Applications of the Situation Calculus To Formalizing Control and Strategic Information: The Prolog Cut Operator

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

We argue that the situation calculus is a natural formalism for representing and reasoning about control and strategic information. As a case study, in this paper we provide a situation calculus semantics for the Prolog cut operator, the central search control operator in Prolog. We show that our semantics is well-behaved when the programs are properly stratified. We also show that according to this semantics, the conventional implementation of the negation-as-failure operator using cut is provably correct with respect to the stable model semantics.

1 Introduction

The situation calculus (McCarthy and Hayes [7]) is a formalism for representing and reasoning about actions in dynamic domains. It is a many-sorted predicate calculus with some reserved predicate and function symbols. For example, to say that block A is initially clear, we write:

$$H(clear(A), S_0),$$

where H is a reserved binary predicate and stands for "holds", and S_0 is a reserved constant symbol denoting the initial situation. As an another example, to say that the action $stack(x, y)$ causes $on(x, y)$ to be true, we write:[1]

$$Poss(stack(x,y), s) \supset H(on(x,y), do(stack(x,y), s)),$$

where the reserved function $do(a, s)$ denotes the resulting situation of doing the action a in the situation s, and $Poss(a, s)$ is the precondition for a to be executable in s. This is an example of how the effects of an action can be represented in the situation calculus. Generally, in the situation calculus:

[1]In this paper, free variables in a displayed formula are assumed to be universally quantified.

- situations are first-order objects that can be quantified over;
- a situation carries information about its history, i.e the sequence of actions that have been performed so far. For example, the history of the situation $do(stack(A, B), do(stack(B, C), S_0))$ is $[stack(B,C), stack(A,B)]$, i.e. the sequence of actions that have been performed in the initial situation to reach this situation. As we shall see later, our foundational axioms will enforce a one-to-one correspondence between situations and sequences of actions.

We believe that these two features of the situation calculus make it a natural formalism for representing and reasoning about control knowledge. For example, in AI planning, a plan is a sequence of actions, thus isomorphic to situations. So control knowledge in planning, which often are constraints on desirable plans, becomes constraints on situations (Lin [4]). Similarly, when we talk about control information in logic programming, we are referring to constraints on derivations, i.e. sequences of actions according to (Lin and Reiter [6]).

Although our long term goal is to develop a general framework for representing and reasoning about control knowledge in problem solving using the situation calculus, our focus in this paper is the Prolog cut operator, the central search control operator in Prolog. We provide a situation calculus semantics for logic programs with cut, and show that our semantics is well-behaved when the programs are properly stratified. We also show that according to this semantics, the conventional implementation of the negation-as-failure operator using cut is provably correct with respect to the stable model semantics of Gelfond and Lifschitz [2]. To the best of our knowledge, this is the first time a connection has been shown between a declarative semantics of negation and that of cut.

This paper is organized as follows. Section 2 briefly reviews the basic concepts in the situation calculus and logic programming. Section 3 reviews the situation cal-

cuius semantics of (Lin and Reiter [6]) for cut-free logic programs. For the purpose of this paper, the key property of this semantics is that derivations in logic programming are identified with situations. Section 4 extends this semantics to logic programs with cut. This is done by an axiom on *accessible* situations, that is, those situations whose corresponding derivations are not "cut off" by cut. Section 5 shows some properties of our semantics, and finally section 6 concludes this paper.

2 Logical Preliminaries

2.1 The Situation Calculus

The language of the situation calculus is a many-sorted second-order one with equality. We assume the following sorts: *situation* for situations, *action* for actions, *fluent* for propositional fluents such as *clear* whose truth values depend on situations, and *object* for everything else. As we mentioned above, we assume that S_0 is a reserved constant denoting the initial situation, *H* a reserved predicate for expressing properties about fluents in a situation, *do* a reserved binary function denoting the result of performing an action, and *Poss* a reserved binary predicate for action preconditions. In addition, we assume the following two partial orders on situations:

- $<$: following convention, we write $<$ in infix form. By $s < s'$ we mean that s' can be obtained from s by a sequence of executable actions. As usual, $s \leq s'$ will be a shorthand for $s < s' \vee s = s'$.
- $\sqsubset$: we also write $\sqsubset$ in infix form. By $s \sqsubset s'$ we mean that s can be obtained from s' by deleting some of its actions. Similarly, $s \sqsubseteq s'$ stands for $s \sqsubset s' \vee s = s'$.

We shall consider only the discrete situation calculus with the following foundational axioms:[2]

$$S_0 \neq do(a, s), \tag{1}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2), \tag{2}$$

$$(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \supset P(do(a, s))) \supset (\forall s)P(s)], \tag{3}$$

$$\neg s < S_0, \tag{4}$$

$$s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s'), \tag{5}$$

$$s \sqsubset s' \equiv s \neq s' \wedge (\exists f)\{(\forall s_1, s_2)(s_1 \leq s_2 \supset f(s_1) \leq f(s_2)) \wedge (\forall a, s_1)(do(a, s_1) \leq s \supset do(a, f(s_1)) \leq s')\}. \tag{6}$$

The first two axioms are unique names assumptions for situations. The third axiom is second order induction.

[2]Except for the one about $\sqsubset$, these axioms are taken from (Lin and Reiter [5]), which also proves some useful properties about them.

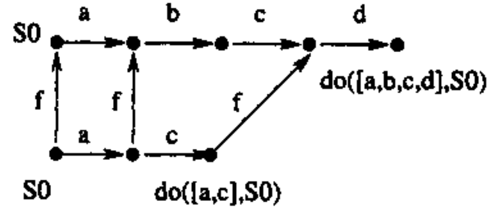

Figure 1: A function f as required by the axiom (6)

It amounts to the domain closure axiom that every situation has to be obtained from the initial one by repeatly applying the function *do*. In other words, it says that every situation is either the initial situation S_0 or the result of performing a sequence of actions in the initial situation, exactly the isomorphism between situations and sequences of actions that we mentioned earlier. As can be expected, induction will play an important role in this paper.

Axioms (4) and (5) define $<$ inductively. Generally, $s \leq s'$ if s' can be obtained from s by performing some executable actions.

In this paper we shall assume that actions are always possible: $(\forall a, s)Poss(a, s)$. Under this assumption, the partial order $<$ reduces to the "prefix" relation:[3] Given a situation $S = do([\alpha_1, ..., \alpha_n], S_0)$, $S' \leq S$ iff there is a $0 \leq k \leq n$ such that $S' = do([\alpha_1, ..., \alpha_k], S_0)$. In particular, we have $(\forall s)S_0 \leq s$.

Axiom (6) defines $\sqsubset$. Informally, $s \sqsubseteq s'$ iff s can be obtained from s' by deleting some of its actions. More precisely, suppose $S' = do([\alpha_1, ..., \alpha_n], S_0)$. Then $S \sqsubseteq S'$ iff there are integers $1 \leq i_1 \leq \cdots \leq i_k \leq n$, $0 \leq k \leq n$, such that $S = do([\alpha_{i_1}, ..., \alpha_{i_k}], S_0)$. Figure 1 illustrates a function f as required by axiom (6) for proving

$$do([a, c], S_0) \sqsubset do([a, b, c, d], S_0).$$

Notice that $\leq$ is a special case of $\sqsubseteq$: if $s \leq s'$ then $s \sqsubseteq s'$. As we shall see, the partial order $\sqsubset$ will play a crucial role in this paper.

In the following, we shall denote by Σ the set of the above axioms.

2.2 Logic Programs

We consider definite logic programs with cut.

An *atom* p is a fluent term $F(t_1, ..., t_n)$, where F is a fluent of arity *object*^{n}, and $t_1, \ldots, t_n$ are terms of sort *object*. A *goal* G is an expression of the form

$$l_1 \ \& \ \cdots \ \& \ l_n$$

[3]Given a sequence of actions $[a_1, ..., a_n]$, we use $do([a_1, ..., a_n], s)$ to denote the resulting situation of performing the sequence of actions in s. Inductively, $do([], s) = s$ and $do([a|L], s) = do(L, do(a, s))$.

where $n \geq 0$, and for each $1 \leq i \leq n$, l_i is either an atom, an equality atom of the form $t = t'$, or $!$.

Without loss of generality, we assume that a *clause (rule)* is an expression of the form

$$F(x_1, ..., x_n) \text{ :- } G$$

where F is a fluent of the arity $object^n$, $x_1, ..., x_m$ are distinct variables of sort *object*, and G is a goal. Notice that the more common form of a clause

$$F(t_1, ..., t_n) \text{ :- } G.$$

can be taken to be a *shorthand* for the following clause:

$$F(x_1, ..., x_n) \text{ :- } x_1 = t_1 \ \& \ \cdots \ \& \ x_n = t_n \ \& \ G.$$

where $x_1, ..., x_n$ are fresh variables not in G and $t_1, ..., t_n$.

Finally, a (definite) *program* is a finite set of clauses. The *definition* of a fluent symbol F in a program P is the set of clauses in P that have F in their heads.

Since a goal is not a situation calculus formulas, we need a way to refer to its truth values. Given a goal $G = l_1 \ \& \ \cdots \ \& \ l_n$, and a situation term 5, we define *H(G,S),* the truth value of G in the situation 5, to be the situation calculus formula

$$H(l_1, S) \wedge \cdots \wedge H(l_n, S),$$

where for each $1 \leq i \leq n$:

1. If l_i is $t = t'$, then $H(l_i, S)$ is l_i.
2. If l_i is $!$, then $H(l_i, S)$ is the tautology *true*.

For example, $H(x = a \ \& \ parent(x, y) \ \& \ !, S_0)$ is

$$x = a \wedge H(parent(x, y), S_0) \wedge true.$$

3 A Logical Semantics

The cut operator in Prolog plays two roles. As a goal, it succeeds immediately. As a search control operator, it prevents a Prolog interpreter from backtracking past it. Consequently, our semantics for programs with cut will come in two stages. First, we consider the "pure logical" semantics of the programs when cut is taken to be a goal that succeeds immediately. For this purpose, we shall use the situation calculus semantics for logic programs without cut proposed by Lin and Reiter ([6]). For our purpose here, the key of this semantics is that program clauses are identified with the effects of actions in the situation calculus, so a branch in a search tree becomes a sequence of actions, thus isomorphically, a situation. This is important because the effect of a cut on the search tree can then be modeled by restrictions on situations. So our second step in formalizing the cut operator is to define a relation call *Acc* on situations so that *Acc(s)* holds with respect to a logic program if the sequence of actions in *s* corresponds to a successful derivation according to the program.

The rest of this section is basically a review of [6] with a minor notational difference: while we reify fluents and use the special predicate H, Lin and Reiter [6] treat fluents as predicate symbols. For example, *H(broken,s)* would be written as *broken(s)* in [6].

According to [6], clauses are treated as rules, so that the application of such a rule in the process of answering a query is like performing an action. Formally, given a clause of the form

$$F(x_1, ..., x_n) \text{ :- } G$$

Lin and Reiter [6] introduce a unique action A of the arity $object^n \rightarrow action$ to name this clause. The only effect of this action is the following:

$$(\exists \vec{\xi}) H(G, s) \supset H(F(\vec{x}), do(A(\vec{x}), s)),$$

where $\vec{x}$ is $(x_1, ..., x_n)$, and $\vec{\xi}$ is the tuple of variables that are in G but not in $\vec{x}$. For example, suppose $gp(x, y)$ is the action that names the following clause:

```
gparent(x,y) :- parent(x,z) & parent(z,y)
```

then we have the following effect axiom:

$$(\exists z)[H(parent(x, z), s) \wedge H(parent(z, y), s)] \supset H(gparent(x, y), do(gp(x, y), s)).$$

Now suppose that P is a program and F a fluent. Suppose the definition of F in P is

$$\begin{array}{ll} A_1(\vec{x}): & F(\vec{x}) \text{ :- } G_1 \\ \vdots & \vdots \\ A_k(\vec{x}): & F(\vec{x}) \text{ :- } G_k \end{array}$$

where $A_1, ..., A_k$ are action names of the corresponding clauses. Then we have the following corresponding effect axioms for the fluent F:

$$\begin{array}{l} (\exists \vec{y_1}) H(G_1, s) \supset H(F(\vec{x}), do(A_1(\vec{x}), s)), \\ \vdots \\ (\exists \vec{y_k}) H(G_k, s) \supset H(F(\vec{x}), do(A_k(\vec{x}), s)), \end{array}$$

where $\vec{y_i}$, $1 \leq i \leq k$, is the tuple of variables in G_i which are not in $\vec{x}$. We then generate the following *successor state axiom* (Reiter [8]) for F:

$$H(F(\vec{x}), do(a, s)) \equiv \{a = A_1(\vec{x}) \wedge (\exists \vec{y_1}) H(G_1, s) \vee \cdots \vee (a = A_k(\vec{x}) \wedge (\exists \vec{y_k}) H(G_k, s)) \vee H(F(\vec{x}), s)\}. \quad (7)$$

Intuitively, this axiom says that F is true in a successor situation iff either it is true initially or the action is one that corresponds to a clause in the definition of F and the body of the clause is satisfied initially. In particular, if the definition of F in the program P is empty, then (7) becomes $H(F(\vec{x}), do(a, s)) \equiv H(F(\vec{x}), s)$.

In the following, we call (7) the *successor state axiom for F wrt to P.*

Given a logic program P, the set of successor state axioms wrt P, together with some domain independent axioms, is then the "pure logical meaning" of P:

Definition 1 *The* basic action theory $\mathcal{D}$ for P *is*

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

where

- Σ *is the set of foundational axioms given in Section 2.1.*
- $\mathcal{D}_{ss}$ *is the set of successor state axioms for the fluents according to* P.
- $\mathcal{D}_{una}$ *is the following set of unique names axioms:*

$$f(\vec{x}) \neq g(\vec{y}) \qquad (8)$$

for every pair f, g *of distinct function symbols, and*

$$f(\vec{x}) = f(\vec{y}) \supset \vec{x} = \vec{y} \qquad (9)$$

for every function symbol f. *Notice that* $\mathcal{D}_{una}$ *includes unique names axioms for actions.*

- $\mathcal{D}_{S_0}$ *is:*

$$\{H(F(\vec{x}), S_0) \equiv false \mid F \text{ is a fluent}\}.$$

Definition 2 *Let* P *be a program, and* $\mathcal{D}$ *its corresponding basic action theory. A situation term* S *is called a* plan *for a goal* G *iff* $\mathcal{D} \models H(G, S)$.

Therefore query answering in logic programs literally becomes planning in the style of (Green [3]) in the situation calculus. This semantics has some nice properties. It is closely related to a recent proposal by Wallace [10], and generalizes the Clark completion semantics. Furthermore, in the propositional case, it is equivalent to Gelfond and Lifschitz's stable model semantics. For details, see (Lin and Reiter [6]).

Example 1 Consider the following program for $max(x, y, z)$, z is the maximum of $\{x, y\}$:[4]

```
A1(x,y,z):   max(x,y,z) :- le(x,y) & ! & z=y
A2(x,y,z):   max(x,y,z) :- z=x
B1(x,y):     le(x,y) :- x=1 & y=2
B2(x,y):     le(x,y) :- x=1 & y=1
B3(x,y):     le(x,y) :- x=2 & y=2
```

Here *le(x,y)* means that *x* is less than or equal to y, and 1 and 2 are constants. We have the following successor state axioms for *max* and *le*:

$$\begin{aligned} &H(max(x,y,z), do(a,s)) \equiv \\ &\quad a = A_1(x,y,z) \wedge H(le(x,y),s) \wedge z = y \vee \\ &\quad (a = A_2(x,y,z) \wedge z = x) \vee H(max(x,y,z),s), \\ &H(le(x,y), do(a,s)) \equiv \{a = B_1(x,y) \wedge x = 1 \wedge y = 2 \vee \\ &\quad a = B_2(x,y) \wedge x = 1 \wedge y = 1 \vee \\ &\quad (a = B_3(x,y) \wedge x = 2 \wedge y = 2) \vee H(le(x,y),s)\}. \end{aligned}$$

From these successor state axioms, it is easy to see that performing first B_1 (1,2), then A_1 (1,2,2) in S_0 will result in a situation satisfying max(l,2,2). Thus we have the following desirable conclusion:

$$\mathcal{D} \models (\exists s) H(max(1,2,2), s).$$

On the other hand, it is also clear that the action $A_2(x,y,x)$ will make $max(x,y,x)$ true, so we also have:

$$\mathcal{D} \models (\forall x, y)(\exists s) H(max(x,y,x), s),$$

which is not so desirable. Of course, the reason is that the basic action theory $\mathcal{D}$ does not take into account the effects of ! on the search space. According to Prolog's search strategy, which attempts rules in the order as they are given, the second rule (A_2) for *max* will not be attempted if the subgoal $le(x,y)$ before ! in the first rule for *max* succeeds. Therefore the derivation corresponding to $do(A_2(x,y,x), S_0)$ will not be considered if there is a derivation for $le(x,y)$, which is the case here if $x = 1$ and $y = 2$.

Let us call a situation *accessible* if the derivation corresponding to this situation[5] is legal, i.e. not ruled out by the cut. Our goal in defining a semantics for the cut operator is then to characterize the set of accessible situations. This is what we are going to do in the next section.

4 A Semantics for Cut

A clause containing cut:

$$cut(\vec{x}): \quad F(\vec{x}) \text{ :- } G_1 \ \& \ ! \ \& \ G_2$$

means that if G_1 succeeds, then any derivation of $F(\vec{x})$ must use either

1. A rule before this one; or
2. This rule with the first derivation of G_1.

We now proceed to formalize this informal reading.

First, notice that we need two ordering relations: one on rules for deciding the precedence of rules, and the other on situations for defining "the first derivation".

[4] This is a typical example of improper uses of cut. We'll return to this point later.

[5] In the following, the terms situations and derivations will be used interchangeably.

In the following, we shall assume that we are given an ordering $\prec$ on actions (rules), and will define an ordering on situations using $\prec$. Intuitively, if $\alpha \prec \beta$, then during the process of search, the action α will be considered before the action β. For instance, according to Prolog's ordering rule, for our *max* example, $(\forall \vec{x}, \vec{y}) A_1(\vec{x}) \prec A_2(\vec{y})$.

Given a partial order on actions, there are many ways situations, i.e. sequences of actions, can be ordered, depending on particular problem solving strategies. In Prolog, a query is answered using a goal-directed search strategy, so if a plan $do([\alpha_1, ..., \alpha_n], S_0)$ is returned, then it must be the case that α_n is first decided, then α_{n-1}, ..., and finally α_1. If we read $s \prec s'$ as that the sequence of actions in s is considered before that in s', then we have the following definition:

Definition 3 *Given a partial order $\prec$ on actions, the derived partial order with the same name $\prec$ on situations is defined by the following axiom:*

$$s \prec s' \equiv (\exists a, b, s_1, s_2)\{s = do(a, s_1) \wedge s' = do(b, s_2) \wedge [a \prec b \vee (a = b \wedge s_1 \prec s_2)]\}. \quad (10)$$

With this partial order on situations, we can then say, roughly, that a situation is a "first derivation" of a goal G if it is a derivation of G that is *minimal* according to $\prec$. However, to make this precise we first need a space of derivations to define the notion of minimality. We do not want it to be the set of all plans for G because according to our definition, if $\mathcal{D}$ is a basic action theory for a logic program, then

$$\mathcal{D} \models (\forall a, s).H(G, s) \supset H(G, do(a, s)).$$

So if s is a plan for G, then for any action a, $do(a, s)$ is also a plan for G. This means that a plan for G can always be made "smaller" according to $\prec$ by appending to it some irrelevant actions. To avoid this problem, we define the notion of *minimal plans*:

Definition 4 *For any situation term S, any goal G, we denote by* $\mathtt{minimal}(G, S)$ *the following formula:*

$$H(G, S) \wedge \neg(\exists s)[s \sqsubset S \wedge H(G, s)].$$

Intuitively, $\mathtt{minimal}(G, S)$ holds if no actions in it can be deleted for it to be a plan of G. For our *max* program in section 3, it is easy to see that

$$\mathcal{D} \models \mathtt{minimal}(le(1, 2), do(B_1(1, 2), S_0)).$$

But

$$\mathcal{D} \not\models \mathtt{minimal}(le(1, 2), do([B_1(1, 2), A_1(1, 2, 2)], S_0)),$$

because $do(B_1(1, 2), S_0) \sqsubset do([B_1(1, 2), A_1(1, 2, 2)], S_0)$. It can be seen that for any goal G, if there is a plan for it, then there is a minimal plan for it:

$$\Sigma \models (\forall s).H(G, s) \supset (\exists s')(s' \sqsubseteq s \wedge \mathtt{minimal}(G, s')).$$

We are now ready to formalize the informal reading of a clause containing ! at the beginning of this section. For this purpose, we introduce a new predicate $Acc(s)$, meaning that s is not "cut off" by cut.

If a program contains the cut rule $cut(\vec{x})$, and the goal G_1 succeeds, then for $Acc(s)$ to hold, it must be the case that for every minimal plan s_1 of F in s, either it uses a rule before this cut rule or it uses this cut rule with the first derivation of G_1:

$$\begin{aligned} Acc(s) \supset (\forall \vec{x})\{(\exists s')[Acc(s') \wedge (\exists \vec{\xi}) H(G_1, s')] \supset \\ (\forall s_1)[s_1 \sqsubseteq s \wedge \mathtt{minimal}(F(\vec{x}), s_1) \supset \\ (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x})) \vee \\ (\exists s_2, s_3)(s_1 = do(cut(\vec{x}), s_2) \wedge s_3 \sqsubseteq s_2 \wedge \\ \mathtt{first\text{-}der}(G_1, s_3, \vec{x}))]\}, \end{aligned} \quad (11)$$

where $\vec{\xi}$ is the tuple of the free variables that are in G_1 but not in $\vec{x}$, and generally, for any goal G, and situation term S, and any tuple $\vec{y}$ of variables, $\mathtt{first\text{-}der}(G, S, \vec{y})$ means that S is the first derivation of G with the variables in $\vec{y}$ fixed:[6]

$$(\exists \vec{v})\mathtt{minimal}(G, S) \wedge \neg(\exists s)[Acc(s) \wedge (\exists \vec{v})\mathtt{minimal}(G, s) \wedge s \prec S],$$

where $\vec{v}$ is the tuple of variables that are in G but not in $\vec{y}$.

Here we notice that in the axiom (11), the formula

$$(\exists s')(Acc(s') \wedge (\exists \vec{\xi}) H(G_1, s'))$$

corresponds to "G_1 succeeds", the case

$$(\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x}))$$

corresponds to "the derivation s_1 of $F(\vec{x})$ uses a rule before the cut rule", and the other case corresponds to "uses this rule with the first derivation of G_1".

Now given a program P, suppose

$$Acc(s) \supset \Psi_1(s),$$
$$\vdots$$
$$Acc(s) \supset \Psi_k(s),$$

are all the axioms about Acc as given above for every occurrence of ! in P (a single clause may have multiple occurrences of !). We call the following *the accessibility axiom* of P:

$$Acc(s) \equiv \Psi_1(s) \wedge \cdots \wedge \Psi_k(s). \quad (12)$$

Notice that this axiom attempts to define Acc recursively since the predicate also occurs in the right hand side of

[6] Notice that if none of G and S contains variables, then $\mathtt{first\text{-}der}(G, st, \vec{y})$ is independent of $\vec{y}$.

the equivalence. This should not be surprising. For example, the logical formalization of negation in logic programs normally requires fixed-point constructions, and negation is usually implemented by cut.

Definition 5 (Extended Action Theory) *Let P be a program, and Δ a given set of axioms about $\prec$ on actions.* The extended action theory *$\mathcal{E}$ of P is the the following set:*

$$\mathcal{E} = \mathcal{D} \cup \Delta \cup \{Acc, (10)\},$$

where $\mathcal{D}$ is the basic action theory for P, and Acc is the accessibility axiom of P of the form (12).

Definition 6 *Let P be a program, $\mathcal{E}$ its extended action theory, and G a goal. A situation term S is an* accessible *plan for G iff $\mathcal{E} \models Acc(S) \wedge H(G, S)$.*

We illustrate the definitions with our *max* example. Suppose we use Prolog's search strategy, and order the actions as:

$$\Delta = \{a \prec b \equiv (\exists \vec{x})a = A_1(\vec{x}) \wedge (\exists \vec{y})b = A_2(\vec{y})\}.$$

Notice that we do not care about how the rules about *le* are ordered. Since the only occurrence of ! is in A_1, the accessibility axiom is

$$\begin{aligned}&Acc(s) \equiv (\forall x, y, z)\{(\exists s')(Acc(s') \wedge H(le(x,y), s')) \supset \\ &\quad (\forall s_1)[s_1 \sqsubseteq s \wedge \mathtt{minimal}(max(x,y,z), s_1) \supset \\ &\quad\quad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec A_1(x,y,z)) \vee \\ &\quad\quad (\exists s_2, s_3)(s_1 = do(A_1(x,y,z), s_2) \wedge s_3 \sqsubseteq s_1 \wedge \\ &\quad\quad\quad \mathtt{first\text{-}der}(le(x,y), s_3, x, y, z))]\}.\end{aligned}$$

Now let $\mathcal{E}$ be the extended action theory of the program. First of all, we can show that

$$\mathcal{E} \models (\forall x, y, s).\mathtt{minimal}(le(x,y), s) \supset Acc(s),$$

thus

$$\begin{aligned}\mathcal{E} \models (\forall x, y)[(\exists s)H(le(x,y), s) \equiv \\ (\exists s)(Acc(s) \wedge H(le(x,y), s))].\end{aligned}$$

This is intuitively right since the only appearance of ! is in the definition of *max,* and the presence of ! has no effect on *le.* For *max,* we can show:

$$\begin{aligned}\mathcal{E} \models (\forall x, y)\{(\exists s)H(le(x,y), s) \wedge x \neq y \supset \\ [(\exists s)(Acc(s) \wedge H(max(x,y,y), s)) \wedge \\ \neg(\exists s)(Acc(s) \wedge H(max(x,y,x), s))]\}.\end{aligned}$$

So the *max* program indeed defines *max* correctly. However since

$$\begin{aligned}\mathcal{E} \not\models (\forall x, y, z)[(\exists s)H(max(x,y,z), s) \equiv \\ (\exists s)(Acc(s) \wedge H(max(x,y,z), s))],\end{aligned}$$

this use of cut is not ideal in the sense that it does more than search control.

In general, we say that the cuts in a program P are for *pure search control* if for any goal *G,*

$$\mathcal{E} \models (\exists s)H(G, s) \equiv (\exists s)(Acc(s) \wedge H(G, s)).$$

Some ramification of this definition for identifying improper uses of cut will be explored further in the full version[7] of this paper.

5 Some Properties

As we have noticed, the accessibility axiom attempts to define *Acc* recursively. A natural question then is if the recursion will yield a unique solution for the predicate. In general, the answer is negative. However, if a program is properly stratified, then the axiom will yield a unique solution.

Let P be a program, and *F* a fluent. We say that the definition of *F* in *P* is *cut-free* if none of the clauses that are *relevant* to *F* contains !. Here a clause is relevant to *F* if, inductively, either it's in the definition of *F* or it's relevant to another fluent that appears in the definition of *F.* For example, the definition of *le* in the *max* example is cut-free. For cut-free fluents, *Acc* does not play a role:

Proposition 1 *Let P be a program, and $\mathcal{E}$ its extended action theory. If the definition of a fluent F in P is cut-free, then we have:*

$$\begin{aligned}&\mathcal{E} \models \mathtt{minimal}(F(\vec{x}), s) \supset Acc(s), \\ &\mathcal{E} \models (\exists s)H(F(\vec{x}), s) \equiv (\exists s)(Acc(s) \wedge H(F(\vec{x}), s)).\end{aligned}$$

Cut-free fluents are the ground case of stratified programs:

Definition 7 *A program P is* stratifiedt/ *there is a function f from fluents in P to natural numbers such that*

1. *If F is cut-free in* P, *then f(F)* = 0.
2. *If F is not cut-free, then f(F) is*

$$1 + max\{f(F') \mid F' \text{ appears in the definition of } F\}.$$

Theorem 1 *Let P be a stratified program, and $\mathcal{E}$ its corresponding extended action theory. There is a formula Φ that does not mention Acc such that*

$$\mathcal{E} \models Acc(s) \equiv \Phi(s).$$

We end this section with a theorem that shows the correctness of the usual implementation of negation using cut.

Let *P* be a logic program with negation (not) but without cut. Suppose that for each fluent *F* in P, *F'*

[7] In preparation. But see http://www.cs.ust.hk/ ~flin

is a new fluent of the same arity. Let P' be the logic program obtained by replacing every literal of the form not $F(\vec{t})$ in P by $F'(\vec{t})$, and by adding, for each new fluent F', the following two clauses:

$$A_F(\vec{x}): \quad F'(\vec{x}) \text{ :- } F(\vec{x}) \;\&\; ! \;\&\; \texttt{fail}.$$
$$A'_F(\vec{x}): \quad F'(\vec{x}).$$

Suppose that for each fluent F, the action *AF* is ordered before the action A'_F

Theorem 2 *Let $\mathcal{E}$ be the extended action theory of P', and V the action theory for P as defined in (Lin and Reiter [6]). For fluent F in P, and any tuple $\vec{t}$ of terms of sort object, we have $\mathcal{D} \models (\exists s)H(F(\vec{t}), s)$ iff $\mathcal{E} \models (\exists s).Acc(s) \wedge H(F(\vec{t}), s)$.*

Prom this theorem, we conclude that the usual implementation of negation using cut is correct with respect to the semantics given in (Lin and Reiter [6]). As noted in (Lin and Reiter [6]), the semantics given there for logic programs with negation yields the same results as that given in (Wallace [10]), and the latter has been shown to be equivalent to the stable model semantics of (Gelfond and Lifschitz [2]) when only Herbrand models are considered. Therefore we can also conclude that the usual implementation of negation in terms of cut is correct with respect to the stable model semantics for logic programs with negation in the propositional case.

6 Concluding Remarks

We have applied the situation calculus to logic programming by giving a semantics to programs with cut. We have shown that this semantics has some desirable properties: it is well-behaved when the program is stratified, and that according to this semantics, the usual implementation of negation-as-failure operator by cut is provably correct with respect to the stable model semantics.

Our more long term goal is to use the situation calculus as a general framework for representing and reasoning about control and strategic information in problem solving. In this regard, we have made some preliminary progress in applying the situation calculus to formalizing control knowledge in planning. As we mentioned in Section 1, in AI planning, a plan is a sequence of actions, thus isomorphic to situations. So control knowledge in planning, which often are constraints on desirable plans, becomes constraints on situations. Based on this idea, in (Lin [4]), we formulate precisely a subgoal ordering in planning in the situation calculus, and show how information about this subgoal ordering can be deduced from a background action theory. We also show for both linear and nonlinear planners how knowledge about this ordering can be used in a provably correct way to avoid unnecessary backtracking.

Regarding our situation calculus semantics for the cut operator, there are many directions for future work. First of all, there is a need to compare our semantics with recent work of [l; 9]. More importantly, we should use this semantics to clarify the proper roles of cut in logic programming, to study the possibility of a better control mechanism, and to do verifications and syntheses of logic programs with cut.

Acknowledgements

Part of this work was done while the author was with the Cognitive Robotics Group of the Department of Computer Science at the University of Toronto. This work is also supported in part by grant DAG96/97.EG34 from the Hong Kong Government.

The author would like to thank Eyal Amir, Yves Lesperance, Hector Levesque, and especially Ray Reiter for helpful discussions relating to the subject of this paper and/or comments on earlier versions of this paper.

References

[l] J. Andrews. A paralogical semantics for the prolog cut. In *Proc. of Int. Logic Programming Symposium,* pages 591-605, 1995.

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming,* pages 1070-1080, 1988.

[3] C. C. Green. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-69),* pages 219-239, 1969.

[4] F. Lin. An ordering on subgoals for planning. *Annals of Mathematics and Artificial Intelligence. Special issue in honor of Professor Michael Gelfond,* To appear. http://www.cs.ust.hk/~flin.

[5] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes,* 4(5):655-678, 1994.

[6] F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. *J. of Logic Programming,* To appear. http://www.cs.toronto.edu/~cogrobo/.

[7] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4,* pages 463-502. Edinburgh University Press, Edinburgh, 1969.

[8] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy,* pages 418-420. Academic Press, San Diego, CA, 1991.

[9] K. Stroetmann and T. Glab. A declarative semantics for the prolog cut operator. In *Proc. of 5th Int. Workshop on Extensions of Logic Programming,* pages 255-271, 1996.

[10] M. G. Wallace. Tight, consistent, and computable completions for unrestricted logic programs. *Journal of Logic Programming,* 15:243-273, 1993.

TEMPORAL REASONING

Temporal Reasoning 1