First Order Golog based on Exact Progression and Limit Reasoning

AAAI Press
Association for the Advancement of Artificial Intelligence
445 Burgess Drive
Menlo Park, California 94025

Abstract

1 An interpreter of $\mathcal{LB}$*Golog*

The interpreter of $\mathcal{LB}$*Golog* is implemented in *PROLOG* language, based on *GOLOG*, with progression of KB and evaluation of formula under B_0 in open world.

Firstly we need the specification of BAT provided by users, as follows:

•`fluent_list(`l`)`: l is the list of fluent declarations;

•`predicate_list(`l`)`: l is the list of predicate declarations;

•`individual_list(`l`)`: l is the list of the known individuals;

•`init_kb(`l`)`: l is the list of initial KB;

•`prim_action(`α`)`: α is a primitive action;

•`sens_action(`β`)`: β is a sensing action;

•`poss(`α`,`ϕ`)`: formula ϕ is the precondition of action α;

•`ssa(`f`,`γ^+`,`γ^-`)`: the successor state axiom of fluent f consists of formulas γ^+ and γ^-;

•`sf(`β`,`ϕ`)`: objective formula ϕ is hold as the result of sensing action β is 1.

As that evaluation and progression are the mostly frequent operations in the interpreter, to increase the efficiency of calculating, we implement the cores of the two operations in *C* language, and then encapsulate them as the primitive operations in *PROLOG*, showed as follows:

•`query(`ϕ`,`s`)`: evaluate subjective formula ϕ in situation s;

•`query(what(`$\vec{c}$`,` $\vec{x}$`,` ϕ`),` 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 situation s to situation s' wrt. primitive action α;

•`sens_prog(`β`,`r`,`s`,`s'`)`: progress situation s to situation s' wrt. sensing action β with sensing result r;

•`del_sit(`s`)`: delete the KB about situation s.

All the operations above are relevant with the KB about situation, which is also stored with the data structures defined in *C*. To noted, the two progression operators yield both the KBs about the new situation and the old one. In this case, we support the deletion of KB to remove the trashy KB from memory.

Basic structures

We define predicates `btrans/3` and `bfinal/1` as the implementation of *BTrans/BFinal* semantics. Predicate `btrans(`δ`,` δ'`,` α`)` implements that program δ executes action α in current situation, and then with program δ' remaining. And predicate `bfinal(`δ`)` checks that program δ terminates in current situation.

We post the the implementations of basic structures, the structures except search operator and planning operator, as follows:

```
bfinal(nil).
bfinal(?(P)):-query(P).
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(if(P,E1,E2)):-query(P)->
    bfinal(E1);bfinal(E2).
bfinal(while(P,E)):-query(P)->bfinal(E);
    true.
bfinal(E):-proc(E,E1),bfinal(E1).

btrans(A,nil,A):-prim_action(A),
    poss(A,P),query(knows(P)).
btrans(B,nil,B):-sens_action(B),
    poss(B,P),query(knows(P)).
btrans(E1:E2,E,A):-btrans(E1,E3,A),
    E=(E3:E2);bfinal(E1),btrans(E2,E,A).
btrans(E1#E2,E,A):-btrans(E1,E,A);
    btrans(E2,E,A).
btrans(pi(L,G,E),E1,A):-
    query(what(L1,L,G)),subl(L,L1,E,E2),
    btrans(E2,E1,A).
btrans(star(E),E1:star(E),A):-
    btrans(E,E1,A).
btrans(if(P,E1,E2),E,A):-query(P)->
    btrans(E1,E,A);btrans(E2,E,A).
btrans(while(P,E),E1,A):-query(P),
    btrans(E:while(P,E),E1,A).
btrans(E,E1,A):-proc(E,E2),
```

```
    btrans(E2,E1,A).

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

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)).
```

As above, predicate `subl(`l_x`, `l_c`, `p_x`, `p_c`)` is to substitute all variables of l_x occurring in program p_x with corresponding constants of l_c, to a new program p_c. And predicate `curr_sit(`s`)` records the current situation s.

Predicate `lbGolog(`δ`)` is the main loop with program δ as input. Predicate `do(`α`)` calls for the progression of action α after execute it. Predicates `query(`ϕ`)`, `prim_prog(`α`)` and `sens_prog(`β`, `r`)` are executed evaluation and progression respectively in current situation. Moreover, predicate `update_sit(`s`)` is provided to update the current situation and delete the KB of old situation after progression.

From the definitions above, the input program, with takes `lbGolog/1` as the entrance of interpreter; then performs `btrans/3` for actions recursively, executing every action by `do/1`; and lastly terminates with the success of `bfinal/1`. In such a case, we can conclude that the program runs online.

And we mark the program only with basic structures as a *basic program*.

Search operator Σ

As the implementation of search operator Σ, we define predicates $bdo/3$ and $ext/4$ as follows:

```
bdo(nil,_,nil).
bdo(A,S,A):-prim_action(A),poss(A,P),
    query(knows(P),S).
bdo(B,S,B):-sens_action(B),poss(B,P),
    query(knows(P),S).
bdo(?(P),S,nil):-query(P,S).
bdo(E1:E2,S,C):-bdo(E1,S,C1),
    ext(C1,E2,S,C).
bdo(E1#E2,S,C):-bdo(E1,S,C);bdo(E2,S,C).
bdo(pi(L,G,E),S,C):-
    query(what(L1,L,G),S),
    subl(L,L1,E,E1),bdo(E1,S,C).
bdo(star(E),S,C):-C=nil;bdo(E,S,C1),
    ext(C1,star(E),S,C).
bdo(if(P,E1,E2),S,C):-query(P,S)->
    bdo(E1,S,C);bdo(E2,S,C).
bdo(while(P,E),S,C):-query(P,S)->
    bdo(E,S,C1),ext(C1,while(P,E),S,C);
    C=nil.
bdo(E,S,C):-proc(E,E1),bdo(E1,S,C).

ext(nil,E,S,C):-bdo(E,S,C).
ext(A,E,S,A:C):-prim_action(A),
    prim_prog(A,S,S1),(bdo(E,S1,C),
    del_sit(S1);del_sit(S1),fail).
ext(B,E,S,C):-sens_action(B),
    sens_prog(B,1,S,ST),(bdo(E,ST,CT),
    del_sit(ST);del_sit(ST),fail),
    sens_prog(B,0,S,SF),(bdo(E,SF,CF),
    del_sit(SF);del_sit(SF),fail),
    sf(B,F),C=(B:if(knows(F),CT,CF)).
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).

btrans(search(E),E1,nil):-curr_sit(S),
    bdo(E,S,E1).
```

As definitions above, predicates `bdo/3` and `ext/4` implement the $\mathcal{C}_\mathcal{D}$ relation and $\mathcal{E}_\mathcal{D}$ relation respectively. Noted that, the situation in searching procedure is changed by the progression of action in "mind". That means the KBs of different situations are concurrently exist in memory. Therefore once search succeeds or backtracks, we should delete the KB of the situation used in the search, except the one of real current situation.

Search implementation guarantees a program with sensing action to work fine, even thought the program is not well formed. For example of catch plane domain, a program δ is formed as:
$sense\_gate\_A; buy\_paper;$
$(goto(gate\_A); buy\_coffee | buy\_coffee; goto(gate\_B));$
$board(plane),$
where actions mentioned are defined literally. To be noted that, $sense\_gate\_A$ is a sensing action to sense whether gate A or gate B to board plane, and $board(plane)$ is successful only if the agent get to the right gate to board plane. δ is not well formed, i.e. it might not work successfully online, because of no control to force the agent to get to the right gate.

This problem can be solved by search operator. The interpretation of $\Sigma\delta$ results in the program δ' as:
$sense\_gate\_A;$
if $Knows(\psi)$
then $buy\_paper; (goto(gate\_A); buy\_coffee; board(plane)$
else $buy\_paper; buy\_coffee; goto(gate\_B)); board(plane),$
where $SF(sense\_gate\_A, \sigma) \equiv \psi[\sigma]$ is hold for an ar-

bitrary situation σ. And the search implementation of Σ operator can guarantee the program δ' always execute online successfully, of which the executions are accepted by δ.

Planning operator Υ

This part is contributed to implement the planning operator Υ, to plan a sequence of actions for a program. The main idea is to construct a planning instance from the BAT and program, and leave the plan task to an existing planner.

A planning instance is a pair, marked as $I = (D, P)$, where D is a domain definition and P is a problem definition. To simplify, we discuss the D and P described in *ADL*. A domain definition contains mainly domain predicates and operators, while a problem definition consists of objects present in I, initial state description and goal.

DCK planning Here we cite the work of Sheila et al. in (Baier, Fritz, and McIlraith 2007), of which the main contribution is providing an algorithm to merge a planning instance and a program into an equivalent planning instance, whose plans adhere to the program.

They define a compiling function $C(\delta, n, E) = (L, L', n')$, as to translate program δ with variables list E into an ε-NFA, and then to feature the auxiliary operators, like $test$, $noop$ and $free$, in list L, and the additional restrictions of domain operators in list L', where integers n and n' represent that δ in state S_n results in state $S_{n'}$. In this way they can get the control information of program.

Given a planning instance $I = (D, P)$ and a program δ, let (L, L', n_f) be the value of $C(\delta, 0, [])$, their algorithm, which we called *compiling algorithm*, is to restrict every operator in D with the operator of the same name in L'; append the auxiliary operators to D; extend domain predicates in D with $bound/1$, $map/2$ and $state/1$, which are the auxiliary ones in compiling; extend objects in P with the ones represent the program variables and automaton's states; add $state(s_0)$ to initial state; and add $state(s_{n_f})$ to goal in P. And the correctness of the compiling algorithm is illustrated in their paper.

Implementation Here the implementation of planning operator Υ is to translate the BAT to a planning instance I wrt. δ, and utilize the compiling algorithm to construct a new planning instance I' with I and program δ. And we mainly introduce the procedure of translating a BAT wrt. a program to a planning instance.

Definition 1.1 Given an action $\alpha(\vec{c})$ and an SSA(Successor State Axiom) formula ψ of the form $F(\vec{x}, do(a, s)) \Leftrightarrow \gamma^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma^-(\vec{x}, a, s)$, define $Eff(\alpha(\vec{c}), \psi)$ as the *eff-formula* for $\alpha(\vec{c})$ with ψ, if $Eff(\alpha(\vec{c}), \psi) \equiv (\gamma^+(\vec{e}, s) \supset F(\vec{e}, do(\alpha(\vec{c}), s))) \wedge (\gamma^-(\vec{e}, s) \supset \neg F(\vec{e}, do(\alpha(\vec{c}), s)))$, where $\gamma^+(\vec{e}, s)$ and $\gamma^-(\vec{e}, s)$ are exactly the $\gamma^+|^a_{\alpha(\vec{c})}$ and $\gamma^-|^a_{\alpha(\vec{c})}$ eliminated all action functions with UNA(Unique Name Axiom), and $\vec{e}$ is the binding of $\vec{x}$ with equality axiom.

Definition 1.1 provides the method of transferring an SSA to an effect formula wrt. a given action, which refers to (Reiter 2001).

Definition 1.2 Given a BAT $\mathcal{D}$ and an action $\alpha(\vec{c})$, define $OP_{\mathcal{D}}(\alpha(\vec{c})) =< a, \vec{x}, Pre(\vec{x}), Eff(\vec{x}) >$ as the *action description of* $\alpha(\vec{c})$ *wrt.* $\mathcal{D}$, if:

1. $a = \alpha$;
2. $\vec{x} = \vec{c}$;
3. $Pre(\vec{c}) \equiv Poss(\alpha(\vec{c}))$, where $Poss(\alpha(\vec{c}))$ is exactly the $Poss(\alpha(\vec{c}), s)$ removed the situation argument s;
4. $Eff(\vec{c}) \equiv \bigwedge_{\psi \in \mathcal{D} \wedge SSA(\psi)} Eff'(\alpha(\vec{c}), \psi)$, where $Eff'(\alpha(\vec{c}), \psi)$ is exactly the $Eff(\alpha(\vec{c}), \psi)$ removed the situation functions.

And now by definition 1.2, we translate the specification of action to an operator description.

Definition 1.3 Given a BAT $\mathcal{D}$ and a program δ, define $P_{\mathcal{D}}(\delta) = S_p \cup S_f$ as the *set of predicates relevant to* δ *wrt.* $\mathcal{D}$, where:

1. S_p is the subset of predicates in $\mathcal{D}$, of which the predicates occur in the formula tests of δ or the action description of actions of $\mathcal{D}$ mentioned in δ;
2. S_f of fluent respectively.

Definition 1.4 Given a BAT $\mathcal{D}$, we mark $OBJ_{\mathcal{D}}$ as the set consisting of the individuals known in $\mathcal{D}$, which means the individuals are either in the individual list defined by users, or mentioned in $\mathcal{D}_{S_0}$.

We restrict that the individuals mentioned in control programs should occur in the individual list.

Definition 1.5 Given a subjective formula ϕ, define an objective formula ϕ' as the *objective form of* ϕ, if ϕ' is exactly the ϕ changing all $Knows(\psi)$ to ψ, where ψ is an objective formula.

Given a program δ, define a program δ' as the *objective form of* δ, if δ' is exactly the δ changing all the mentioned subjective formulas to their objective forms.

Definition 1.6 Given a BAT $\mathcal{D}$, a ground situation σ and a basic program δ with no sensing actions or procedure calls, define $I_{\mathcal{D}}(\delta, \sigma)$ as the planning instance wrt. $\mathcal{D}$ and δ in σ, if:

1. operations: $\{OP_{\mathcal{D}}(\alpha(\vec{t}))|\alpha(\vec{t})\ is\ mentioned\ in\ \delta\}$;
2. domain predicates: $P_{\mathcal{D}}(\delta)$;
3. objects: $OBJ_{\mathcal{D}}$;
4. initial state: D_{init}, which is the subset of D_σ, containing the literals about predicates and fluent of $P_{\mathcal{D}}(\delta)$ with individuals of $OBJ_{\mathcal{D}}$;
5. goal: $True$.

So we can translate a BAT wrt. a program to a planning instance with above definitions. And plan implementation of planning operator Σ is:

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

As showed above, the predicate `planning(`δ`, `σ`, `δ'`)` implements that, given a BAT $\mathcal{D}$, a basic program δ with no sensing actions or procedure calls, and current situation σ:

1. Firstly calculate planning instance $I_{\mathcal{D}}(\delta,\sigma) = (D,P)$; and use the compiling algorithm in (Baier, Fritz, and McIlraith 2007), to construct a new planning instance $I_{\mathcal{D},\delta_o}(\delta,\sigma)$ from $I_{\mathcal{D}}(\delta,\sigma)$ with δ_o, where δ_o is the objective form of δ. Noted that, we compile δ_o with parameters $x_i (1 \leq i \leq k)$, and so we call $C(\delta_o, 0, [x_i]_{i=1}^k)$ instead of $C(\delta_o, 0, [])$. What's more, the initial state is appended to $\{bound(x_i), map(x_i, r_i)\}_{i=1}^k$ correspondingly, given the value r_i of parameters x_i for every i.
2. And then call *FF* planner with $I_{\mathcal{D},\delta_o}(\delta,\sigma)$ for a plan $\vec{\alpha}$.
3. Finally let δ' be the sequence of $Filter(\vec{\alpha}, D)$, which is the function defined in (Baier, Fritz, and McIlraith 2007) to filter the actions not in domain D.

To accelerate the generation of planning instance $I_{\mathcal{D},\delta_o}(\delta,\sigma) = (D_{\delta_o}, P_{\delta_o})$ at the runtime, before running the program, the interpreter will pretreat to generate the planning domain D_{δ_o} corresponding to the program dominated by Υ, i.e. only planning problem P_{δ_o} would be dynamically generated.

Correctness of the interpreter

Theorem 1.7 (Correctness of basic constructs) *Let $\mathcal{D}$ be a BAT, σ a ground situation, and δ a basic program. Then we have*

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

The theorem 1.7 shows that the implementation of basic constructs is correct. Since the implementation and semantics are in direct correspondence, the proof of this theorem is trivial by structure induction.

Theorem 1.8 (Soundness and weak completeness of search) *Let $\mathcal{D}$ be a BAT, σ a ground situation, and δ a basic program. Then we have*

1. `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.*

The weak completeness of search comes from the running orders of nondeterministic choice. Consider the program $\delta = (\delta_1|\delta_2)$ in situation σ, in the case that δ_2 in search finitely succeeds, so $(\delta,\sigma,\delta') \in \mathcal{C}_{\mathcal{D}}$ is hold for some δ'. As δ_1 would be searched first, however, once δ_1 loops forever, e.g. $\delta_1 = (\alpha^*; false?)$, δ would not terminate. And the theorem of search is also proved by structure induction.

Definition 1.9 [Just-in-time] Given a BAT $\mathcal{D}$, a ground situation σ, and a basic program δ, we say that σ is *just-in-time* for δ wrt. $\mathcal{D}$, if $p(\vec{c}) \in UP(prog(\Sigma[s_0],\sigma))$ iff $\neg p(\vec{c}) \notin UP(prog(\Sigma[s_0],\sigma))$, where $p \in P_{\mathcal{D}}(\delta)$, and $c_i \in OBJ_{\mathcal{D}}$ for $1 \leq i \leq n$ such that $\vec{c} = c_1, \ldots, c_n$.

In order to evaluate quantified formulas with finite objects, we construct predicate $known(O)$, with the meaning that object O is already known. Given a BAT $\mathcal{D}$ and a main program, we so append $\bigwedge_{O \in OBJ_{\mathcal{D}}} known(O)$ to KB.

Definition 1.10 [Objects-all-known program] Given a program δ, we say that δ is an *objects-all-known program*, if mentioned in δ:

1. every quantified formula is of either the form $\forall \vec{x} \bigwedge_{i=1}^n knows(known(x_i)) \supset \phi$ or $\exists \vec{x} \bigwedge_{i=1}^n knows(known(x_i)) \wedge \phi$, marked as *objects-all-known formula*,
2. and every ϕ in $\pi\vec{x}.\phi \wedge \delta$ constructs, is of the form $\bigwedge_{i=1}^n knows(known(x_i)) \wedge \phi'$,

where $\vec{x} = x_1, \ldots, x_n$.

Theorem 1.11 (Correctness of planning) *Let $\mathcal{D}$ be a BAT, σ a ground situation, and δ a basic program with no sensing actions or procedure calls. Suppose that δ is also an objects-all-known program, and σ is just-in-time for δ. Then we have*

1. *if* `btrans(planning(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(planning(δ), P, A)` *wrt. σ succeeds.*

The theorem 1.11 shows the correctness of searching an action sequence by calling planner, under dynamic complete knowledge case. And the proof of this theorem is to firstly prove the equivalence of interpreting programs between the transition function Δ defined in (Baier, Fritz, and McIlraith 2007) and the online semantics $\rightarrow_{\mathcal{D}}$ and $\mathcal{F}_{\mathcal{D}}$; and then construct the corresponding relation between online semantics and offline semantics $\mathcal{C}_{\mathcal{D}}$. And the theorem is proved with the two relations mentioned above.

2 Experiments

Wumpus world

We use our interpreter to construct the control programs for the famous domain Wumpus world. We take the safety as the first place to construct the controller, that's to say, the agent would not die under the controller. On such base, the program would control the agent to get as more golds as she can. And the experiment results are showed as Table 1, with map size 8×8 and 3000 random maps.

Table 1. Wumpus world in our interpreter(8×8, 3000)

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

In Table 1, *Prob* is the probability of pit for every grid; *Gold* is the number of the got golds; *IMP* is the number of maps that is impossible, i.e. the first grid is felt smell or stench so the agent has to climb out the dungeon; *Reward* and *Moves* are separately the averages of the reward and move times for every map; *Time* is the average cost time in the experiment with 3000 maps; and *Calls* is the average times of calling the plan implementation of planning operator, in other words, calling a planer (*FF*).

As showed above, with the higher *Prob*, all the indexes go lower, except the *IMP* climbs up. In other words, the lower

Prob is, the more possibilities the agent explores the dungeon. Noted that, even in the lowest *Prob*, the agent cost only averagely less than 0.4 seconds for a map, which shows the good efficiency of our interpreter.

Open service robot

This domain is the extension of service robot, which is hold for a competition by USTC in China every year, with incomplete knowledge case. The service robot domain describes several small objects, like cup and bottle, at the locations of big objects, like desk and teapot. And the tasks for agent are, to take some small objects for human or on appointed big objects, to putdown or catch some small objects, and to goto some location.

In service robot, the knowledge about locations of small objects is complete, i.e. any small objects' locations can be inferred from KB, while in open service robot, the knowledge is incomplete, with the sense that some small objects' locations cannot be inferred from KB. The knowledge about locations of small objects is of the clause form, instead of literal form, e.g. $location(N_{10}, L_5) \vee location(N_{10}, L_7)$, $location(N_{10}, L_8) \vee location(N_{11}, L_4)$ and $location(N_{12}, L_5)$.

And now we can use our interpreter to construct control programs for the agent to complete the tasks given. And the main control is to use sensing actions to find out the specific locations of target objects, then execute the tasks online. Noted that, we implement an operator that calculates a set of assignments, such that at least one of them satisfies the given unbound literal, in which way we get the possible locations of the small objects. However, this operator is not so important in the interpreter, and we omit the details here.

Now we post an example with the tasks [give(human, book), puton(red, can, desk)], given that the book is marked N_{10} and the red can is marked N_{11}, and the knowledge about locations of them is the conjunction of three clauses $location(N_{10}, L_5) \vee location(N_{10}, L_7)$, $location(N_{10}, L_8) \vee \neg location(N_{11}, L_5)$ and $location(N_{11}, L_4) \vee location(N_{11}, L_5)$. Actually, the real locations of them are $location(N_{10}, L_7)$ and $location(N_{11}, L_4)$. And the executions of the control program are as follows:

```
move(l0,l8)
sense_loc(n10,l8)
<sense_loc(n10,l8)(yes/no)>:no.
move(l8,l7)
sense_loc(n10,l7)
<sense_loc(n10,l7)(yes/no)>:yes.
catch(n10,l7)
move(l7,l1)
putdown(n10,l1)
move(l1,l4)
catch(n11,l4)
move(l4,l9)
putdown(n11,l9)
true.
```

As above, the agent completes the tasks without any sensing actions about the location of the red can N_{11}. That's because by unit propagation, $location(N_{11}, L_5)$ is inferred false according to the second clause with the negative sensing result of $sense_loc(N_{10}, L_8)$, and then $location(N_{11}, L_4)$ is inferred true according to the third clause. From this case, we can test the intelligence of our interpreter.

3 Related work

Based on situation calculus, there has been several high-level programming languages, like *GOLOG* (Levesque et al. 1997), *sGOLOG* (Lakemeyer 1999) and *IndiGolog* (De Giacomo, Levesque, and Sardiña 2001). They offer a useful way to put some simple control information into low-level programming.

GOLOG, as the basic programming language, requires *CWA* (Closed World Assumption) for formula evaluations, with *regression* mechanism. And to make a successful nondeterministic choice, it calculates the complete execution steps of the entire program *offline*.

In *incomplete knowledge case*, *sGOLOG* extends *GOLOG* by incorporating sensing actions, supporting complete inference based on *possible-world* semantics and *domain closure*. And it yields a tree of actions branching with the results of sensing actions, instead of a linear sequence of actions.

To adapt *open world* case, *IndiGolog* extends *GOLOG* by serving *online* execution of program with $Trans/Final$ semantics. As the ground literal form of KB, JIT (Just-In-Time) is asked for preserving the complete inference. Moreover, it proposes *search operator* to guarantee the successful executions of sub-program as offline planning.

Considering incomplete knowledge case, Jens and Gerhard (Claßen and Lakemeyer 2009) extend the expression and reasoning of knowledge in *GOLOG* with disjunctive, by the logic of limited belief $\mathcal{SL}$ (Liu, Lakemeyer, and Levesque 2004). Within a fixed *belief level*, they support a program semantics following online semantics, and a search operator as offline planning. Further, they propose another search operator considering all belief levels iteratively.

4 Programs in Wumpus world

The primitive actions:
$move/4$
$shoot\_up/0$
$shoot\_down/0$
$shoot\_left/0$
$shoot\_right/0$
$grab/0$
$climb/0$

The sensing actions:
$sense\_stench/0$
$sense\_breeze/0$
$sense\_gold/0$
$sense\_scream/0$

The control programs:
proc $main$

$senseAll$;
if $K(gold(n_1, n_1))$
then $grab; climb$
else $explore; \Upsilon(moveLoc(n_1, n_1)); climb$
endIf;
endProc

proc $senseAll$
$sense\_stench$;
$sense\_breeze$;
$sense\_gold$
endProc

proc $explore$
while $K(\neg getsGold) \wedge$
$\exists[x, y].(K((\neg wumpus(x, y) \vee \neg wumpusAlive) \wedge$
$\neg pit(x, y)) \wedge \neg K(explored(x, y)))$
do $(\pi[x, y].(K((\neg wumpus(x, y) \vee \neg wumpusAlive) \wedge$
$\neg pit(x, y)) \wedge \neg K(explored(x, y))) \wedge$
$(\Upsilon moveLoc(x, y));$
$senseAll$;
if $K(\exists[x_0, y_0].agent(x_0, y_0) \wedge gold(x_0, y_0))$
then $grab$
else
if $\exists[x_1, y_1].K(wumpusAlive \wedge$
$wumpus(x_1, y_1))$
then $shootWumpus$
else nil
endIf
endIf
endWhile
endProc

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

proc $shootWumpus$
$(\pi[x_1, y_1, x_2, y_2].K(wumpus(x_1, y_1) \wedge$
$explored(x_2, y_2) \wedge$
$(x_1 = x_2 \wedge succ(y_1, y_2)) \vee$
$(x_1 = x_2 \wedge succ(y_2, y_1)) \vee$
$(y_1 = y_2 \wedge succ(x_1, x_2)) \vee$
$(y_1 = y_2 \wedge succ(x_2, x_1))) \wedge$
$(\Upsilon moveLoc(x_2, y_2));$
$(succ(y_2, y_1)?; shoot\_up | succ(y_1, y_2)?; shoot\_dowm |$
$succ(x_1, x_2)?; shoot\_left | succ(x_2, x_1)?; shoot\_right));$
$sense\_scream$
endProc

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.; 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.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. Golog: A logic programming language for dynamic domains. *J. of Logic Programming* 31:59–84.

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.

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

5 Proof

In the following proofs, we do not discuss the conditionals, while-loops and procedure cases that are not necessary. And the programs in proof are ignored the difference in syntax from implementation.

1. Theorem 1.7

Proof: Use structure induction to prove the correctness.

(1) Base case: For $\mathcal{F}_{\mathcal{D}}$, consider empty program nil, test $\phi?$ and iteration δ^*. They are trivial according to the definitions except $\phi?$.

For the case $\phi?$, by the [query theorem], `query(`ϕ`, `σ`)` succeeds iff $wclosure(B_0 prog(\Sigma[S_0], \sigma)) \models B_0\phi[\sigma]$. And the proof is finished by definitions.

For $\rightarrow_{\mathcal{D}}$, consider primitive action α and sensing action β. The proofs of them are similar, so take α case to prove.

`query(knows(`Π_α`), `σ`)` succeeds, iff, by the [query theorem], $wclosure(B_0 prog(\Sigma[S_0], \sigma)) \models B_0\Pi_\alpha[\sigma]$, where Π_α is the precondition of α. And then the proof is over by definitions.

(2) Induction step: Assume that the two parts of the theorem are hold.

For $\mathcal{F}_{\mathcal{D}}$, consider $\delta_1; \delta_2$, $\delta_1|\delta_2$ and $\pi\vec{x}.\phi \wedge \delta$; and for $\rightarrow_{\mathcal{D}}$, consider $\delta_1; \delta_2$, $\delta_1|\delta_2$, $\pi\vec{x}.\phi \wedge \delta$ and δ^*. The proof for $\mathcal{F}_{\mathcal{D}}$ is simple by definition and [query theorem]. And for $\rightarrow_{\mathcal{D}}$, prove the $\delta_1; \delta_2$ case.

The one to prove is that: Given a ground situation σ, `btrans(`$\delta_1; \delta_2$`, `P`, `A`)` wrt. σ succeeds with $P = \delta'$ and $A = \alpha$, iff $(\delta_1; \delta_2, \sigma) \rightarrow_{\mathcal{D}} (\delta', do(\alpha, \sigma))$.

By definitions, inductively prove that:

(A) T_{A1}: [`btrans(`$\delta_1; \delta_2$`, `P_1`, `A_1`)` wrt. σ] succeeds with $P_1 = \gamma; \delta_2$ and $A_1 = \alpha$ s.t. `btrans(`δ_1`, `γ`, `α`)` wrt. σ succeeds, iff T_{A2}: [$(\delta_1; \delta_2, \sigma) \rightarrow_{\mathcal{D}} (\gamma; \delta_2, do(\alpha, \sigma))$], s.t. $(\delta_1, \sigma) \rightarrow_{\mathcal{D}} (\gamma, do(\alpha, \sigma))$.

(B) T_{B1}: [`btrans(`$\delta_1; \delta_2$`, `P_1`, `A_1`)` wrt. σ] succeeds with $P_1 = \gamma$ and $A_1 = \alpha$ s.t. `btrans(`δ_2`, `γ`, `α`)` wrt. σ succeeds, iff T_{B2}: [$(\delta_1; \delta_2, \sigma) \rightarrow_{\mathcal{D}} (\gamma, do(\alpha, \sigma))$] s.t. $(\delta_2, \sigma) \rightarrow_{\mathcal{D}} (\gamma, do(\alpha, \sigma))$.

For case (A), T_{A1} succeeds iff T_{A3}: [`btrans(`δ_1`, `γ`, `α`)` wrt. σ] succeeds; and T_{A2} is hold iff T_{A4}: [$(\delta_1, \sigma) \rightarrow_{\mathcal{D}} (\gamma, do(\alpha, \sigma))$] is hold. By the induction hypothesis, T_{A3} succeeds iff T_{A4} is hold.

For case (B), T_{B1} succeeds iff both T_{B3}: [`bfinal(`δ_1`)` wrt. σ] and T_{B4}: [`btrans(`δ_2`, `γ`, `α`)` wrt. σ] succeed; and T_{B2} is hold iff both T_{B5}: [$(\delta_1, \sigma) \in \mathcal{F}_{\mathcal{D}}$] and T_{B6}: [$(\delta_2, \sigma) \rightarrow_{\mathcal{D}} (\gamma, do(\alpha, \sigma))$] are hold. It's easy to get that, T_{B3} succeeds iff T_{B5} is hold, and T_{B4} succeeds iff T_{B6} is hold, by the induction hypothesis.

Hence, the proof of $\delta_1; \delta_2$ case for $\rightarrow_{\mathcal{D}}$ is finished. It's by the similar way to prove other cases.

Therefore, the implementation of basic constructs is correct. ■

2. Theorem 1.8

Proof: Use structure induction to prove the soundness and weak completeness.

(Soundness)

(1) Base case: Consider the cases of empty program nil, primitive action α, sensing action β, test $\phi?$ and iteration δ^*. The nil and δ^* cases are trivial, and the proofs of the other cases are similar each other. So take the $\phi?$ case to prove.

That's to prove that: `btrans(search(`$\phi?$`), P, A)` wrt. σ succeeds with $P = \delta'$, then $(\phi?, \sigma, \delta') \in \mathcal{C}_\mathcal{D}$

The success of `btrans(search(`$\phi?$`), P, A)` wrt. σ implies that `bdo(`$\phi?$`, σ, P)` with $P = nil$, i.e. `query(`ϕ`, σ)` succeeds. By [query theorem], $wclosure(B_0 prog(\Sigma[S_0], \sigma)) \models B_0\phi[\sigma]$ is hold, hence $(\phi?, \sigma, nil) \in \mathcal{C}_\mathcal{D}$.

(2) Induction step: Consider $\delta_1; \delta_2$, $\delta_1|\delta_2$, $\pi\vec{x}.\phi\wedge\delta$ and δ^*. Since the proofs of $\delta_1|\delta_2$ and $\pi\vec{x}.\phi\wedge\delta$ are trivial by the assumption $A_{\mathcal{C}_\mathcal{D}}$, prove $\delta_1; \delta_2$ case as δ^* case is proved similarly.

Assume that: $A_{\mathcal{C}_\mathcal{D}}$: [Given a situation s, if `bdo(`δ_1`, s, P1)` succeeds with $P_1 = \rho^*$, then $(\delta_1, s, \rho_1) \in \mathcal{C}_\mathcal{D}$; if `bdo(`$\delta_2$`, s, P2)` succeeds with $P_2 = \rho_2$, then $(\delta_2, s, \rho_2) \in \mathcal{C}_\mathcal{D}$.]

By assumption $A_{\mathcal{C}_\mathcal{D}}$, the success of first sub-goal `bdo(`δ_1`, s, ρ1)` implies $(\delta_1, s, \rho_1) \in \mathcal{C}_\mathcal{D}$. And we now prove that: if `ext(`ρ`, δ2, σ, P)` succeeds with $P = \rho'$, then $(\rho, \delta_2, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$.

We use structure induction here.

(2-1) Base case: Consider the empty program nil. From definitions this case is hold simply, by the assumption $A_{\mathcal{C}_\mathcal{D}}$, as that the success of `bdo(`δ_2`, σ, ρ')` implies $(\delta_2, \sigma, \rho') \in \mathcal{C}_\mathcal{D}$.

(2-2) Induction step:

Consider the cases of $\alpha; \rho$, $\beta; \rho$ and **if**ϕ**then**ρ_1**else**ρ_2**endIf**. Here take the $\beta; \rho$ case to prove.

Assume that: $A_{\mathcal{E}_\mathcal{D}}$: [if `ext(`$\rho$`, δ2, σ, P)` succeeds with $P = \rho'$, then $(\rho, \delta_2, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$.]

(pause here)

It'a going to prove that: if `ext(`$\beta;\rho$`, δ2, σ, P)` succeeds with $P = \rho'$, then $(\beta; \rho, \delta_2, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$.

Since `sens_prog(`β`, 1, σ, σT)` succeeds by definition of `ext/4`, it means that $\sigma_T = do(\beta_T, \sigma)$, i.e. `ext(`$\rho$`, δ2, `$do(\beta_T, \sigma)$`, ρT)` succeeds. For the record, `del_sit(`σ_T`)` is to delete the KB of σ_T, whenever the search in σ_T succeeds or backtracks. Hence by the assumption $A_{\mathcal{E}_\mathcal{D}}$, $(\rho, \delta_2, do(\beta_T, \sigma), \rho_T) \in \mathcal{E}_\mathcal{D}$ is hold, and $(\rho, \delta_2, do(\beta_F, \sigma), \rho_F) \in \mathcal{E}_\mathcal{D}$ is hold respectively. As `sf(`β`, ψ)` succeeds such that $SF(\beta, \sigma) \equiv \psi[\sigma]$, get that $\rho' = \beta;$**if**ϕ**then**ρ_T**else**ρ_F**endIf**. Therefore $(\rho, \delta_2, \sigma, \rho') \in \mathcal{E}_\mathcal{D}$ is hold.

The other cases are proved similarly. △

Return to proof of the $\delta_1; \delta_2$ case, $(\rho_1, \delta_2, \sigma, \rho) \in \mathcal{E}_\mathcal{D}$ is hold with the success of `ext(`ρ_1`, δ2, σ, ρ)` by the conclusion above. Therefore, $(\delta_1; \delta_2, \sigma, \rho) \in \mathcal{C}_\mathcal{D}$ is hold. □

(Weak completeness)

The proof is almost the anti-direction of the one for soundness. However, notice that the cases of loop forever leads that the search does not terminate.

Consider the program $(\alpha_1^*; false?|\alpha_2)$ in search, where both α_1 and α_2 are possible in any situation, and then easily know that $(\alpha_1^*; false?|\alpha_2, \sigma, \alpha_2) \in \mathcal{C}_\mathcal{D}$ given a ground situation σ. With the codes of `bdo/3`in ***PROLOG***, `bdo(`$\alpha_1^*; false?$`, σ, ρ)` is to test first. And this task loops forever, since the the test of iteration α_1^* lasts forever until $false?$ turns true or α is not possible in some situation, impossibly. Hence the search would neither finitely succeed nor fail, i.e. not terminate.

Therefore, get the weak completeness. ■

3. Theorem 1.11

Proof: Using the correctness theorem in (Baier, Fritz, and McIlraith 2007), it's to prove that: $\vec{\alpha}$ is accepted by ε-NFA $A_{\delta_o, I_\mathcal{D}}$, iff $(\delta, \sigma, \vec{\alpha}) \in \mathcal{C}_\mathcal{D}$, considered $\vec{\alpha}$ as the sequence constructs for simplicity. Here point out that $\vec{\alpha}$ is also the plan in $I_\mathcal{D}(\delta, \sigma)$, as of which the goal is always true.

To prove the transformation above, firstly need several lemmas, with the same premise as theorem 1.11, and let $Init$ the initial state of $I_{\mathcal{D},\delta_o}(\delta, \sigma)$, s a state, ϕ a ground or objects-all-known subjective formula, and ϕ_o is the objective form of ϕ. Then:

Lemma 5.1 *$Init \models \phi_o$, iff $wclosure(B_0 prog(\Sigma[S_0], \sigma)) \models \phi$.*

Lemma 5.2 *$Succ(Init, \vec{\alpha}, s)$ and $s \models \phi_o$, iff $\vec{\alpha}$ is legal in σ and $wclosure(B_0 prog(\Sigma[S_0], do(\vec{\alpha}, \sigma))) \models \phi$.*

This two lemmas above guarantee the equivalent abilities of evaluation between planning instance $I_{\mathcal{D},\delta_o}(\delta, \sigma)$ and BAT $\mathcal{D}$.

Let γ be a basic and objects-all-known program with no sensing actions and procedure calls, in which the predicates mentioned are in $P_\mathcal{D}(\delta)$, and the objects mentioned are in $OBJ_\mathcal{D}$, and γ_o the objective form of γ. Then:

Lemma 5.3 *$[nil, s] \in \Delta([\gamma_o, s], \varepsilon^*)$ with $Succ(Init, \vec{\alpha}, s)$, iff $(\gamma, \sigma') \in \mathcal{F}_\mathcal{D}$ with $\sigma' = do(\vec{\alpha}, \sigma)$ and $\vec{\alpha}$ is legal in σ.*

Lemma 5.4 *Let β be an action s.t. $\beta \neq \varepsilon$. Then*

1. *for every program γ_o' s.t. $[\gamma_o', s'] \in \Delta([\gamma_o, s], \beta)$, with $Succ(Init, \vec{\alpha}, s)$ and $Succ(s, \beta, s')$, there exists a program γ' s.t. $(\gamma, \sigma') \rightarrow_\mathcal{D} (\gamma', do(\beta, \sigma'))$, with $\sigma' = do(\vec{\alpha}, \sigma)$ and $\vec{\alpha}$ is legal in σ, and the objective form of γ' is γ_o';*
2. *for every program γ' s.t. $(\gamma, \sigma') \rightarrow_\mathcal{D} (\gamma', do(\beta, \sigma'))$, with $\sigma' = do(\vec{\alpha}, \sigma)$ and $\vec{\alpha}$ is legal in σ, we have that $[\gamma_o', s'] \in \Delta([\gamma_o, s], \beta)$, with $Succ(Init, \vec{\alpha}, s)$ and $Succ(s, \beta, s')$, where γ_o' is the objective form of γ'.*

This two lemmas above shows the equivalence of interpreting programs between the transition function Δ and the semantics $\rightarrow_\mathcal{D}$ and $\mathcal{F}_\mathcal{D}$.

Lemma 5.5 *Let $\vec{\alpha}$ be an action sequence. Then there exists a program δ', s.t. $(\delta, \sigma) \rightarrow_\mathcal{D}^* (\delta', do(\vec{\alpha}, \sigma))$ and $(\delta', do(\vec{\alpha}, \sigma)) \in \mathcal{F}_\mathcal{D}$, iff $(\delta, \sigma, \vec{\alpha}) \in \mathcal{C}_\mathcal{D}$.*

The lemma above shows the correctness of offline semantics $\mathcal{C}_\mathcal{D}$ for an action sequence.

Lemma 5.1 and lemma 5.2 are proved by utilizing the procedure of constructing $I_\mathcal{D}(\delta, \sigma)$. And as the first two lemmas are hold, lemma 5.3 and lemma 5.4 are proved by structure induction. Lastly, lemma 5.5 is also proved by structure induction.

For soundness, by lemma 5.3 and lemma 5.4, we firstly get that the action sequence $\vec{\alpha}$, accepted by $A_{\delta_o, I_\mathcal{D}}$, can be transited iteratively by $\rightarrow_\mathcal{D}$ from configuration (δ, σ) to a configuration $(\delta', do(\vec{\alpha}, \sigma)) \in \mathcal{F}_\mathcal{D}$, for some δ'. Then by lemma 5.5 the soundness is proved. And the completeness is proved similarly on anti-direction.