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

Cognitive robotics, Reasoning about actions,

Abstract

While founded on the situation calculus, current implementations of Golog are based on the closed-world assumption (CWA), dynamic versions of CWA, or the domain closure assumption. Also, they are 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. The interpreter is offline but the programmer can use two operators to specify offline execution for parts of programs. The search operator returns a conditional plan as execution of the operand program. The planning operator is used when locally complete information is available and calls a state-of-the-art planner to generate a plan.

Introduction

?? where to mention LBGolog

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. Golog is theoretically based on the situation calculus (Reiter 2001a), which is a first-order language. However, current implementations of Golog offer limited first-order capabilities. For example, implementation of classic Golog is based on the closed-world assumption (CWA). Although variants of Golog have been proposed to handle incomplete information and sensing, their implementations resort to dynamic versions of CWA or at least the domain closure assumption (DCA). For example, implementation 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 as proposed by Reiter (2001b) is based on DCA and reduces first-order reasoning to propositional one. However, in many real-world applications, it is inappropriate to have CWA, or dynamic CWA, or domain closure axiom.

An essential component of any Golog interpreter is a query evaluation module, which solves the projection problem, that is, decide if a formula holds after a sequence of actions have been performed. Two powerful methods to solve the projection problem are *regression* and *progression*. Roughly, regression reduces a query about the future to a query about the initial knowledge base (KB). Progression, on the other hand, changes the initial KB according to the effects of each action and then checks whether the formula holds in the resulting KB. One 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, as in the case of a robot operating for an extended period of time, regression simply becomes unmanageable. However, current implementations of Golog are exclusively based on regression. This might be due to the negative result that in general progression is not first-order definable. However, recently, Liu and Lakemeyer (2009) showed that for the so-called local-effect actions, progression is always first-order definable and computable.

Golog interpreters can be put into three categories: online, offline, and a combination of the two. An online interpreter is incomplete because no backtracking is allowed, while an offline interpreter is computationally expensive because of the much larger search space. In the presence of sensing, Reiter's interpreter for knowledge-based Golog is online, while the one for sGolog (Lakemeyer 1999) is offline, generating a conditional plan. IndiGolog combines online execution with offline execution of parts of programs, specified by the programmer with a search operator. However, unlike sGolog, IndiGolog ignores sensing results during offline execution of programs. To improve efficiency of Golog interpreters, there has been work on exploiting state-of-the-art planners. For example, Baier *et al.* (2007) developed an approach for compiling procedural domain control knowledge written in a Golog-like program into a planning instance which can then be solved by a planner. Claßen *et al.* (**?**) proposed the idea of calling a planner to achieve a goal during the execution of a Golog program.

Our work in this paper is based on existing work on the so-called proper$^+$ KBs for representing first-order disjunctive information. Intuitively, a proper$^+$ KB is equivalent to

a possibly infinite set of ground clauses. Our solution to the projection problem is based on exact progression and limited reasoning, and we exploit existing results regarding proper$^+$ KBs. We call our version of Golog LBGolog (Limited-Belief-based Golog). Since the deduction problem for proper$^+$ KBs is undecidable, Liu, Lakemeyer and Levesque (2004) proposed a logic of limited belief called the subjective logic $\mathcal{SL}$, and proved that $\mathcal{SL}$-based reasoning with proper$^+$ KBs is decidable. 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. On the other hand, Liu and Lakemeyer (2009) showed that for a restricted class of local-effect actions and proper$^+$ KBs, progression is not only first-order definable but also efficiently computable.

In this paper, we propose an interpreter for knowledge-based Golog with first-order incomplete information and sensing. We do not assume any of the CWA, dynamic CWA and DCA. The incomplete information is in the form of the so-called proper$^+$ KBs (**?**); To improve efficiency, we implement the reasoning (right now, we do not do any case analysis) and progression procedures by grounding. The trick here is to use an appropriate number of special constants, which actually carries first-order information. We provide a search operator, which, unlike in indiGolog, returns a conditional program where branchings are conditioned on the results of sensing actions. We also provide a planning operator, which is used with a program when locally complete information is available, and calls a state-of-the-art planner to generate a sequence of actions constituting a legal execution of the program. We have experimented our interpreter with Wumpus world, blocks world, Unix domain, and service robot domains; the results showed 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 start with 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, identified with their disjunction. 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 ϕ be a formula, and let μ and μ' be two expressions. We denote by $\phi(\mu/\mu')$ the result of replacing every occurrence of μ in ϕ with μ'. We write ϕ^x_d to denote ϕ with all free occurrences of variable x replaced by constant d.

Proper$^+$ KBs

Proper$^+$ KBs were proposed by Lakemeyer and Levesque (**?**) for representing first-order disjunctive information. Intuitively, a proper$^+$ KB is equivalent to a (possibly infinite) set of ground clauses. To formally define proper$^+$ KBs, 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 use θ to range over substitutions of all variables by constants, and write $\phi\theta$ as the result of applying the substitution θ to ϕ.

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

Situation calculus

The language $\mathcal{L}_{sc}$ of the situation calculus (Reiter 2001a) is a many-sorted first-order language suitable for describing dynamic worlds. There are three disjoint sorts: $action$ for actions, $situation$ for situations, and $object$ for everything else. $\mathcal{L}_{sc}$ has the following components: a constant S_0 denoting the initial situation; a binary function $do(a, s)$ denoting the successor situation to s resulting from performing action a; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; action functions, *e.g.*, $move(x, y)$; a finite number of relational fluents, *i.e.*, predicates taking a situation term as their last argument, *e.g.*, $ontable(x, s)$; and a finite number of situation-independent predicates. We ignore functional fluents in this paper.

We relate $\mathcal{L}_{sc}$ to $\mathcal{L}$ as follows: There is a set of constants of sort object which are constants of $\mathcal{L}$. The situation-independent predicates and relational fluents are predicates from $\mathcal{L}$. That is, if $P(\vec{x})$ is a situation-independent predicate, and $F(\vec{x}, s)$ is a relational fluent, then $P(\vec{x})$ and $F(\vec{x})$ are predicates from $\mathcal{L}$.

Often, we need to restrict our attention to formulas that refer to a particular situation. For this purpose, we say that a formula ϕ is uniform in a situation term τ, if ϕ does not mention any other situation terms except τ, does not quantify over situation variables, and does not mention $Poss$.

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

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

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 $F(\vec{x}, do(a, s)) \equiv \gamma^+_F(\vec{x}, a, s) \vee (F(\vec{x}) \wedge \neg\gamma^-_F(\vec{x}, a, s))$, where $\gamma^+_F(\vec{x}, a, s)$ and $\gamma^-_F(\vec{x}, a, s)$ are uniform in s.
4. ??$\mathcal{D}_{una}$ is the set of unique names axioms for actions: $A(\vec{x}) \neq A'(\vec{y})$, and $A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}$, where A and A' are distinct action functions.
5. $\mathcal{D}_{S_0}$, the initial KB, is a set of sentences uniform in S_0.

Following Levesque (1996), we extend the situation calculus to accommodate sensing actions. Assume that in addition to ordinary actions (called physical actions) that change the world, we also have binary sensing actions that do not change the world but tell the agent whether some condition ϕ holds in the current situation. We use the predicate $SF(a, s)$ to characterize what the sensing action tells the

agent about the world. Now our BAT has an extra component $\mathcal{D}_{sf}$, which is a set of sensed fluent axioms (SFAs), one for each 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. To simplify presentation, 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.

Progression

Lin and Reiter (**?**) formalized the notion of progression. Let $\mathcal{D}$ be a basic action theory, and α a ground action. We denote by S_α the situation term $do(\alpha, S_0)$.

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 $M', \sigma \models F(\vec{x}, S_\alpha)$.

We denote by $\mathcal{L}^2_{sc}$ the second-order extension of $\mathcal{L}_{sc}$. The notion of uniform formulas carries over to $\mathcal{L}^2_{sc}$.

Definition 3 Let $\mathcal{D}_{S_\alpha}$ be a set of sentences in $\mathcal{L}^2_{sc}$ 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. Their proof is a very simple one via the concept of forgetting. For a restricted class of local-effect actions and proper$^+$ KBs, they showed that progression is not only first-order definable but also efficiently 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 $\vec{c}$.

Definition 4 An SSA is local-effect if both $\gamma^+_F(\vec{x}, a, s)$ and $\gamma^-_F(\vec{x}, a, s)$ are disjunctions of formulas of the form $\exists \vec{z}[a = A(\vec{u}) \wedge \phi(\vec{u}, s)]$, where A is an action function, $\vec{u}$ contains $\vec{x}$, $\vec{z}$ is the remaining variables of $\vec{u}$. An action theory is local-effect if each SSA is local-effect.

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.

It is well known that forgetting an atom from a first-order formula can be done by a simple syntactic operation, and the result is a first-order formula.

Definition 5 An SSA is essentially quantifier-free if for each ground action α, by using $\mathcal{D}_{una}$, both $\gamma^+_F(\vec{x}, \alpha, s)$ and $\gamma^-_F(\vec{x}, \alpha, s)$ can be simplified to quantifier-free formulas.

Thus, when $\mathcal{D}_{ss}$ is essentially quantifier-free, its instantiation wrt a ground action α and S_0 is definable as a proper$^+$ KB. Liu and Lakemeyer (2009) proved the following:

Theorem 2 *Suppose that $\mathcal{D}$ is local-effect, $\mathcal{D}_{ss}$ is essentially quantifier-free, and $\mathcal{D}_{S_0}$ is proper$^+$. Then progression of $\mathcal{D}_{S_0}$ wrt any ground action α 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 now extend the notion of progression to accommodate sensing actions.

Definition 6 Let α be a ground sensing action and $\mu \in \{T, F\}$. Let $\mathcal{D}_{S_\alpha}$ be a set of sentences in $\mathcal{L}^2_{sc}$ uniform in S_α. $\mathcal{D}_{S_\alpha}$ is a progression of the initial KB $\mathcal{D}_{S_0}$ wrt α_μ if for any structure M, M is a model of $\mathcal{D}_{S_\alpha}$ iff there is a model M' of $\mathcal{D} \cup \{(\neg)SF(\alpha, S_0)\}$ such that $M \sim_{S_\alpha} M'$, where there is $\neg$ in front of SF iff $\mu = T$.

It is easy to show the following:

Theorem 3 *Let $\alpha = A(\vec{c})$ be a ground sensing action. Then $(\mathcal{D}_{S_0} \cup (\neg)\Psi_A(\vec{c}, S_0))(S_0/S_\alpha)$ is a progression of $\mathcal{D}_{S_0}$ wrt α_μ, where there is $\neg$ in front of Ψ_A iff $\mu = T$.*

If Ψ_A is quantifier-free, then $(\neg)\Psi_A$ can be converted into a CNF. Thus we have the following result for proper$^+$KBs:

Theorem 4 *If $\mathcal{D}_{sf}$ is quantifier-free and $\mathcal{D}_{S_0}$ is proper$^+$, then progression of $\mathcal{D}_{S_0}$ wrt any ground sensing action is definable as a proper$^+$ KB and can be efficiently computed.*

The subjective logic $\mathcal{SL}$

A popular way of specifying a limited reasoning service is through a logic of belief. 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) propose 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. To save space, later we will refer to (Liu, Lakemeyer, and Levesque 2004) by (LLL04).

The language $\mathcal{SL}$ is a first-order logic with equality whose atomic formulas are belief atoms of the form $\boldsymbol{B}_k\phi$ where ϕ is a formula of $\mathcal{L}$ 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 B_j for $j \leq k$. We call formulas of $\mathcal{L}$ objective formulas, and formulas of $\mathcal{SL}$ subjective formulas

To define the semantics of $\mathcal{SL}$, we introduce some notation. 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 $(\boldsymbol{B}_k\neg\exists x\phi)\downarrow = \forall x\boldsymbol{B}_k\neg\phi$.

Sentences of $\mathcal{SL}$ are interpreted via a *setup*, which is a set of non-empty *ground clauses*. Let s be a setup. For any sentence $\varphi \in \mathcal{SL}$, $s \models \varphi$ (read "s satisfies φ") 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 some $c \in s$ such that for all $\rho \in c$, $s \cup \{\rho\} \models \boldsymbol{B}_{k-1}\phi$.

As usual, a set Γ of sentences entails a sentence φ, written $\Gamma \models \varphi$, if for every setup s such that s satisfies every sentence of Γ, we have that s satisfies φ.

The $\mathcal{SL}$-based reasoning problem is to decide if $\boldsymbol{B}_0\Sigma \models \boldsymbol{B}_k\phi$, given a KB Σ and a query ϕ. In (LLL04), the authors showed that $\mathcal{SL}$-based reasoning for proper$^+$ KBs is decidable by presenting a decision procedure called W. Later, Liu and Levesque (**?**) showed that $\mathcal{SL}$-based reasoning with proper$^+$ KBs is not only decidable but also tractable when both the KB and the query use a bounded number of variables.

Closed-world assumption on knowledge

In his work on knowledge-based programming, Reiter (**?**) 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 7 $bcl(B_0\Sigma) \stackrel{def}{=} B_0\Sigma \cup \{\neg B_k\phi \mid B_0\Sigma \not\models B_k\phi\}$.

By an important result from (LLL04) that $\models \boldsymbol{B}_0\Sigma \supset \boldsymbol{B}_k\phi$ iff $\text{gnd}(\Sigma) \models \boldsymbol{B}_k\phi$, we have $bcl(B_0\Sigma) = B_0\Sigma \cup \{\neg B_k\phi \mid \text{gnd}(\Sigma) \models \neg B_k\phi\}$. Thus it is easy to prove

Theorem 5 *Let Σ be a proper$^+$ KB. Then*

1. *$bcl(B_0\Sigma)$ is satisfiable.*
2. *For any $\varphi \in \mathcal{SL}$, $bcl(B_0\Sigma) \models \varphi$ or $bcl(B_0\Sigma) \models \neg\varphi$.*
3. *For any $\varphi \in \mathcal{SL}$, $gnd(\Sigma) \models \varphi$ iff $bcl(B_0\Sigma) \models \varphi$.*

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 $B_k\phi$ 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 $\mathcal{E} \cup \Sigma$ classically entails φ_o. Thus we have

Theorem 6 *Let Σ be a complete proper KB. Then $bcl(B_0\Sigma) \models \varphi$ iff $\mathcal{E} \cup \Sigma$ classically entails φ_o.*

LBGolog: syntax and semantics

In this section, we introduce the syntax and semantics of LBGolog, and give examples of programs in LBGolog.

The following are the programming constructs of LBGolog. A difference with normal Golog is that all tests ϕ are $\mathcal{SL}$ formulas with only B_0 modal operators. For readability, we also write $B_0\psi$ 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.	$(\pi\vec{x}.\phi \wedge \delta)$	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 $\pi\vec{x}.\phi \wedge \delta$, 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 program 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. Unlike IndiGolog, the search operator returns a conditional program ready to be executed online, 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, every constant appearing in δ must be of type τ, δ is a basic program without sensing actions, and either δ is a procedure call itself, or δ does not contain any procedural call. $\Upsilon(\tau, \delta)$ 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 the latter requirement when we present the implementation of the planning operator.

We assume a local-effect BAT $\mathcal{D}$ such that $\mathcal{D}_{ss}$ is essentially quantifier, $\mathcal{D}_{sf}$ is quantifier-free, and $\mathcal{D}_{S_0}$ is proper$^+$. We call such a BAT well-formed. In the rest of this paper, 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 transi-

tion 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, which means that 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 $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi[\sigma]$. 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:

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

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

We first give the formal semantics for basic constructs. The set of final configurations wrt $\mathcal{D}$, denoted $\mathcal{F}_\mathcal{D}$, is inductively defined as follows:

1. $(nil, \sigma) \in \mathcal{F}_\mathcal{D}$.
2. $(\delta_1; \delta_2, \sigma) \in \mathcal{F}_\mathcal{D}$ if $(\delta_1, \sigma) \in \mathcal{F}_\mathcal{D}$ and $(\delta_2, \sigma) \in \mathcal{F}_\mathcal{D}$.
3. $(\delta_1 \mid \delta_2, \sigma) \in \mathcal{F}_\mathcal{D}$ if $(\delta_1, \sigma) \in \mathcal{F}_\mathcal{D}$ or $(\delta_2, \sigma) \in \mathcal{F}_\mathcal{D}$.
4. $(\pi\vec{x}.\phi \wedge \delta, \sigma) \in \mathcal{F}_\mathcal{D}$ if there exist constants $\vec{c}$ such that $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma) \in \mathcal{F}_\mathcal{D}$.
5. $(\delta^*, \sigma) \in \mathcal{F}_\mathcal{D}$.

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

1. $(\alpha, \sigma) \rightarrow_\mathcal{D} (nil, do(\alpha, \sigma))$ if $\alpha = A(\vec{c})$ is a physical action and $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models B_0\Pi_A(\vec{c}, \sigma)$. Recall that the action precondition axiom for $A(\vec{x})$ is $Poss(A(\vec{x}) \equiv \Pi_A(\vec{x}, s)$.
2. $(\alpha, \sigma) \rightarrow_\mathcal{D} (nil, do(\alpha_\mu, \sigma))$ if $\alpha = A(\vec{c})$ is a sensing action, the sensing result is μ, and $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models B_0\Pi_A(\vec{c}, \sigma)$.
3. $(\phi?, \sigma) \rightarrow_\mathcal{D} (nil, \sigma)$ if $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi[\sigma]$.
4. $(\delta_1; \delta_2, \sigma) \rightarrow_\mathcal{D} (\gamma; \delta_2, \sigma')$ if $(\delta_1, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$.
5. $(\delta_1; \delta_2, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$ if $(\delta_1, \sigma) \in \mathcal{F}_\mathcal{D}$ and $(\delta_2, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$.
6. $(\delta_1 \mid \delta_2, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$ if $(\delta_1, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$ or $(\delta_2, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$.
7. $(\pi\vec{x}.\phi \wedge \delta, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$ if there exist constants $\vec{c}$ s.t. $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$.
8. $(\delta^*, \sigma) \rightarrow_\mathcal{D} (\gamma; \delta^*, \sigma')$ if $(\delta, \sigma) \rightarrow_\mathcal{D} (\gamma, \sigma')$.

To define the semantics of search and planning operators, we define a relation $\mathcal{C}_\mathcal{D}$: intuitively, $(\delta, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ means that an offline-execution of program δ in situation σ results in conditional program ρ. To define $\mathcal{C}_\mathcal{D}$, we introduce an auxiliary relation $\mathcal{E}_\mathcal{D}$: intuitively, $(\rho, \delta, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$ means that in situation σ, executing conditional program ρ and then program δ, leads to conditional program ρ'. Formally, $\mathcal{C}_\mathcal{D}$ is inductively defined as follows:

1. $(nil, \sigma, nil) \in \mathcal{C}_\mathcal{D}$.
2. $(\alpha, \sigma, \alpha) \in \mathcal{C}_\mathcal{D}$ if α is $A(\vec{c})$ and $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models B_0\Pi_A(\vec{c}, \sigma)$.
3. $(\phi?, \sigma, nil) \in \mathcal{C}_\mathcal{D}$ if $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi[\sigma]$.
4. $(\delta_1; \delta_2, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ if there exists ρ' such that $(\delta_1, \sigma, \rho') \in \mathcal{C}_\mathcal{D}$ and $(\rho', \delta_2, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$.
5. $(\delta_1 \mid \delta_2, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ if $(\delta_1, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ or $(\delta_2, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$.
6. $(\pi\vec{x}.\phi \wedge \delta, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ if there exist constants $\vec{c}$ such that $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi(\vec{c})[\sigma]$ and $(\delta(\vec{c}), \sigma, \rho) \in \mathcal{C}_\mathcal{D}$.
7. $(\delta^*, \sigma, nil) \in \mathcal{C}_\mathcal{D}$; and $(\delta^*, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ if there exists ρ' such that $(\delta^*, \sigma, \rho') \in \mathcal{C}_\mathcal{D}$ and $(\rho', \delta, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$.

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

1. $(nil, \delta, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$ if $(\delta, \sigma, \rho') \in \mathcal{C}_\mathcal{D}$.
2. $(\alpha; \rho, \delta, \sigma, \alpha; \rho') \in \mathcal{E}_\mathcal{D}$ if α is a physical action and $(\rho, \delta, do(\alpha, \sigma), \rho') \in \mathcal{E}_\mathcal{D}$.
3. $(\alpha; \rho, \delta, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$ if α is a sensing action and there exist ρ_1 and ρ_2 such that $(\rho, \delta, do(\alpha_T, \sigma), \rho_1) \in \mathcal{E}_\mathcal{D}$ and $(\rho, \delta, do(\alpha_F, \sigma), \rho_2) \in \mathcal{E}_\mathcal{D}$ and ρ' is: α; **if Knows**$\Psi_A(\vec{c})$ **then** ρ_1 **else** ρ_2 **endIf**. Recall that the sensored fluent axiom for $A(\vec{x})$ is $SF(A(\vec{x}), s) \equiv \Psi_A(\vec{x}, s)$.
4. (**if** ϕ **then** ρ_1 **else** ρ_2 **endIf**$, \delta, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$ if the following holds: if $bcl(B_0 prog(\mathcal{D}_{S_0}, \sigma)) \models \phi[\sigma]$, then $(\rho_1, \delta, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$, otherwise $(\rho_2, \delta, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$.

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 $\forall x.$**Knows**$\tau(x) \supset \phi$, and $\exists x\phi$ with $\exists x.$**Knows**$\tau(x) \wedge \phi$, and replace each construct of the form $\pi\vec{x}.\phi \wedge \delta$ with $\pi\vec{x}.\bigwedge$**Knows**$\tau(x_i) \wedge \phi \wedge \delta$. We require that the initial KB $\mathcal{D}_{S_0}$ contains a sentence of the form $\forall x.\tau(x) \equiv x = d_1 \vee \ldots \vee x = d_n$.

We can now expand the definition of the transition relation with the following items:

8. $(\Sigma\delta, \sigma) \rightarrow_\mathcal{D} (\rho, \sigma)$ if $(\delta, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$.
9. $(\Upsilon(\tau, \delta), \sigma) \rightarrow_\mathcal{D} (\rho, \sigma)$ if $(\delta_\tau, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$.

Finally, an online execution of a LBGolog program δ_0 starting from a situation σ_0 is a sequence of configurations $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$, such that for $i < n$, $(\delta_i, \sigma_i) \rightarrow_\mathcal{D} (\delta_{i+1}, \sigma_{i+1})$. The online execution is successful if $(\delta_n, \sigma_n) \in \mathcal{F}_\mathcal{D}$.

We now illustrate programming in LBGolog with the Wumpus World (**?**). Below is the main program, where n_1 is a constant for coordinate 1. The agent first senses the environment. If she knows that the gold is at location $(1, 1)$, she grabs the gold and climbs out of the dungeon. Otherwise, she explores the dungeon, 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** $grab; climb$
  else $explore; \Upsilon moveLoc(n_1, n_1); climb$ **endIf**
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 $moveLoc(X, Y)$
$[\pi x_0, y_0, x_1, y_1.\textbf{Knows}(agt(x_0, y_0) \wedge explored(x_1, y_1)) \wedge move(x_0, y_0, x_1, y_1)]^*$;
$\pi x_2, y_2.\textbf{Knows}(agt(x_2, y_2)) \wedge move(x_2, y_2, X, Y)$
endProc

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

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

Finally, the procedure for shooting the wumpus. Here $succ(x, y)$ means that y is the successor coordinate of x. The agent picks an explored location (x_2, y_2) which is adjacent to the wumpus' location, moves to (x, y), shoots and senses if there is a scream.

proc $shootWumpus$
$[\pi x_1, y_1, x_2, y_2.\textbf{Knows}(wp(x_1, y_1) \wedge explored(x_2, y_2) \wedge$
$(x_1 = x_2 \wedge succ(y_1, y_2)) \vee (x_1 = x_2 \wedge succ(y_2, y_1))$
$(y_1 = y_2 \wedge succ(x_1, x_2)) \vee (y_1 = y_2 \wedge succ(x_2, x_1))) \wedge$
$\Upsilon moveLoc(x_2, y_2)$;
$(succ(y_2, y_1)?; shoot\_up \mid succ(y_1, y_2)?; shoot\_down \mid$
$succ(x_1, x_2)?; shoot\_left \mid succ(x_2, x_1)?; shoot\_right)]$;
$sense\_scream$
endProc

Implementing progression and query evaluation by grounding

As shown in the last section, to implement LBGolog, we need to implement progression and evaluation of a $\mathcal{SL}$ formula against the closure of $B_0\mathcal{D}_\sigma$, where $\mathcal{D}_\sigma$ is the current KB. Initially, we implemented the progression algorithm from (Liu and Lakemeyer 2009) and the query evaluation algorithm from (LLL04). However, the implementation was not efficient. So we decided to implement progression and query evaluation by grounding. But we have an infinite domain. The trick is to use an appropriate number of special constants as representatives of those infinitely many constants not mentioned by the KB.

Here is the general picture. We first ground the initial KB, perform unit propagation on it. When an action is performed, if the action mentions new constants, we extend the current ground KB with these constants, then we progress the ground KB, and perform unit propagation on it. Whenever we need to evaluate a query, we use the current ground KB to answer the query. In the following, we present grounding, progression, and query evaluation in sequence.

Grounding

We begin with initial 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. To simplify presentation, we assume that there are 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 let U denote the set of these constants, and 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 8 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.

The intuition is that constants not appearing in Σ behave the same, and we take U constants as their representatives. 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 9 We define $FO(\Sigma_p)$ as follows: replace each c in Σ_p with $FO(c)$, denoting $\forall(e \supset c(u_1/x_1, \ldots, u_j/x_j))$, 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 $\bigwedge_{d \in N} x \neq d$.

Let Γ_1 and Γ_2 be two sets of sentences. We write $\Gamma_1 \Leftrightarrow_{\mathcal{E}} \Gamma_2$ if $\mathcal{E} \cup \Gamma_1 \models \Gamma_2$ and $\mathcal{E} \cup \Gamma_2 \models \Gamma_1$.

Theorem 7 $FO(prop(\Sigma, N)) \Leftrightarrow_{\mathcal{E}} \Sigma$.

We now define extended grounding.

Definition 10 Let B be a finite set of normal constants s.t. $B \cap H(\Sigma_p) = \emptyset$. We define $egnd(\Sigma_p, B)$ inductively as follows:

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. $egnd(\Sigma_p, \{d\} \cup B) = egnd(egnd(\Sigma_p, \{d\}), B)$.

The following shows correctness of extended grounding.

Theorem 8 $egnd(prop(\Sigma, N), B) \Leftrightarrow_{\mathcal{E}} prop(\Sigma, N \cup B)$.

Note that the way we do grounding is brute-force. For example, if Σ contains $\forall x P(x)$, then its ground KB contains $P(u_1), \ldots, P(u_j)$, 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 that it is equivalent to progression of the original KB. Recall the notation from the background work section on progression.

Definition 11 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 a physical action, and $\Sigma_p(S_0/S_\alpha) \cup (\neg)\Psi_A(\vec{c}, S_\alpha)$ if α is a sensing action.

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. The following lemma establishes connection between forgetting a ground atom from a proper$^+$ KB Σ and from its ground KB.

Lemma 9 *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, 7, 8 and Lemma 9, we have

Theorem 10 $FO(pprog(prop(\Sigma, N), \alpha)) \Leftrightarrow_{\mathcal{E}} prog(\Sigma, \alpha)$.

Query Evaluation

We say that a query ϕ is suitable for Σ_p if for each clause c in ϕ, the total number of variables in c and constants in c but not Σ_p is no more than the number 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) where $k = 0$ except for the case of evaluating clauses.

$$G[\Sigma_p, \phi] = \begin{cases} 1 & \text{if one of the following conditions holds.} \\ 0 & \text{otherwise.} \end{cases}$$

1. ϕ is a clause c and there exists a clause $c' \in \mathsf{UP}(\Sigma_p)$ 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 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 $G[\Sigma_p, \psi^x_d] = 1$ for some $d \in H^+(\Sigma_p \cup \{\psi\})$.
8. $\phi = \neg\exists x\psi$ and $G[\Sigma_p, \neg\psi^x_d]$ for all $d \in H^+(\Sigma_p \cup \{\psi\})$.

Based on G, we now define an evaluation procedure $F[\Sigma_p, \varphi]$, where $\varphi \in \mathcal{SL}_0$ is suitable for Σ_p.

$$F[\Sigma_p, \varphi] = \begin{cases} 1 & \text{if one of the following conditions holds.} \\ 0 & \text{otherwise.} \end{cases}$$

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. $\varphi = \varphi_1 \vee \varphi_2$, and $F[\Sigma_p, \varphi_1] = 1$ or $F[\Sigma_p, \varphi_2] = 1$.
5. $\varphi = \exists x\omega$, and $F[\Sigma_p, \omega^x_d] = 1$ 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 11 $F[\Sigma_p, \varphi] = 1$ *iff* $gnd(FO(\Sigma_p)) \models \varphi$.

By Lemma 11 and Theorem 5(3), we have

Theorem 12 $F[\Sigma_p, \varphi] = 1$ *iff* $bcl(B_0 FO(\Sigma_p)) \models \varphi$.

An interpreter

We have implemented an interpreter for *$\mathcal{LB}$Golog* 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, ϕ`)`: formula ϕ is the precondition for action α;
- `ssa(`F, γ^+, γ^-`)`: γ^+ is the condition for making F true, and γ^- is the condition for making F false;
- `sf(`β, ϕ`)`: β senses whether ϕ holds.

Since progression and query evaluation are the most frequent operations during the execution of the interpreter, to improve efficiency, we implement the core parts of the two operations in C, and provide the following primitive predicates in Prolog:

- `query(`ϕ, s`)`: evaluate formula ϕ in situation s;
- `query(what(`$\vec{c}, \vec{x}, \phi), s$`)`: return $\vec{c}$ such that subjective formula $\phi^{\vec{x}}_{\vec{c}}$ is evaluated true in situation s;
- `prim_prog(`α, s, s'`)`: progress the KB of situation s to situation s' wrt primitive action α;
- `sens_prog(`β, r, s, s'`)`: progress the KB of situation s to situation s' wrt sensing action β with sensing result r;
- `del_sit(`s`)`: delete the KB about situation s.

All KBs are stored as data structures in *C*. The two progression operators yield the new KBs while keeping the old ones. Thus we need the `del_sit` predicate.

In the following, we present the implementation of the basic constructs, search and planning operators in sequence, and end with correctness theorems of the interpreter.

Basic constructs

We define predicates `btrans/3` and `bfinal/1` to implement the $\mathcal{F}$ and $\rightarrow$ relations in the semantic definition. Note that we omit the current situation σ_c from the arguments. Thus `bfinal(`δ`)` represents $(\delta, \sigma_c) \in \mathcal{F}$, and `btrans(`$\delta, \delta', \alpha$`)` means $(\delta, \sigma_c) \rightarrow_{\mathcal{D}} (\delta', do(\alpha, \sigma_c))$. 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 `subl(`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.

```
bfinal(nil).
bfinal(E1:E2):-bfinal(E1),bfinal(E2).
bfinal(E1#E2):-bfinal(E1);bfinal(E2).
```

```
bfinal(pi(L,G,E)):-query(what(L1,L,G)),
    subl(L,L1,E,E1),bfinal(E1).
bfinal(star(_)).
bfinal(E):-proc(E,E1),bfinal(E1).

btrans(A,nil,A):-prim_action(A),
    poss(A,P),query(knows(P)).
btrans(?(P),nil,nil):-query(P).
btrans(E1:E2,E,A):-btrans(E1,E3,A),
    E=(E3:E2);bfinal(E1),btrans(E2,E,A).
btrans(star(E),E1:star(E),A):-
    btrans(E,E1,A).
```

The top part of the interpreter uses `btrans` and `bfinal` to determine the next action to perform or to terminate. To perform an action, do the corresponding input/output actions, and then do progression. Predicate `curr_sit`(s) maintains the current situation, and `update_sit`(s) updates the current situation and deletes the KB of the old situation.

```
lbGolog(E):-btrans(E,E1,A),
    (A=nil->true;do(A)),!,lbGolog(E1).
lbGolog(E):-bfinal(E),!.

do(A):-execute(A),prim_prog(A).
do(B):-execute(B,R),sens_prog(B,R).
execute(A):-prim_action(A),writeln(A).
execute(B,R):-sens_action(B),write(B),
    write(':(y/n)'),read(R).

query(P):-curr_sit(S),query(P,S).
prim_prog(A):-curr_sit(S),
    prim_prog(A,S,S1),update_sit(S1).
sens_prog(B,R):-curr_sit(S),
    sens_prog(B,R,S,S1),update_sit(S1).
update_sit(S):-retract(curr_sit(S0)),
    del_sit(S0),assert(curr_sit(S)).
```

Search operator Σ

We define predicates $bdo/3$ and $ext/4$ to implement relations $\mathcal{C}$ and $\mathcal{E}$ respectively. We present only some of the clauses. Note that when implementing search, we explore a tree of situations, and maintain KBs of different situations. Once search succeeds or backtracks, we delete KBs accordingly.

```
bdo(B,S,B):-sens_action(B),poss(B,P),
    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),(ext(C,E,S1,C1),
    del_sit(S1);del_sit(S1),fail).
ext(B:C,E,S,C1):-sens_action(B),
    sens_prog(B,1,S,ST),(ext(C,E,ST,CT),
    del_sit(ST);del_sit(ST),fail),
    sens_prog(B,0,S,SF),(ext(C,E,SF,CF),
    del_sit(SF);del_sit(SF),fail),
    sf(B,F),C1=(B:if(knows(F),CT,CF)).
ext(if(P,C1,C2),E,S,C):-query(P,S)->
    ext(C1,E,S,C);ext(C2,E,S,C).
```

Then the `btrans` clause for the search operator is:

```
btrans(search(E),E1,nil):-curr_sit(S),
    bdo(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 δ below for catching a plane:
$sense\_gate\_A$; buy_paper;
$(goto(gate\_A); buy\_coffee \mid buy\_coffee; goto(gate\_B))$;
$board(plane)$
Assume that there are only two gates A and B, and $sense\_gate\_A$ tells the agent which gate to take. Note that $board(plane)$ is executable only if the agent gets to the right gate. So an online execution of δ might not be successful. This problem can be solved by using the search operator. The interpretation of $\Sigma\delta$ results in the following program, whose online execution is guaranteed to be successful.
$sense\_gate\_A$;
if Knows$(it\_is\_gate\_A)$
then $buy\_paper; goto(gate\_A); buy\_coffee; board(plane)$
else $buy\_paper; buy\_coffee; goto(gate\_B); board(plane)$,

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 functions, operators, and domain objects. A problem consists of an initial state and a goal. An operator is a tuple $\langle O(\vec{x}), Pre(\vec{x}), Eff(\vec{x})\rangle$, where $O(\vec{x})$ is the operator name, $Pre(\vec{x})$ is the precondition axiom, and $Eff(\vec{x})$ is a list of conditional effect axioms. Not that quantifiers are allowed in the axioms.

Baier *et al.* define a translating 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 ADL operator $O(A(\vec{x})) = \langle A(\vec{x}), Pre(\vec{x}), Eff(\vec{x})\rangle$, where $Pre(\vec{x})$ is directly obtained from the action precondition axiom, and $Eff(x)$ is obtained from the SSAs. We omit the details here. We use $\mathcal{P}_\mathcal{D}(\delta)$ to denote the set of predicates relevant

to program δ wrt BAT $\mathcal{D}$, that is, the set of predicates that occur in tests of δ or precondition or effect axioms for actions that occur in δ.

Definition 12 (Translation function $\mathcal{T}$) 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 $\mathcal{P}_{\mathcal{D}}(\delta)$;
2. the operations 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. $P \in \mathcal{P}_{\mathcal{D}}(\delta)$, $\vec{c} \in \tau$ and $\mathcal{D}_\sigma$ is the KB of σ.
5. the goal is $true$.

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

Definition 13 We say that a ground situation σ is just-in-time for $\Upsilon(\tau, \delta)$ wrt $\mathcal{D}$, if for every ground atom $P(\vec{c})$ such that $P \in \mathcal{P}_{\mathcal{D}}(\delta)$ and $\vec{c} \in \tau$, $B_0 prog(\mathcal{D}_{S_0}, \sigma) \models B_0 P(\vec{c})$ or $B_0 prog(\mathcal{D}_{S_0}, \sigma) \models B_0 \neg P(\vec{c})$.

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. The semantics of LBGolog corresponds to that of the Golog-like language by Baier *et al.* except that we apply $\mathcal{SL}$-based reasoning to tests and they evaluate tests wrt databases. However, under the just-in-time assumption, by a generalize version of Theorem 6, $\mathcal{SL}$-based reasoning coincides with database query evaluation. So we get (Recall that δ_τ denotes the restriction of δ to τ.)

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

We implement a predicate `plan`$(\tau, \delta, \sigma, \delta')$ which does the following: First, apply $\mathcal{T}$ on $(\mathcal{D}, \tau, \delta, \sigma)$ to generate a planning instance I. Then apply Baier *et al.* ś translation (with a slight modification) on (I, δ_o) to obtain a planning instance I', call FF planner (**?**) on I' to get a plan ρ. Finally, filter out the auxiliary actions from ρ. Then the `btrans` clause for the planning operator is:

```
btrans(planning(T,E),E1,nil):-
    curr_sit(S),proc(E,E2)->
    plan(T,E2,S,E1);plan(T,E,S,E1).
```

Actually, the domain part of the planning instance I' 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

Due to correctness of progression and query evaluation (Theorems 10 and 12), by induction on the program, it is easy to prove the following two theorems:

Theorem 14 (Correctness of basic constructs) *Let $\mathcal{D}$ be a well-formed BAT, δ a basic program, and σ a ground situation. Then*

1. bfinal*(δ) wrt σ succeeds iff $(\delta, \sigma) \in \mathcal{F}_{\mathcal{D}}$;*
2. *if* btrans*(δ, P, A) wrt σ succeeds with $P = \delta'$ and $A = \alpha$, then $(\delta, \sigma) \rightarrow_{\mathcal{D}} (\delta', do(\alpha, \sigma))$, where $do(nil, \sigma) = \sigma$;*
3. *if $(\delta, \sigma) \rightarrow_{\mathcal{D}} (\delta', \sigma')$, then* btrans*($\delta$, P, A) wrt σ succeeds.*

Theorem 15 (Soundness and weak completeness of search) *Let $\mathcal{D}$ be a well-fromed BAT, δ a basic program, and σ a ground situation. Then we have*

1. *if* btrans*(search(δ), P, A) wrt σ succeeds with $P = \delta'$, then $(\delta, \sigma, \delta') \in \mathcal{C}_{\mathcal{D}}$;*
2. *if $(\delta, \sigma, \delta') \in \mathcal{C}_{\mathcal{D}}$ for some δ', then* btrans*(*search*(δ), P, A) wrt σ succeeds or does not terminate.*

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

The theorem below follows from Lemma 13:

Theorem 16 (Correctness of planning operator) *Suppose σ is just-in-time for $\Upsilon(\tau, \delta)$ wrt well-formed BAT $\mathcal{D}$. Then*

1. *if* btrans*(*planning*(τ, δ), P, A) wrt σ succeeds with $P = \delta'$, then $(\delta_\tau, \sigma, \delta') \in \mathcal{C}_{\mathcal{D}}$;*
2. *if $(\delta_\tau, \sigma, \delta') \in \mathcal{C}_{\mathcal{D}}$ for some δ', then* btrans*(*planning*(τ, δ), P, A) wrt σ succeeds.*

Experiments

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

Wumpus world

We have written a control program for Wumpus world and executed it with our interpreter. When writing the program, we take a cautious strategy and ensure that the agent gets out the dungeon alive. We assume that there is only one piece of gold. On the premise of safety, the program would control the agent to get the gold as much as possible. Table 1 shows the experiment results for 8×8 maps. Each row represents a setting, and for each setting, we tested our control program on 3000 random maps.

Prob	Gold	IMP	Reward	Moves	Time	Calls
10%	1412	695	437	34	0.398	16
15%	890	917	275	22	0.246	11
20%	567	1171	175	14	0.147	7
30%	263	1581	82	6	0.070	3
40%	182	1924	58	3	0.040	2

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

In Table 1, *Prob* is the probability of a location containing a pit; *Gold* is the number of maps where the agent got the gold; *IMP* is the number of maps for which it is impossible to get the gold, that is, the agent sensed smell or stench at the initial location. The rest of the columns show the average of the reward, the number of moves, the running time in seconds, and the number of calling the FF planner.

As the probability goes up, *IMP* goes up, but all the other indexes go down. In other words, the lower the probability is, the more likely it is for the agent to explore the dungeon

and get the gold. Note that, even for the lowest probability, the average running time is less than 0.4 seconds, which shows the efficiency of our interpreter.

Blocks World

In this domain we assume a number of blocks on or above the table, and the goal of the agent is to make clear some of them. The only primitive action is $move(x, y, z)$, moving block x from the top of y to the top of z. There are 2 fluents: $clear(x)$, denoting that there are no blocks on top of x, and $on(x, y)$, denoting that x is on top of y.

The relevant axioms as below:

$Poss(move(x, y, z), s) \equiv on(x, y) \wedge clear(x) \wedge clear(y)$,
$Poss(sense\_on(x, y), s) \equiv true$,
$Poss(sense\_clear(x), s) \equiv true$,
$clear(x, do(a, s)) \equiv (\exists y)a = moveToTable(y, x) \vee clear(x, s)$,
$on(x, y, do(a, s)) \equiv (\exists z)a = move(x, z, y) \vee on(x, y, s) \wedge (\neg\exists z)a = move(x, y, z)$,
$SF(sense\_on(x, y), s) \equiv on(x, y, s)$.

The initial KB is as follows:

$\forall x.x \neq a \wedge x \neq b \wedge x \neq c \wedge x \neq d \supset clear(x)$,
$\forall x, y.x \neq a \wedge x \neq b \wedge x \neq c \wedge x \neq d \wedge x \neq y \supset \neg on(x, y)$,
$\forall x, y.x \neq y \supset (on(x, y) \supset \neg clear(y))$,
$\forall x.\neg on(x, x)$,
$\forall x, y.x \neq y \supset (on(x, y) \supset \neg on(y, x))$,
$\forall x, y, z.y \neq z \supset \neg(on(x, y) \wedge on(x, z))$,
$\forall x, y, z.y \neq z \supset \neg(on(y, x) \wedge on(z, x))$.

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 $\neg\forall b_1$.**Knows**$(b_1 \in L) \supset$ **Knows**$(clear(b_1))$
then πb_2.**Knows**$(b_2 \in L) \wedge \neg$**Knows**$(clear(b_2))$
$\wedge(make\_clear(b_2, L); make\_clear\_all(L))$ **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 πb_2.**Knows**$(on(b_2, x))$
$\wedge(make\_clear(b_2, L); move\_away(b_2, x, L))$
else $\pi b_3.\neg$**KWhether**$(on(b_3, x))$
$\wedge(sense\_on(b_3, x); make\_clear(x, L))$ **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 πb_2.**Knows**$(y \neq b_2 \wedge clear(b_2) \wedge b_2 \notin L)$
$\wedge move(y, x, b_2)$
else πb_3.**Knows**$(b_3 \notin L) \wedge \neg$**KWhether**$(clear(b_3))$
$\wedge(sense\_clear(b_3); move\_away(y, x, L))$ **endIf**
endProc

An example execution is given below:

```
?- 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 mentioned block, the agent attempts to find an extra block and at last, she successfully accomplishes her tasks with two extra blocks `c1` and `c2`.

Conclusions

We now discuss related work other than those we have mentioned in the introduction. Petrick and Bacchus (2002) proposed a planning system with incomplete information and sensing called **PKS**. Their system is also based on the domain closure assumption. It can only handle a special form of disjunctive knowledge in the form of exclusive disjunctive knowledge. Their system is based on approximate progression reasoning procedures without semantic characterizations. However, **PKS** has support for functions. Classen *et al.* proposed an integration of Golog and planning. However, their implementation of Golog is also based on CWA. Classen and Lakemeyer proposed Golog with disjunctive knowledge bases where their reasoing is also based on SL and they provide a search operator that finds plans within only a fixed belief level k. However, their system is also based on regression.

To summarize, in this paper, we have presented 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 incomplete information in the form of proper$^+$ KBs. Our limited reasoning is based on the subjection logic $\mathcal{SL}$ and so far we only implement reasoning at the B_0 level. Our progression is based on the progression algorithm from (LL09). To improve efficiency, we implement these two algorithms by grounding. The interpreter is offline but the programmer can . In the future, we would like to implement reasoning at the B_1 level and explore the support of state constraints in our interpreter.

References

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

Claßen, J., and Lakemeyer, G. 2009. Tractable first-order golog with disjunctive knowledge bases. In *Proceedings of the Ninth International Symposium on Logical Formalizations of Commonsense Reasoning (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.

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

Levesque, H. J. 1996. What is planning in the presence of sensing? In *AAAI*, 1139–1146.

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. of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, 212–221.

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

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