\}\).
In the sequel, we let $\Sigma_p$ denote a ground proper $+$ KB with U constants. To prove correctness of grounding, we first define the first-order KB represented by $\Sigma_p$.

**Definition 10** We define $FO(\Sigma_p)$ as follows: replace each $c$ in $\Sigma_p$ with $FO(c)$, denoting $\forall(c \supset c(u_1/x_1, \ldots, u_j/x_j))$, where $c$ is the eflf $\bigwedge_{i=1}^k x_i \notin H(\Sigma_p) \wedge \bigwedge_{i=k}^l x_i \neq x_k$, and $x \notin N$ is the abbreviation for $\bigwedge_{d\in N} x \neq d$.

**Theorem 6** $FO(prop(\Sigma, N)) \equiv \exists_\Sigma$.

We now define extended grounding and show its correctness.

**Definition 11** Let $B$ be a finite set of normal constants not occurring in $\Sigma_p$. We define $egnd(\Sigma_p, B)$ inductively:
1. $egnd(\Sigma_p, \emptyset) = \Sigma_p$;
2. $egnd(\Sigma_p, \{d\}) = \Sigma_p \cup \{c(u_k/d) \mid c \in \Sigma_p, 1 \leq k \leq j\}$;
3. $egnd(\Sigma_p, \{d\} \cup B) = egnd(egnd(\Sigma_p, \{d\}), B)$.

**Theorem 7** $egnd(prop(\Sigma, N), B) \equiv prop(\Sigma, N \cup B)$.

Note that the way we do grounding is brute-force. For example, if $\Sigma$ contains $\forall x P(x)$, then its ground KB contains $P(u_1), \ldots, P(u_j)$, each of which carries the same information. However, brute-force grounding will facilitate later progression operation.

**Progression**

We now define progression of a ground KB, and show its equivalence to progression of the original KB.

**Definition 12** Let $D$ be a well-formed BAT, and $\alpha = A(\bar{c})$ a ground action. Let $B$ be the set of constants appearing in $\bar{c}$ but not in $\Sigma_p$. We define $pprog(\Sigma_p, \alpha)$ as

$$forget(egnd(\Sigma_p, B) \cup D_{ss}(\bar{\Omega}), \Omega(S_0))(S_0/S_0),$$

if $\alpha$ is an ordinary primitive action, and $\Sigma_p(S_0/S_0) \cup \{\neg A(\bar{c}, S_0)\}$ if $\alpha$ is a sensing action. We use $upprog(\Sigma_p, \alpha)$ to represent $UP(pprog(\Sigma_p, \alpha))$.

Forgetting a ground atom $q$ from a set of ground clauses can be done by computing all resolvents wrt $q$ and then removing all clauses containing $q$. We generalize the notation to $pprog(\Sigma_p, \sigma)$ and $upprog(\Sigma_p, \sigma)$, where $\sigma$ is a ground situation. The lemma below establishes connection between forgetting a ground atom from a proper $+$ KB and from its ground KB.

**Lemma 8** Let $q$ be a ground atom. Let $N$ be a set of normal constants containing those that appear in $\Sigma$ or $q$. Then $forget(\Sigma, q) \equiv FO(forget(prop(\Sigma, N), q))$.

By Theorems 1, 6, 7 and Lemma 8, we have

**Theorem 9** $FO(pprog(prop(\Sigma, N), \alpha)) \equiv\exists_\Sigma\ prog(\Sigma, \alpha)$.

**Query Evaluation**

We say a query $\phi$ is suitable for $\Sigma_p$ if for each clause $c$ in $\phi$, the number of variables in $c$ and constants in $c$ but not $\Sigma_p$ is no more than that of $U$ constants in $\Sigma_p$.

We first define an evaluation procedure $G[\Sigma_p, \phi]$ where $\phi \in L$ is suitable for $\Sigma_p$. It is the same as the $W[\Sigma, k, \phi]$ procedure from (LLL04) to decide if $B_0 \Sigma \models B_k \phi$ where $k = 0$ except for the case of evaluating clauses. $G[\Sigma_p, \phi] = 1$ if one of the following conditions holds, and 0 otherwise.

1. $\phi$ is a clause $c$ and there exists a clause $c' \in \Sigma_p$ such that $c' \subseteq c(d_1/u_1, \ldots, d_k/u_k)$, where $\{d_1, \ldots, d_k\}$ is the set of normal constants that appear in $c$ but not $\Sigma_p$.
2. $\phi = (d = d')$ and $d$ is identical to $d'$.
3. $\phi = (d \neq d')$ and $d$ is distinct from $d'$.
4. $\phi = \neg\psi$ and $G[\Sigma_p, \psi] = 1$.
5. $\phi = (\psi \lor \eta)$, $\psi$ or $\eta$ is not a clause, and $G[\Sigma_p, \psi] = 1$ or $G[\Sigma_p, \eta] = 1$.
6. $\phi = (\psi \lor \eta)$, $G[\Sigma_p, \psi] = 1$ and $G[\Sigma_p, \eta] = 1$.
7. $\phi = \exists x \psi$ and $G[\Sigma_p, \psi_d] = 1$ for some $d \in H^+(\Sigma_p \cup \{\psi\})$.
8. $\phi = \exists x \psi$ and $G[\Sigma_p, \neg\psi_d]$ for all $d \in H^+(\Sigma_p \cup \{\psi\})$.

Based on $G$, we now define an evaluation procedure $F[\Sigma_p, \varphi]$ where $\varphi \in SS_0$ is suitable for $\Sigma_p$. $F[\Sigma_p, \varphi] = 1$ if one of the following conditions holds, and 0 otherwise.

1. $\varphi = B_0\phi$ and $G[\Sigma_p, \phi] = 1$.
2. $\varphi = (t_1 = t_2)$ and $t_1$ is identical to $t_2$.
3. $\varphi = \neg\omega$ and $F[\Sigma_p, \omega] = 0$.
4. $\varphi = \varphi_1 \lor \varphi_2$, and $F[\Sigma_p, \varphi_1] = 1$ or $F[\Sigma_p, \varphi_2] = 1$.
5. $\varphi = \exists x \omega$, and $F[\Sigma_p, \omega_d] = 1$ for some $d \in H^+(\Sigma_p \cup \{\omega\})$.

By exploiting that $U$ constants serve as representatives of constants not appearing in $\Sigma_p$, we can prove

**Lemma 10** $F[UP(\Sigma_p), \varphi] = 1$ iff $gnd(FO(\Sigma_p)) \models \varphi$.

By Lemma 10 and Theorem 4(3), we have

**Theorem 11** $F[UP(\Sigma_p), \varphi] = 1$ iff $bcl(B_0 FO(\Sigma_p)) \models \varphi$.

We now obtain the main conclusion of this section, which shows the correctness of our implementation of progression and query evaluation by grounding:

**Theorem 12** $F[upprog(upprop(DS_0, H(DS_0)), \sigma), \varphi] = 1$ iff $bcl(B_0 upprog(DS_0, \sigma)) \models \varphi$, where $\sigma$ is a situation.

**An interpreter**

We have implemented an interpreter for LBGolog in Prolog. We assume the user provides the following set of clauses corresponding to the background basic action theory:

- $init_kb(l)$: $l$ is a list of $\forall$-clauses of the initial KB;
- $poss(\alpha, \sigma, \phi)$: formula $\phi$ is the precondition for action $\alpha$ in situation $\sigma$;
- $ssa(F, \gamma^+, \gamma^-)$: $\gamma^+$ (resp. $\gamma^-$) is the condition for making fluent $F$ true (resp. false);
- $st(\beta, \sigma, \phi)$: $\beta$ senses if formula $\phi$ holds in situation $\sigma$.

To improve efficiency, we implement the core parts of progression and query evaluation operations in C, and provide the following primitive predicates in Prolog:

- $bool_query(\phi, \sigma)$: $\phi$ is evaluated true in situation $\sigma$;
- $open_query(F, \phi, \sigma, \bar{c})$: formula $\phi\bar{c}$ is evaluated true in situation $\sigma$;
- $prim_prog(\alpha, \sigma, \sigma')$: progress the KB of situation $\sigma$ to situation $\sigma'$ wrt ordinary primitive action $\alpha$. 
• \( \text{sens\_prog}(\beta, \mu, \sigma, \sigma') \): progress the KB of situation \( \sigma \)
to situation \( \sigma' \) wrt sensing action \( \beta \) with sensing result \( \mu \);
• \( \text{del\_sit}(\sigma) \): delete the KB about situation \( \sigma \).

The two progression operators yield the new KBs while keeping the old ones. Thus we need the \text{del\_sit} predicate.

The top part of the interpreter uses \text{btrans} and \text{bfinal} to determine the next action to perform or to terminate.

\[
lb\text{Golog}(E, S) :- \text{bfinal}(E, S), !.
\]
\[
lb\text{Golog}(E, S) :- \text{btrans}(E, S, E_1, S_1), !.
\]

\text{bfinal}(E, S) :- \text{bfinal}(E, S_1).

\[
\text{bfinal}(E, S) :- \text{bfinal}(E_1, S) \\
\text{bfinal}(E, S) :- \text{bfinal}(E_2, S).
\]

\[
\text{bfinal}(E, S) :- \text{bfinal}(E_1, S), \text{bfinal}(E_2, S).
\]

\[
\text{bfinal}(\text{star}(E), S) :- \text{btrans}(\text{star}(E), S, E_1, S_1).
\]

\[
\text{bfinal}(\text{star}(E), S) :- \text{btrans}(\text{star}(E), S, E, S_1).
\]

\[
\text{bfinal}(\text{star}(E), S) :- \text{bfinal}(\text{star}(E), S_1).
\]

Basic constructs

We define predicates \( \text{bfinal}/2 \) and \( \text{btrans}/4 \) to implement the \( F \) and \( \rightarrow \) relations. The Prolog syntax for the constructs are: \( E_1 \# E_2 \) for sequence, \( E_1 \# E_2 \) for nondet. choice; \( \text{pi}(L, G, E) \) for guarded nondet. choice of arguments, \( \text{star}(E) \) for nondet. iteration, \( A \) for primitive action, and \( B \) for sensing action. For illustration, we present only some of the clauses. We use predicate \( \text{sub\_arg}(I_x, I_c, p_x, p_c) \) to substitute all variables of \( I_c \) occurring in program \( p_x \) with corresponding constants of \( I_c \), resulting in program \( p_c \).

Predicate \( \text{po}(\alpha, \sigma, \phi') \) (resp. \( \text{sfns}(\alpha, \sigma, \phi') \)) holds iff \( \text{poss}(\alpha, \sigma, \phi) \) (resp. \( \text{sf}(\alpha, \sigma, \phi) \)) holds and \( \phi' \) is \( \phi \) with situation arguments suppressed.

\[
\text{bfinal}((P), S) :- \text{bool\_query}(P, S).
\]
\[
\text{bfinal}(E_1 \# E_2, S) :- \text{bfinal}(E_1, S), \text{bfinal}(E_2, S).
\]

\[
\text{bfinal}(\text{pi}(L, G, E), S) :- \text{open\_query}(L, G, E, S).
\]

\[
\text{bfinal}(L_1, E_1, E_1, E_1) :- \text{bfinal}(L_1, E_1).
\]

\[
\text{bfinal}(\text{star}(E), S) :- \text{bfinal}(\text{star}(E), S).
\]

\[
\text{btrans}(B, S, \text{nil}, S_1) :- \text{sens\_action}(B), \text{po}(B, S, P),
\]

\[
\text{bool\_query}(\text{knows}(P), S).
\]

\[
\text{btrans}(B, S, \text{nil}, S_1) :- \text{btrans}(B, S, \text{nil}, S_1).
\]

\[
\text{btrans}(B, S, \text{nil}, S_1) :- \text{exec}(B, R),
\]

\[
\text{del\_sit}(S).
\]

\[
\text{btrans}(B, S, \text{nil}, S_1) :- \text{exec}(B, R), \text{write}\(':'(y/n)'), \text{read}(T),
\]

\[
(T=y->\text{true}; \text{false}).
\]

Search operator \( \Sigma \)

We define predicates \( \text{bdo}/3 \) and \( \text{ext}/4 \) to implement relations \( C \) and \( E \) respectively. During search, we maintain KBs of different situations. Once search succeeds or backtracks, we delete KBs accordingly. Hence we design predicates \( \text{add\_extra\_sit}(\sigma) \), \( \text{del\_extra\_sit}(\sigma) \) and \( \text{clear\_extra\_sits} \) for managing KBs of situations explored in search.

\[
\text{bdo}(B, S, B) :- \text{sens\_action}(B), \text{po}(B, P),
\]

\[
\text{bool\_query}(\text{knows}(P), S).
\]

\[
\text{bdo}(E_1 \# E_2, S, C) :- \text{bdo}(E_1, S, C), \text{ext}(C_1, E_2, S, C). \]

\[
\text{bdo}(\text{star}(E), S, C) :- C=\text{nil}; \text{bdo}(E, S, C), \text{ext}(C_1, \text{star}(E), S, C).
\]

\[
\text{ext}(\text{nil}, E, S, C) :- \text{bdo}(E, S, C).
\]

\[
\text{ext}(A \# E, S, A \# C) :- \text{prim\_prog}(A), \text{prim\_prog}(A, S, C), \text{add\_extra\_sit}(S),
\]

\[
\text{del\_extra\_sit}(S_1), \text{fail}.
\]

Planning operator \( \Upsilon \)

The main idea of our implementation of the planning operator \( \Upsilon(\tau, \delta) \) is this: we construct a planning instance from the BAT \( D, \tau, \delta \), and call an existing planner to solve the instance. Our implementation is based on the work by (Baier, Fritz, and McIlraith 2007) on compiling procedural domain control knowledge written in a Golog-like program into a planning instance.

A planning instance is a pair \( I = (D, P) \), where \( D \) is a domain definition and \( P \) is a problem. We assume that \( D \) and \( P \) are described in ADL. A domain definition consists of domain predicates and operators. A problem consists of domain objects, an initial state and a goal.

Baier et al. define a compiling function which, given a planning instance \( I \) and a program \( \delta \), outputs a new instance \( I_\delta \) such that planning for the generated instance \( I_\delta \) is equivalent to planning for the original instance \( I \) under the control of \( \delta \), except that plans for \( I_\delta \) contain some auxiliary actions.

We now present a translation function which, given a well-formed BAT \( D \), a program \( \Upsilon(\tau, \delta) \) and a ground situation \( \sigma \), outputs a planning instance \( I \) such that \( \text{safe}\_\text{plan}(\Upsilon(\tau, \delta), I) \) from \( D_{\Upsilon(\tau, \delta)} \) to \( D_{\Upsilon(\tau, \delta)} \) is equivalent to \( \Upsilon(\tau, \delta) \) and \( \sigma \).

Definition 13 (Translation function \( T \)) Given a well-formed BAT \( D \), a program \( \Upsilon(\tau, \delta) \) and a ground situation \( \sigma \), we define a planning instance \( I \) as follows:
1. the domain predicates are elements of $P(\delta)$;
2. the operators are $O(A(x))$ where $A$ appears in $\delta$;
3. the objects are elements of the type predicate $\tau$;
4. the initial state consists of ground atoms $P(\tilde{c}) \in \text{UP}(D_\alpha)$ s.t. $P \in P(\delta)$ and $\tilde{c} \in \tau$, $P(\tilde{c}) \in \text{UP}(D_\alpha)$ or $\neg P(\tilde{c}) \in \text{UP}(D_\alpha)$.
5. the goal is true.

Note that this planning instance where “the goal is true” will later be fed to Baier et al.’s compiling algorithm to generate the final planning instance where the goal is not true any more but to reach the final state of the control program.

To prove property of $T$, we define a just-in-time assumption:

**Definition 14** We say that a ground situation $\sigma$ is just-in-time for $\Upsilon(\tau, \delta)$ wrt $D$ if for each ground atom $P(\tilde{c})$ s.t. $P \in P(\delta)$ and $\tilde{c} \in \tau$, $P(\tilde{c}) \in \text{UP}(D_\alpha)$ or $\neg P(\tilde{c}) \in \text{UP}(D_\alpha)$.

Thus the just-in-time assumption ensures complete information regarding the execution of $\delta$. Let $\delta$ be a program. We define its objective program, denoted by $\delta_n$, as the program obtained from $\delta$ by replacing each test with its objective formula. Under the just-in-time assumption, SC-based reasoning coincides with database query evaluation (by a generalization of Theorem 5), and projection coincides with database update. So we get:

**Lemma 13** If $\sigma$ is just-in-time for $\Upsilon(\tau, \delta)$, then $(\delta_n, \rho, \sigma) \in C_D$ iff $\rho$ is a plan for $I = T(D, \tau, \delta, \sigma)$ under control of $\delta_n$.

We implement a predicate $\text{plan}(\tau, \delta, \sigma)$ which does the following: First, apply $T$ on $(D, \tau, \delta, \sigma)$ to generate a planning instance $I$. Then apply Baier et al.’s compiling function (with a slight modification) on $(I, \delta_n)$ to obtain a planning instance $I_{\delta_n}$, call FF planner (Hoffmann and Nebel 2001) on $I_{\delta_n}$ to get a plan $\rho$. Finally, filter out the auxiliary actions from $\rho$. Actually, the domain part of the planning instance $I_{\delta_n}$ does not depend on the current situation, and is generated during preprocessing of the program. Only the problem part is generated each time $\Upsilon(\tau, \delta)$ is called.

**Correctness of the interpreter**

Since the implementation of basic constructs and search operator are in direct correspondence with their semantics, based on correctness of progression and query evaluation (Theorems 12), by induction on the program, it is easy to prove Theorems 14 and 15 below.

**Theorem 14 (Correctness of basic constructs)** Let $D$ be well-formed, $\delta$ a basic program, and $\sigma$ a ground situation. Then $b\text{final}(\delta, \sigma)$ succeeds iff $(\delta, \sigma) \in F$, and $b\text{trans}(\delta, \sigma, \delta', \sigma')$ succeeds iff $(\delta, \sigma) \rightarrow (\delta', \sigma')$.

**Theorem 15 (Soundness and weak completeness of search)** Let $D$ be well-formed, $\delta$ a basic program and $\sigma$ a ground situation. Then if $b\text{trans}(\delta, \sigma, \rho, S)$ succeeds with $P=\rho$, then $(\delta, \sigma, \rho) \in C$; and if $(\delta, \sigma, \rho) \in C$ for some $\rho$, then $b\text{trans}(\delta, \sigma, \rho, S)$ succeeds or does not terminate.

To see why we get weak completeness, consider program $\delta = (\alpha; \text{false})$ where $\alpha$ true. Although $(\delta, \sigma, \text{nil}) \in C$, to search $\delta$, we first search $\alpha; \text{false}$ and wouldn’t terminate.

By Lemma 13 and correctness of the compiling function of Baier et al., we have:

**Theorem 16 (Correctness of planning operator)** Suppose $\sigma$ is just-in-time for $\Upsilon(\tau, \delta)$ wrt well-formed $\text{BAT} D$. We have: if $b\text{trans}(\delta, \sigma, \rho, S)$ succeeds with $P=\rho$, then $(\delta, \sigma, \rho) \in C$; and if $(\delta, \sigma, \rho) \in C$ for some $\rho$, then $b\text{trans}(\delta, \sigma, \rho, S)$ succeeds.

**Experiments**

We have experimented our interpreter with Wumpus world, blocks world, Unix domain, and service robot domain. Here we present experimental data about Wumpus world and give an example execution of a program in the blocks world.

Table 1 shows our experimental data about Wumpus world where we use the control program presented earlier. In Table 1, $\text{Prob}$ is the probability of a location containing a pit. For each probability, we tested on 3000 random $8 \times 8$ maps. $\text{IMP}$ is the number of maps for which it is impossible to explore. The rest of the columns show the average of the reward, the number of moves, the running time in seconds, and the number of calls of the FF planner. Note that the average running time is less than 0.7 seconds, which shows the efficiency of our interpreter.

<table>
<thead>
<tr>
<th>Prob</th>
<th>Gold</th>
<th>IMP</th>
<th>Reward</th>
<th>Moves</th>
<th>Time</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>1412</td>
<td>695</td>
<td>437</td>
<td>33</td>
<td>0.57</td>
<td>16</td>
</tr>
<tr>
<td>15%</td>
<td>909</td>
<td>1171</td>
<td>275</td>
<td>22</td>
<td>0.43</td>
<td>11</td>
</tr>
<tr>
<td>20%</td>
<td>567</td>
<td>1171</td>
<td>175</td>
<td>14</td>
<td>0.25</td>
<td>7</td>
</tr>
<tr>
<td>30%</td>
<td>263</td>
<td>1581</td>
<td>82</td>
<td>6</td>
<td>0.11</td>
<td>3</td>
</tr>
<tr>
<td>40%</td>
<td>182</td>
<td>1924</td>
<td>58</td>
<td>5</td>
<td>0.06</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1. Experimental results for Wumpus world ($8 \times 8$, 3000)

In the blocks world domain, we consider a program which makes clear a list of blocks. The only primitive action is $\text{move}(x, y, z)$, moving block $x$ from block $y$ to block $z$. There are 2 fluents: $\text{clear}(x)$, block $x$ has no blocks on top of it; and $\text{on}(x, y)$, block $x$ is on block $y$.

The following are relevant axioms:

$\text{Poss}(\text{move}(x, y, z), s) \equiv \text{on}(x, y) \land \text{clear}(x) \land \text{clear}(y)$.

$\text{Poss}(\text{on}(x, y), s) \equiv \text{true}$.

$\text{Poss}(\text{clear}(x), s) \equiv \text{true}$.

$\text{clear}(x, \text{do}(a, s)) \equiv (\exists y, z) a = \text{move}(y, x, z) \lor \text{clear}(x, z) \land \neg (\exists y, z) a = \text{move}(y, x, z)$.

$\text{on}(x, y, \text{do}(a, s)) \equiv (\exists z) a = \text{move}(x, y, z) \lor \text{on}(x, y, s) \land \neg (\exists z) a = \text{move}(x, y, z)$.

$\text{SF}(\text{clear}(x), s) \equiv \text{clear}(x)$.

$\text{SF}(\text{on}(x, y), s) \equiv \text{on}(x, y, s)$.

The initial KB is a proper $\text{KB}$ as follows:

$\forall x. x \neq a \land x \neq b \land x \neq c \land x \neq d \lor \text{clear}(x)$.

$\forall x, x \neq a \land x \neq b \land x \neq c \land x \neq d \land \neg y \lor \text{on}(x, y)$.

$\forall x, \text{on}(x, x)$.

$\forall x, \text{on}(x, y) \lor \text{on}(y, x)$.

$\forall x, \text{on}(x, y) \lor \text{on}(y, x)$.

$\forall x, \text{on}(x, y) \lor \text{on}(y, x)$.

$\forall x, \text{on}(x, y) \lor \text{on}(y, x)$.

Note that 4 blocks $a$, $b$, $c$ and $d$ appear in the initial KB.

The program is as below:

```
proc make_clear_all(L)
if ¬b1.Knows(b1 ∈ L) ∨ Knows(clear(b1))
then πb2.Knows(b2 ∈ L) ∧ ¬Knows(clear(b2))
   ¬(make_clear(b2, L); make_clear_all(L)) endIf
endProc
```
procmake\_clear(x, L)
if not KWhether(clear(x)) then sense\_clear(x) endIf;
if $\exists b_1$ Knows(clear(x)) then
  if $\exists b_2$ Knows(on(b_1, x))
    then $\pi b_2$.Knows(on(b_2, x))
    $\rightarrow$(make\_clear(b_2, L); move\_away(b_2, x, L))
  else $\pi b_2$.¬KWhether(on(b_3, x))
    $\rightarrow$(sense\_clear(b_3, x); make\_clear(x, L)) endIf
endProc
procmove\_away(y, x, L)
if $\exists b_1$.Knows(y $\neq b_1 \land$ clear(b_1) $\land$ b_1 $\not\in L$)
  then $\pi b_2$.Knows(y $\neq b_2 \land$ clear(b_2) $\land$ b_2 $\not\in L$)
    $\rightarrow$move(y, x, b_2)
  else $\pi b_3$.Knows(b_3 $\not\in L$) $\land$ ¬KWhether(clear(b_3))
    $\rightarrow$(sense\_clear(b_3); move\_away(y, x, L)) endIf
endProc

An example execution is given as follows:

?¬ 1bGolog(make\_clear\_all([a, b, c, d])).
  sense\_clear(a): no.
  sense\_on(b, a): no.
  sense\_on(c, a): no.
  sense\_on(d, a): yes.
  sense\_clear(d): no.
  sense\_on(b, d): no.
  sense\_on(c, d): yes.
  sense\_clear(c): yes.
  move(c, d, c1)
  move(d, a, c2)
  sense\_clear(b): yes.
  true.

Note how clever our agent is. Having discovered that c is clear and on top of d, she considers to move c away. Realizing that she cannot put c on any block in [a, b, c, d], the agent attempts to find an extra block and at last, she accomplishes her task with two extra blocks c1 and c2.

Conclusions

Other than those related work mentioned in the introduction, Petrick and Bacchus (2002) proposed a planning system with incomplete information and sensing called PKS. The form of incomplete knowledge they consider is mainly a set of ground literals but also exclusive disjunctive knowledge. It does offline execution based on incomplete progression and incomplete reasoning; but the incomplete procedures do not come with semantic characterizations. PKS has a limited support for functions, which we do not support. Now we summarize the contribution of this paper. First of all, we propose an implementation of Golog based on exact progression of first-order incomplete information. Secondly, we make the unique name but not the closed-world or domain closure assumption; we also make the dynamic CWA on knowledge; and we do limited reasoning with first-order incomplete information. The only other similar system we are aware of is the one by (Claßen and Lakemeyer 2009), but it is based on regression. Thirdly, we implement the progression and limited reasoning algorithms by Liu, Lakemeyer and Levesque by grounding, and we provide theoretical foundation for it. Fourthly, we provide a planning operator based on the work by (Baier, Fritz, and McIlraith 2007); however, they transform a program execution task into a single planning task while for us, a planning problem is dynamically generated each time the planner is called during a single program execution task. Lastly, we provide a search operator which returns a conditional plan, and it is different from the one in an extension of IndiGolog (Sardina 2001) in that ours does not rely on special branching actions specified by the programmer.

However, we have only implemented limited reasoning at the $B_0$ level, that is, full unit propagation but no case analysis. At this level, our reasoning is complete in the presence of the closed-world assumption or its dynamic versions, or in the presence of Horn disjunctive knowledge only. In the future, we would like to implement reasoning at the $B_1$ level. Also, we would like to investigate the support of procedure calls in the scope of planning operators, and explore the support of state constraints in our system.

Acknowledgments

We thank the anonymous reviewers for helpful comments. This work was supported by the Natural Science Foundation of China under Grant No. 61073053.

References


