A Situation Calculus Semantics for the Prolog Cut Operator

Fangzhen Lin
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
Email: fl@cs.toronto.edu
Tel. (416) 978 6277 Fax (416) 978 1455

Abstract

Given a definite logic program with cut, we proceed in two steps to give it a semantics. First, we delete cut from the program, and construct the basic action theory for the resulting cut-free program in the situation calculus according to (Lin and Reiter [3]). We then formalize the effects of cut by adding to the basic action theory a situation calculus sentence that characterizes the set of situations (derivations) that are not eliminated by cut. We show that our semantics is well-behaved when the logic program is properly stratified. We also show that according to this semantics, the usual implementation of the negation-as-failure operator using cut is provably correct.

Keywords. Semantics of logic programs. Cut. The situation calculus. Control strategies. Negation as failure.

A Situation Calculus Semantics for the Prolog Cut Operator[1]

Fangzhen Lin
University of Toronto

Abstract. Given a definite logic program with cut, we proceed in two steps to give it a semantics. First, we delete cut from the program, and construct the basic action theory for the resulting cut-free program in the situation calculus according to (Lin and Reiter [3]). We then formalize the effects of cut by adding to the basic action theory a situation calculus sentence that characterizes the set of situations (derivations) that are not eliminated by cut. We show that our semantics is well-behaved when the logic program is properly stratified. We also show that according to this semantics, the usual implementation of the negation-as-failure operator using cut is provably correct.

1 Introduction

Given a definite logic program P that contains cut (`!`), we proceed as follows to provide a semantics for P. First, we ignore cut, and delete all occurrences of `!` in P. This will give us a program that does not mention `!`, so a situation calculus action theory $\mathcal{D}$ for it can be constructed according to (Lin and Reiter [3]). In $\mathcal{D}$, situations are like proofs and derivations. However, due to the presence of `!`, some situations may not be reachable. A logical characterization of cut is then achieved by adding to $\mathcal{D}$ a situation calculus sentence that axiomatizes the set of reachable situations.

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 calculus semantics of (Lin and Reiter [3]) for cut-free logic programs. Section 4 considers the semantics of cut. Section 5 proves some properties of our semantics for cut. Specifically, we show that our semantics is well behaved for properly stratified logic programs, and that the usual implementation of negation by cut is provably correct according to our semantics. Section 6 concludes this paper.

2 Logical Preliminaries

2.1 The Language of the Situation Calculus

The language $\mathcal{L}$ of the situation calculus (McCarthy and Hayes [6]) is a many-sorted first-order one with equality. We assume the following sorts: *state* for situations, *action* for actions, and *object* for everything else. We also assume the following domain independent predicates and functions:

[1]Thanks to Ray Reiter for useful discussions about the subject of this paper, and for his comments on an earlier draft of this paper. This work was supported by grants from the NSERC of Canada, the IRIS of Canada, and the ITRC of Ontario.

- A constant S_0 of sort *state* denoting the initial state.
- A binary function *do* - $do(a,s)$ denotes the state resulting from performing action a in state s.
- A binary predicate *Poss* - $Poss(a,s)$ means that action a is possible (executable) in state s. In this paper we shall assume that actions are always executable, i.e. $(\forall a,s)Poss(a,s)$. So technically, there is no real need for this predicate in this paper. We keep it however in order to be consistent with the general framework of (Reiter [7] and Lin and Reiter [4]).
- A binary predicate (partial order) $<$ over states. 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 $s < s' \vee s = s'$.
- Another binary predicate (partial order) $\sqsubset$ over states. 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 $s \sqsubset s' \vee s = s'$.

Following [6], we define *fluents* to be predicate symbols of arity $object^n \times state$, $n \geq 0$.

2.2 The Discrete Situation Calculus

We shall consider only the discrete situation calculus with the following foundational axioms:[2] [3]

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

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

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

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

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

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

The first two axioms are unique names assumptions for states. The third axiom is second order induction. It amounts to the domain closure axiom that every situation has to be obtained from the initial one by repeatly applying the function *do*.[4] As we shall see, induction plays an important role in this paper.

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

[2] Except for the one about $\sqsubset$, these axioms are taken from (Lin and Reiter [4]), which also proves some useful properties about them.

[3] In this paper, free variables in displayed formulas are assumed to be universally quantified.

[4] For a detailed discussion of the use of induction in the situation calculus, see (Reiter [8]).

have assumed $(\forall a, s)Poss(a, s)$, the partial order $<$ in this paper reduces to the "prefix" relation: Given a state $S = do(\alpha_n, do(\alpha_{n-1}, ..., do(\alpha_1, S_0)...))$, $S' \leq S$ iff there is a $1 \leq k \leq n$ such that $S' = do(\alpha_k, do(\alpha_{k-1}, ..., do(\alpha_1, S_0)...))$. In particular, we have $(\forall s)S_0 \leq s$.

The axiom (6) is a second-order definition of $\sqsubset$. Informally, $S \sqsubseteq S'$ iff S can be obtained from S' by deleting some of its actions. More precisely, suppose $S' = do(\alpha_n, ..., do(\alpha_1, S_0)...)$. Then $S \sqsubseteq S'$ iff there are integers $1 \leq i_1 \leq \cdots \leq i_k \leq n$ such that $S = do(\alpha_{i_k}, ..., do(\alpha_{i_1}, S_0)...)$. For instance, the following is the set of states $\sqsubset do(\alpha_3, do(\alpha_2, do(\alpha_1, S_0)))$:

$$\{do(\alpha_3, do(\alpha_2, S_0)),\ do(\alpha_3, do(\alpha_1, S_0)),\ do(\alpha_2, do(\alpha_1, S_0)),$$
$$do(\alpha_1, S_0),\ do(\alpha_2, S_0),\ do(\alpha_3, S_0),\ S_0\}.$$

To see that the axiom (6) formalizes this partial order, consider

$$do(\alpha_3, do(\alpha_1, S_0)) \sqsubset do(\alpha_3, do(\alpha_2, do(\alpha_1, S_0))).$$

Any function f that is monotonic wrt $<$, and satisfies

$$f(S_0) = S_0,\ f(do(\alpha_1, S_0)) = do(\alpha_2, do(\alpha_1, S_0)),$$
$$f(do(\alpha_3, do(\alpha_1, S_0))) = do(\alpha_3, do(\alpha_2, do(\alpha_1, S_0)))$$

can be used to prove this relation using the axiom (6). 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.

Notice the similarity between some of these axioms and the Peano foundational axioms for number theory. However, unlike Peano arithmetic which has a unique successor function, we have a class of successor functions here represented by the function *do*. In the following, we shall denote by Σ the set of the above axioms.

2.3 Logic Programs

We consider definite logic programs with cut.

An *atom* p is an expression of the form $F(t_1, ..., t_n)$, where F is a fluent of arity $object^n \times state$, and $t_1, \ldots, t_n$ are terms of sort *object*. Notice that an atom is not a formula in the situation calculus. It is an expression obtained from an atomic formula by suppressing its state argument.

A *goal* G is an expression of the form

$$l_1 \ \texttt{\&} \ \cdots \ \texttt{\&} \ l_n$$

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 `!`.

A *clause (rule)* is an expression of the form

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

where F is a fluent of the arity $object^n \times state$, $x_1, ..., x_m$ are distinct variables of sort $object$, and G is a goal. In the following, for any terms $t_1, ..., t_n$ of sort $object$, we shall also write and call a clause an expression of the form

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

Formally, however, this expression is taken to be a *shorthand* for the following clause:

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

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 $G = l_1 \;\texttt{\&}\; \cdots \;\texttt{\&}\; l_n$, and a state term st, we define $G[st]$, the truth value of G at the state st, to be the situation calculus formula

$$l_1[st] \wedge \cdots \wedge l_n[st],$$

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

1. If l_i is $F(t_1, ..., t_n)$, then $l_i[st]$ is $F(t_1, ..., t_n, st)$.
2. If l_i is $t = t'$, then $l_i[st]$ is l_i.
3. If l_i is `!`, then $l_i[st]$ is the tautology $true$.

For example,

$$(x = a \;\texttt{\&}\; y = b \;\texttt{\&}\; parent(x, y) \;\texttt{\&}\; \texttt{!} \;\texttt{\&}\; parent(x, z))[S_0]$$

is

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

3 A Logical Semantics

This section considers the semantics of a logic program when `!` is ignored. It is basically a review of that in (Lin and Reiter [3]).

We mentioned that the first step in our efforts to provide a semantics for cut is to delete it from the underlying program. However, since we have defined $\texttt{!}[s]$ to be a tautology for any state s, as we shall see, there is no need to do the physical deletions.

Given a clause of the form

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

Suppose that A is an action (function) symbol of the arity $object^n \rightarrow action$ in our situation calculus language. Then the clause is like a description of the following effect of A on F: if G holds initially, then F will hold after the action is performed. In the situation calculus, this effect of A can be formalized by the following axiom:

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

where $\vec{x}$ is $(x_1, ..., x_n)$, and $\vec{\xi}$ is the tuple of variables that are in G but not in $\vec{x}$.

Consider the following clause:

```
ancestor(x,y) :- parent(x,z) & parent(z,y).
```

Suppose that $gp(x, y)$ is the action for this clause, then we have the following effect axiom:

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

In the following, we shall write on the left hand side of a clause its corresponding action. For instance, the above *ancestor* clause and its corresponding action can be written as:

$gp(x, y)$: `ancestor(x,y) :- parent(x,z) & parent(z,y).`

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

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

Then we have the following corresponding effect axioms for the fluent F:

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

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 have the following *successor state axiom* (Reiter [7]) for F:

$$\begin{array}{l} F(\vec{x}, do(a, s)) \equiv \{a = A_1(\vec{x}) \wedge (\exists \vec{y_1})G_1[s] \vee \cdots \vee \\ \qquad (a = A_k(\vec{x}) \wedge (\exists \vec{y_k})G_k[s]) \vee F(\vec{x}, s)\}. \end{array} \tag{7}$$

Notice the similarity between this axiom and Clark's completion of the predicate F in P.

Intuitively, the successor state axiom for F says that the fluent is true in a successor state 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

$$F(\vec{x}, do(a, s)) \equiv F(\vec{x}, s).$$

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 2.2.*
- $\mathcal{D}_{ss}$ *is the set of successor state axioms for the fluents according to* P.
- $\mathcal{D}_{una}$ *is the following set of unique names axioms:*

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

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

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

  for every function symbol f. *Notice that* $\mathcal{D}_{una}$ *includes unique names axioms for actions.*
- $\mathcal{D}_{S_0}$ *is:*

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

Definition 2 *Let* P *be a program, and* $\mathcal{D}$ *its corresponding basic action theory. A state term* st *is called a* plan *for a goal* G *iff* $\mathcal{D} \models G[st]$. *A substitution* σ *is an* answer for G *iff* $\mathcal{D} \models (\exists s) G\sigma[s]$, *where* $G\sigma$ *is the goal resulting from* G *by simultaneously substituting for its variables according to* σ.

Therefore query answering in logic programs literally becomes planning in the style of (Green [2]) in the situation calculus. This semantics has some nice properties. It is closely related to a recent proposal by Wallace [9], and generalizes the Clark completion semantics. For details, see (Lin and Reiter [3]).

Notice that for any program P, if P' is the resulting program of deleting all occurrences of `!` in P, then the basic action theories for P and P' are equivalent, assuming that the corresponding rules in P and P' have the same action name. This is because by postulating $(\forall s)\,!\,[s]$, occurrences of cut are effectively removed.

Example 1 Consider the following well-known example of an efficient encoding of $max(x, y, z)$ (z is the maximum of $\{x, y\}$) using `!` (Lloyd [5]):

$A_1(x, y, z)$:	`max(x,y,z) :- le(x,y) & ! & z=y.`
$A_2(x, y, z)$:	`max(x,y,z) :- z=x.`
$B_1(x, y)$:	`le(x,y) :- x=1 & y=2.`
$B_2(x, y)$:	`le(x,y) :- x=1 & y=1.`
$B_3(x, y)$:	`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 two effect axioms about max:

$$le(x, y, s) \wedge true \wedge z = y \supset max(x, y, z, do(A_1(x, y, z), s)),$$
$$z = x \supset max(x, y, z, do(A_2(x, y, z), s)).$$

Thus we have the following successor state axioms for max:

$$max(x, y, z, do(a, s)) \equiv \{a = A_1(x, y, z) \wedge le(x, y, s) \wedge z = y \vee$$
$$(a = A_2(x, y, z) \wedge z = x) \vee max(x, y, z, s)\}.$$

Similarly, we have the following successor state axiom for le:

$$le(x, y, do(a, s)) \equiv \{a = B_1(x, y) \wedge x = 1 \wedge y = 2 \vee$$
$$(a = B_2(x, y) \wedge x = 1 \wedge y = 1) \vee (a = B_3(x, y) \wedge x = 2 \wedge y = 2) \vee le(x, y, s)\}.$$

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 state satisfying $max(1, 2, 2)$. Thus we have the following desirable conclusion:

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

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:

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

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 plan $do(A_2(x, y, x), S_0)$ will not be considered if there is a plan for $le(x, y)$, which is the case here if $x = 1$ and $y = 2$. So the state $do(A_2(1, 2, 1), S_0)$, which is a plan for $max(1, 2, 1)$ in $\mathcal{D}$, is actually not reachable according to Prolog's search strategy due to the presence of `!` in the first rule.

Our goal in defining a semantics for cut is then to characterize the set of reachable (*accessible*) states. This is what we are going to do in the next section.

4 A Semantics for Cut

A clause containing cut:

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

means that if there is a proof of G_1, then any proof of $F(\vec{x})$ must use either

1. A rule before this one; or

2. This rule with the first proof 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 states for defining "the first proof".

In the following, we shall assume that we are given an ordering $\prec$ on actions (rules), and will define an ordering on states using $\prec$. Intuitively, if $\alpha \prec \beta$, then during the search for a plan for a goal, 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 states, 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 the plan $do(\alpha_n, ..., do(\alpha_1, S_0)...)$ 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', we then have the following definition:

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

$$\begin{aligned} s \prec s' \equiv s = S_0 \wedge s' \neq S_0 \vee & \\ & (\exists a, b, s_1, s_2).s = do(a, s_1) \wedge s' = do(b, s_2) \wedge a \prec b \vee \\ & (\exists a, s_1, s_2).s = do(a, s_1) \wedge s' = do(a, s_2) \wedge s_1 \prec s_2. \end{aligned} \tag{10}$$

With this partial order on states, we can then say, roughly, that a state S is a "first proof" of a goal G if it is a "proof" of G that is minimal according to $\prec$. However, to make precise this statement, we have to decide over which space of alternative "proofs" S is compared to. 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

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

So if S is a plan for G, then for any action α, $do(\alpha, S)$ is also a plan for G. Since the partial order $\prec$ on states (sequences of actions) compares first actions at the end of sequences, this means that a plan for G can always be made "better" according to $\prec$ by appending to it some irrelevant actions. To avoid this pitfall, we define the notion of *minimal plans*:

Definition 4 *For any state term st, any goal G, we denote by* $\texttt{minimal}(G, st)$ *the following formula:*

$$G[st] \wedge \neg(\exists s) s \sqsubset st \wedge G[s].$$

Intuitively, if $\texttt{minimal}(G, st)$ holds, then st is a minimal plan for G because it is a plan for G without containing any redundant actions. For our *max* program in section 3, it is easy to see that

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

But

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

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

$$\Sigma \models (\forall s).G[s] \supset (\exists s')(s' \sqsubseteq s \wedge \texttt{minimal}(G, s')).$$

We are now ready to formalize the informal reading of a clause containing cut at the beginning of this section. We introduce a new predicate *Accessible*. Intuitively, $Accessible(s)$ holds if s is not "cut off" by `!`.

Given the rule $cut(\vec{x})$ at the beginning of the section, its effect on *Accessible* is captured with the following axiom:

$$\begin{aligned}
&Accessible(s) \supset (\forall \vec{x})\{(\exists s')(Accessible(s') \wedge (\exists \vec{\xi})G_1[s']) \supset \\
&\quad (\forall s_1)[s_1 \sqsubseteq s \wedge \texttt{minimal}(F(\vec{x}), s_1) \supset \\
&\qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x})) \vee \\
&\qquad (\exists s_2)s_1 = do(cut(\vec{x}), s_2) \wedge (\exists s_3)(s_3 \sqsubseteq s_1 \wedge \texttt{first-proof}(G_1, s_3, \vec{x}))]\},
\end{aligned} \tag{11}$$

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 state term st, and any tuple $\vec{y}$ of variables, $\texttt{first-proof}(G, st, \vec{y})$ stands for the following formula:[5]

$$(\exists \vec{\nu})\texttt{minimal}(G, st) \wedge \neg(\exists s)(Accessible(s) \wedge (\exists \vec{\nu})\texttt{minimal}(G, s) \wedge s \prec st),$$

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

Refering back to the informal interpretation of the rule $cut(\vec{x})$ at the beginning of the section, we notice that in the axiom (11), the formula

$$(\exists s')(Accessible(s') \wedge (\exists \vec{\xi})G_1[s'])$$

corresponds to "there is a proof of G_1", the formula

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

corresponds to "the proof s_1 of $F(\vec{x})$ uses a rule before the cut rule", and the expansion of the macro $\texttt{first-proof}(G_1, s_3, \vec{x})$:

$$(\exists \vec{\xi})\texttt{minimal}(G_1, s_3) \wedge \neg(\exists s)(Accessible(s) \wedge (\exists \vec{\xi})\texttt{minimal}(G_1, s) \wedge s \prec s_3)$$

corresponds to "s_3 is the first proof of G_1".

Now given a program P, suppose

$$\begin{gathered}
Accessible(s) \supset \Psi_1(s), \\
\vdots \\
Accessible(s) \supset \Psi_k(s),
\end{gathered}$$

[5]Notice that if none of G and st contains variables, then $\texttt{first-proof}(G, st, \vec{y})$ expands to a sentence for any $\vec{y}$.

are all the axioms about *Accessible* 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:

$$Accessible(s) \equiv \Psi_1(s) \wedge \cdots \wedge \Psi_k(s). \tag{12}$$

Notice that this axiom attempts to define *Accessible* recursively since the predicate also occurs in the right hand side of the equivalence. This should not be surprising since the logical formalization of negation in logic programs normally requires fixed-point constructions, and, as it is well known, negation can be easily implemented using 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 state term st is an* accessible *plan for G iff*

$$\mathcal{E} \models Accessible(st) \wedge G[st].$$

A substitution σ is an accessible answer *for G iff*

$$\mathcal{E} \models (\exists s).Accessible(s) \wedge G\sigma[s].$$

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. Notice that by the unique names axioms in $\mathcal{D}_{una}$ (section 3), we have

$$\mathcal{D}_{una} \cup \Delta \models (\forall \vec{x})\neg(\exists a)a \prec A_1(\vec{x}).$$

Since the only occurrence of `!` is in A_1, so the accessibility axiom is

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

Now let $\mathcal{E}$ be the extended action theory of the program. Notice first that by the successor state axiom for *max* in $\mathcal{E}$,

$$\mathcal{E} \models (\forall x, y, z, s)[\neg max(x, y, z, s) \supset (\forall s')(s' \sqsubseteq s \supset \neg max(x, y, z, s'))].$$

Therefore

$$\mathcal{E} \models (\forall s)[\neg(\exists x, y, z) max(x, y, z, s) \supset Accessible(s)].$$

This is intuitively right since the only appearance of `!` is in the definition of max. Thus

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

Thus

$$\mathcal{E} \models (\forall x, y)[(\exists s) le(x, y, s) \equiv (\exists s)(Accessible(s) \wedge le(x, y, s))].$$

This means that the presence of `!` has no effect on le, and proving the existence of accessible plans for le in $\mathcal{E}$ is equivalent to proving the existence of plans for le in $\mathcal{D}$, the basic action theory for the max program.

Now let $S_1 = do(B_1(1, 2), S_0)$, and $S_2 = do(A_1(1, 2, 2), S_1)$. Clearly

$$\mathcal{E} \models le(1, 2, S_1) \wedge \texttt{minimal}(max(1, 2, 2), S_2).$$

We claim that $\mathcal{E} \models Accessible(S_2)$ so that S_2 is an accessible plan for $max(1, 2, 2)$. Notice that

$$\begin{aligned}
&\mathcal{E} \models (\forall x, y, z).max(x, y, z, S_2) \supset (x = 1 \wedge y = 2 \wedge z = 2),\\
&\mathcal{E} \models (\exists s).Accessible(s) \wedge le(1, 2, s),\\
&\mathcal{E} \models \neg(\exists a, s).S_2 = do(a, s) \wedge a \prec A_1(1, 2, 2).
\end{aligned}$$

Therefore

$$\mathcal{E} \models Accessible(S_2) \equiv (\exists s_3)(s_3 \sqsubseteq S_2 \wedge \texttt{first-proof}(le(1, 2), s_3, x, y, z)).$$

But $S_1 \sqsubset S_2$, so $\mathcal{E} \models Accessible(S_2)$ if

$$\mathcal{E} \models \texttt{first-proof}(le(1, 2), S_1, x, y, z),$$

that is

$$\mathcal{E} \models \texttt{minimal}(le(1, 2), S_1) \wedge \neg(\exists s).Accessible(s) \wedge \texttt{minimal}(le(1, 2), s) \wedge s \prec S_1.$$

This follows from

$$\mathcal{E} \models (\forall s)(\texttt{minimal}(le(1, 2), s) \supset s = S_1).$$

On the other hand, there are no accessible plans for $max(1, 2, 1)$. For suppose $\mathcal{E} \models max(1, 2, 1, S)$. Then since

$$\mathcal{E} \models (\forall s).\texttt{minimal}(max(1, 2, 1), s) \supset s = do(A_2(1, 2, 1), S_0)$$

and

$$\mathcal{E} \models (\exists s).Accessible(s) \wedge le(1, 2, s),$$

it follows that for $Accessible(S)$ to be true, it must be the case that

$$\mathcal{E} \models (\exists s).s \sqsubseteq do(A_2(1, 2, 1), S_0) \wedge le(1, 2, s),$$

which is obviously impossible.

Generalizing the above reasoning, we have

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

This means that according to our semantics for `!`, the *max* program indeed defines *max* correctly. Notice that this depends critically on the ordering of actions. We can show that

$$\mathcal{E} \models (\forall x, y)(\exists s)(Accessible(s) \wedge max(x,y,x,s))$$

if the action A_2 is ordered before A_1.

5 Some Properties

As we have noticed, the accessibility axiom attempts to define *Accessible* recursively. A natural question then is if the recursion will yield a unique solution for the predicate. In general, the answer is negative. We shall see an example later. 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, *Accessible* 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:*

$$\mathcal{E} \models \texttt{minimal}(F(\vec{x}), s) \supset Accessible(s),$$
$$\mathcal{E} \models (\exists s)F(\vec{x}, s) \equiv (\exists s)(Accessible(s) \wedge F(\vec{x}, s)).$$

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

Definition 7 *A program P is* stratified[6] *if 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*

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

[6]This notion of stratification is more restrictive than necessary. In particular, if F is not cut free, then F cannot appear in the body of any clause in its definition. We are working on ways to relax this.

It is clear that if P is stratified, then for any fluent F in P, $f(F)$ is uniquely determined by the above two rules. In such case, we shall call the uniquely determined number the *rank* of F in P, and the maximum of the ranks of the fluents in P will be called the rank of the program P, For instance, in our *max* example, the rank of *le* is 0, and the rank of *max* is 1. So the rank of the program is 1.

We now show that if P is stratified, then its accessibility axiom can be written equivalently in a recursion-free form. To that end, we introduce a new set of unary predicates $Acc_0, Acc_1, .., Acc_n, ...$ Intuitively, $Acc_n(s)$ holds if s is accessible (reachable) when all rules that mention a fluent of rank $> n$ are deleted from P. Formally, they are defined as follows:

$$Acc_0(s) \equiv true. \tag{13}$$

For $n > 0$, suppose

$$\begin{array}{ll} cut_1(\vec{x_1}): & F_1(\vec{x_1}) \texttt{ :- } G_1 \texttt{ \& ! \& } G'_1. \\ \vdots & \vdots \\ cut_k(\vec{x_k}): & F_k(\vec{x_1}) \texttt{ :- } G_k \texttt{ \& ! \& } G'_k. \end{array}$$

are all of the occurrences of `!` in clauses whose heads contain fluents of rank n. Then Acc_n is defined by the following axiom:

$$Acc_n(s) \equiv Acc_{n-1}(s) \wedge \Psi_1(s) \wedge \cdots \wedge \Psi_k(s), \tag{14}$$

where for any $1 \leq i \leq k$, $\Psi_i(s)$ is obtained as follows: Suppose $\Phi_i(s)$ is a formula such that

$$(\forall s).Accessible(s) \supset \Phi_i(s)$$

is the axiom of the form (11) for the rule $cut_i(\vec{x_i})$. Then $\Psi_i(s)$ is the result of replacing every occurrence of *Accessible* in $\Phi_i(s)$ by Acc_{n-1}.

Now since the right hand of equivalence in the axiom (14) mentions only Acc_{n-1}, we have define Acc_n, $n \geq 0$, inductively.

Theorem 1 *Let P be a stratified program, M its rank, and $\mathcal{E}$ its corresponding extended action theory. We have*

$$\mathcal{E} \models Accessible(s) \equiv Acc_M(s).$$

In general, however, there may be multiple interpretations of *Accessible*. Consider the following program:

```
r1:    p :- q'.
r2:    q :- p'.
r3:    p' :- p & ! & fail.
r4:    p'.
r5:    q' :- q & ! & fail.
r6:    q'.
```

Notice that the definition of $fail$ is empty, so $(\forall s)\neg fail(s)$. This program is clearly not stratified. Suppose the ordering on actions is

$$\Delta = \{(\forall a, b)[a \prec b \equiv (a = r_3 \wedge b = r_4) \vee (a = r_5 \wedge b = r_6)]\}.$$

It can be shown that the accessibility axiom of this program is equivalent to:

$$Accessible(s) \equiv$$
$$[p'(s) \supset \neg(\exists s')(Accessible(s') \wedge p(s'))] \wedge [q'(s) \supset \neg(\exists s')(Accessible(s') \wedge q(s'))].$$

Let $\mathcal{E}$ be the extended action theory of this program, we can then show that

$$\begin{aligned}\mathcal{E} \quad \models \quad & (\exists s)(Accessible(s) \wedge p(s)) \wedge \neg(\exists s)(Accessible(s) \wedge q(s)) \vee \\ & (\exists s)(Accessible(s) \wedge q(s)) \wedge \neg(\exists s)(Accessible(s) \wedge p(s)).\end{aligned}$$

So there is an accessible plan for p iff there is not an accessible plan for q, and the other way around as well.

Notice that the program is a rendition of the following logic program:

```
p :- not q.
q :- not p.
```

with negation implemented by cut as:

```
not F :- F & ! & fail.
not F.
```

For this program, our semantics yields equivalent results as that of the stable model semantics of (Gelfond and Lifschitz [1]) for logic programs with negation. Our following theorem shows that this equivalence holds for arbitrary normal programs as well.

Let P be a logic program with negation but without cut. Suppose that for each fluent F in P, F' 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:

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

Suppose that for each fluent F, the action A_F is ordered before the action A'_F.

Theorem 2 *Let $\mathcal{E}$ be the extended action theory of P', and $\mathcal{D}$ the action theory for P as defined in (Lin and Reiter [3]). For fluent F in P, and any tuple $\vec{t}$ of terms of sort object, we have $\mathcal{D} \models (\exists s)F(\vec{t}, s)$ iff $\mathcal{E} \models (\exists s).Accessible(s) \wedge F(\vec{t}, s)$.*

From this theorem, we conclude that the usual implementation of negation using cut is correct with respect to the semantics given in (Lin and Reiter [3]). As noted in (Lin and Reiter [3]), the semantics given there for logic programs with negation yields the same results as that given in (Wallace [9]). The latter has been shown to be equivalent to the stable model semantics 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.

6 Concluding Remarks

It is often said that the cut operator in logic programming languages is like the goto statements in the conventional programming languages. Unstructured use of it leads to programs that are hard to understand, so are prone to errors. The recursive nature of the accessibility axiom (12) certainly supports this point of view. On the other hand, in view of Theorem 1, the use of cut can be made safe if we insist on stratified programs.

This paper also supports the claims made in (Lin and Reiter [3]) that the general framework set out there is useful for formalizing search control operators, and it is worthwhile to go beyond the proposal of Wallace [9], and introduce actions and their axiomatizations in the situation calculus.

References

[1] 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.

[2] 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.

[3] F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. Submitted to this symposium.

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

[5] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[6] 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.

[7] 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.

[8] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

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