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 Sardiña 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 $\mathcal{SL}$ and showed that $\mathcal{SL}$-based reasoning with proper$^+$ KBs is decidable. Recently, Claßen and Lakemeyer (2009) proposed an $\mathcal{SL}$-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 $\mathcal{LB}Golog$ (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 execution 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 $\mathcal{L}$ 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 $\mathcal{E}$ 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 Γ and Γ' be two sets of sentences. We write $\Gamma \models_{\mathcal{E}} \Gamma'$ to mean $\mathcal{E} \cup \Gamma$ classically entails Γ', and we write $\Gamma \Leftrightarrow_{\mathcal{E}} \Gamma'$ to mean $\Gamma \models_{\mathcal{E}} \Gamma'$ and vice versa. Let ϕ be a formula, and let μ and μ' be two terms/formulas. We denote by $\phi(\mu/\mu')$ the result of replacing every occurrence of μ in ϕ with μ'. We let ϕ^x_d denote ϕ 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 ϕ. We let θ range over substitutions of all variables by constants, and write $\phi\theta$ as the result of applying θ to ϕ.

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 Σ, gnd(Σ) is defined as $\{c\theta \mid \forall(e \supset c) \in \Sigma \text{ and } \models_{\mathcal{E}} e\theta\}$.

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 $\mathcal{L}_{sc}$ 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 τ, and we call such formulas uniform in τ. We ignore functional fluents in this paper.

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

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{sf} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}, \text{ where}$$

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

We use $\mathcal{D}$ to represent a BAT throughout this paper.

Progression

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

Definition 2 Let M and M' be structures with the same domains for sorts action and object. We write $M \sim_{S_\alpha} M'$ 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_α: For every relational fluent F, and every variable assignment σ, $M, \sigma \models F(\vec{x}, S_\alpha)$ iff $M', \sigma \models F(\vec{x}, S_\alpha)$.

Definition 3 Let $\mathcal{D}_{S_\alpha}$ be a set of sentences uniform in S_α. $\mathcal{D}_{S_\alpha}$ is a progression of the initial KB $\mathcal{D}_{S_0}$ wrt α if for any structure M, M is a model of $\mathcal{D}_{S_\alpha}$ iff there is a model M' of $\mathcal{D}$ such that $M \sim_{S_\alpha} M'$.

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 $A(\vec{c})$ changes the truth value of an atom $F(\vec{d}, s)$, then $\vec{d}$ is contained in $\vec{c}$. 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 α:*

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

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

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

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

It is easy to prove the following:

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

We say that $\mathcal{D}_{sf}$ is quantifier-free if for each action function $A(\vec{x})$, Ψ_A is quantifier-free. We say that $\mathcal{D}_{ss}$ is essentially quantifier-free if for each action function $A(\vec{x})$ and each fluent F, by using $\mathcal{D}_{una}$, $\Phi_F(\vec{x}, A(\vec{x}), s)$ 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 $\mathcal{D}_{ss}$ is essentially quantifier-free, $\mathcal{D}_{sf}$ is quantifier-free, and $\mathcal{D}_{S_0}$ 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 $prog(\mathcal{D}_{S_0}, \alpha)$ to denote a proper$^+$ KB which is a progression of $\mathcal{D}_{S_0}$ wrt α. It is straightforward to generalize the notation to $prog(\mathcal{D}_{S_0}, \sigma)$, where σ is a ground situation. We also use $\mathcal{D}_\sigma$ to denote $prog(\mathcal{D}_{S_0}, \sigma)$.

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 $\boldsymbol{B}_k\phi$ where $\phi \in \mathcal{L}$ and $\boldsymbol{B}_k$ is a modal operator for any $k \geq 0$. $\boldsymbol{B}_k\phi$ is read as "ϕ is a belief at level k". We let $\mathcal{SL}_k$ denote the set of $\mathcal{SL}$-formulas whose only modal operators are $\boldsymbol{B}_j$ 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 $\mathsf{UP}(s)$ 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 $\{\overline{\rho}\} \cup c \in s'$, where $\overline{\rho}$ denotes the complement of ρ, then $c \in s'$. Let $\phi \in \mathcal{L}$. The notation $(\boldsymbol{B}_k\phi)\!\downarrow$, called belief reduction, is defined as follows:

1. $(\boldsymbol{B}_k c)\!\downarrow \,=\, \boldsymbol{B}_k c$, where c is a clause;
2. $(\boldsymbol{B}_k e)\!\downarrow \,=\, e$, where e is an equality literal;
3. $(\boldsymbol{B}_k\neg\neg\phi)\!\downarrow \,=\, \boldsymbol{B}_k\phi$;
4. $(\boldsymbol{B}_k(\phi \vee \psi))\!\downarrow \,=\, (\boldsymbol{B}_k\phi \vee \boldsymbol{B}_k\psi)$, where ϕ or ψ is not a clause; and $(\boldsymbol{B}_k\neg(\phi \vee \psi))\!\downarrow \,=\, (\boldsymbol{B}_k\neg\phi \wedge \boldsymbol{B}_k\neg\psi)$;
5. $(\boldsymbol{B}_k\exists x\phi)\!\downarrow \,=\, \exists x\boldsymbol{B}_k\phi$; and $(\boldsymbol{B}_k\neg\exists x\phi)\!\downarrow \,=\, \forall x\boldsymbol{B}_k\neg\phi$.

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

1. $s \models (d = d')$ iff d and d' are the same constant;
2. $s \models \neg\varphi$ iff $s \not\models \varphi$;
3. $s \models \varphi \vee \omega$ iff $s \models \varphi$ or $s \models \omega$;
4. $s \models \exists x\varphi$ iff for some constant d, $s \models \varphi^x_d$;
5. $s \models \boldsymbol{B}_k\phi$ iff one of the following holds:
   (a) *subsume*: $k = 0$, ϕ is a clause c, and there is $c' \in \mathsf{UP}(s)$ s.t. $c' \subseteq c$;
   (b) *reduce*: ϕ is not a clause and $s \models (\boldsymbol{B}_k\phi)\!\downarrow$;
   (c) *split*: $k > 0$ and there is $c \in s$ s.t. for all $\rho \in c$, $s \cup \{\rho\} \models \boldsymbol{B}_{k-1}\phi$.

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

Reiter (2001b) introduced the closed-world assumption on knowledge that a given $\mathcal{K}$ of axioms about what an agent knows captures everything that the agent knows; any knowledge sentences not following logically from $\mathcal{K}$ 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 Σ be a proper$^+$ KB. We adapt Reiter's idea to $\mathcal{SL}$ as follows:

Definition 8 $bcl(\boldsymbol{B}_0\Sigma) \overset{def}{=} \boldsymbol{B}_0\Sigma \cup \{\neg\boldsymbol{B}_k\phi \,|\, \boldsymbol{B}_0\Sigma \not\models \boldsymbol{B}_k\phi\}$.

By an important result from (LLL04) that $\boldsymbol{B}_0\Sigma \models \boldsymbol{B}_k\phi$ iff $\text{gnd}(\Sigma) \models \boldsymbol{B}_k\phi$, we have $bcl(\boldsymbol{B}_0\Sigma) = \boldsymbol{B}_0\Sigma \cup \{\neg\boldsymbol{B}_k\phi \mid \text{gnd}(\Sigma) \models \neg\boldsymbol{B}_k\phi\}$. Thus it is easy to prove

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

1. *$bcl(\boldsymbol{B}_0\Sigma)$ is satisfiable.*
2. *For any $\varphi \in \mathcal{SL}$, $bcl(\boldsymbol{B}_0\Sigma) \models \varphi$ or $bcl(\boldsymbol{B}_0\Sigma) \models \neg\varphi$.*
3. *For any $\varphi \in \mathcal{SL}$, $gnd(\Sigma) \models \varphi$ iff $bcl(\boldsymbol{B}_0\Sigma) \models \varphi$.*

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

Theorem 5 *Let Σ be a complete proper KB. Then $bcl(\boldsymbol{B}_0\Sigma) \models \varphi$ iff $\Sigma \models_{\mathcal{E}} \varphi_o$.*

$\mathcal{LBG}olog$: syntax and semantics

The following are the programming constructs of $\mathcal{LBG}olog$. A difference with normal Golog is that all tests ϕ are $\mathcal{SL}_0$ formulas. For readability, we also write $\boldsymbol{B}_0\psi$ as **Knows**ψ.

1.	α	primitive action
2.	$\phi?$	test action
3.	$(\delta_1; \delta_2)$	sequence
4.	$(\delta_1 \mid \delta_2)$	nondeterministic choice of actions
5.	$(\pi\vec{x}.\phi \rightarrow \delta)$	guarded nondet. choice of arguments
6.	δ^*	nondeterministic iteration
7.	**if** ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
8.	**while** ϕ **do** δ **endWhile**	while loop
9.	**proc** $P(\vec{x})$ δ **endProc**	procedure definition
10.	$P(\vec{c})$	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 $\pi\vec{x}.\phi \rightarrow \delta$, where any variable of $\vec{x}$ must appear in ϕ, and it is executed by nondeterministically picking $\vec{x}$ such that $\phi(\vec{x})$ holds and then performing $\delta(\vec{x})$. We call a program basic if it uses only basic constructs. As in IndiGolog, there is a search operator $\Sigma\delta$, where δ is a basic program, which specifies that lookahead should be performed over δ to ensure that nondeterministic choices are resolved in a way that guarantees its successful completion. We allow δ 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 τ is a type predicate with a finite domain, that is, the initial KB $\mathcal{D}_{S_0}$ contains a sentence of the form $\forall x.\tau(x) \equiv x = d_1 \vee \ldots \vee x = d_n$, where $d_1, \ldots, d_n$ are constants. Any use of $\Upsilon(\tau, \delta)$ must satisfy the following restrictions: every constant appearing in δ is of type τ, δ is a basic program without sensing actions, and δ does not support procedural calls other than the simple case that δ 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 δ where objects are restricted to elements of type τ. Thus we require that when executing $\Upsilon(\tau, \delta)$, the agent should have complete knowledge regarding the execution of δ restricted to τ. 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 (δ, σ), where δ is a program (that remains to be executed) and σ 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 progression, $\mathcal{SL}$-based limited reasoning, and closed-world assumption on knowledge: when we evaluate a test ϕ wrt a configuration (δ, σ), we check if $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi[\sigma]$, where $\mathcal{D}_\sigma$ denotes $prog(\mathcal{D}_{S_0}, \sigma)$. Note that ϕ is a situation-suppressed formula, and $\phi[\sigma]$ denotes the formula obtained from ϕ by taking σ as the situation arguments of all fluents.

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

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \overset{def}{=} [\phi?; \delta_1] \mid [\neg\phi?; \delta_2],$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \overset{def}{=} [\phi?; \delta]^*; \neg\phi?.$$

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

1. $(nil, \sigma) \in \mathcal{F}$.
2. $(\phi?, \sigma) \in \mathcal{F}$ if $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi[\sigma]$.
3. $(\delta_1; \delta_2, \sigma) \in \mathcal{F}$ if $(\delta_1, \sigma) \in \mathcal{F}$ and $(\delta_2, \sigma) \in \mathcal{F}$.
4. $(\delta_1 \mid \delta_2, \sigma) \in \mathcal{F}$ if $(\delta_1, \sigma) \in \mathcal{F}$ or $(\delta_2, \sigma) \in \mathcal{F}$.
5. $(\pi\vec{x}.\phi \rightarrow \delta, \sigma) \in \mathcal{F}$ if there exist constants $\vec{c}$ such that $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma) \in \mathcal{F}$.
6. $(\delta^*, \sigma) \in \mathcal{F}$.

The transition relation between configurations wrt $\mathcal{D}$, denoted $\rightarrow_\mathcal{D}$, is inductively defined as follows:

1. $(\alpha, \sigma) \rightarrow (nil, do(\alpha, \sigma))$ if $\alpha = A(\vec{c})$ is an ordinary primitive action and $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \boldsymbol{B}_0\Pi_A(\vec{c}, \sigma)$.
2. $(\alpha, \sigma) \rightarrow (nil, do(\alpha_\mu, \sigma))$ if $\alpha = A(\vec{c})$ is a sensing action with result μ, and $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \boldsymbol{B}_0\Pi_A(\vec{c}, \sigma)$.
3. $(\delta_1; \delta_2, \sigma) \rightarrow (\gamma; \delta_2, \sigma')$ if $(\delta_1, \sigma) \rightarrow (\gamma, \sigma')$.
4. $(\delta_1; \delta_2, \sigma) \rightarrow (\gamma, \sigma')$ if $(\delta_1, \sigma) \in \mathcal{F}$ and $(\delta_2, \sigma) \rightarrow (\gamma, \sigma')$.
5. $(\delta_1 \mid \delta_2, \sigma) \rightarrow (\gamma, \sigma')$ if $(\delta_1, \sigma) \rightarrow (\gamma, \sigma')$ or $(\delta_2, \sigma) \rightarrow (\gamma, \sigma')$.
6. $(\pi\vec{x}.\phi \rightarrow \delta, \sigma) \rightarrow (\gamma, \sigma')$ if there exist constants $\vec{c}$ s.t. $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma) \rightarrow (\gamma, \sigma')$.
7. $(\delta^*, \sigma) \rightarrow (\gamma; \delta^*, \sigma')$ if $(\delta, \sigma) \rightarrow (\gamma, \sigma')$.

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

1. $(nil, \sigma, nil) \in \mathcal{C}$.
2. $(\alpha, \sigma, \alpha) \in \mathcal{C}$ if α is $A(\vec{c})$ and $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \boldsymbol{B}_0\Pi_A(\vec{c}, \sigma)$.
3. $(\phi?, \sigma, nil) \in \mathcal{C}$ if $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi[\sigma]$.
4. $(\delta_1; \delta_2, \sigma, \rho) \in \mathcal{C}$ if there exists ρ' such that $(\delta_1, \sigma, \rho') \in \mathcal{C}$ and $(\rho', \delta_2, \sigma, \rho) \in \mathcal{E}$.
5. $(\delta_1 \mid \delta_2, \sigma, \rho) \in \mathcal{C}$ if $(\delta_1, \sigma, \rho) \in \mathcal{C}$ or $(\delta_2, \sigma, \rho) \in \mathcal{C}$.
6. $(\pi\vec{x}.\phi \rightarrow \delta, \sigma, \rho) \in \mathcal{C}$ if there exist constants $\vec{c}$ such that $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma, \rho) \in \mathcal{C}$.
7. $(\delta^*, \sigma, nil) \in \mathcal{C}$; and $(\delta^*, \sigma, \rho) \in \mathcal{C}$ if there exists ρ' such that $(\delta, \sigma, \rho') \in \mathcal{C}$ and $(\rho', \delta^*, \sigma, \rho) \in \mathcal{E}$.

The formal definition of $\mathcal{E}$ is as follows:

1. $(nil, \delta, \sigma, \rho') \in \mathcal{E}$ if $(\delta, \sigma, \rho') \in \mathcal{C}$.

2. $(\alpha;\rho,\delta,\sigma,\alpha;\rho') \in \mathcal{E}$ if α is an ordinary primitive action and $(\rho,\delta,do(\alpha,\sigma),\rho') \in \mathcal{E}$.

3. $(\alpha;\rho,\delta,\sigma,\rho') \in \mathcal{E}$ if α is a sensing action and there exist ρ_1 and ρ_2 such that $(\rho,\delta,do(\alpha_T,\sigma),\rho_1) \in \mathcal{E}$ and $(\rho,\delta,do(\alpha_F,\sigma),\rho_2) \in \mathcal{E}$ and ρ' is: α; **if Knows**$\Psi_A(\vec{c})$ **then** ρ_1 **else** ρ_2 **endIf**.

4. (**if** ϕ **then** ρ_1 **else** ρ_2 **endIf**, $\delta,\sigma,\rho) \in \mathcal{E}$ if the following holds: if $bcl(\boldsymbol{B}_0\mathcal{D}_\sigma) \models \phi[\sigma]$, then $(\rho_1,\delta,\sigma,\rho) \in \mathcal{E}$, otherwise $(\rho_2,\delta,\sigma,\rho) \in \mathcal{E}$.

To define the semantics of $\Upsilon(\tau,\delta)$, we define the restriction of δ to τ, denoted by δ_τ. Intuitively, δ_τ is δ where objects are restricted to elements of type τ. Formally, δ_τ is obtained from δ as follows: replace each formula of the form $\forall x\phi$ with $\forall x.\mathbf{Knows}\tau(x) \supset \phi$, and $\exists x\phi$ with $\exists x.\mathbf{Knows}\tau(x) \wedge \phi$, and replace each construct of the form $\pi\vec{x}.\phi \rightarrow \delta$ with $\pi\vec{x}.\bigwedge \mathbf{Knows}\tau(x_i) \wedge \phi \rightarrow \delta$.

We can now expand the definition of $\rightarrow$ as follows:

8. $(\Sigma\delta,\sigma) \rightarrow (\rho,\sigma)$ if $(\delta,\sigma,\rho) \in \mathcal{C}$.

9. $(\Upsilon(\tau,\delta),\sigma) \rightarrow (\rho,\sigma)$ if $(\delta_\tau,\sigma,\rho) \in \mathcal{C}$.

Finally an online execution of an $\mathcal{LBG}olog$ program δ_0 starting from a situation σ_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 $(\delta_n,\sigma_n) \in \mathcal{F}$.

We now illustrate programming in $\mathcal{LBG}olog$ 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 $coor(x)$ 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.

```
proc  $main$
  $sense\_stench; sense\_breeze; sense\_gold;$
  if Knows$(gold(n_1,n_1))$ then $grab$
  else $explore$ endIf
  $(climb \mid \Upsilon(coor, moveLoc(n_1,n_1)); climb)$
endProc
```

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

```
proc  $moveLoc(X,Y)$
  $[\pi x_0,y_0,x_1,y_1.$Knows$(agt(x_0,y_0) \wedge explored(x_1,y_1))$
      $\rightarrow move(x_0,y_0,x_1,y_1)]^*;$
  $\pi x_2,y_2.$Knows$(agt(x_2,y_2)) \rightarrow move(x_2,y_2,X,Y)$
endProc
```

The procedure below explores the dungeon. Here $wp(x,y)$ means that the wumpus is at location (x,y), and $pit(x,y)$ 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.

```
proc  $explore$
  while  Knows$(\neg getsGold) \wedge$
  $\exists x,y.$Knows$((\neg wp(x,y) \vee \neg wpAlive) \wedge \neg pit(x,y)) \wedge$
        $\neg$Knows$(explored(x,y))$  do
  $\pi x,y.$Knows$((\neg wp(x,y) \vee \neg wpAlive) \wedge \neg pit(x,y)) \wedge$
        $\neg$Knows$(explored(x,y)) \rightarrow$
      $\Upsilon(coor, moveLoc(x,y));$
      $sense\_stench; sense\_breeze; sense\_gold;$
      if  Knows$(\exists x_0,y_0.agt(x_0,y_0) \wedge gold(x_0,y_0))$
      then  $grab$  else
        if  $\exists x_1,y_1.$Knows$(wpAlive \wedge wp(x_1,y_1))$
        then  $shootWumpus$  endIf endIf endWhile
endProc
```

Implementing progression and query evaluation by grounding

To implement $\mathcal{LBG}olog$, we need to implement progression and evaluation of an $\mathcal{SL}$ formula against the closure of $\boldsymbol{B}_0\mathcal{D}_\sigma$, where $\mathcal{D}_\sigma$ 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 Σ as the maximum number of distinct variables in a $\forall$-clause of Σ. Let j be the width of Σ. 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 Γ of formulas, we use $H(\Gamma)$ to denote the set of normal constants appearing in Γ, and let $H^+(\Gamma)$ represent $H(\Gamma)$ extended with a normal constant not appearing in Γ.

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

The intuition is that constants not appearing in Σ behave the same, and we take U constants as their representatives.

Example 1 Consider a simple proper$^+$ KB $\Sigma = \{\forall x.\neg on(x,A), \forall x,y.x \neq y \supset \neg on(x,y) \vee \neg clear(y)\}$. Then Σ has width 2, and $U = \{u_1,u_2\}$. Let $N = \{A\}$. So $prop(\Sigma,N) = \{\neg on(A,A), \neg on(u_1,A), \neg on(u_2,A),$ $\neg on(A,u_1) \vee \neg clear(u_1), \neg on(A,u_2) \vee \neg clear(u_2),$ $\neg on(u_1,A) \vee \neg clear(A), \neg on(u_1,u_2) \vee \neg clear(u_2),$ $\neg on(u_2,A) \vee \neg clear(A), \neg on(u_2,u_1) \vee \neg clear(u_1)\}$.

In the sequel, we let Σ_p denote a ground proper$^+$ KB with U constants. To prove correctness of grounding, we first define the first-order KB represented by Σ_p.

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

Theorem 6 $FO(prop(\Sigma, N)) \Leftrightarrow_{\mathcal{E}} \Sigma$.

We now define extended grounding and show its correctness.

Definition 11 Let B be a finite set of normal constants not occurring in Σ_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 $egnd(prop(\Sigma, N), B) \Leftrightarrow_{\mathcal{E}} prop(\Sigma, N \cup B)$.

Note that the way we do grounding is brute-force. For example, if Σ contains $\forall x P(x)$, then its ground KB contains $P(u_1), \dots, P(u_j)$, 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 $\mathcal{D}$ be a well-formed BAT, and $\alpha = A(\vec{c})$ a ground action. Let B be the set of constants appearing in $\vec{c}$ but not Σ_p. We define $pprog(\Sigma_p, \alpha)$ as

$$forget(egnd(\Sigma_p, B) \cup \mathcal{D}_{ss}[\Omega], \Omega(S_0))(S_0/S_\alpha),$$

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

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 σ 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 Σ or q. Then $forget(\Sigma, q) \Leftrightarrow_{\mathcal{E}} FO(forget(prop(\Sigma, N), q))$.*

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

Theorem 9 $FO(pprog(prop(\Sigma, N), \alpha)) \Leftrightarrow_{\mathcal{E}} prog(\Sigma, \alpha)$.

Query Evaluation

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

We first define an evaluation procedure $G[\Sigma_p, \phi]$ where $\phi \in \mathcal{L}$ is suitable for Σ_p. It is the same as the $W[\Sigma, k, \phi]$ procedure from (LLL04) to decide if $\boldsymbol{B}_0\Sigma \models \boldsymbol{B}_k\phi$ 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. ϕ is a clause c and there exists a clause $c' \in \Sigma_p$ such that $c' \subseteq c(d_1/u_1, \dots, d_k/u_k)$, where $\{d_1, \dots, d_k\}$ is the set of normal constants that appear in c but not Σ_p.
2. $\phi = (d = d')$ and d is identical to d'.
3. $\phi = (d \neq d')$ and d is distinct from d'.
4. $\phi = \neg\neg\psi$ and $G[\Sigma_p, \psi] = 1$.
5. $\phi = (\psi \vee \eta)$, ψ or η is not a clause, and $G[\Sigma_p, \psi] = 1$ or $G[\Sigma_p, \eta] = 1$.
6. $\phi = \neg(\psi \vee \eta)$, $G[\Sigma_p, \neg\psi] = 1$ and $G[\Sigma_p, \neg\eta] = 1$.
7. $\phi = \exists x\psi$ and $G[\Sigma_p, \psi_d^x] = 1$ for some $d \in H^+(\Sigma_p \cup \{\psi\})$.
8. $\phi = \neg\exists x\psi$ and $G[\Sigma_p, \neg\psi_d^x]$ for all $d \in H^+(\Sigma_p \cup \{\psi\})$.

Based on G, we now define an evaluation procedure $F[\Sigma_p, \varphi]$ where $\varphi \in \mathcal{SL}_0$ is suitable for Σ_p. $F[\Sigma_p, \varphi] = 1$ if one of the following conditions holds, and 0 otherwise.

1. $\varphi = \boldsymbol{B}_0\phi$ 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. $\varphi = \varphi_1 \vee \varphi_2$, and $F[\Sigma_p, \varphi_1] = 1$ or $F[\Sigma_p, \varphi_2] = 1$.
5. $\varphi = \exists x\omega$, and $F[\Sigma_p, \omega_d^x] = 1$ for some $d \in H^+(\Sigma_p \cup \{\omega\})$.

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

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

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

Theorem 11 $F[\mathsf{UP}(\Sigma_p), \varphi] = 1$ *iff* $bcl(\boldsymbol{B}_0 FO(\Sigma_p)) \models \varphi$.

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 $F[upprog(uprop(\mathcal{D}_{S_0}, H(\mathcal{D}_{S_0})), \sigma), \varphi] = 1$ *iff* $bcl(\boldsymbol{B}_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \varphi$*, where σ is a situation.*

An interpreter

We have implemented an interpreter for $\mathcal{LBG}olog$ 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`(α, σ, ϕ): formula ϕ is the precondition for action α in situation σ;
- `ssa`(F, γ^+, γ^-): γ^+ (resp. γ^-) is the condition for making fluent F true (resp. false);
- `sf`(β, σ, ϕ): β senses if formula ϕ holds in situation σ.

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`(ϕ, σ): ϕ is evaluated true in situation σ;
- `open_query`($\vec{x}, \phi, \sigma, \vec{c}$): formula $\phi_{\vec{c}}^{\vec{x}}$ is evaluated true in situation σ;
- `prim_prog`(α, σ, σ'): progress the KB of situation σ to situation σ' wrt ordinary primitive action α;

- `sens_prog`($\beta, \mu, \sigma, \sigma'$): progress the KB of situation σ to situation σ' wrt sensing action β with sensing result μ;
- `del_sit`(σ): delete the KB about situation σ.

The two progression operators yield the new KBs while keeping the old ones. Thus we need the `del_sit` predicate.

The top part of the interpreter uses `btrans` and `bfinal` to determine the next action to perform or to terminate.

```
lbGolog(E,S):-bfinal(E,S),!.
lbGolog(E,S):-btrans(E,S,E1,S1),!,
    lbGolog(E1,S1).
```

Basic constructs

We define predicates `bfinal/2` and `btrans/4` to implement the $\mathcal{F}$ and $\rightarrow$ relations. The Prolog syntax for the constructs are: `E1:E2` for sequence, `E1#E2` for nondet. choice; `pi(L,G,E)` for guarded nondet. choice of arguments, `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 `sub_arg`(l_x, l_c, p_x, p_c) to substitute all variables of l_x occurring in program p_x with corresponding constants of l_c, resulting in program p_c. Predicate `po`(α, σ, ϕ') (resp. `sfns`(α, σ, ϕ')) holds iff `poss`(α, σ, ϕ) (resp. `sf`(α, σ, ϕ)) holds and ϕ' is ϕ with situation arguments suppressed.

```
bfinal(?(P),S):-bool_query(P,S).
bfinal(E1#E2,S):-bfinal(E1,S);bfinal(E2,S).
bfinal(pi(L,G,E),S):-open_query(L,G,S,L1),
   sub_arg(L,L1,E,E1),bfinal(E1,S).
bfinal(star(_),_).
btrans(B,S,nil,S1):-sens_action(B),po(B,S,P),
    bool_query(knows(P),S),do(B,S,S1).
btrans(E1:E2,S,E,S1):-btrans(E1,S,E3,S1),
    E=(E3:E2);bfinal(E1,S),btrans(E2,S,E,S1).
btrans(star(E),S,E1:star(E),S1):-
    btrans(E,S,E1,S1).
```

To perform an action, do the corresponding input/output actions, and then do progression.

```
do(B,S,S1):-sens_action(B),exec(B,R),
    sens_prog(B,R,S,S1),del_sit(S).
exec(B,R):-write(B),write(':(y/n)'),read(T),
    (T=y->R=true;R=false).
```

Search operator Σ

We define predicates $bdo/3$ and $ext/4$ to implement relations $\mathcal{C}$ and $\mathcal{E}$ respectively. During search, we maintain KBs of different situations. Once search succeeds or backtracks, we delete KBs accordingly. Hence we design predicates `add_extra_sit`(σ), `del_extra_sit`(σ) and `clear_extra_sits` for managing KBs of situations explored in search.

```
bdo(B,S,B):-sens_action(B),po(B,P),
    bool_query(knows(P),S).
bdo(E1:E2,S,C):-bdo(E1,S,C1),ext(C1,E2,S,C).
bdo(star(E),S,C):-C=nil;bdo(E,S,C1),
    ext(C1,star(E),S,C).
ext(nil,E,S,C):-bdo(E,S,C).
ext(A:C,E,S,A:C1):-prim_action(A),
    prim_prog(A,S,S1),add_extra_sit(S1),
    (ext(C,E,S1,C1);del_extra_sit(S1),fail).
ext(B:C,E,S,C1):-sens_action(B),
    sens_prog(B,true,S,ST),add_extra_sit(ST),
    (ext(C,E,ST,CT);del_extra_sit(ST),fail),
    sens_prog(B,0,S,SF),add_extra_sit(SF),
    (ext(C,E,SF,CF);del_extra_sit(SF),fail),
    sfns(B,S,F),C1=(B:if(knows(F),CT,CF)).
ext(if(P,C1,C2),E,S,C):-bool_query(P,S)->
    ext(C1,E,S,C);ext(C2,E,S,C).
search(E,S,C):-bdo(E,S,C),clear_extra_sits.
btrans(search(E),S,E1,S):-search(E,S,E1).
```

The implementation of the search operator ensures that nondeterministic choices are resolved in a way that guarantees the successful completion of the program. To see an example, consider the program `E` below for catching a plane:

```
sense_gate_A : buy_paper :
(goto(gate_A) : buy_coffee #
 buy_coffee : goto(gate_B)) : board
```

Assume that there are only two gates `A` and `B`, and `sense_gate_A` tells the agent which gate to take. Note that `board` is executable only if the agent gets to the right gate. So an online execution of `E` might fail. This problem can be solved by using the search operator. The interpretation of `search(E)` results in the following program, whose online execution is guaranteed to be successful.

```
sense_gate_A :
if(knows(it_is_gate_A),
   buy_paper:goto(gate_A):buy_coffee:board,
   buy_paper:buy_coffee:goto(gate_B):board)
```

Planning operator Υ

The main idea of our implementation of the planning operator $\Upsilon(\tau, \delta)$ is this: we construct a planning instance from the BAT $\mathcal{D}$, τ and δ, and call an existing planner to solve the instance. Our implementation is based 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 δ, outputs a new instance I_δ such that planning for the generated instance I_δ is equivalent to planning for the original instance I under the control of δ, except that plans for I_δ contain some auxiliary actions.

We now present a translation function which, given a well-formed BAT $\mathcal{D}$, a program $\Upsilon(\tau, \delta)$ and a ground situation σ, outputs a planning instance I. Since $\mathcal{D}$ is local-effect, for any action $A(\vec{x})$, we can generate an operator $O(A(\vec{x}))$ from $\mathcal{D}_{ap}$ and $\mathcal{D}_{ss}$. We omit the details here. We let $\mathcal{P}(\delta)$ denote the set of predicates relevant to δ (wrt $\mathcal{D}$), *i.e.*, the set of predicates that occur in tests of δ or precondition or effect axioms for actions that occur in δ.

Definition 13 (Translation function T) Given a well-formed BAT $\mathcal{D}$, a program $\Upsilon(\tau, \delta)$ and a ground situation σ, we define a planning instance I as follows:

1. the domain predicates are elements of $\mathcal{P}(\delta)$;
2. the operators are $O(A(\vec{x}))$ where A appears in δ;
3. the objects are elements of the type predicate τ;
4. the initial state consists of ground atoms $P(\vec{c}) \in \mathsf{UP}(\mathcal{D}_\sigma)$ s.t. $P \in \mathcal{P}(\delta)$, $\vec{c} \in \tau$ and $\mathcal{D}_\sigma$ is the KB of σ.
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 σ is just-in-time for $\Upsilon(\tau, \delta)$ wrt $\mathcal{D}$, if for each ground atom $P(\vec{c})$ s.t. $P \in \mathcal{P}(\delta)$ and $\vec{c} \in \tau$, $P(\vec{c}) \in \mathsf{UP}(\mathcal{D}_\sigma)$ or $\neg P(\vec{c}) \in \mathsf{UP}(\mathcal{D}_\sigma)$.

Thus the just-in-time assumption ensures complete information regarding the execution of δ. Let δ be a program. We define its objective program, denoted by δ_o, as the program obtained from δ by replacing each test with its objective formula. Under the just-in-time assumption, $\mathcal{SL}$-based reasoning coincides with database query evaluation (by a generalize version of Theorem 5), and progression coincides with database update. So we get:

Lemma 13 *If σ is just-in-time for $\Upsilon(\tau, \delta)$, then $(\delta_\tau, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ iff ρ is a plan for $I = T(\mathcal{D}, \tau, \delta, \sigma)$ under control of δ_o.*

We implement a predicate $\texttt{plan}(\tau, \delta, \sigma, \delta')$ which does the following: First, apply T on $(\mathcal{D}, \tau, \delta, \sigma)$ to generate a planning instance I. Then apply Baier *et al.*'s compiling function (with a slight modification) on (I, δ_o) to obtain a planning instance I_{δ_o}, call FF planner (Hoffmann and Nebel 2001) on I_{δ_o} to get a plan ρ. Finally, filter out the auxiliary actions from ρ. Actually, the domain part of the planning instance I_{δ_o} 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 $\mathcal{D}$ be well-formed, δ a basic program, and σ a ground situation. Then* bfinal(δ, σ) *succeeds iff* $(\delta, \sigma) \in \mathcal{F}$, *and* btrans$(\delta, \sigma, \delta', \sigma')$ *succeeds iff* $(\delta, \sigma) \to (\delta', \sigma')$.

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

To see why we get weak completeness, consider program $\delta = (\alpha^*; false?)\,|\,true?$. Although $(\delta, \sigma, nil) \in \mathcal{C}$, to search δ, we first search $\alpha^*; false?$ and wouldn't terminate.

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

Theorem 16 (Correctness of planning operator) *Suppose σ is just-in-time for $\Upsilon(\tau, \delta)$ wrt well-formed BAT $\mathcal{D}$. We have: if* btrans(planning$(\tau, \delta), \sigma, P, S)$ *succeeds with* $P = \rho$, *then* $(\delta, \sigma, \rho) \in \mathcal{C}$; *and if* $(\delta, \sigma, \rho) \in \mathcal{C}$ *for some* ρ, *then* btrans(planning$(\tau, \delta), \sigma, P, S)$ *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, *Prob* is the probability of a location containing a pit. For each probability, we tested on 3000 random 8×8 maps. *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.

Prob	Gold	IMP	Reward	Moves	Time	Calls
10%	1412	695	437	33	0.670	16
15%	890	917	275	22	0.430	11
20%	567	1171	175	14	0.254	7
30%	263	1581	82	6	0.112	3
40%	182	1924	58	3	0.064	2

Table 1. Experimental results for Wumpus world (8×8, 3000)

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

The following are relevant axioms:

$$
\begin{aligned}
&Poss(move(x, y, z), s) \equiv on(x, y) \wedge clear(x) \wedge clear(y),\\
&Poss(sense\_on(x, y), s) \equiv true,\\
&Poss(sense\_clear(x), s) \equiv true,\\
&clear(x, do(a, s)) \equiv (\exists y, z)a = move(y, x, z) \vee\\
&\qquad clear(x, s) \wedge (\neg\exists y, z)a = move(y, z, x),\\
&on(x, y, do(a, s)) \equiv (\exists z)a = move(x, z, y) \vee\\
&\qquad on(x, y, s) \wedge (\neg\exists z)a = move(x, y, z),\\
&SF(sense\_clear(x), s) \equiv clear(x, s),\\
&SF(sense\_on(x, y), s) \equiv on(x, y, s).
\end{aligned}
$$

The initial KB is a proper$^+$ KB as follows:

$$
\begin{aligned}
&\forall x. x \neq a \wedge x \neq b \wedge x \neq c \wedge x \neq d \supset clear(x),\\
&\forall x, y. x \neq a \wedge x \neq b \wedge x \neq c \wedge x \neq d \wedge x \neq y \supset \neg on(x, y),\\
&\forall x, y. x \neq y \supset \neg on(x, y) \vee \neg clear(y),\\
&\forall x. \neg on(x, x),\\
&\forall x, y. x \neq y \supset \neg on(x, y) \vee \neg on(y, x),\\
&\forall x, y, z. y \neq z \supset \neg on(x, y) \vee \neg on(x, z),\\
&\forall x, y, z. y \neq z \supset \neg on(y, x) \vee \neg on(z, x).
\end{aligned}
$$

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 $\neg\forall b_1.$Knows$(b_1 \in L) \supset$ Knows$(clear(b_1))$
then $\pi b_2.$Knows$(b_2 \in L) \wedge \neg$Knows$(clear(b_2))$
        $\to (make\_clear(b_2, L); make\_clear\_all(L))$ endIf
endProc
```

```
proc $make\_clear(x, L)$
if $\neg$KWhether$(clear(x))$ then $sense\_clear(x)$ endIf;
if $\neg$Knows$(clear(x))$ then
    if $\exists b_1.$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 $\pi b_3.\neg$KWhether$(on(b_3, x))$
            $\rightarrow(sense\_on(b_3, x); make\_clear(x, L))$ endIf
endIf endProc
```

```
proc $move\_away(y, x, L)$
if $\exists b_1.$Knows$(y \neq b_1 \wedge clear(b_1) \wedge b_1 \notin L)$
then $\pi b_2.$Knows$(y \neq b_2 \wedge clear(b_2) \wedge b_2 \notin L)$
        $\rightarrow move(y, x, b_2)$
else $\pi b_3.$Knows$(b_3 \notin L) \wedge \neg$KWhether$(clear(b_3))$
        $\rightarrow(sense\_clear(b_3); move\_away(y, x, L))$ endIf
endProc
```

An example execution is given as follows:

```
?- lbGolog(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 $\boldsymbol{B}_0$ 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 $\boldsymbol{B}_1$ 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

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. ICAPS-07*, 26–33.

Claßen, J., and Lakemeyer, G. 2009. Tractable first-order golog with disjunctive knowledge bases. In *Proc. Commonsense 2009*.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.* 121(1-2):109–169.

De Giacomo, G.; Levesque, H. J.; and Sardiña, S. 2001. Incremental execution of guarded theories. *ACM Trans. Comput. Log.* 2(4):495–525.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.* 14:253–302.

Lakemeyer, G. 1999. On sensing and off-line interpreting in Golog. In *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter*.

Lin, F., and Reiter, R. 1994. Forget it! In *Working Notes of AAAI Fall Symposium on Relevance*.

Lin, F., and Reiter, R. 1997. How to progress a database. *Artificial Intelligence* 92(1–2):131–167.

Liu, Y., and Lakemeyer, G. 2009. On first-order definability and computability of progression for local-effect actions and beyond. In *Proc. IJCAI-09*.

Liu, Y.; Lakemeyer, G.; and Levesque, H. J. 2004. A logic of limited belief for reasoning with disjunctive information. In *Proc. KR-04*, 587–597.

Petrick, R. P. A., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. AIPS-02*, 212–221.

Reiter, R. 2001a. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Reiter, R. 2001b. On knowledge-based programming with sensing in the situation calculus. *ACM Trans. Comput. Log.* 2(4):433–457.

Sardina, S. 2001. Local conditional high-level robot programs. In *Proc. LPAR-01*.