A Feasible Approach to Disjunctive Knowledge in Situation Calculus

by

Stavros Vassos

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

A Feasible Approach to Disjunctive Knowledge in Situation Calculus

Stavros Vassos

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

In this thesis we present $\mathcal{L}_P$, a reinterpretation of situation calculus based on intuitions from many-valued logics. The key difference is that the notion of truth is based on the fact that a term is interpreted into a set of objects rather than one single object and equality is interpreted as "possibly equals". $\mathcal{L}_P$ is suitable for defining action theories that capture fluent-based disjunctive knowledge, which means that any incomplete knowledge the theory captures is limited to be about the value of one fluent each time. This essentially enforces an independence assumption on the fluents which allows for efficient evaluation mechanisms. We show that like situation calculus a similar regression theorem holds in $\mathcal{L}_P$. Furthermore, we prove that $\mathcal{L}_P$ can be embedded in situation calculus and show how a special form of $\mathcal{L}_P$ theories can be soundly implemented in Prolog and the agent programming language Indigolog.

To my parents

Στους γονείς μου

Acknowledgements

In the University of Toronto I had the unique opportunity to meet and have as my advisor, Professor Hector Levesque to whom I am more than grateful for his continuous guidance and support. I want to deeply thank him for being considerate, patient and understanding in my first steps in research.

I would like to thank Professor Stathis Zachos and Professor Pavlos Peppas who introduced me to the field of logic and artificial intelligence when I was an undergraduate student in the National Technical University of Athens. Furthermore, I want to thank them and also Professor Timos Sellis for encouraging me to pursue graduate studies far away from my home country.

I want to deeply thank Rosalia and Niki for all the support at the times I needed it most, as well as all my friends in Greece who have always been close to me.

Many thanks to the Greek community of UofT and all of my new friends in Canada for the warm welcome. In particular, I would like to say thank you to Kassius, Sebastian, Matei, Lap Chi and all the people involved in F.E.T.A. Productions for the great times of creativity and joy that we shared in tackling music, sports, life.

Last but not least, I would like to thank my family for always doing as much as they can to help me in every possible way.

Contents

Chapter 1

Introduction

In the field of knowledge representation and reasoning about actions we are interested in logical systems which are capable of representing dynamic domains. We view this problem from the standpoint that these formalisms are intended to be implemented and applied to real robotic systems or disembodied agents in the internet, providing a representation of the world and the means to reason about it. There are two main requirements:

- We want a logical formalization which is rich enough to express most of the intuitive facts about dynamic domains.

- We want it to be efficient enough so that even computationally modest middle-ware devices can use it to do reasoning.

In this classical trade-off between expressiveness and efficiency, it is not difficult to see that disjunctive knowledge plays a key role. The representation of disjunctive knowledge is crucial in terms of which domains can be represented and what kind of the reasoning can be performed, but is always related with some form of exponential complexity in the theorem proving task. Also, note that we are interested in approaches that essentially account for a dynamic epistemic state of the agent, in the sense that the agent's model of the world is affected by directly sensing the real world.

The situation calculus [MH69] is a standard formalism used in the field of reasoning about actions and much research has been done on approaches which provide various levels of expressiveness, while solving some of the classical identified problems (qualification, frame and ramification problem). In this thesis we explore some *alternative semantics* for the situation calculus, based on intuitions from *many-valued logics*. We present $\mathcal{L}_P$, a logical formalism which features extended expressiveness in the way the disjunctive knowledge is treated.

The key differences are that the notion of truth is based on the fact that a term is interpreted into a *set of objects* rather than one single object and that equality is interpreted as "*possibly equals*". In this setting, unlike standard semantics, disjunctive knowledge can *also* be captured *inside* the structures themselves. This feature allows for expressiveness, regarding disjunctive knowledge, which resembles the distinction between $Know(\alpha \vee \beta)$ and $Know(\alpha) \vee Know(\beta)$, as appears in a modal setting of knowledge.

We utilize this expressiveness to define theories of action which capture incomplete knowledge, but also allow for efficient evaluation mechanisms. We define theories such that the general disjunctive knowledge which $\mathcal{L}_P$ allows for, is bounded to be *fluent-based* only; that is, any incomplete knowledge the theory captures, refers to one fluent of the language only.

In recent results [PL02], Petrick and Levesque show that in order for a situation calculus theory to capture this kind of incomplete knowledge, it is not enough to ensure that the initial situation S_0 has this intended property. On the contrary, it must be seriously restricted to exclude some intuitive behaviors, such as successor state axioms which relate the value of one fluent with the value of some other fluent. Unlike the situation calculus, $\mathcal{L}_P$ is suitable for refining the effect of formulas, such as the aforementioned successor state axioms, so that fluent-based disjunctive knowledge persists in all situations. In fact, this refinement is mainly based on the non-standard $\mathcal{L}_P$ semantics rather than the syntax, which is kept very similar to the syntax of situation calculus.

We define $\mathcal{L}_P$ theories of action and we show that a restricted form of these theories capture fluent-based disjunctive knowledge only and can therefore be efficiently implemented in Prolog. When these theories capture no incomplete knowledge at all, then they reduce to basic action theories. What is more interesting is that the implementation of these theories can be done naturally on top of the existing *Golog interpreters [LRL$^+$97], [DGLL00], [DGL99].

We note that $\mathcal{L}_P$ is implicitly modal and is intended to represent the epistemic state of the agent. $\mathcal{L}_P$ theories are used to describe *only what the agent knows about the world without modeling what is happening to the real world as well.* The interaction with the real world can be achieved via sensing actions which affect the epistemic state of the agent. The sensing results are intended to be acquired by the real world, "online" and whenever needed, augmenting properly the initial theory. In what follows we will be talking about expressing knowledge and what the agent knows, even if $\mathcal{L}_P$ does not include any $Know$ symbol. The adopted stance implies that whatever can be proved is known and whatever cannot be proved is not known. In fact, in most of the cases we consider, the symbol of logical consequence $\models$ can be understood as proving what the agent *knows*.

The rest of the thesis is organized as follows. In chapter 2, we present the technical background needed for the following chapters and briefly discuss approaches in the literature which focus on efficient ways to deal with special forms of incomplete knowledge.

In chapter 3, we present the syntax and semantics of $\mathcal{L}_P$ languages and discuss some important properties which illustrate the intuitions behind the definitions of $\mathcal{L}_P$ languages.

In chapter 4, we present the *DK theories of action* as the analog of the basic action theories in situation calculus [Rei91], [Rei01] in $\mathcal{L}_P$. We discuss regression in these theories and show that a similar regression theorem holds.

In chapter 5, we present the *closed form DK* action theories as the analog of basic action theories that can be soundly implemented in Prolog [Rei01]. These *DK* action theories are very restricted, but nevertheless extend the basic action theories to account for fluent-based disjunctive knowledge. We show how they can be correctly implemented in Prolog and present a detailed example.

The thesis ends with some conclusions and observations about future work in chapter 6.

Chapter 2

Literature Review

This chapter presents some technical background needed for the following chapters and discusses ideas found in the literature, that inspired and influenced the work for this thesis. In the first part, we present the situation calculus basic action theories and discuss the logic agent programming languages Golog and Indigolog which are based on them. Then, we present some characteristic approaches in the literature about representing the epistemic state of an agent and the intuitions behind them.

2.1 Situation Calculus

The situation calculus [MH69] is a logic language specifically designed for representing dynamically changing worlds. As presented in [Rei01], it is a second order many-sorted language with equality, featuring three disjoint sorts: one for *actions*, one for *situations* and one for everything else needed. We briefly discuss the basic elements of the situation calculus language.

Actions and situations are first order terms of the corresponding sort. The dynamics of the world is captured as the result of actions happening at situations: a situation represents a possible world history, as a sequence of actions. The constant S_0 is used to denote the *initial situation* where no action has been performed. The sequences of

actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s.

The dynamic elements of the world are captured by the *fluents*. The *relational fluents* represent relations that their truth values can change from situation to situation and *functional fluents* represent functions with dynamic behavior. A fluent is denoted by including a situation argument as its last argument, indicating the value of the fluent at that situation.

The language also includes a binary predicate $Poss(a, s)$ which is used to state that it is possible to perform an action a in situation s. The formal definitions and the details missing can be found in [Rei01].

2.1.1 Basic Action Theories

Following earlier ideas in the literature, Reiter presents the *basic action theories* [Rei91] which are situation calculus theories with appealing properties, such as providing solution for the frame and qualification problems [MH69].

A basic action theory $\mathcal{D}$ is a collection of axioms of the following form

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

where

- For each fluent fl, the set $\mathcal{D}_{ss}$ includes a *successor state axiom* of the form

  $$fl(\vec{x}, do(a, s)) \equiv \Phi(\vec{x}, a, s)$$

  if it is a relational fluent or

  $$fl(\vec{x}, do(a, s)) = y \equiv \phi(\vec{x}, a, s, y)$$

  if it is a functional fluent. These axioms characterize the conditions under which the fluent has a specific (truth) value at situation $do(a, s)$ as a function of situation s.

- For each action A, the set D_{ap} includes a *action precondition axiom* of the form

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

  These axioms characterize the conditions under which some action $A(\vec{x})$ can be performed in situation s.

- D_{S_0} is a set of first order formulas describing the initial situation that syntactically only mention situation S_0.

- Σ is a set of domain independent foundational axioms which define the legal situations and $\mathcal{D}_{una}$ includes the uniqueness of names axioms for actions.

We briefly present some important notions regarding these action theories, which we will be referring to in the next chapters.

A situation calculus formula is *uniform in some situation s* iff it does not mention the predicate $Poss$, it does not quantify over variables of sort situation, it does not mention equality on situations and whenever it mentions a term of sort situation in the situation argument position of a fluent, then that term is s. Also, by $do([a_1, ..., a_n], S_0)$ we denote the situation term which we get, when starting from S_0, action a_1 is performed, then action a_2 is performed and so on up to action a_n. That is the term $do(a_n, do(a_{n-1}, ...do(a_1, S_0)...))$.

Regression is a central computational mechanism in the situation calculus which is based on the intuition that, by regressing the effect of the actions performed in some situation, we can determine a logical equivalent formula in the initial situation S_0. Reiter discusses regression in basic action theories [Rei91] and defines an intuitive *regression operator* $\mathcal{R}$ on *regressable formulas*. The *regression theorem*, which is proved in [PR99], shows that entailment of a formula with respect to a basic action theory $\mathcal{D}$ can be reduced to entailment of the regressed formula by $\mathcal{D}_{S_0} \cup \mathcal{D}_{una}$.

Basic action theories can be restricted to have a *closed initial database* [Rei01] so that to be soundly implemented in the logic programming language Prolog. The restrictions

needed are such that the basic action theory is a *definitional* theory which satisfies the requirements of Clark's theorem [Cla78] for implementing first order theories in Prolog. Intuitively, these restrictions force uniqueness of names to all terms and force the theory capture no incomplete knowledge of any kind. [Rei01] discusses in detail an implementation method and proves the *implementation theorem* for basic action theories.

2.1.2 Golog and Indigolog Logic Programming Languages

Basic action theories can be used as the representation of real-world environments for embodied or software agents. One of the basic requirements for such agents is the ability to reason about the outcome of a sequence of actions. This is known as the *projection task*. Furthermore, it is common that the main functionality of the agents relies on planning techniques which compute the necessary sequence of actions, such that after the execution of the actions a given goal holds. It is not difficult to see that this quickly becomes infeasible as the domains become non-trivial.

Levesque et al [LRL$^+$97] suggest an alternative to planning for specifying agent behavior. Instead of taking into account only the goal and then perform search among all possible sequences of actions, domain-dependant high-level programs are specified which essentially *guide* the planning procedure to follow certain action paths in trying to satisfy the goal. Such a high-level program can be variably non-deterministic and resembles a sketch of plan that gives strong clues-restrictions about the intended solution. This is very important, as the search space is drastically reduced and planning on complex domains can be done in variable efficiency.

Golog [LRL$^+$97] is a logic programming language built on top of Prolog according to this intuition. It provides a generic way of describing a domain as a situation calculus basic action theory and also the functionality of *complex actions* that are suitable for defining intuitive high-level execution plans on the domain. The complex actions act as abbreviations for logical expressions in the language of situation calculus and can

be though of as macros which expand to normal situation calculus formulas. This is achieved with the aid of the abbreviation $Do(\delta, s, s')$, whose definition follows. Intuitively, $Do(\delta, s, s')$ holds whenever the situation s' is a terminating situation of an execution of the complex action δ starting in situation s.

- *Primitive actions*: $Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$.

  The primitive actions are the actual actions that can be executed in the represented domain. By the notation $a[s]$ we mean the result of restoring the situation variable s to any functional fluent mentioned by the action term a.

- *Test actions:* $Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s' = s$.

  Here, ϕ is a pseudo-fluent expression which stands for a formula in the language of the situation calculus, but with all situation arguments suppressed. By the notation $\phi[s]$ we mean the result of restoring the situation variable s to any fluent (relational or functional) mentioned in ϕ.

- *Sequence:* $Do([\delta_1; \delta_2], s, s') \stackrel{def}{=} \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.

- *Non-deterministic choice of two actions:*
  $Do(\delta_1 \mid \delta_2, s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.

- *Non-deterministic choice of action arguments:* $Do(\pi x.\delta(x), s, s') \stackrel{def}{=} \exists x.Do(\delta(x), s, s')$

  This means to nondeterministically pick an individual x, and for that x, perform $\delta(x)$.

- *Non-deterministic iteration:* Execute δ zero or more times.
  $Do(\delta^*, s, s') \stackrel{def}{=} \forall P.\{\forall s_1 P(s_1, s_1) \wedge \forall s_1 \forall s_2 \forall s_3.[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \supset P(s, s')$.

  Note that second order functionality is needed to express this.

- *Conditional:* $Do(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, s, s') \stackrel{def}{=} Do([\phi?; \delta_1] \mid [\neg\phi?; \delta_2], s, s')$.

- *While-loops:* $Do(\text{while } \phi \text{ do } \delta, s, s') \stackrel{def}{=} Do([\phi?; \delta^*] \mid [\neg\phi?; \delta_2], s, s')$.

Using the presented Golog functionality, one can specify high-level programs in the context of the domain that is modeled. The Golog interpreter searches *off-line* in the restricted search space so that to provide an instance of the program, as a valid sequence of primitive actions (that is, actions that can be executed in the domain). One can view these programs as specifying a high-level behavior for the agent, or as a guided way to do planning for a desired set of goals. For example, the goals can be captured in the Golog program using test actions to ensure that the intended properties hold.

Note that the final plan to be executed by the agent is computed off-line. So, the interpreter is required to look all the way until the last action, before a single action can actually be executed in the world by the agent. In some cases this is the desired approach, as the total success of the plan can depend strongly in the choices made in very early steps of the sequence of actions. Nevertheless, this is a serious problem when large programs are considered and moreover in the presence of sensing actions, which provide essential information only after they are executed in the world.

To deal with this issue, De Giacomo and Levesque [DGL99] propose a new *incremental way* of interpreting such high-level programs and a new high-level language construct which together provide refined control over the actual execution of the primitive actions. Indigolog executes each complex action *on-line*, which means that at each step of the high-level program, the agent commits to appropriate primitive actions that are actually executed in the world. Furthermore, off-line planning is triggered using a new *search operator* Σ. The complex action $\Sigma\delta$ forces the interpreter to search *off-line* for a successful terminating state for program δ. Indigolog functionality is very useful in the presence of sensing actions, which are intended to be executed on-line. It is then in the hands of the user to exploit the language constructs so that to specify programs which rely on appropriate off-line planning and on-line execution of actions.

Similarly to Golog, Indigolog programs refer to some situation calculus basic action

theory which accounts for sensing actions too. Sensing is treated as in [Lev96], using the special predicate SF and sensed fluent axioms, one for each primitive action, which characterize SF.

Indigolog functionality is based on the special predicates $Trans$ and $Final$, which were first introduced in [DGLL97]. $Trans(\delta, s, \delta', s')$ expresses that, by executing program δ in situation s, one can get to situation s' in one elementary step, with the program δ' remaining to be executed. In a sense, this captures the fact that there is a possible transition from the configuration (δ, s) to the configuration (δ', s'). $Final(\delta, s)$ expresses that program δ can successfully terminate in situation s, or equivalently that the configuration (δ, s) is final. An appropriate $Trans$ and $Final$ axiom is defined for each Golog complex action [DGLL97] and the new search operator Σ [DGL99]. The incremental execution of the programs is then determined by $Trans$ and $Final$. De Giacomo and Levesque also provide an incremental interpreter in Prolog.

We do not proceed into more details regarding the theory and implementation of Indigolog. For the scope of this thesis, we are only interested in the functionality that Indigolog provides: incremental execution of high-level programs in the context of domains which are represented in the Indigolog framework as situation calculus basic action theories. In chapter 5, we present some details on how domains are represented in the Indigolog language, as we discuss that Indigolog functionality can be used as is to represent our theories.

2.2 Reasoning about Actions and Knowledge

We now focus on approaches in the literature that account for the epistemic state of the agent. Essentially, we are interested in logical formalisms that can capture *incomplete knowledge* about the represented domains and provide ways of updating it with the use of sensing.

2.2.1 An Epistemic Fluent in the Situation Calculus

The situation calculus formalism represents the world and does not account for the epistemic state of some agent. Scherl and Levesque [SL03] formalize knowledge in the situation calculus by adapting a standard possible worlds model of knowledge [Hin62], as was done by Moore [Moo85]. A similar approach is adopted by Thielscher in the fluent calculus [Thi00] and by Lobo et al [LMT97] in the $\mathcal{A}$ language [SB01].

Following the possible worlds approach, a binary relation $K(s', s)$ is introduced which captures that *situation s' is accessible from s*. Note that the "official" situation argument for the fluent is the last one. $K(s', s)$ is treated like any other fluent and informally $K(s', s)$ holds when, as far as an agent in situation s knows, it could be in situation s'. The expression $\mathbf{Knows}(W, s)$ is an abbreviation for a formula that uses K and is used to state that W is known in situation s. For example:

$$\mathbf{Knows}(Broken(x), s) = \forall s\ K(s', s) \rightarrow Broken(x, s')$$

Note the convention regarding the situation argument.

Like any other fluent, the truth value of $K(s', s)$ changes due to actions and there is a successor state axiom which captures the fluent's dynamics among situations. The possible situations forming the epistemic state of the agent are affected by knowledge producing actions. In order that all actions are treated uniformly, there is a sensing result function SR and for each action a there is a sensing result axiom of the following form.

$$SR(a(\vec{x}), s) = r \equiv \phi_a(\vec{x}, r, s)$$

For ordinary actions the result is always the same with the specific result not being significant. The successor state axiom for the epistemic fluent K is then the following.

$$K(s'', do(a, s)) \equiv (\exists s'\ s'' = do(a, s') \wedge K(s', s) \wedge Poss(a, s') \wedge SR(a, s) = SR(a, s'))$$

A basic action theory also includes axioms that define the possible world alternatives for

K in the initial situation S_0. These specifications essentially define what is known and what is not known initially.

Scherl and Levesque discuss many interesting properties of the theories presented. They show that a particular modal logic can be modeled by including axioms that place restrictions on the accessibility relation. For instance, the S4 modal logic is modeled by including reflexivity and transitivity axioms. Provided these properties hold for the K relation in initial situations, then they will also hold in every situation resulting from an executable sequence of actions. Finally, they also talk about regression in this setting, where the define a regression operator $\mathcal{R}_\Theta$ and show that a similar regression theorem as in normal situation calculus holds.

Theories of action and knowledge presented in [SL03] offer a large expressive power, but suffer from the difficulties found in all possible worlds approaches to modeling knowledge. The main issue is efficiency, as theorem proving of knowledge formulas may need to check all possible worlds, leading easily to intractability. As a result, there are many approaches in the literature which account for knowledge, but under strong restrictions that guarantee the intended efficiency. We now focus on approaches which do not follow the possible worlds paradigm, but capture only some key features that the possible worlds account for, in a standard first order or propositional setting.

2.2.2 Restricted Knowledge Approaches

In treating theories of action efficiently, *disjunctive knowledge* plays a key role. We refer to disjunctive knowledge as the modeling of arbitrary disjunctions that cannot be broken apart into knowledge of the individual disjuncts. For instance, disjunctive knowledge is captured by the following sentences in the Scherl and Levesque basic action theories of knowledge.

$$\mathbf{Knows}(Door1IsOpen \vee \neg Door2IsOpen, S_0)$$

$$\mathbf{Knows}(Dice = 1 \vee Dice = 2 \vee Dice = 3 \vee Dice = 4, S_0)$$

The first sentence expresses that in the initial situation the agent knows that either door 1 or door 2 is open, while the second expresses that in the initial situation the agent knows that some dice has one of the four possible values 1,2,3 or 4. Note that the second sentence captures disjunctive knowledge which is about a single fluent $Dice(s)$.

In reasoning about action, we are mainly concerned in expressing disjunctive knowledge about the values of fluents. The representation of disjunctive knowledge is a source of intractability, as it leads to exponential number of possibilities for the values, for a fixed number of fluents. One way of limiting the expressiveness so that to avoid the exponential blow-up, is to force fluents to be *independent*. In this way, any disjunctive knowledge captured is about one fluent each time, which is what we call *fluent-based* disjunctive knowledge. Disjunctions can then be broken apart and this brings theorem proving back to tractability. This is called in [PL02] as the *weak disjunctive knowledge property*.

Results in [PL02] have shown that basic action theories of knowledge cannot be forced to capture only fluent-based disjunctive knowledge, unless they are seriously restricted to exclude very common and intuitive behaviors. Even if it would seem logical that if we restrict the description of the initial situation to capture only fluent-based disjunctive knowledge then this property should persist in all situations, unfortunately it is not true. We examine this point in detail.

Consider the following example, as presented in [PL02].

$$F(do(a,s)) \equiv (a = A \wedge G(s)) \vee F(s)$$

$$G(do(a,s)) \equiv G(s)$$

$$\begin{aligned} \exists s_1 \exists s_2 \exists s_3 \exists s_4 \; & K(s_1, S_0) \wedge K(s_2, S_0) \wedge K(s_3, S_0) \wedge K(s_4, S_0) \wedge \\ & F(s_1) \wedge G(s_1) \wedge F(s_2) \wedge \neg G(s_2) \wedge \\ & \neg F(s_3) \wedge G(s_3) \neg F(s_4) \wedge \neg G(s_4) \end{aligned}$$

In the initial situation nothing is known about the truth values of the relational fluents

F and G. The fluents' truth values are not related in any way and it is easy to see that the disjunctive knowledge captured in the initial situation is fluent-based. The successor state axioms are straightforward, capturing that the only change that can happen is that action A makes the truth value of F be the same as G. Nevertheless, even in this very simple setting the intended property does not hold: In the situation $S = do(A, S_0)$ $\mathbf{Knows}(F \vee \neg G, S)$ holds, but neither $\mathbf{Knows}(F, S)$ nor $\mathbf{Knows}(\neg G, S)$ hold.

The point is that the successor state axioms actually relate the values of the fluents in ways which form general disjunctive knowledge. The only safe way out seems to be to avoid context-sensitive successor state axioms that mention more than one fluent, which is a serious and not wanted restriction.

Note that this issue also arises in normal situation calculus. In this case, we refer to the normal disjunctions and whether theorem proving can be broken down to proving that either of the disjuncts holds. Disjunctive knowledge as a term, makes more sense in a modal setting, but we will be using it in settings that do not explicitly account for knowledge, such as normal situation calculus.

We now briefly present two characteristic approaches in the literature which capture some form of incomplete knowledge in an efficient way. These are based on the situation calculus and have influenced our work for this thesis. The section ends with a short discussion on other logical approaches for dealing with incomplete knowledge which are not based in the situation calculus. In some of these approaches it is not the case that both the real world and what the agent knows are represented. In this case, the formalism captures the epistemic state of the agent only, in an essentially implicit modal setting.

2.2.3 Modal Fluents in Situation Calculus

Demolombe and Pozos Parra [DPP00] present a different approach of modeling knowledge in the situation calculus, which forces knowledge be about fluent literals only. They introduce a modal operator K which is combined syntactically with a relational fluent

literal P to form a knowledge fluent KP. Informally, $KP(s)$ is a fluent capturing that P is known to be true in situation s.

In this way, the modal fluents are used to explicitly model literal-based knowledge, without manipulating possible worlds. The epistemic state of the agent is then captured with the aid of pairs of modal fluents, KF and $K\neg F$ for each relational fluent F. Functional fluents are encoded into relational fluents with one extra argument. In addition to specifying a standard successor state axiom for F, a theory includes successor knowledge state axioms for both KF and $K\neg F$. These axioms have the same form as normal successor state axioms. For example,

$$KF(\vec{x}, do(a, s)) \equiv \gamma^{+}_{KF}(\vec{x}, a, s) \vee KF(\vec{x}, a, s) \wedge \gamma^{-}_{KF}(\vec{x}, a, s)$$
$$K\neg F(\vec{x}, do(a, s)) \equiv \gamma^{+}_{K\neg F}(\vec{x}, a, s) \vee K\neg F(\vec{x}, a, s) \wedge \gamma^{-}_{K\neg F}(\vec{x}, a, s)$$

Note that the theory should ensure that both $KF(\vec{x}, s)$ and $K\neg F(\vec{x}, s)$ do not hold in the same situation s, so that knowledge remains consistent.

Demolombe and Pozos Parra also discuss regression in this setting. A regression operator $\mathcal{R}_T$ is defined and it is proven that a regression theorem similar to the one presented in [SL03] holds. Finally, they prove that if the theory includes axioms which define KF and $K\neg F$ in the initial situation S_0, then this results to a complete description of the epistemic state in every situation.

2.2.4 Interval Arithmetic in Situation Calculus

In [Fun99], Funge presents a method of incorporating interval arithmetic into the situation calculus. Interval arithmetic is used to account for knowledge by providing means of expressing incomplete knowledge about the value of fluents.

The approach distinguishes between normal fluents and *sensory fluents* whose value can be sensed through a knowledge producing action. For each sensory fluent f, a corresponding interval-valued epistemic (IVE) fluent $\mathcal{I}_f$ is introduced. This epistemic

fluent captures the possible values for the corresponding normal fluent, using interval arithmetic to represent them in a restricted but concise way.

In more detail, the possible values can be an interval of some number system $\mathbb{X}$, for which a new number system of sort $\mathcal{I}_{\mathbb{X}}$ is defined. The constants of $\mathcal{I}_{\mathbb{X}}$ are the set of pairs $\langle u, v\rangle$ such that $u, v \in \mathbb{X}$ and $u \leq u$. For an interval $\mathbf{x} = \langle u, v\rangle$, $\overline{\mathbf{x}} = u$ and $\underline{\mathbf{x}} = v$ are the lower and upper bound respectively. A value x is contained in $\mathbf{x} = \langle u, v\rangle$ iff $u \leq x \leq v$ and interval $\mathbf{y}$ contains $\mathbf{x}$, $\mathbf{x} \supseteq \mathbf{y}$ iff $\overline{\mathbf{y}} \leq \overline{\mathbf{x}}$ and $\underline{\mathbf{x}} \leq \underline{\mathbf{y}}$. Also, for two intervals $\mathbf{x}, \mathbf{y}$ we say that $\mathbf{x} \leq \mathbf{y}$ iff $\overline{\mathbf{x}} \leq \underline{\mathbf{y}}$.

This approach to knowledge is mainly motivated by the need to express incomplete knowledge about fluents that have real numbers as values, using for example intervals of $\mathcal{I}_{\mathbb{R}^+}$. Nevertheless, any number system can be defined so that to capture the needs of specific domains. Note for instance the case of $\mathcal{I}_{\mathbb{B}}$. This number system includes exactly three intervals $\langle 0, 0\rangle$, $\langle 0, 1\rangle$ and $\langle 1, 1\rangle$ which have the properties that $\langle 0, 0\rangle \leq \langle 0, 1\rangle \leq \langle 1, 1\rangle$ and also that $\langle 0, 0\rangle \subseteq \langle 0, 1\rangle$ and $\langle 1, 1\rangle \subseteq \langle 0, 1\rangle$. 1 and 0 can be used to represent true and false and the intervals $\langle 1, 1\rangle$, $\langle 0, 1\rangle$ and $\langle 0, 0\rangle$ can be used to represent "known to be true", "unknown" and "known to be false" respectively, which imitates the behavior of a three-valued logic.

The intention is to use the interval-valued epistemic fluents to replace the epistemic K fluent of the Scherl and Levesque approach. In this setting, the agent knows that a fluent has a specific value iff $\mathcal{I}_f$ has collapsed into a "thin" interval.

$$\mathbf{Knows}'(f = c, s) \equiv \mathcal{I}_f = \langle c, c\rangle$$

This notion is extended to account for knowledge about arbitrary formulas using $\mathcal{I}_{Knows}(t, s)$ as uniform mechanism to express the "interval value" of any term t in situation s. Finally, the necessary state constraints are presented so that $\mathcal{I}_{Knows}$ subsumes the epistemic K fluent.

2.2.5 Other Approaches to Incomplete Knowledge

We now mention two approaches in the literature which account for some limited form of incomplete knowledge and are not based on the situation calculus.

The first one borrows functionality and intuitions from *many-valued logics* or *multi-valued logics*. In such formalisms, the standard semantics are extended to include more than the two common truth values, true and false [Bel77], [Urq86], [Gin88]. One way this can be utilized to represent lack of knowledge is by the use of an "unknown" truth value added to the usual ones.

The second approach utilizes database functionality to do reasoning about actions with incomplete knowledge. Similar approaches to planning in presence of incomplete knowledge and sensing are [EGW97] and [PS92].

Proper Knowledge Bases

Levesque [Lev98] presents a way of dealing with incomplete knowledge in first-order theories, based on the introduction of a third truth value standing for "unknown". *Proper knowledge bases* are theories of first order languages with equality that include no function symbols. These theories include uniqueness of names axioms for the possibly countably infinite constant symbols and a finite consistent set of formulas of the form

$$\forall(e \rightarrow \rho) \quad \text{or} \quad \forall(e \rightarrow \neg\rho)$$

where e is a quantifier free formula whose only predicate is equality and ρ ranges over predicate atoms. $\forall(\alpha)$ denotes the universal closure of formula α.

The problem of determining whether a sentence is logically entailed by a proper KB is undecidable, even though only a very limited form of incomplete knowledge is allowed. Levesque proposes the reasoning procedure V which always terminates and is logically sound. V evaluates formulas into 1,0 or $\frac{1}{2}$ which correspond to false, true and unknown respectively. This is done in an intuitive way by evaluating the atoms in a standard way

and extending this to arbitrary formulas using min and max appropriately.

A syntactic normal form $\mathcal{NF}$ is also defined and it is proved that for queries in this normal form V is also complete. It is also shown that in the propositional case for every formula there exists a logical equivalent one in $\mathcal{NF}$. Finally, in [LL03] it is shown that V can be efficiently implemented using database techniques.

Planning with Knowledge and Sensing

The PKS planning system [BP98] is a generalization of STRIPS [FN71]. In STRIPS, the state of the world is represented by a database and actions are represented as updates to that database. Instead of a single database, the PKS system uses a set of four databases that collectively represent the epistemic state of the agent rather than the state of the world. Updates in the databases are modeled to modify agent's knowledge appropriately.

Each of the databases capture a different type of knowledge which corresponds to a formal interpretation in a first-order modal logic of knowledge.

- The K_f database stores positive and negative ground literals which are known to the agent. This is not complete knowledge, as no closed world assumption is applied.

- The K_w database stores conjunctions of ground atomic formulas whose truth value is known by the agent. Intuitively, if formula a is in K_w then at planning time either the agent knows a or it knows $\neg a$; which one holds will be resolved at execution time. This database is used for plan-time modeling of the effects of sensing actions.

- The K_v database stores un-nested function terms whose values are known to the agent at execution time. K_v is used for plan-time modeling of the effect of sensing actions that return numeric values.

- The K_x database captures information about "exclusive or" disjunctive knowledge of ground literals of the form $(l_1|l_2|...|l_n)$. Intuitively, this formula represents the

fact that the agent knows that exactly one of the l_i is true.

Note that the incomplete knowledge that can be expressed in PKS is limited to complete lack of knowledge about an atom, by leaving it out of K_f, and knowledge that only one of a finite set of literals are true, using K_x.

Chapter 3

$\mathcal{L}_P$ Languages

The syntax of an $\mathcal{L}_P$ language is similar to the syntax of Situation Calculus as presented in [Rei01]. Predicate symbols and relational fluent symbols are missing in $\mathcal{L}_P$, but their effect can be emulated by function symbols.

$\mathcal{L}_P$ semantics are based on standard first order Tarskian semantics. The main difference is that the notion of truth in a structure is based on the fact that a term is interpreted as a (non-empty and possibly singleton) *set of objects* rather than one object. This is a result of allowing the interpretation of *fluent symbols* to range over the powerset of the object domain, while also providing a consistent way of "propagating" these sets of objects to the interpretation of terms built upon them.

This way, the semantics intend to provide a refined way of treating disjunctive knowledge. Each structure inherently captures a special form of incomplete knowledge: the set of objects which a term is interpreted into are intended to represent the *possible values* for that term. Essentially, it is a restricted form of disjunctive knowledge about that term, which we will be calling *fluent-based disjunctive knowledge*.

In this setting, equality gets a different interpretation than the standard one. Based on this, we will be using some useful macros for capturing specific notions regarding fluent-based disjunctive knowledge. To emphasize the difference and to avoid confusion,

we will be using $\approx$ as the equality symbol.

We now move to the formal definitions and some examples that give the intuitions behind them.

3.1 Syntax

An $\mathcal{L}_P$ language $\mathcal{L}$ is a first order language with three disjoint sorts: A for actions, S for situations and O for all the rest of the objects. $\mathcal{L}$ has the following alphabet:

- Countably infinite variable symbols for each sort. We will use $x, y, ...$ for variables of any sort, $a, a_1, ...$ for variables of sort A and $s, s_1, ...$ for variables of sort S.

- Three function symbols of sort S,

  - A constant symbol S_0, denoting the initial situation.

  - A binary function symbol $do : A \times S \mapsto S$, denoting the successor situation function.

  - A binary function symbol $Poss : A \times S \mapsto 2^O$, which is intended to capture that it is possible to perform an action a at situation s and will be forced to map to $\top, \bot$.

- Two constant symbols $\top, \bot$ of sort O, denoting true and false. These will be used to emulate predicates and relational fluents using functional fluent symbols.

- Countably infinite function symbols $(A \cup O)^n \mapsto O$ which are used to denote situation independent functions. We use $h, h_1, h_2...$ to denote these function symbols.

- Countably infinite function symbols $(A \cup O)^n \mapsto A$ which are called *action functions*. We use $A, A_1, A_2, ...$ to denote action function symbols.

- Countably infinite function symbols $(A \cup O)^n \times S \mapsto 2^A$ or $(A \cup O)^n \times S \mapsto 2^O$, which are called *functional fluents*. We use $fl, fl_1, fl_2, ...$ to denote functional fluent symbols.

- the logical symbols $\wedge, \neg, \exists, \approx, (,)$.

All function symbols of arity zero are also called *constant symbols*. Also, we will use $f, f_1, f_2, ...$ to denote any function symbol of the above. Note that there are *no predicate* and *no relational fluent* symbols. Relational fluents can be treated as functional fluents with two possible values, $\top, \bot$ and similarly predicates can be treated as functions. We will see an example of that in section 3.3. Functional fluent symbols will be printed in a different font, like *dice,weather*, while non-fluent function symbols will be printed in normal math font, like $sunny, rainy, 1, 2$.

All terms appearing in formulas are assumed to be of the correct sort, according to the definition of the predicate or function in which they appear. We will use $t, t_1, t_2, ...$ to denote terms, and the vector notation to denote tuples of terms, $\vec{t}, \vec{t_1}, \vec{t_2}, ...$ Also, we will use $\alpha, \beta, ...$ to denote arbitrary formulas.

The formal definitions follow for an $\mathcal{L}_P$ language $\mathcal{L}$.

Definition 3.1.1 ($\mathcal{L}$-term)**.**

- Every variable is an $\mathcal{L}$-term of the same sort.

- If f is an n-ary function symbol of sort $\langle i_1, ..., i_n, i_{n+1} \rangle$ and each t_j is an $\mathcal{L}$-term of sort i_j, then $f(t_1, ..., t_n)$ is an $\mathcal{L}$-term of sort i_{n+1}.

An $\mathcal{L}$-term that includes no variable symbols is a *ground* $\mathcal{L}$-term. An $\mathcal{L}$-term that includes no fluent symbols is a *rigid* $\mathcal{L}$-term. □

Example 3.1.1. $x, f(x), senses(fl_1), t, t_1, dice(1, s), 1, rollDice$ are all $\mathcal{L}$-terms of various sorts. □

Definition 3.1.2 ($\mathcal{L}$-formula)**.**

- If t_1, t_2 are $\mathcal{L}$-terms, then $t_1 \approx t_2$ is an *atomic* $\mathcal{L}$-formula.
- If α and β are $\mathcal{L}$-formulas, then $\neg\alpha, (\alpha \wedge \beta)$ are also $\mathcal{L}$-formulas.
- If x is a variable and α is an $\mathcal{L}$-formula, then $\exists x\alpha$ is a $\mathcal{L}$-formula.

An $\mathcal{L}$-formula that includes no variable symbols is a *ground* $\mathcal{L}$-formula. An $\mathcal{L}$-formula that includes no fluent symbols is a *rigid* $\mathcal{L}$-formula. □

Example 3.1.2. $\mathit{dice}(1, s) \approx 1$, $\mathit{dice}(2, s) \approx 5$, $\mathit{dice}(1, s) \approx \mathit{dice}(2, S_0)$, $succ(1) \approx x$, $succ(\mathit{dice}(1, s)) \approx \mathit{dice}(2, s)$ are atomic $\mathcal{L}$-formulas. □

The logical symbols $\vee, \rightarrow, \leftrightarrow, \equiv$ and the universal quantifier $\forall$ are defined as macros based on $\neg, \wedge, \exists$, in the standard way. Furthermore, we define three macros that capture intuitive notions regarding fluent-based disjunctive knowledge and which will be useful in the next sections. Since their use and effect will become clear after we see the semantics of $\mathcal{L}_P$ languages, we postpone their definition until after the semantic definitions.

Finally, in $\mathcal{L}_P$ syntax we will be using the standard rules for omitting parenthesis. We now move to the semantics of $\mathcal{L}$.

3.2 Semantics

$\mathcal{L}_P$ semantics extends the standard semantics using intuitions from many-valued logics. As usual, a structure for some $\mathcal{L}_P$ language $\mathcal{L}$ has a domain of objects for each of the sorts. The important difference with standard semantics is that the interpretation of *fluent symbols* ranges over the powerset of the object domain, excluding the empty set. Intuitively, this captures the fact that it may not be definite which is the value of a fluent, but it is definite what the options are, even if this is an infinite number of possibilities. Since having no possible value is essentially inconsistent, the empty set is not allowed as

an interpretation of a fluent. The rest of the parameters of the language get the standard interpretation.

In general, the interpretation of an $\mathcal{L}$-term is a set of objects of the correct sort, as the fluent-based disjunctive knowledge propagates to form incomplete knowledge about the terms built upon the fluent symbols. This form of incomplete knowledge is inherently captured *inside the structures* and provides an extra way of talking about disjunctive knowledge. In chapter 5 we will see that we can define action theories which capture only this form of fluent-based disjunctive knowledge and because of that, evaluation can be done efficiently.

Before we move to the formal definitions, we emphasize once more that $\mathcal{L}_P$ semantics is not standard in many aspects: every structure can capture fluent-based disjunctive knowledge by itself and functional fluents are interpreted into non-standard functions to name a few. Also, note that we will follow the same notation as in [End72].

Definition 3.2.1 ($\mathcal{L}$-structure)**.** An $\mathcal{L}$-structure $\mathcal{M}$ consists of the following:

- A non-empty set M_i for each of the sorts of the language, called the universe of $\mathcal{M}$ of sort i. We use $|\mathcal{M}|$ to denote the union of the universes of all sorts. Variables in an $\mathcal{L}$-formula range over the singleton sets of the corresponding M_i and terms get their interpretation from 2^{M_i} for the corresponding i.

- For each n-ary functional fluent symbol fl in $\mathcal{L}$ of sort $\langle i_1, ..., i_n, S\rangle$, an associated function $fl^{\mathcal{M}} : M_{i_1} \times \cdots \times M_{i_n} \mapsto 2^{M_S} - \emptyset$.

- For each n-ary function symbol f in $\mathcal{L}$ of sort $\langle i_1, ..., i_n, i_{n+1}\rangle$ that is not a functional fluent, an associated function $f^{\mathcal{M}}(c_1, ..., c_n) = \{g(c_1, ..., c_n)\}$ for all $c_j \in M_{i_j}$, where $g : M_{i_1} \times \cdots \times M_{i_n} \mapsto M_{i+1}$. □

Before proceeding to the next semantic definitions, we need to see some notions about these non-standard structures. The form of the fluent symbols' interpretation implies a partial ordering among the structures.

Definition 3.2.2 (Order relation on $\mathcal{L}$-structures)**.** Let $\mathcal{M}_1, \mathcal{M}_2$ be $\mathcal{L}$-structures, identical except for the interpretation of functional fluent symbols. $\mathcal{M}_1 \preceq \mathcal{M}_2$ iff for all fluents in $\mathcal{L}$ and for all possible arguments $\vec{c}$, $f^{\mathcal{M}_1}(\vec{c}) \subseteq f^{\mathcal{M}_2}(\vec{c})$.
Note, that the relation $\preceq$ is a partial order on the set of $\mathcal{L}$-structures. □

The structures which have the property that they capture no fluent-based disjunctive knowledge, are called *simple structures*.

Definition 3.2.3 (Simple structure)**.** A *simple structure* is a structure that captures no fluent-disjunctive knowledge. That is, all fluent symbols are interpreted into functions that always return singleton sets: for all fluents in $\mathcal{L}$ and for all possible arguments $\vec{c}$, $|f^{\mathcal{M}}(\vec{c})| = 1$. □

Now we can see the definition of term denotation, which relies on these notions. $\mathcal{L}_P$ semantics provides a consistent way of propagating the fluent-based disjunctive knowledge to form incomplete knowledge about the terms built upon the fluent symbols. In detail, the interpretation of a function-term $f(\vec{t})$ is the set of the objects you get if you apply $f^{\mathcal{M}}$ to all possible tuples $\vec{t}$, produced by the cross-product of the arguments' interpretation, subject to the restriction that *in each tuple, syntactically equal terms are bound to the same interpretation*.

Definition 3.2.4 ($\mathcal{L}$-term denotation in a simple structure)**.** Let $\mathcal{M}$ be a *simple* $\mathcal{L}$-structure and let μ be a variable assignment for $\mathcal{M}$ which assigns an object of the corresponding M_i to every variable. Each $\mathcal{L}$-term t is then assigned a *singleton set* of the corresponding universe M_i. This is defined by induction on terms t, as follows:

- For any variable x, $x^{\mathcal{M}}[\mu]$ is $\{\mu(x)\}$
- For any n-ary function symbol f and $\mathcal{L}$-terms $t_1, ..., t_n$,

$$f(t_1, ..., t_n)^{\mathcal{M}}[\mu] = f^{\mathcal{M}}(c_1, ..., c_n)$$

  where $t_i^{\mathcal{M}}[\mu] = \{c_i\}$. □

Definition 3.2.5 ($\mathcal{L}$-term denotation)**.** Let $\mathcal{M}$ be an $\mathcal{L}$-structure and let μ be a variable assignment for $\mathcal{M}$ which assigns an object of the corresponding M_i to every variable. Each $\mathcal{L}$-term t is then assigned a *non-empty subset* of the corresponding universe M_i. Formally, $t^{\mathcal{M}}[\mu]$ ranges over the powerset of M_i. This is defined on the partial ordering of structures, as follows:

$$t^{\mathcal{M}}[\mu] = \bigcup t^{\mathcal{M}'}[\mu] \text{ for all simple } \mathcal{M}' \text{ such that } \mathcal{M}' \preceq \mathcal{M}$$

□

So, in this setting, fluents range over the powerset of the object domain rather than the domain itself. As mentioned earlier, this is exactly how fluent-based disjunctive knowledge is captured. We explore this a little more in the following example.

We consider a fluent $\mathit{dice}(t, s)$ which expresses which face of the dice t is facing upwards at situation s. It is reasonable to expect $\mathit{dice}(1, s)^{\mathcal{M}}[\mu]$ to be $\{1, 2, 3, 4, 5, 6\}$, if dice 1 is a six-sided dice or $\mathit{dice}(2, s)^{\mathcal{M}}[\mu]$ to be $\{1, 2, 3, 4\}$, if dice 2 is a four-sided dice. On the other hand, when we consider the non-fluent function symbol that is intended to express the successor function, $succ(t)$, it does not seem right to have $succ(1)^{\mathcal{M}}[\mu]$ to be anything more than $\{2\}$, or at least some *singleton* set. That is the reason why functional fluents are interpreted into functions which range over the powerset of M_i, while the other function symbols are interpreted in the standard way as functions which range over M_i, with the particularity that they actually return the singleton set of the object, so that to be compatible with the set machinery.

According to the term denotation definition, any term that contains some fluent can capture fluent-based disjunctive knowledge, as it propagates at term composition. For instance in the dice example, it seems quite reasonable to get that $succ(\mathit{dice}(2, s))^{\mathcal{M}}[\mu]$ is $\{2, 3, 4, 5\}$ and this is what the formalism would actually do if $succ^{\mathcal{M}}$ is interpreted as the successor function.

The observation to be made here is that $\mathcal{L}$-structures can capture fluent-based dis-

junctive knowledge inherently and functional fluents are the basic elements which make that possible.

Note also that non-simple structures need special treatment in the way the complex terms' interpretation is treated and the fluent-based disjunctive knowledge is propagated. This is because the recursive rule that works for the simple structures would lead to counter-intuitive behavior, when syntactically equivalent terms appear as arguments of some complex term.

For example, consider the term

$$sum(\mathsf{dice}(2,s),\mathsf{dice}(2,s))^{\mathcal{M}}[\mu]$$

If $sum^{\mathcal{M}}$ is interpreted as the sum function, then according to the term denotation definitions 3.2.4 and 3.2.5, the interpretation of the sum term is the intuitive one

$$sum(\mathsf{dice}(2,s),\mathsf{dice}(2,s))^{\mathcal{M}}[\mu] = \{2,4,6,8\}$$

On the other hand, the (FOL) recursive rule for term interpretation would give

$$sum(\mathsf{dice}(2,s),\mathsf{dice}(2,s))^{\mathcal{M}}[\mu] = \{2,3,4,5,6,7,8\}$$

It is easy to see that some of the values come from the application of $sum^{\mathcal{M}}$ to *different possible values* of $\mathsf{dice}(2,s)^{\mathcal{M}}[\mu]$. This is counter-intuitive, as we expect the interpretation of each term to rely only on *one possible value for each distinct sub-term that appears in it.* In this example, we want the interpretation of the sum term *not to consider different possible values for the two appearances of the term* $\mathsf{dice}(2,s)^{\mathcal{M}}[\mu]$. We now proceed to the definition of truth for $\mathcal{L}$-formulas.

Definition 3.2.6 (Truth in a structure)**.** For an $\mathcal{L}$-formula α, an $\mathcal{L}$-structure $\mathcal{M}$ and a variable assignment μ, the notion of truth in the structure $\mathcal{M} \models \alpha[\mu]$ is defined by structural induction on formulas α as follows:

- For $\mathcal{L}$-terms t_1, t_2, $\mathcal{M} \models t_1 \approx t_2[\mu]$ iff t_1, t_2 are of the same sort and there exists a simple $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$ and $t_1^{\mathcal{M}'}[\mu] = t_2^{\mathcal{M}'}[\mu]$. Intuitively, this means that *it is possible* that the two terms have the same value.

- For an $\mathcal{L}$-formula α, $\mathcal{M} \models \neg\alpha[\mu]$ iff $\mathcal{M} \not\models \alpha[\mu]$

- For $\mathcal{L}$-formulas α, β, $\mathcal{M} \models \alpha \wedge \beta[\mu]$ iff $\mathcal{M} \models \alpha[\mu]$ and $\mathcal{M} \models \beta[\mu]$

- For an $\mathcal{L}$-formula α and a variable x of sort i,

$$\mathcal{M} \models \exists x \alpha(x)[\mu] \text{ iff } \mathcal{M} \models \alpha(x)[\mu(c/x)]$$

  for some object $c \in M_i$. □

From the definition of truth of atomic $\mathcal{L}$-formulas we see that logical connective $\approx$ gets a very different interpretation than true equality on sets, which is nevertheless appropriate for expressing what we need in respect to this setting of incomplete knowledge. Here, equality should be interpreted as capturing that two $\mathcal{L}$-terms are *possibly equal*. In some $\mathcal{L}$-structure $\mathcal{M}$, which inherently captures fluent-based disjunctive knowledge, that is true when there exists some simple structure $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$ and interprets the terms to be the same singleton set.

This way of definition, that relies on the notion of simple structure, is necessary for a similar reason that we used it in the term denotation definition: It is necessary that syntactically equal sub-terms are bound to the same interpretation, so that no counter-intuitive results occur. Continuing the dice example, when considering

$$\mathcal{M} \models prec(\mathsf{dice}(2, s)) \approx succ(\mathsf{dice}(2, s))[\mu] \tag{3.1}$$

we want *both instances* of $\mathsf{dice}(2, s)^{\mathcal{M}}[\mu]$ to be bound to *a single possible value* when considering all possibilities for the values of $prec(\mathsf{dice}(2, s))^{\mathcal{M}}[\mu]$ and $succ(\mathsf{dice}(2, s))^{\mathcal{M}}[\mu]$. In this example, by considering the simple structures we prevent that (3.1) gets true, as a result of

$$2 \in succ(\mathsf{dice}(2, s))^{\mathcal{M}}[\mu] \text{ and } 2 \in prec(\mathsf{dice}(2, s))^{\mathcal{M}}[\mu]$$

Also, note that this interpretation of equality is enough for expressing all we may need regarding incomplete knowledge. We will see that in detail in the next section, where we define some useful macros for dealing with fluent-based disjunctive knowledge and we discuss their effect in some examples.

We now move to the last semantic definitions.

Definition 3.2.7 (Satisfiability, validity and entailment $\mathcal{L}$)**.** *Satisfiability, validity and entailment* in $\mathcal{L}$ are defined in the same way as in first order logic. Let α, γ be $\mathcal{L}$-sentences, $\mathcal{M}$ an $\mathcal{L}$-structure and μ a variable assignment.

- γ is satisfiable iff $\mathcal{M} \models \gamma[\mu]$ for some $\mathcal{M}$ and μ.
- γ is valid iff $\mathcal{M} \models \gamma[\mu]$ for all $\mathcal{M}$ and μ.
- $\gamma \models \alpha$ iff for all $\mathcal{M}$ and μ, if $\mathcal{M} \models \gamma[\mu]$ then also $\mathcal{M} \models \alpha[\mu]$.

The same definitions hold if γ is a set of formulas. □

We are only interested in the truth of *sentences*, that is formulas with no free occurrences of variables and we will not consider variable assignment μ in general. When the variable assignment μ is absent, it means that it could actually be anything, without affecting the results we consider.

Closing this section, it is important to note that $\mathcal{L}_P$ also accounts for general disjunctive knowledge expressed by the logical connective $\vee$. As we will see, $\mathcal{L}_P$ provides nice functionality to express exactly the kind of incomplete knowledge that it is desired. In the next section we will see among others that there are $\mathcal{L}_P$ properties which allow us to do that easily. In chapter 5, we will be using these to define theories that capture *only* fluent-based disjunctive knowledge.

3.3 Properties of $\mathcal{L}_P$

In this section we discuss the behavior of $\mathcal{L}_P$ languages. First, we focus on the non-standard interpretation of equality in $\mathcal{L}_P$ languages and present some useful macros which utilize it. Then, we proceed to showing some useful lemmas about $\mathcal{L}_P$ semantics.

3.3.1 $\mathcal{L}_P$ macros and expressiveness of $\mathcal{L}_P$

Here, we see three useful macros that we will be using in the next chapters. These macros are based on the interpretation of equality in $\mathcal{L}_P$ as *possibly equals* and capture notions which resemble notions expressed in situation calculus theories of action and knowledge. We discuss this non-standard interpretation of equality and the effect of the macros in a few illustrative examples.

The t-formulas intend to capture all the disjunctive knowledge about the value of a term t.

Definition 3.3.1 (t-formula macros)**.** If $t, t_1, ..., t_n$ are $\mathcal{L}$-terms then

$$t = [t_1 ..., t_n]$$

is called a *strong* t-formula and stands for the $\mathcal{L}$-formula [1]

$$\forall x\ t \approx x \equiv x \approx t_1 \vee ... \vee x \approx t_n$$

Also, we will find useful a weaker notion of capturing the disjunctive knowledge about the value of an $\mathcal{L}$-term t.

$$t \sqsubseteq [t_1 ..., t_n]$$

is called a *weak* t-formula and stands for the $\mathcal{L}$-formula

$$\forall x\ t \approx x \rightarrow x \approx t_1 \vee ... \vee x \approx t_n$$

[1] Here, we do not expand the macros for $\vee, \equiv, \forall$ for the sake of readability. The actual $\mathcal{L}$-sentence is the one we get after replacing all macros according to their definition. The same remarks applies to the definition of the other macros.

When it is not specified if we are talking about a strong or weak t-formula, we mean a strong t-formula by default. □

Example 3.3.1. $\mathit{dice}(1, s) = [1, 2, 3, 4]$ is a $\mathit{dice}(1, s)$-formula, $\mathit{weather}(s) = [sunny, rainy]$ is a $\mathit{weather}(s)$-formula and $fl(\vec{t}) = [t_1, ..., t_n]$ is a $fl(\vec{t})$-formula. □

The K macro is intended to capture the fact that there is no incomplete knowledge about the value of an $\mathcal{L}$-term t.

Definition 3.3.2 (K macro)**.** For an $\mathcal{L}$-term t, $\mathsf{K}t$ stands for the $\mathcal{L}$-formula

$$\exists x\ t = [x]$$

□

A strong t-formula is true when the uncertainty captured by term t is *exactly* the uncertainty defined by the interpretation of terms t_i appearing in brackets and note that the terms t_i need not be rigid. The strong t-formula macro can be intuitively seen as checking or assigning a specific *disjunctive knowledge "value"* to a term t. The list in the t-formula can be intuitively seen as that value.

The weak t-formula has similar effect, it is true when the uncertainty captured by term t is *at most as much as* the uncertainty defined by the interpretation of terms t_i. The notion that a term has a definite value and there is no incomplete knowledge about its value is captured by the K macro. $\mathsf{K}t$ is true in a structure when t is interpreted into some singleton set.

We first discuss the behavior of equality and the macros we defined in regard to truth in a structure[2].

Example 3.3.2 (Satisfaction)**.** Consider an $\mathcal{L}_P$ language $\mathcal{L}$ which includes the symbols $1, 2, 3, S_0, fl$ of the intuitive sort and arity. Let $\mathcal{M}$ be an $\mathcal{L}$-structure such that

$$fl(S_0)^{\mathcal{M}}[\mu] = \{1, 2\}, \quad 1^{\mathcal{M}}[\mu] = \{1\}, \quad 2^{\mathcal{M}}[\mu] = \{2\}, \quad 3^{\mathcal{M}}[\mu] = \{3\}$$

[2]Here, we abuse notation a little by using the number symbols both as syntactic and as semantic objects. The context is always enough to distinguish which notion is the intended one

This means that there is incomplete knowledge about the value of $fl(S_0)$, it could be 1 or it could be 2.

First of all, let us see in this simple setting that the logical symbol of equality expresses that *it is possible that terms have the same value*. It is easy to verify that according to the definition of truth in a structure 3.2.6,

$$\mathcal{M} \models fl(S_0) \approx 1[\mu] \text{ and } \mathcal{M} \models fl(S_0) \approx 2[\mu] \text{ and } \mathcal{M} \models fl(S_0) \not\approx 3[\mu]$$

Note that since equality expresses that terms are *possibly* equal, we can use it in formulas in ways which do not seem intuitive at first glance. The previous equation actually implies that

$$\mathcal{M} \models fl(S_0) \approx 1 \wedge fl(S_0) \approx 2[\mu]$$

On the other hand, the strong $fl(S_0)$-formula macro expresses that the term on the left hand side has exactly the disjunctive knowledge expressed in the right hand side. According to the definition of truth in a structure 3.2.6 and the definition of the macro 3.3.2,

$$\mathcal{M} \not\models fl(S_0) = [1][\mu] \text{ and } \mathcal{M} \not\models fl(S_0) = [2][\mu]$$

which should be interpreted as "it is not true that $fl(S_0)$ has the value 1 (respectively 2)".

According to the same definitions,

$$\mathcal{M} \models fl(S_0) = [1,2][\mu] \text{ and } \mathcal{M} \not\models fl(S_0) = [1,2,3][\mu]$$

which should be interpreted as "it is true that the value of $fl(S_0)$ is exactly either 1 or 2 (respectively not true that it is exactly 1 or 2 or 3)".

One can view this as if the interpretations of the term $fl(S_0)$ and the list encapsulate the disjunctive knowledge into a "*disjunctive knowledge value*" in the intuitive way. According to this view, the previous reasoning is as follows:

If it is true that $fl(S_0)$ has the disjunctive knowledge value $[1,2]$ then it is not true that it has the disjunctive knowledge value $[1]$, or $[1,2,3]$.

In a similar way, the weak version of the $fl(S_0)$-formula has the intuitive behavior:

$$\mathcal{M} \not\models fl(S_0) \sqsubseteq [1][\mu],\ \mathcal{M} \models fl(S_0) \sqsubseteq [1,2][\mu] \text{ and } \mathcal{M} \models fl(S_0) \sqsubseteq [1,2,3][\mu]$$

which adopting the previously described point of view is as follows:

If it is true that $fl(S_0)$ has the disjunctive knowledge value $[1,2]$ then it is not true that it's disjunctive knowledge value is *less than* $[1]$.

If it is true that $fl(S_0)$ has the disjunctive knowledge value $[1,2]$ then it is also true that it's disjunctive knowledge value is *less than* $[1,2]$ or $[1,2,3]$.

Also, it is interesting to see how the notion of *having the same value with no uncertainty* can be captured in $\mathcal{L}$. First we see the effect of K in the truth of the terms. According to the definition of truth in a structure 3.2.6 and the definition of the K macro 3.3.2,

$$\mathcal{M} \not\models \mathsf{K}fl(S_0)[\mu] \quad \text{but } \mathcal{M} \models \mathsf{K}1[\mu], \quad \mathcal{M} \models \mathsf{K}2[\mu], \quad \mathcal{M} \models \mathsf{K}3[\mu]$$

Formula $t_1 \approx t_2 \wedge \mathsf{K}t_1 \wedge \mathsf{K}t_2$ expresses that t_1 has the same value with t_2 with no uncertainty. The next formula expresses that notion for terms $fl(S_0)$ and 1, which is not true in $\langle \mathcal{M}, \mu \rangle$.

$$\mathcal{M} \not\models fl(S_0) \approx 1 \wedge \mathsf{K}fl(S_0) \wedge \mathsf{K}1[\mu]$$

Note that $t_1 \approx t_2 \wedge \mathsf{K}t_1 \wedge \mathsf{K}t_2$ is logical equivalent to $t_1 = [t_2] \wedge \mathsf{K}t_2$ and $t_2 = [t_1] \wedge \mathsf{K}t_1$. $\square$

In this example we focused on evaluating various formulas in a specific interpretation. We now make an important remark on the effect of strong fl-formulas when included in $\mathcal{L}_P$ theories.

For any fluent symbol fl, strong fl-formulas are of special interest, as they directly restrict the interpretation of fl and thus the fluent-based disjunctive knowledge of the

theories they are part of. In the same way as in first order logic a sentence $f = 1$ forces the interpretation of the function symbol f, $f^{\mathcal{M}}$ to be $1^{\mathcal{M}}$ in all of its models $\mathcal{M}$, here a strong $fl(S_0)$-formula

$$fl(S_0) = [1, 2]$$

forces the interpretation $fl^{\mathcal{M}}(c_{S_0})$ to be $1^{\mathcal{M}} \cup 2^{\mathcal{M}}$, where $S_0^{\mathcal{M}} = \{c_{S_0}\}$. This effect can be generalized so that an axiom can define $fl^{\mathcal{M}}$ entirely. This strong fl-formula functionality will be used in chapter 5 to define theories which restrict the disjunctive knowledge of the initial situation to be fluent-based.

Until now, we focused on expressing facts about incomplete knowledge regarding one fluent only. We now briefly discuss the expressiveness of $\mathcal{L}_P$ regarding general disjunctive knowledge.

Example 3.3.3 (Disjunctive knowledge)**.** $\mathcal{L}_P$ theories can capture disjunctive knowledge about different fluents in many ways. A sentence of the following form

$$fl_1(S_0) \approx 1 \vee fl_2(S_0) \approx 1$$

expresses a *weak form* of disjunctive knowledge (*It is possible that* $fl_1(S_0)$ has the value 1, or *it is possible that* $fl_2(S_0)$ has the value 1). We can also express in $\mathcal{L}_P$ what in normal situation calculus would be

$$fl_1(S_0) = 1 \vee fl_2(S_0) = 1$$

by the $\mathcal{L}$-sentence

$$fl_1(S_0) = [1] \vee fl_2(S_0) = [1]$$

The fact that in $\mathcal{L}_P$ we deal with the possible values of the terms, proves to be very important in cases where the disjunctive knowledge is propagated through constraints. Consider the simple situation calculus theory D_{SC}, which consists of the two following sentences.

$$fl_1(S_0) = 1 \vee fl_1(S_0) = 2 \qquad fl_2(S_0) = fl_1(S_0)$$

This has the effect that there are two classes of models which satisfy D_{SC}, one such that both $fl_1(S_0)$ and $fl_2(S_0)$ have the value 1 and one such that both fluents have the value 2. Note that the theory captures disjunctive knowledge which is not fluent-based. Intuitively, this is so, because there are no sets of possible values for each fluent that can have the same effect, or equivalently this effect cannot be captured inside a single (non-simple) $\mathcal{L}_P$ structure.

This effect can be expressed in $\mathcal{L}_P$ by the following theory $\mathcal{D}_1$.

$$fl_1(S_0) = [1] \vee fl_1(S_0) = [2] \qquad \forall x \forall y \; fl_2(S_0) \approx x \wedge fl_1(S_0) \approx y \rightarrow x \approx y$$

The theory is not as trivial as D_{SC}, because we need to talk about the possible values of the two fluents. This illustrates nicely, though, the connection between the two fluents which according to our intuition, correctly results to disjunctive knowledge that is not fluent-based. Similar to $\mathcal{D}_{SC}$, there are two classes of models which satisfy D_1, one such that both $fl_1(S_0)$ and $fl_2(S_0)$ have the value 1 and one such that both fluents have the value 2.

Nevertheless, we can force the disjunctive knowledge be fluent-based by losing some information. This can be done if, instead of posing the constraint that the fluents have the *same value*, we pose the constraint that they have the *same possible values*. This effect can be expressed in $\mathcal{L}_P$ by the following theory $\mathcal{D}_2$.

$$fl_1(S_0) = [1, 2] \qquad \forall x \; fl_2(S_0) \approx x \equiv fl_1(S_0) \approx x$$

Clearly, theory $\mathcal{D}_2$ does not express the same thing as $\mathcal{D}$, but captures fluent-based disjunctive knowledge, losing the least possible information: fluent fl_1 has two possible values 1 and 2 and fl_2 has the same possible values. In this way, the fluents' values are independent and D_2 is satisfied by a class of non-simple models such that both fluents are interpreted to have the possible values 1 and 2. Such a model summarizes four classes of simple models, one for each combination of values 1 and 2 for the two fluents.

This is important because we intend to deal with theories which capture fluent-based disjunctive knowledge only, so that fluents are independent and theorem proving can be done efficiently. Note that in general, this effect cannot be expressed in situation calculus. We can express that two terms have the same possible values only in a constructive manner when we can identify the possible values. □

The last two examples illustrate the intuition behind the design of $\mathcal{L}_P$ semantics. Even if it allows for general disjunctive knowledge that is *implied* by the *many models* satisfying the theory in question, there is functionality which can force all incomplete knowledge to be fluent-based disjunctive knowledge, summarized and captured *inside the models themselves*. Example 3.3.2 gives the intuition on how to restrict the initial situation to capture fluent-based disjunctive knowledge only and 3.3.3 gives insight on how we can define the successor state axioms so that this persists in all situations. We postpone the detailed discussion of this until next chapter and proceed to some more remarks about the expressiveness of $\mathcal{L}_P$.

Even in this very simple example with $fl(S_0)$, we can see that there are many interesting and useful notions which can be expressed in $\mathcal{L}_P$, some of which cannot be captured by the situation calculus and normal first order logic. The interpretation of the equality symbol as *possibly equals* provides ways to capture notions which can be expressed only in a modal logic setting of knowledge. One can view $\mathcal{L}_P$ theories as if they represent the epistemic state of some agent and that the theorems entailed are implicity referring to what the agent *knows*. In what follows, we will be talking about what is known, even if there is no explicit **Knows** element in the language.

Example 3.3.4 (Epistemic)**.** The strong fl-formula

$$fl(S_0) = [1, 2]$$

can be viewed as expressing that *it is known* that the value of $fl(S_0)$ is *either one* of the options but *it is not known which one*. Notice though, that if a theory entails this

fl-formula, it cannot entail $\mathsf{K}fl(S_0) \wedge fl(S_0) \approx 1$ or $\mathsf{K}fl(S_0) \wedge fl(S_0) \approx 2$ because that would be inconsistent. This is because $fl(S_0) = [1, 2]$ is a strong $fl(S_0)$-formula which uses $\equiv$ to define the fluent-based disjunctive knowledge about fl. On the other hand, the weak $fl(S_0)$-formula

$$fl(S_0) \sqsubseteq [1, 2]$$

captures that it is known that the value of $fl(S_0)$ is either one of the options and in general it is not known which one is the actual value (i.e. it also allows for the possibility that we can actually prove that one of them is the value of fl). That is exactly what is expressed by the formula

$$\mathbf{Knows}(fl = 1 \vee fl = 2, s)$$

in a situation calculus theory of action and knowledge that uses the epistemic fluent **Knows** [SL03].

This is something that by itself is not very important, since even in normal situation calculus, we can view disjunctions as implicitly being the argument of an appropriate **Knows** formula. It is important though, if we can also express that it is either known that $fl(S_0)$ has the value or it is known that it has the value 2, which in a situation calculus theory of action and knowledge is captured by the following sentence.

$$\mathbf{Knows}(fl = 1, s) \vee \mathbf{Knows}(fl = 2, s)$$

Indeed, this can be expressed in $\mathcal{L}_P$ as

$$fl = [1] \vee fl = [2]$$

or using the K macro

$$\mathsf{K}fl(S_0) \wedge (fl(S_0) \approx 1 \vee fl(S_0) \approx 2)$$

In fact, the notation of K macro was chosen due to this relation it has with knowledge. Note, that in a similar way that $\mathbf{Knows}(fl = 1 \vee fl = 2, s)$ does not imply $\mathbf{Knows}(fl = 1, s) \vee \mathbf{Knows}(fl = 2, s)$, $fl(S_0) \sqsubseteq [1, 2]$ does not imply $fl = [1] \vee fl = [2]$[3].

[3]Consider a non-simple model $\mathcal{M}$ such that $fl^{\mathcal{M}} = \{1, 2\}$

It is out of the scope of the work presented here, but it is not difficult to see that the correlation of $\mathcal{L}_P$ theories and situation calculus theories with or without the epistemic fluent **Knows** can be made precise. □

We now illustrate some issues regarding predicates and $\mathcal{L}_P$ behavior. First, we see a simple example on how predicates can be modeled in some $\mathcal{L}_P$ theory.

Example 3.3.5 (Predicate emulation, *dice*)**.** Suppose that a predicate $IsDice(x)$ is needed, which classifies which objects of the domain O are dice. We can express this in $\mathcal{L}_P$ using a function $isdice(x)$ and constants $\top, \bot$. Our theory should include a uniqueness of names axiom for $\top, \bot$ and we have to use atomic formulas of the form $isdice(x) \approx \top$ and $isdice(x) \approx \bot$ instead of $IsDice(x)$ and $\neg IsDice(x)$. Furthermore, if $isdice(x)$ is forced to have as a value exactly one of $\top$ or $\bot$ for each object in the domain, then we need only use $isdice(x) \approx \top$ in formulas other than the one(s) defining it. An axiom that defines $isdice(x)$ in this manner is the following.

$$\forall x\ isdice(x) \approx y \equiv (x \approx 1 \vee x \approx 2 \vee x \approx 3) \wedge y \approx \top \vee \neg(x \approx 1 \vee x \approx 2 \vee x \approx 3) \wedge y \approx \bot$$

In some appropriate theory which includes this axiom, the following formula expresses that 1 is a possible value for all dice in the domain in the initial situation.

$$\forall x\ isdice(x) \approx \top \rightarrow \mathit{dice}(x, S_0) \approx 1$$

□

Things get a little complicated when functions that intend to capture predicates (like $isdice(x)$) mention fluents as arguments. Then, we get results which do not seem intuitive, but are nevertheless correct according to the interpretation of equality in $\mathcal{L}_P$.

Example 3.3.6 (Predicate emulation, *odd*)**.** Suppose that we have a theory that includes a definition for the function $odd(x)$ which is intended to express the functionality of predicate $Odd(x)$. If the theory also includes the formula

$$\mathit{dice}(2, S_0) = [1, 2, 3, 4]$$

then, it is easy to verify that this theory entails

$$odd(\mathsf{dice}(2, S_0)) \approx \top$$

That is because it is actually *possible* that the value of $\mathsf{dice}(2, S_0)$ is an odd number. We can also express that it is *definite* that the value of $\mathsf{dice}(2, S_0)$ is an odd number, by the formula

$$\forall x \; x \approx \mathsf{dice}(2, S_0) \rightarrow odd(x) \approx \top$$

or

$$odd(\mathsf{dice}(2, S_0)) = [\top]$$

which are not entailed by the theory. □

3.3.2 Lemmas about $\mathcal{L}_P$ semantics

In this section, we present some lemmas about the semantics of $\mathcal{L}_P$ which will prove useful in the proofs we will be doing on the following chapters.

First, we see some easy lemmas regarding term interpretation. These are included for clarity, since the basic semantic definitions rely on the notions of *simple* and *non-simple structure* and this makes things not obvious, even for simple facts.

Lemma 3.3.1 (Variable interpretation in non-simple structures)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$. For any variable x and any variable assignment μ, $x^{\mathcal{M}}[\mu] = \{\mu(x)\}$.*

Proof: By the definition of term denotation 3.2.5,

$$x^{\mathcal{M}}[\mu] = \bigcup x^{\mathcal{M}'}[\mu]$$

for all simple structures $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$. By the definition of term denotation in simple structures 3.2.4, that is true iff

$$x^{\mathcal{M}}[\mu] = \bigcup \{\mu(x)\}$$

for all simple structures $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$. Therefore, $x^{\mathcal{M}}[\mu] = \{\mu(x)\}$. □

Lemma 3.3.2 (Term interpretation in simple structures)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and a simple $\mathcal{L}$-structure $\mathcal{M}$. For any $\mathcal{L}$-term t and any variable assignment μ, $t^{\mathcal{M}}[\mu]$ is a singleton set.*

Proof: By induction on the construction of $\mathcal{L}$-terms. □

Lemma 3.3.3 (Term interpretation in non-simple structures)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$. For any $\mathcal{L}$-term t and any variable assignment μ, $c \in t^{\mathcal{M}}[\mu]$ iff there exists a simple structure $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$ and $t^{\mathcal{M}'}[\mu] = \{c\}$.*

Proof: Directly from the definition of term denotation in simple and non-simple structures, 3.2.4,3.2.5. □

Lemma 3.3.4 (Term interpretation is a non empty set)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$. For any $\mathcal{L}$-term t and any variable assignment μ, $t^{\mathcal{M}}[\mu]$ is a non-empty set.*

Proof: If $\mathcal{M}$ is a simple structure, then the lemma follows by lemma 3.3.2. If $\mathcal{M}$ is a non-simple structure, then by the definition of term denotation 3.2.5

$$t^{\mathcal{M}}[\mu] = \bigcup t^{\mathcal{M}'}[\mu] \text{ for all simple } \mathcal{M}' \text{ such that } \mathcal{M}' \preceq \mathcal{M}$$

It easy to see that there exists at least one such structure $\mathcal{M}'$, so by lemma 3.3.2, the lemma follows. □

Lemma 3.3.5 (Term interpretation and atomic formulas)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$. For any $\mathcal{L}$-term t, variable x not appearing in t, object c of the corresponding domain and variable assignment μ,*

$$\mathcal{M} \models x \approx t[\mu(c/x)] \text{ iff } c \in t^{\mathcal{M}}[\mu]$$

Proof: By lemma 3.3.3, $c \in t^{\mathcal{M}}[\mu]$ iff there exists a simple structure $\mathcal{M}'$ such that

$$\mathcal{M}' \preceq \mathcal{M} \text{ and } t^{\mathcal{M}'}[\mu] = \{c\}$$

That holds iff there exists a simple structure $\mathcal{M}'$ such that

$$\mathcal{M}' \preceq \mathcal{M} \text{ and } x^{\mathcal{M}'}[\mu(c/x)] = t^{\mathcal{M}'}[\mu(c/x)]$$

By the definition of truth in a structure 3.2.6, that holds iff $\mathcal{M} \models x \approx t[\mu(c/x)]$. □

Corrolary 3.3.6 (For every term there exists a possible value)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$. For any $\mathcal{L}$-term t, any variable x not appearing in t and any variable assignment μ,*

$$\mathcal{M} \models x \approx t[\mu(c/x)]$$

for some c in the corresponding domain.

Proof: By lemmas 3.3.4 and 3.3.5. □

Before we move to the next less obvious lemmas, we clarify a notation issue. In general, when W is an $\mathcal{L}$-formula and t, t' are $\mathcal{L}$-terms, then $W|^t_{t'}$ denotes the formula obtained by W by replacing all occurrences of t by t'. This can be viewed as if W is a function of t, $W(t)$ and $W|^t_{t'} = W(t')$. We choose to use the $W|^t_{t'}$ notation because $W(t)$ can be ambiguous with the syntax of $\mathcal{L}_P$.

It is easy to see that in some simple structure, the (singleton) interpretation of a term t is the same with the interpretation of $t|^{t'}_y$ when y has the same interpretation with t'. This is a straightforward fact which is true in first order logic and therefore in simple $\mathcal{L}_P$-structures which have the same behavior. In the next lemma, we extend and prove this notion for any $\mathcal{L}$ structure.

Lemma 3.3.7 (Term substitution in terms)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any $\mathcal{L}$-terms t, t', $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$t^{\mathcal{M}}[\mu] = \bigcup t|^{t'^{\mathcal{M}}}_y[\mu(c/y)], \text{ for all } c \in t'^{\mathcal{M}}[\mu]$$

where y does not appear in t, t'.

Proof: By induction on the construction of $\mathcal{L}$-term t.

- **Base case:** t is a variable x. If t' is the same variable x, the lemma is true iff

$$x^{\mathcal{M}}[\mu] = \bigcup y^{\mathcal{M}}[\mu(c/y)], \qquad c \in x^{\mathcal{M}}[\mu]$$

  Otherwise, the lemma is true iff

$$x^{\mathcal{M}}[\mu] = \bigcup x^{\mathcal{M}}[\mu(c/y)], \qquad c \in t'^{\mathcal{M}}[\mu]$$

  By lemma 3.3.1, it is easy to see that in both cases, the lemma holds.

- **Induction hypothesis:** Assume that the lemma holds for the $\mathcal{L}$-terms $t_1, ..., t_n$. That is, for any $\mathcal{L}$-term t', $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,

$$t_i^{\mathcal{M}}[\mu] = \bigcup t_i|_y^{t'^{\mathcal{M}}}[\mu(c/y)], \text{ for all } c \in t'^{\mathcal{M}}[\mu]$$

  and y does not appear in $t_1, ..., t_n, t'$.

- **Induction Step:** We prove the lemma for the $\mathcal{L}$-term $f(t_1, ..., t_n)$, where f is an n-ary functional fluent of language $\mathcal{L}$. We want to prove that

$$f(t_1, ..., t_n)^{\mathcal{M}}[\mu] = \bigcup f(t_1, ..., t_n)|_y^{t'^{\mathcal{M}}}[\mu(c/y)], \text{ for all } c \in t'^{\mathcal{M}}[\mu]$$

  If t' is $f(t_1, ..., t_n)$, then the lemma is true iff

$$f(t_1, ..., t_n)^{\mathcal{M}}[\mu] = \bigcup y^{\mathcal{M}}[\mu(c/y)], \qquad c \in f(t_1, ..., t_n)^{\mathcal{M}}[\mu]$$

  By lemma 3.3.1, it is easy to see that the lemma holds.

  Now we consider the case where t' is not $f(t_1, ..., t_n)$. We start by the right hand side of the equality.

$$RHS = \bigcup f(t_1, ..., t_n)|_y^{t'^{\mathcal{M}}}[\mu(c/y)], \text{ for all } c \in t'^{\mathcal{M}}[\mu]$$

By lemma 3.3.3,

$$RHS = \bigcup f(t_1, ..., t_n)|_y^{t'\mathcal{M}}[\mu(c/y)], \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

where

$$t'^{\mathcal{M}'}[\mu] = \{c\}$$

By the definition of term denotation 3.2.5,

$$RHS = \bigcup f(t_1, ..., t_n)|_y^{t'\mathcal{M}'}[\mu(c/y)], \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

where

$$t'^{\mathcal{M}'}[\mu] = \{c\}$$

(Here, we omit the second union over all simple structures $\mathcal{M}' : \mathcal{M}' \preceq \mathcal{M}$, since it does not have any additional effect.) Now, since t' is not $f(t_1, ..., t_n)$,

$$RHS = \bigcup f(t_1|_y^{t'}, ..., t_n|_y^{t'})^{\mathcal{M}'}[\mu(c/y)], \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

where

$$t'^{\mathcal{M}'}[\mu] = \{c\}$$

By the definition of term denotation in simple structures 3.2.4,

$$RHS = \bigcup f^{\mathcal{M}'}(c_1, ..., c_n), \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

where

$$t_i|_y^{t'\mathcal{M}'}[\mu(c/y)] = \{c_i\}, \quad t'^{\mathcal{M}'}[\mu] = \{c\}$$

Now, by induction hypothesis,

$$t_i^{\mathcal{M}'}[\mu] = \bigcup t_i|_y^{t'\mathcal{M}'}[\mu(c/y)], \text{ for all } c \in t'^{\mathcal{M}'}[\mu]$$

and since $\mathcal{M}'$ is a simple structure, the induction hypothesis becomes,

$$t_i^{\mathcal{M}'}[\mu] = t_i|_y^{t'\mathcal{M}'}[\mu(c/y)], \text{ for any } c \text{ such that } t'^{\mathcal{M}'}[\mu] = \{c\}$$

So,

$$RHS = \bigcup f^{\mathcal{M}'}(c_1, ..., c_n), \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

where

$$t_i^{\mathcal{M}'}[\mu] = \{c_i\}$$

By the definition of term denotation in simple structures again, 3.2.4,

$$RHS = \bigcup f(t_1, ..., t_n)^{\mathcal{M}'}[\mu], \text{ for all simple } \mathcal{M}', \mathcal{M}' \preceq \mathcal{M}$$

By the definition of term denotation 3.2.5,

$$RHS = f(t_1, ..., t_n)^{\mathcal{M}}[\mu]$$

So, we have proved that the lemma holds for $f(t_1, ...t_n)$.

Thus, induction follows and the lemma holds for all terms t. □

Corrolary 3.3.8 (Term substitution in terms, simple)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any $\mathcal{L}$-terms t, t', simple $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$t^{\mathcal{M}}[\mu] = t|_y^{t'^{\mathcal{M}}}[\mu(c/y)], \text{ where } t'^{\mathcal{M}}[\mu] = \{c\}$$

and y does not appear in t, t'. □

Note that we can view these results as substituting y and not t' in a term t. This is easy to verify if we think the term t as a function of t', as stated earlier. The above corollary can be viewed also in the following way.

Corrolary 3.3.9 (Term substitution in terms, simple)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any $\mathcal{L}$-terms t, t', simple $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$t|_{t'}^{y\,\mathcal{M}}[\mu] = t^{\mathcal{M}}[\mu(c/y)], \text{ where } t'^{\mathcal{M}}[\mu] = \{c\}$$

□

We close this section with a remark about entailment in $\mathcal{L}_P$. The definition of the entailment relation in $\mathcal{L}_P$ is exactly the same as in first order logic. Therefore, it is a direct consequence that $\mathcal{L}_P$ is *monotonic* in the following sense.

Remark 3.3.1 ($\mathcal{L}_P$ is monotonic)**.** Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any consistent sets of $\mathcal{L}$-formulas Γ, Δ such that $\Gamma \subseteq \Delta$ and any $\mathcal{L}$-formula α,

$$\text{If } \Gamma \models \alpha \text{ then } \Delta \models \alpha$$

□

Chapter 4

DK Action Theories

In this chapter, we present the *DK theories of action*. Syntactically, *DK* theories of action are essentially the same as the situation calculus *basic action theories (BAT)* in the $\mathcal{L}_P$ setting. Due to the non-standard $\mathcal{L}_P$ semantics, though, the behavior of the theories regarding disjunctive knowledge differentiates them from that of the basic action theories. We will be showing that a similar *regression operator* as in the basic action theories can be used in *DK* theories and that a similar *regression theorem* holds. At the end of the chapter, we also discuss how sensing can be treated in *DK* theories.

The definitions in this section extend the ones of basic action theories, as presented in [Rei91],[PR99] and [Rei01]. In particular, the regression theorem proof for *DK* action theories follows closely the one for basic action theories found in [PR99].

DK action theories is the basis for next chapter, where we present a restricted version which captures fluent-based disjunctive knowledge only. Note that this is a "semantic-oriented" way of restricting the effect of action theories. In the light of the results in [PL02], this approach is a result of the choice taken not to syntactically restrict the basic action theories of situation calculus to achieve such an effect, but to consider a useful way of semantically forcing the intended behavior, so that we can achieve an efficient way of doing theorem proving.

4.1 *DK* Action Theories

In the *DK* action theories setting, the *rigid terms* are intended to capture the *objects and states* of the domain which is represented and the *fluent terms* are intended to capture *notions about those objects and states*. The emphasis is on how disjunctive knowledge is treated and, as we mentioned before, the intention is to define *DK* theories that we can force to capture only fluent-based disjunctive knowledge.

The basic elements for a *DK* theory of action are the same as the basic action theories. We will see the definitions of the *successor state axiom* for a functional fluent symbol fl, as well as the *action precondition axiom* for an action function symbol A, and discuss their effect. We start the discussion about *DK* action theories by introducing the *initial specification axiom* which gives some insight on issues that arise in the $\mathcal{L}_P$ context of possible values. These axioms will be used in the next chapter to ensure that disjunctive knowledge is limited appropriately in the initial situation.

In what follows, we will be using $\forall\vec{x}$ and $\exists\vec{x}$ to denote $\forall x_1...\forall x_n$ and $\exists x_1...\exists x_n$, for some natural number n which will be implicit by the context.

Definition 4.1.1 (Initial specification axiom)**.** An *initial specification axiom (ISA)* for the $n+1$-ary functional fluent symbol fl is an $\mathcal{L}_P$-sentence of the form:

$$\forall\vec{x}\forall y\ fl(\vec{x}, S_0) \approx y \equiv \Psi_{fl}(\vec{x}, y)$$

where Ψ_{fl} mentions only rigid terms and all its free variables are among $\vec{x}, y$. □

The initial specification axiom for a fluent fl is intended to force all fluent terms of the form $fl(\vec{t}, S_0)$ to a specific interpretation defined by $\Psi_{fl}(\vec{x}, y)$. Intuitively, this is because the axiom acts like a generic strong $fl(\vec{x}, S_0)$-formula which defines exactly the interpretation of the fluent symbol fl for the initial situation, for any structure $\mathcal{M}$: it essentially defines $fl^{\mathcal{M}}(\vec{c}, c_{S_0})$ for all $\vec{c}$, where $S_0^{\mathcal{M}} = \{c_{S_0}\}$.

Note that fluent terms are not allowed in Ψ_{fl}, so that to avoid ISAs to form counter-intuitive loops in the way the values of fluents are defined. Some examples of ISAs for

fluents follow.

Example 4.1.1 (Initial specification axiom)**.** The next sentence is an ISA for the unary fluent *weather* which states that it is definite that the value of $\mathit{weather}(S_0)$ in the initial situation is $rainy$.

$$\forall y\ \mathit{weather}(S_0) \approx y \equiv y \approx rainy$$

The next sentences are ISAs for the binary fluent *location* which is intended to represent the location of objects. The first ISA states that for all objects x, the value of $\mathit{location}(x, S_0)$ is exactly either $place_1$ or $place_2$ or $place_3$.

$$\forall x \forall y\ \mathit{location}(x, S_0) \approx y \equiv y \approx place_1 \vee y \approx place_2 \vee y \approx place_3$$

The axiom acts as a strong $\mathit{location}(x, S_0)$-formula of the form

$$\forall x\ \mathit{location}(x, S_0) = [place_1, place_2, place_3]$$

This ISA states that for all objects x, the possible values for $\mathit{location}(x, S_0)$ are exactly $\{place(z)\}$.

$$\forall x \forall y\ \mathit{location}(x, S_0) \approx y \equiv \exists z\ y \approx place(z)$$

The axiom acts again as a a strong $\mathit{location}(x, S_0)$-formula which now captures an infinite number of possibilities for $\mathit{location}(x, S_0)$.

Following the notation of the *dice* examples we saw in chapter 3 and specifically example 3.3.6, the next formula expresses that all objects in the domain which are dice, have the value 1,2,3 or 4 in the initial situation. Note that the value of $\mathit{dice}(x, S_0)$ is also defined for the case that x is not a dice and in that case the term has all possible values.

$$\forall x \forall y\ \mathit{dice}(x, S_0) \approx y \equiv isdice(x) \approx \top \wedge (y \approx 1 \vee y \approx 2 \vee y \approx 3 \vee y \approx 4) \vee isdice(x) \approx \bot \wedge true$$

□

The successor state axiom for the fluents has the same syntax as in basic action theories.

Definition 4.1.2 (Successor state axiom)**.** A *successor state axiom (SSA)* for the $n+1$-ary functional fluent symbol fl is an $\mathcal{L}_P$-sentence of the form:

$$\forall x_1, ..., \forall x_n \forall a \forall s \forall y \; fl(x_1, ..., x_n, do(a, s)) \approx y \;\equiv\; \Phi_{fl}(x_1, ..., x_n, y, a, s)$$

where Φ is a formula uniform in s and all its free variables are among $x_1, ..., x_n, a, s$. $\square$

Note that in $\mathcal{L}_P$, this standard successor state axiom syntax also achieves a similar closure effect as the strong $fl(\vec{t})$-formula macro. In the same way that the ISA of a fluent defines its possible values for the initial situation, the SSA defines its possible values for all the rest of the situations.

In basic action theories there is one value y such that $fl(\vec{x}, s) = y$ for each instantiation of $\vec{x}, s$ and the universal quantification for y has the completeness effect that the atomic formula $fl(\vec{x}, s) = y$ is not true for any other y. Here, there are possible many values y such that $fl(\vec{x}, s) \approx y$ for each instantiation of $\vec{x}, s$ and the quantification of y has also the effect that for each instantiation of $\vec{x}, s$, the interpretation of $fl(\vec{x}, s)$ is specified constructively in the same way a strong $fl(\vec{x}, s)$-formula would do it.

Note also that the successor state axiom does not *relate* the value of the fluent between the two situations, but merely *defines* the possible values for the next situation. As discussed also in chapter 3 in example 3.3.3, this is weaker than the effect of successor state axioms in situation calculus, but we appeal to it because it ensures that the successor state axiom preserves fluent-based disjunctive knowledge.

It is interesting to examine the interpretation of $fl(\vec{t}, do(A, S))$, when the arguments of fl also capture disjunctive knowledge. This may be disjunctive knowledge about the value of some arguments $\vec{t}$, as well as disjunctive knowledge about the action performed, A. In both cases the possible values for the fluent fl are treated by the semantics in the standard way.

We explore further these remarks in the following detailed examples.

Example 4.1.2 (Successor state axiom, $dice(t, s)$)**.** The following sentence is a SSA for fluent $\mathsf{dice}(t, s)$.

$$\begin{aligned}\forall x \forall a \forall s \forall y \ \mathsf{dice}(x, do(a, s)) \approx y \ \equiv \ & a \approx rollDice(x) \wedge (y \approx 1 \vee y \approx 2 \vee y \approx 3 \vee y \approx 4) \\ \vee \ & a \approx setDiceValueTo1(x) \wedge y \approx 1 \\ \vee \ & \neg(a \approx rollDice(x) \vee a \approx setDiceValueTo1(x)) \wedge y \approx \mathsf{dice}(x, s)\end{aligned}$$

This SSA is intended to force the following behavior for the value of dice t, $\mathsf{dice}(t, do(A, s))$.

- When an action $A \stackrel{def}{=} rollDice(t)$ is performed at situation s, then the possible values for $\mathsf{dice}(t, do(rolldice(t), s))$ are exactly 1,2,3 or 4.
- When action $A \stackrel{def}{=} setDiceValueTo1(t)$ is performed at situation s, then it is definite that $\mathsf{dice}(t, do(rolldice(t), s))$ has the value 1.
- When the action performed in s is none of the above, then the possible values for $\mathsf{dice}(t, do(rolldice(t), s))$ are exactly the possible values of $\mathsf{dice}(t, s)$.

We can see that the disjunctive knowledge regarding the value of a fluent can both become more specific (e.g. by action $setDiceValueTo1(x)$) and less specific (e.g. by action $rollDice(x)$), depending on the form of the SSA.

It is interesting to see the effect of the SSA when there is disjunctive knowledge captured by the terms appearing in the fluent term in question. The arguments' disjunctive knowledge is propagated to the disjunctive knowledge about the possible values of $\mathsf{dice}(t, do(A, S))$ in the standard way the semantics define. In detail, according to the definition of term denotation 3.2.5, the interpretation of $\mathsf{dice}(t, do(rolldice(t), S))$ is the union of the interpretation of the term for all simple structures. Even if sometimes there is no clear intuition about what the results should be, we claim that the results this formalism entails are intuitive.

Suppose that the theory entails

$$\mathsf{dice}(1, S) = [3], \quad \mathsf{dice}(2, S) = [4] \ \text{ and } \ t = [1, 2]$$

Then it also entails

$$\mathit{dice}(t, S) = [3, 4]$$

which is what we expect for the possible values of $\mathit{dice}(t, S)$. We get similar results when action A captures disjunctive knowledge. In this case, though, a little more thought is needed in order to see that the results are indeed intuitive. In the same setting, we consider action $A \stackrel{def}{=} setDiceValueTo1(t)$. Then, according to the SSA about fluent fl, the theory also entails

$$\mathit{dice}(t, do(setDiceValueTo1(t), S)) = [1]$$

but also

$$\mathit{dice}(1, do(setDiceValueTo1(t), S)) = [1, 3]$$

$$\mathit{dice}(2, do(setDiceValueTo1(t), S)) = [1, 4]$$

We do expect that it is definite that the value of dice t is 1 after performing action A, since, regardless of which dice t refers to (1 or 2), after the action is performed, its value will be 1. On the other hand, regarding dice 1, we expect that it has two possible values: one because it is possible that action A refers to that dice and therefore its value is set to 1, and another one because it is possible that the action does not refer to that dice and so its value remains the same, 3. The same argument applies for dice 2.

Following similar reasoning it is not difficult to verify that the theory also entails the following results.

$$\mathit{dice}(t, do(setDiceValueTo1(1), S)) = [1, 4]$$

$$\mathit{dice}(1, do(setDiceValueTo1(1), S)) = [1]$$

$$\mathit{dice}(2, do(setDiceValueTo1(1), S)) = [4]$$

□

Example 4.1.3 (Successor state axiom, $location(t, s)$)**.** The following sentence is a SSA for fluent $\mathsf{location}(x, s)$.

$$\forall x \forall a \forall s \forall y \ \mathsf{location}(x, do(a, s)) \approx y \ \equiv \ a \approx moveToLocation(x, y)$$
$$\vee \ \neg \exists z \ a \approx moveToLocation(x, z) \wedge y \approx \mathsf{location}(x, s)$$

This SSA is intended to capture the possible values for the location of any object x. In this simple form, it states that the location of an object remains the same, unless there is a $moveToLocation$ action which involves that object. In that case, the object's new location is the one mentioned in the action. As in the previous example, the possible disjunctive knowledge about the arguments of the fluent $\mathsf{location}$ is propagated appropriately.

The successor state axioms in $\mathcal{L}_P$ have a non-standard effect which is easy to identify when they are *context-sensitive*. Consider the following SSA for fluent $\mathsf{location}(x, s)$ which, instead of action $moveToLocation(x, y)$, includes the action $moveToObject(x, y)$. This action captures that object x is moved to the location of object y.

$$\forall x \forall a \forall s \forall y \ \mathsf{location}(x, do(a, s)) \approx y \ \equiv \ \exists z \ a \approx moveToObject(x, z) \wedge y \approx \mathsf{location}(z, s)$$
$$\vee \ \neg \exists z \ a \approx moveToObject(x, z) \wedge y \approx \mathsf{location}(x, s)$$

Unlike situation calculus, it is not the case that *the value* of the location of the one object *is the same as the value* of the location of the other, in the resulting situation after a $moveToObject$ action. Instead, when there is a $moveToObject(x, y)$ action, it is *the possible values* for the location of object x that are the same. For example, suppose that the following hold for the initial situation.

$$\mathsf{location}(box1, S_0)) = [room1]$$

$$\mathsf{location}(box2, S_0)) = [room1, room2, room3]$$

Then, according to the previous successor state axiom,

$$\mathsf{location}(box1, do(moveToObject(box1, box2), S_0))) = [room1, room2, room3]$$

This is one of the key features in the behavior of *DK* action theories. As mentioned before, this keeps the disjunctive knowledge fluent-based, in contrast to the situation calculus behavior. In a similar way as we saw in example 3.3.3, this SSA fails to capture that the actual values of the fluents should be the same. In this way, in the resulting situation, both of the fluents can have *any of the possible values* which is something that can be *summarized* in an $\mathcal{L}_P$ structure utilizing the fluents' interpretation to sets and the cross-product-like behavior for the possible values. On the contrary, the effect that a situation calculus SSA captures, rules out some of these possibilities so that only the ones that the fluents have the same values persist. This effect cannot be summarized in an $\mathcal{L}_P$ structure and expresses a form of general disjunctive knowledge which we intend to avoid. □

Before moving on to the next definitions, there is last remark on successor state axioms and the dynamics of the possible values. A SSA is essentially a set of rules which define, for each possible value of a fluent fl in situation s, what the next possible value(s) should be in situation $do(a, s)$. In this way, the SSA defines all the possible values for $fl(\vec{x}, do(a, s))$. Note though, that $fl(\vec{x}, s)$ and $fl(\vec{x}, do(a, s))$ are two terms that are only related through the SSA and apart from that they are treated by the semantics as two different and independent entities. This can result in counter-intuitive behavior when both terms appear in the same formula, as the semantics take into account any combination of possible values for the terms, even if they may not be justified by some rule of the SSA. We do not discuss this in detail and we note that it does not affect our results, as we focus on formulas which are uniform in some situation s.

The action precondition axiom is a little different than in basic action theories, as here $Poss$ is a function symbol.

Definition 4.1.3 (Action precondition axiom)**.** An *action precondition axiom (APA)* for

the n-ary action function symbol A is an $\mathcal{L}_P$-sentence of the form:

$$\forall x_1, ..., \forall x_n \forall s \forall y \; Poss(A(x_1, ..., x_n), s) \approx y \;\equiv\; \Pi_A(x_1, ..., x_n, s) \wedge y \approx \top \\ \vee \neg\Pi_A(x_1, ..., x_n, s) \wedge y \approx \bot$$

where Π_A is a formula uniform in s that does not mention $Poss$ and all its free variables are among $x_1, ..., x_n, s$. □

As we saw in section 3.3 and specifically in examples 3.3.5, 3.3.6, the atomic formulas involving fluents which emulate predicates need some attention. We discuss that for the case of $Poss$ in the next examples. Note also that for reasons that will become clear in chapter 5, we appeal to action precondition axioms which use $\top$ and $\bot$ as presented in the definition, so that all possible values for $Poss$ are always exactly at most $\top$ and $\bot$. An APA of the form

$$\forall x_1, ..., \forall x_n \forall s \; Poss(A(x_1, ..., x_n), s) \approx \top \;\equiv\; \Pi_A(x_1, ..., x_n, s)$$

does not ensure that there are at most two possible values for $Poss$.

Example 4.1.4 (Action precondition axiom)**.** The following sentence is a very simple precondition axiom for action $rollDice(x)$ which is intended to capture that the action can be performed, if the agent is holding object x in situation s and that object is in fact a dice.

$$\forall x \forall y \; Poss(rollDice(x), s) \approx y \;\equiv\; \Pi_A(x, s) \wedge y \approx \top \vee \neg\Pi_A(x, s) \wedge y \approx \bot$$

where

$$\Pi_A(x, s) \equiv isdice(x) \approx \top \wedge \mathit{holding}(x, s) \approx \top$$

This APA relies on some appropriate definition of the function symbols $isdice(x)$ and $\mathit{holding}(x, s)$ so as to emulate predicate functionality (see discussion on example 3.3.5).□

Note that like all atomic formulas in $\mathcal{L}_P$, $Poss(A, S) \approx \top$ expresses that *it is possible* that action A can be performed in situation S. This is important when the action term A captures disjunctive knowledge. In that case we can distinguish between saying that *it is possible* that the action can be performed and that *it is definite* that the action can be performed.

Example 4.1.5 (Action precondition axiom 2)**.** Suppose that the theory entails

$$t = [1, 2], \quad isdice(1) \approx \top, \quad isdice(2) \approx \top$$

$$\mathsf{holding}(isdice(1), S) \approx \top, \quad \mathsf{holding}(isdice(2), S) \approx \bot$$

and action $rollDice(t)$ is performed at situation S. Then the theory also entails

$$Poss(rollDice(t), S) \approx \top$$

as it is indeed possible that the action can be performed. We can also express that the action can definitely be performed by the next sentence

$$\forall a \; a \approx rollDice(t) \rightarrow Poss(a, S) \approx \top$$

which is not entailed by the theory, as action $rollDice(2)$ cannot be performed. □

There is one more set of axioms we need to specify, before we get to the definition of *DK* action theories, the *foundational axioms for situations*. This is defined in [Rei01] using a second order logic axiom and has the effect of limiting the sort situation to the smallest set containing S_0 and closed under the application of the function *do* to an action and a situation. Here, instead of the second order axiom, we prefer an infinite first order axiomatization for situations. This choice is made because we find that it would be complicated and maybe unclear to talk about second order $\mathcal{L}_P$ languages, as $\mathcal{L}_P$ semantics are non-standard in many ways. Note, that this infinite set of axioms *is not equivalent with the second order one*, but has the intended effect for the formulas we will be dealing with.

Definition 4.1.4 (First order foundational axioms for situations)**.** The first order foundational axioms for situations are the countable infinite sentences, summarized by the following induction schema for any $\mathcal{L}$-formula α:

$$\forall s\ \alpha(s) \wedge \forall a \forall s'\ \alpha(s') \rightarrow \alpha(do(a, s')) \rightarrow \forall s\ \alpha(s)$$

and the following formula:

$$\forall a_1 \forall a_2 \forall s_1 \forall s_2\ do(a_1, s_1) \approx do(a_2, s_2) \rightarrow a_1 \approx a_2 \wedge s_1 \approx s_2$$

□

DK action theories are defined similarly as the situation calculus basic action theories.

Definition 4.1.5 (*DK* action theories)**.** Let $\mathcal{L}$ be an $\mathcal{L}_P$ language. A *DK action theory* $\mathcal{D}$ for $\mathcal{L}$ is a collection of axioms of the following form:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{SSA} \cup \mathcal{D}_{APA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{S_0}$$

where

- Σ are the (first order) foundational axioms for situations.
- $\mathcal{D}_{SSA}$ is a set of successor state axioms for the functional fluents. The set includes one SSA for each functional fluent symbol in $\mathcal{L}$.
- D_{APA} is a set of action precondition axioms for the action function symbols. The set includes one APA for each action function symbol.
- $\mathcal{D}_{UNA}$ is the set of unique names axioms for actions function symbols.
- $\mathcal{D}_{S_0}$ is a set of $\mathcal{L}$-sentences that are uniform in S_0. $\mathcal{D}_{S_0}$ will function as the initial theory of the world, before any actions are executed. Following the standard naming conventions in the literature, we will often call $\mathcal{D}_{S_0}$ as the initial database. Note that the incomplete knowledge that the initial database captures can be restricted

according to our needs using initial specification axioms or any sentences uniform in S_0. [1]

Furthermore, $\mathcal{D}$ must satisfy the following functional fluent consistency property. For each $n+1$-ary fluent fl

$$\mathcal{D}_{UNA} \cup \mathcal{D}_{S_0} \models \forall a \forall s \forall x_1, ..., \forall x_n \exists y \ \Phi_{fl}(x_1, ..., x_n, y, a, s)$$

□

The functional fluent consistency property is sufficient for preventing a source of inconsistency in the successor state axioms. Note that, unlike the basic action theories, here is only needed that there exists *some* value y for each fluent for each situation and not that it is also unique. This is a result of dealing with the possible values of the fluents.

4.2 Regression in *DK* Action Theories

In this section we discuss regression in *DK* action theories. We will see that the syntactic regression operator is similar to the one used in basic action theories and that the corresponding regression theorem holds.

The basis of the regression operator for *DK* action theories is the same as in basic action theories but needs some tuning due to the non-standard interpretation of equality and terms. Similarly, the structure of the regression theorem proof for *DK* action theories is the same as the one about basic action theories, but since some steps of the proof are not trivial in $\mathcal{L}_P$, we need to prove some helping lemmas. The essential difference has to do with the treatment of functional fluent symbols and the notion of possible values. We now move on to the necessary definitions.

[1] We will see more about limiting the initial database using ISAs in the next chapter, where we discuss about implementing *DK* action theories in Prolog.

Regressable formulas and the regression operator are defined similarly as in basic action theories. Also, in what follows, we will be using the $do([a_1, ..., a_n], S_0)$ notation for situation terms.

Definition 4.2.1 (Regressable Formula)**.** An $\mathcal{L}$-formula W is *regressable* iff

- Every term of sort situation mentioned by W has the form $do([a_1, ..., a_n], S_0)$ for some $n \geq 0$ and for terms $a_1, ..., a_n$ of sort action.
- The function symbol $Poss$ appears only in atoms of the form $Poss(\alpha, S) \approx t$, where α has the form $A(t_1, ..., t_n)$ for some n-ary action function symbol A of $\mathcal{L}$.
- W does not quantify over situations.

□

Definition 4.2.2 (Prime functional fluent term)**.** A fluent term is *prime* iff it has the syntactic form $fl(\vec{t}, do([a_1, ..., a_n], S_0))$ for $n \geq 1$ and each of the terms $\vec{t}, a_1, ..., a_n$ is uniform in S_0.

As proved in [PR99], any regressable fluent term mentions a prime functional fluent term. □

We now move to the definition of the regression operator for *DK* theories. It is easy to see that the main idea is the same as in basic action theories; the operator is only tuned to comply with the $\mathcal{L}_P$ semantics of possible values.

Definition 4.2.3 (Regression operator)**.** For a *DK* action theory $\mathcal{D}$ of some $\mathcal{L}_P$ language $\mathcal{L}$ and a regressable $\mathcal{L}$-formula W, the *regression operator* $\mathcal{R}$ is defined by induction as follows:

- W is a regressable atomic formula. Then it is of the form $t_1 \approx t_2$. We consider all the possible cases for t_1, t_2.

1. W is such that S_0 is the only term of sort situation (if any) mentioned in t_1, t_2. Then,

   $\mathcal{R}[W] = W$.

2. t_1, t_2 are situation terms, so W is $do([a_1, ..., a_m], S_0) \approx do([a'_1, ..., a'_n], S_0)$ for some m, n. The case $m = n = 0$ is covered by 1. If $m = n \geq 1$, then

   $\mathcal{R}[W] = \mathcal{R}[a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m]$.

   Otherwise, if $m \neq n$, then

   $\mathcal{R}[W] = false$.

3. W is a regressable $Poss$ atomic formula of the form $Poss(A(\vec{t}), S) \approx t'$. By the definition of DK action theories, 4.1.5, there must exist in D an action precondition axiom for A of the following form:

   $\forall \vec{x} \forall s \forall y \; Poss(A(\vec{x}), s) \approx y \;\equiv\; \Pi_A(\vec{x}, s) \wedge y \approx \top \vee \neg \Pi_A(\vec{x}, s) \wedge y \approx \bot$

   Without loss of generality, assume that all quantifiers (if any) of $\Pi_A(\vec{x}, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $Poss(A(\vec{t}), S)$. Then,

   $\mathcal{R}[W] = \mathcal{R}[\exists x_1 ... \exists x_n \exists a \; \bigwedge x_i \approx t_i \wedge a \approx A(\vec{x})$

   $\wedge (\Pi_A(\vec{x}, S) \wedge t' \approx \top \;\vee\; \neg \Pi_A(\vec{x}, S) \wedge t' \approx \bot) \;]$.

   Here, the regression operator replaces the $Poss$ atomic formula by a logically equivalent formula (in the context of theory D) and regresses the resulting formula. Note that we pick $x_1, ..., x_n, a$ to be variables that does not appear free in $W, \vec{t}, A, S$.

4. W mentions a term of the form $fl'(\vec{t'}, do(A', S'))$ for some functional fluent fl'. Then, according to definitions 4.2.1 and 4.2.2, $fl'(\vec{t'}, do(A', S'))$ mentions a prime fluent term $fl(\vec{t}, do(A, S))$. By the definition of DK action theories, 4.1.5, there must exist in D a successor state axiom for fl of the following form:

   $\forall \vec{x} \forall a \forall s \forall y \; fl(\vec{x}, do(a, s)) \approx y \;\equiv\; \Phi_{fl}(\vec{x}, y, a, s)$

Without loss of generality, assume that all quantifiers (if any) of $\Phi_A(\vec{x}, y, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $fl(\vec{t}, do(A, S))$. Then,

$\mathcal{R}[W] = \mathcal{R}[\exists x_1 ... \exists x_n \exists a \exists y \bigwedge x_i \approx t_i \wedge a \approx A$

$\wedge \Phi_A(\vec{t}, y, a, S) \wedge W|_y^{fl(\vec{t}, do(A,S))}]$

Note that we pick $x_1, ..., x_n, a, y$ to be variables that do not appear free in $W, \vec{t}, A, S$. Like the previous step, the regression operator replaces W by a logically equivalent formula (in the context of theory D) and regresses the resulting formula.

- $\mathcal{R}[\neg W] = \neg \mathcal{R}[W]$
- $\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$
- $\mathcal{R}[\exists x W] = \exists x \mathcal{R}[W]$ □

We now move on to prove that this regression operator has the intended effect in the $\mathcal{L}_P$ setting and *DK* action theories. In order to do that, we follow the proof of the regression operator for basic action theories. The non-standard interpretation of $\mathcal{L}_P$-terms poses some complication and in order to apply that proof, we need to prove the *functional fluent regression principle* and the *Poss regression principle*.

This is basically the proof that the regression operator replaces the atomic *Poss* formulas and the functional fluent terms with logically equivalent sentences. Due to the non-standard behavior of the fluents in $\mathcal{L}_P$, it is not obvious that it is correct.

Lemma 4.2.1 (Functional fluent regression principle)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$, a DK theory D and a regressable atomic $\mathcal{L}$-formula W which mentions the fluent term $fl(t_1, ..., t_n, do(A, S))$. By the definition of DK action theories, 4.1.5, there must exist in D a successor state axiom for fl of the following form:*

$$\forall \vec{x} \forall a \forall s \forall y \; fl(\vec{x}, do(a, s)) \approx y \; \equiv \; \Phi_{fl}(\vec{x}, y, a, s)$$

Without loss of generality, assume that all quantifiers (if any) of $\Phi_{fl}(\vec{x}, y, a, s)$ *have had their quantified variables renamed to be distinct from the free variables (if any) of* $fl(\vec{t}, do(A, S))$*. Then,*

$$\mathcal{D} \models W \equiv \exists x_1 ... \exists x_n \exists a \exists s \exists y \bigwedge x_i \approx t_i \wedge a \approx A \wedge \Phi_A(\vec{t}, y, a, S) \wedge W|_y^{fl(\vec{t}, do(A,S))}$$

Note that we pick $x_1, ..., x_n, a, y$ *to be variables that do not appear free in* $W, \vec{t}, A, S$*.*

Proof: W is atomic, so it is an $\mathcal{L}$-formula of the form $t_1 \approx t_2$. Let $\mathcal{M}$ be an arbitrary $\mathcal{L}$-model and variable assignment μ such that $\mathcal{M} \models \mathcal{D}[\mu]$. By the definition of truth in a structure 3.2.6, $\mathcal{M} \models W[\mu]$ iff there exists some simple structure $\mathcal{M}'$ such that

$$\mathcal{M}' \preceq \mathcal{M} \quad \text{and} \quad \mathcal{M}' \models W[\mu] \tag{4.1}$$

Since $\mathcal{M}'$ is a simple structure and by lemma 3.3.2 about the interpretation of terms in a simple structure, the interpretation of $fl(\vec{t}, do(A, S))$ is a singleton set,

$$fl(\vec{t}, do(A, S))^{\mathcal{M}'}[\mu] = \{c\}, \text{ for some } c$$

By applying twice the corollary 3.3.8 for the terms $t_1, fl(\vec{t}, do(A, S))$ and $t_2, fl(\vec{t}, do(A, S))$, we get that (4.1) holds iff there exists some simple structure $\mathcal{M}'$ such that

$$\mathcal{M}' \preceq \mathcal{M}, \quad \mathcal{M}' \models W|_y^{fl(\vec{t}, do(A,S))}[\mu(c/y)]$$

where

$$fl(\vec{t}, do(A, S))^{\mathcal{M}'}[\mu] = \{c\}, \text{ for some } c$$

By the definition of truth in a structure 3.2.6 and lemma 3.3.3, that holds iff

$$\mathcal{M} \models W|_y^{fl(\vec{t}, do(A,S))}[\mu(c/y)], \text{ and } c \in fl(\vec{t}, do(A, S))^{\mathcal{M}}[\mu]$$

By lemma 3.3.5, that is true iff

$$\mathcal{M} \models W|_y^{fl(\vec{t}, do(A,S))}[\mu(c/y)] \text{ and } \mathcal{M} \models fl(\vec{t}, do(A, S)) \approx y[\mu(c/y)] \tag{4.2}$$

for some c in the corresponding domain.

By corollary 3.3.6 we also get that,

$$\mathcal{M} \models x_i \approx t_i[\mu(c_j/x_j)]$$

$$\mathcal{M} \models a \approx A[\mu(c_A/a)]$$

for some $c_1, ..., c_n, c_A$ in the corresponding domains. So, since we carefully chose $x_1, ..., x_n, a, y$ to be variables that do not appear free in $W, \vec{t}, A, S$, (4.2) holds iff

$$\begin{aligned}
\mathcal{M} &\models W|_y^{fl(\vec{t},do(A,S))}[\mu(c/y)(c_j/x_j)(c_A/a)] \\
\mathcal{M} &\models fl(\vec{t}, do(A,S)) \approx y\ [\mu(c/y)(c_j/x_j)(c_A/a)] \\
\mathcal{M} &\models x_i \approx t_i\ [\mu(c/y)(c_j/x_j)(c_A/a)] \\
\mathcal{M} &\models a \approx A\ [\mu(c/y)(c_j/x_j)(c_A/a)]
\end{aligned} \tag{4.3}$$

for some $c, c_1, ...c_n, c_A$ in the corresponding domains.

Now, since $\mathcal{M} \models \mathcal{D}$, then[2]

$$\mathcal{M} \models \forall\vec{x}\forall a\forall s\forall y\ fl(\vec{x}, do(a,s)) \approx y\ \equiv\ \Phi_{fl}(\vec{x}, y, a, s)[\mu]$$

and by the definition of truth in a structure 3.2.6,

$$\mathcal{M} \models fl(\vec{x}, do(a,s)) \approx y\ \equiv\ \Phi_{fl}(\vec{x}, y, a, s)[\mu(d/y)(d_j/x_j)(d_a/a)(d_s/s)]$$

for all d, d_j, d_a, d_s in the corresponding domains. Therefore, this also holds for $d = c$, $d_j = c_j$, $d_a = c_A$ and $d_s = c_S$, where $S^{\mathcal{M}}[\mu] = \{c_S\}$

$$\mathcal{M} \models fl(\vec{x}, do(a,s)) \approx y\ \equiv\ \Phi_{fl}(\vec{x}, y, a, s)[\mu(c/y)(c_j/x_j)(c_S/s)(c_A/a)]$$

or equivalently

$$\mathcal{M} \models fl(\vec{x}, do(a,S)) \approx y\ \equiv\ \Phi_{fl}(\vec{x}, y, a, S)[\mu(c/y)(c_j/x_j)(c_A/a)]$$

Therefore

$$\mathcal{M} \models fl(\vec{x}, do(a,S)) \approx y\ [\mu(c/y)(c_j/x_j)(c_A/a)]$$

[2]Note that we may need to do some change of variable names in order for the SSA to quantify over the same variables $(\vec{x}, a, s, y)$, but this is not a problem.

iff

$$\mathcal{M} \models \Phi_{fl}(\vec{x}, y, a, S)[\mu(c/y)(c_j/x_j)(c_A/a)]$$

So, (4.3) holds, iff

$$\mathcal{M} \models W|_y^{fl(\vec{t},do(A,S))}[\mu(c/y)(c_j/x_j)(c_A/a)]$$
$$\mathcal{M} \models \Phi_{fl}(\vec{x}, y, a, S)[\mu(c/y)(c_j/x_j)(c_A/a)]$$
$$\mathcal{M} \models x_i \approx t_i \ [\mu(c/y)(c_j/x_j)(c_A/a)]$$
$$\mathcal{M} \models a \approx A \ [\mu(c/y)(c_j/x_j)(c_A/a)]$$

for some $c, c_1, ...c_n, c_A$ in the corresponding domains. By the definition of truth in a structure 3.2.6, that is true iff

$$\mathcal{M} \models \bigwedge x_i \approx t_i \wedge W|_y^{fl(\vec{t},do(A,S))} \wedge \Phi_{fl}(\vec{x}, y, a, S) \wedge a \approx A \ [\mu(c/y)(c_j/x_j)(c_A/a)]$$

for some $c, c_1, ...c_n, c_A$ in the corresponding domains. By the same definition, that is true iff

$$\mathcal{M} \models \exists x_1 ... \exists x_n \exists a \exists y \bigwedge x_i \approx t_i \wedge a \approx A \wedge \Phi_{fl}(\vec{x}, y, a, S) \wedge W|_y^{fl(\vec{t},do(A,S))}[\mu]$$

So,

$$\mathcal{M} \models W \equiv \exists x_1 ... \exists x_n \exists a \exists y \bigwedge x_i \approx t_i \wedge a \approx A \wedge \Phi_{fl}(\vec{x}, y, a, S) \wedge W|_y^{fl(\vec{t},do(A,S))}[\mu]$$

Since $\mathcal{M}$ is an arbitrary model which satisfies $\mathcal{D}$, the lemma holds. □

Lemma 4.2.2 (*Poss* regression principle)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$, a DK theory D and a regressable atomic $\mathcal{L}$-formula W of the form $Poss(A(\vec{t}), S) \approx t'$. By the definition of DK action theories, 4.1.5, there must exist in D an action precondition axiom for A of the following form:*

$\forall \vec{x} \forall s \forall y \ Poss(A(\vec{x}), s) \approx y \ \equiv \ \Pi_A(\vec{x}, s) \wedge y \approx \top \vee \neg \Pi_A(\vec{x}, s) \wedge y \approx \bot$

Without loss of generality, assume that all quantifiers (if any) of $\Pi_A(\vec{x}, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $Poss(A(\vec{t}), S)$.

Then,

$\mathcal{D} \models W \equiv \exists x_1 ... \exists x_n \exists a \ \bigwedge x_i \approx t_i \wedge a \approx A(\vec{x})$

$\wedge \Pi_A(\vec{x}, S) \wedge t' \approx \top \vee \neg \Pi_A(\vec{x}, S) \wedge t' \approx \bot]$

Note that we pick $x_1, ..., x_n, a$, *to be variables that do not appear free in* W.

Proof: The proof procedure is similar to the proof of the functional fluent regression principle 4.2.1. □

The proof of the regression theorem is done by induction on $index(W)$ for a formula W. $index(W)$ is exactly the same as in [PR99] except for that it does not account for atomic sentences of the form $t_1 \sqsubset t_2$. Intuitively, it is defined so that to give precedence to the regression of *Poss* atomic formulas, then situation atomic formulas and then regression of functional fluents starting from the primal fluent terms. We copy the definition here, for completeness.

Definition 4.2.4 ($index(W)$)**.** Consider the set Λ of all countably infinite sequences of natural numbers with a finite number of non-zero elements and the following reverse lexicographic order relation on this set:

$$(\lambda_1, \lambda_2, ...) \prec (\lambda'_1, \lambda'_2, ...) \text{ iff for some } m, \lambda_m < \lambda'_m \text{ and for all } n > m, \lambda_n = \lambda'_n$$

$(\Lambda, \prec)$ is well founded with minimal element (0,0,...). Next, let **2** be the set of all 2-tuples of natural numbers. We overload the relation $\prec$ by defining a reverse lexicographic ordering on **2**:

$$(m_1, m_2) \prec (n_1, n_2) \text{ iff } m_2 < n_2 \text{ or } m_2 = n_2 \text{ and } m_1 < n_1$$

Finally, we overload $\prec$ again to define an ordering on $\Lambda \times \mathbf{2}$:

$$(\lambda, f) \prec (\lambda', f') \text{ iff } f < f' \text{ or } f = f \text{ and } \lambda < \lambda'$$

The relation $\prec$ on $\Lambda \times \mathbf{2}$ is well founded with minimal element ((0,0,...),(0,0)) and therefore can serve as a basis for an inductive proof.

For $n \geq 0$, we define the *length* of the situation term $do([a_1, ..., a_n], S_0)$ to be n. Also, whenever $g(t_1, ..., t_n)$ is an $\mathcal{L}$-term, $t_1, ..., t_n$ are said to be *proper subterms* of $g(t_1, ..., t_n)$. An occurrence of a situation term in an $\mathcal{L}$-formula W is *maximal* iff its occurrence is not a proper subterm of some situation term.

Now, given a regressable formula W, we can define the *index* of the formula

$$index(W) = ((C, \lambda_1, \lambda_2, ...), (P, E))$$

where

- C is the total number of connectives and quantifiers mentioned in W.
- For $m > 1$, λ_m is the number of occurrences in W of maximal situation terms of length m.
- P is the number of atoms of the form $Poss(A, S) \approx \top$ or $Poss(A, S) \approx \bot$ and E is the number of atoms of the form $S_1 \approx S_2$ mentioned in W, where S_1, S_2 are terms of sort situation. □

The proof of the regression theorem is by complete induction on the index of regressable formulas, relative to the ordering $\prec$. As we mentioned earlier, we follow the proof of the regression operator for basic action theories as in [PR99], tuned to comply with $\mathcal{L}_P$ and using the necessary lemmas 4.2.1 and 4.2.2 which we proved in this section.

Theorem 4.2.3 (Regression theorem)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. Suppose W is a regressable $\mathcal{L}$-formula and $\mathcal{D}$ is a DK action theory of $\mathcal{L}$. Then, $\mathcal{R}[W]$ is a formula uniform in S_0 and*

$$\mathcal{D} \models W \equiv \mathcal{R}[W].$$

Proof: By complete induction on the index of regressable $\mathcal{L}$-formulas W, $index(W)$.

- **Base case:** When the index is $((0, 0, ...), (0, 0))$, then W is an atomic formula uniform in S_0 which does not mention $Poss$ or equality between situation. By the

definition of the regression operator, $R[W] = W$ and it is easy to see that the theorem holds.

- **Induction hypothesis:** The theorem holds for all regressable $\mathcal{L}$-formulas W such that $((0,0,...),(0,0)) \preceq index(W) \prec \nu$.

- **Induction step:** We prove the theorem for formula W such that $index(W) = \nu$. Following the definition of the regression operator, there are the following cases:

  - W is a regressable atomic formula that mentions equality between situations. In other words W has the form $do([a_1, ..., a_m], S_0) \approx do([a'_1, ..., a'_n], S_0)$, for some m, n.

    If $m = n = 0$, then W is $S_0 \approx S_0$, so $\mathcal{R}[W] = W$ and the theorem holds.

    If $m \neq n$, then $\mathcal{R}[W] = false$ and $\Sigma \models W \equiv false$, so the theorem holds.

    If $m = n \geq 1$, then

    $$\Sigma \models W \equiv a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m \tag{4.4}$$

    Moreover, $a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m$ is regressable and has some index $\prec v$. Therefore, by induction hypothesis $\mathcal{R}[a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m]$ is uniform in S_0 and

    $$\mathcal{D} \models a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m \equiv \mathcal{R}[a_1 \approx a'_1 \wedge ... \wedge a_m \approx a'_m] \tag{4.5}$$

    The theorem follows from (4.4) and (4.5).

  - W is a regressable $Poss$ atomic formula of the form $Poss(A(\vec{t}), S) \approx t'$. By the definition of DK action theories, 4.1.5, there must exist in D an action precondition axiom for A of the following form:

    $$\forall \vec{x} \forall s \forall y \; Poss(A(\vec{x}), s) \approx y \;\equiv\; \Pi_A(\vec{x}, s) \wedge y \approx \top \vee \neg \Pi_A(\vec{x}, s) \wedge y \approx \bot$$

    Without loss of generality, assume that all quantifiers (if any) of $\Pi_A(\vec{x}, s)$ have had their quantified variables renamed to be distinct from the free variables

(if any) of $Poss(A(\vec{t}), S)$. Then, by the definition of the regression operator

$$\mathcal{R}[W] = \mathcal{R}[W'] \tag{4.6}$$

where W' is

$$\exists x_1 ... \exists x_n \exists a \bigwedge x_i \approx t_i \wedge a \approx A(\vec{x}) \wedge \Pi_A(\vec{x}, S) \wedge t' \approx \top \vee \neg \Pi_A(\vec{x}, S) \wedge t' \approx \bot$$

Note that we pick $x_1, ..., x_n, a$ to be variables that do not appear free in W. According to the $Poss$ regression principle 4.2.2,

$$\mathcal{D} \models W \equiv W' \tag{4.7}$$

Now, W' is uniform in S and in particular it does not mention $Poss$, nor it mentions equality between situation terms. Therefore,

$$index(W') \prec index(W)$$

Furthermore, because W is regressable and W' is uniform in S, W' is also regressable and by so by induction hypothesis $\mathcal{R}[W']$ is a formula uniform in S_0 and

$$\mathcal{D} \models W' \equiv \mathcal{R}[W'] \tag{4.8}$$

The theorem follows from (4.6), (4.7) and (4.8).

- W is a regressable atomic formula that mentions a term of the form $fl'(\vec{t}', do(A', S'))$ for some functional fluent fl'. Then, according to definitions 4.2.1 and 4.2.2, $fl'(\vec{t}', do(A', S'))$ mentions a prime fluent term $fl(\vec{t}, do(A, S))$. By the definition of DK action theories, 4.1.5, there must exist in D a successor state axiom for fl of the following form:

$$\forall \vec{x} \forall a \forall s \forall y \; fl(\vec{x}, do(a, s)) \approx y \;\equiv\; \Phi_{fl}(\vec{x}, y, a, s)$$

Without loss of generality, assume that all quantifiers (if any) of $\Phi_A(\vec{x}, y, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $fl(\vec{t}, do(A, S))$. Then, by the definition of the regression

operator

$$\mathcal{R}[W] = \mathcal{R}[W'] \tag{4.9}$$

where W' is

$$\exists x_1 ... \exists x_n \exists a \exists y \bigwedge x_i \approx t_i \wedge a \approx A \wedge \Phi_A(\vec{t}, y, a, S) \wedge W|_y^{fl(\vec{t}, do(A,S))}$$

Note that we pick $x_1, ..., x_n, a, y$ to be a variables that do not appear free in $W, \vec{t}, A, S$. According to the fluent regression principle 4.2.1,

$$\mathcal{D} \models W \equiv W' \tag{4.10}$$

$\Phi(\vec{t}, y, A, S)$ is uniform in S and in particular, it does not mention $Poss$ or equality between situation terms, or the function symbol do. Also, $fl(\vec{t}, y, A, S)$ is a prime functional fluent term. Therefore,

$$index(W') \prec index(W)$$

Furthermore, W' is regressable because W is and W' is uniform in S, so by induction hypothesis, $\mathcal{R}[W']$ is uniform in S_0 and

$$\mathcal{D} \models W' \equiv \mathcal{R}[W'] \tag{4.11}$$

The theorem follows from equations (4.9), (4.10) and (4.11)

- W is a regressable formula of the form $W = \neg W_1$. By the definition of the regression operator,

$$\mathcal{R}[\neg W_1] = \neg \mathcal{R}[W_1] \tag{4.12}$$

Clearly, $index(W_1) \prec index(W)$ and formula W_1 is regressable. So, by induction hypothesis, $\mathcal{R}[W_1]$ is uniform in S_0 and

$$\mathcal{D} \models W_1 \equiv \mathcal{R}[W_1]$$

Therefore,

$$\mathcal{D} \models \neg W_1 \equiv \neg \mathcal{R}[W_1]$$

and by (4.12)

$$\mathcal{D} \models \neg W_1 \equiv \mathcal{R}[\neg W_1]$$

and the theorem holds.

- W is a regressable formula of the form $W = W_1 \wedge W_2$. Then,

$$\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$$

and the theorem is proved similarly, as in the case where $W = \neg W_1$.

- W is a regressable formula of the form $W = \exists v W_1$. Then,

$$\mathcal{R}[\exists v W] = \exists v \mathcal{R}[W]$$

and the theorem is proved similarly, as in the case where $W = \neg W_1$.

So, we have proved that the theorem holds for all the possible cases for W.

Thus, induction follows and the regression theorem holds. □

4.3 Sensing and *DK* Action Theories

We now briefly discuss the issue of sensing in DK action theories. $\mathcal{L}_P$ and DK action theories are defined in such way that the sensing approaches for situation calculus found in the literature can be incorporated in a straightforward manner. The approach presented in [Lev96] is based on an epistemic fluent K which models the fact that the agent is unsure about which situation it is in and the use of a special predicate SF which captures the effect of a sensing action a in some situation s.

Nevertheless, the intention is to use $\mathcal{L}_P$ theories to model the epistemic state of the agent and therefore the effect of SF should affect what the theory entails and not some other epistemic fluent. Using intuitions from [SL03], we present a simple way to treat sensing in DK theories with the use of a function $SR(a, s)$, which expresses the sensing result of the action a in situation s.

All sensing actions are actions are of the form $senses(fl(\vec{t}))$. This is intended to be the action that senses the value of some fluent $fl(\vec{t}, s)$ at some situation s that the action is performed. This action, instead of affecting some epistemic fluent K, it directly affects the possible values of the sensed fluent. A simple way to do that is to let sensing actions affect the value of the fluents in the same way as the non-sensing actions do, by *setting their value to the sensed input*. This is similar to the approach used in *Golog languages [LRL+97], [DGLL00], [DGL99].

Furthermore, because of the similarity with regular actions, we chose to embed the sensing actions into the successor state axiom. The successor state axiom for some fluent fl has then the following form:

$$\forall \vec{x} \forall a \forall s \forall y \; fl(\vec{x}, do(a, s)) \approx y \;\equiv\; \Phi_{fl}(\vec{x}, y, a, s)$$

where Φ_{fl} is

$$\begin{aligned} \Phi_f(\vec{x}, y, a, s) \;\equiv\; & a \approx a_1 \wedge ... \\ & \vee \quad ... \\ & \vee a \approx a_n \wedge ... \\ & \vee a \approx sense(f(\vec{x})) \wedge y \approx SR(a, s) \end{aligned}$$

Note that similarly to any other action, there is no restriction on what the next value of a fluent can be. What we know about a fluent in some situation s does not affect its possible value after some action, sensing or not. So, the sensing might increase or decrease the number of possible values, make the value precise or make it even unknown, depending on the type of sensing.

This simple approach is problematic in the sense that the sensing results cannot be used to reason about the situations before the sensing occurred. A sensing action has more the effect of a *setting* action, which sets the disjunctive value of a fluent to the sensed

one. Nevertheless, our intention is to use *DK* theories of action which are carefully defined so that to capture only fluent-based disjunctive knowledge. In such theories, the fluents are essentially *independent* and therefore such backward reasoning would not contribute to proving anything about other fluents in previous situations, but only that the sensed value of the fluent "persists" in the past.

Example 4.3.1 (Sensing)**.** Consider the SSA for fluent $\mathsf{location}(x, s)$ we saw in example 4.1.3, tuned to include the sensing action $sense(location(x))$.

$$\begin{aligned}\forall x \forall a \forall s \forall y \ \mathsf{location}(x, do(a, s)) \approx y \ \equiv \ & a \approx moveToLocation(x, y) \\ \vee \ & a \approx sense(location(x)) \wedge y \approx SR(a, s) \\ \vee \ & \neg(\exists z \ a \approx moveToLocation(x, z) \vee a \approx sense(location(x))) \wedge y \approx \mathsf{location}(x, s)\end{aligned}$$

Like example 4.1.3, suppose that the following hold for the initial situation.

$$\mathsf{location}(box1, S_0)) = [room1]$$

$$\mathsf{location}(box2, S_0)) = [room1, room2, room3]$$

Then, according to the previous successor state axiom,

$$\mathsf{location}(box1, do(moveToObject(box1, box2), S_0))) = [room1, room2, room3]$$

Now suppose that the sensing action $sense(location(box2))$ occurs with the sensing result

$$SR(sense(location(box2), do(moveToObject(box1, box2), S_0))) = [room3]$$

Then, for the resulting situation

$$S \stackrel{def}{=} do(sense(location(box2)), do(moveToObject(box1, box2), S_0)))$$

the successor state axiom entails

$$\mathsf{location}(box2, S)) = [room3]$$

but

$$\mathsf{location}(box1, S)) = [room1, room2, room3]$$

□

The way sensing is treated here is simple but can represent a large variety of domains in terms of dynamics. Even if it is restricted in sensing the value of some fluent, there are ways of encoding other sorts of sensing into that, as for example sensing whether a fluent's value has some property. Also, sensing involving many fluents is broken down into sensing each of the fluents involved.

We close this section saying once more that a DK action theory with sensing is not intended to model the real world too. It models only the epistemic state of an agent and the sensing of the real world is intended to happen by direct interaction with it. The sensing results are intended to be acquired by the real world, "online" and whenever needed, augmenting properly the initial theory with SR atoms. DK theories with sensing are intended to be implemented into frameworks like Indigolog and the SR functionality is intended to be provided by real sensors which interact with the world in real time.

Chapter 5

Implementation of *DK* Action Theories

This chapter presents a method for soundly implementing a special form of *DK* action theories in the logic programming language Prolog. The soundness of the implementation relies on the embedding of $\mathcal{L}_P$ in situation calculus and the soundness of the implementation of a special form of situation calculus basic action theories, as presented in [Rei01].

In section 5.1, we define *$\mathcal{E}$-normal form* which is a syntactic normal form for $\mathcal{L}_P$-formulas, such that in each atom there is at most one appearance of the symbols that need special attention for the implementation. These are the fluent symbols and the function symbols which intend to capture predicate functionality which we call *predicate function symbols*. We define a transformation for the syntax and semantics of $\mathcal{L}_P$ to situation calculus and prove that for theories and formulas in $\mathcal{E}$-normal form, entailment in $\mathcal{L}_P$ can be reduced to entailment in situation calculus.

In section 5.2, we define the *closed form DK action theories* as the analog of the restricted situation calculus basic action theories which can be soundly implemented in Prolog, as presented in [Rei01]. These carefully defined $\mathcal{L}_P$ theories capture fluent-based

disjunctive knowledge only and moreover, according to the defined transformation, they are embedded in situation calculus as appropriate basic action theories such that the *implementation theorem* [Rei01] for sound implementation in Prolog applies. In this way, we get *the implementation theorem for DK action theories*, which states that closed form *DK* action theories can be soundly implemented in Prolog.

The last section presents a simple example of implementing such a *DK* action theory in Prolog and also discusses how this can be done in the Indigolog framework using Indigolog primitives.

We close this short summary of the technical work that follows, with a general remark about the limitations of the theories we consider in this chapter. Closed form *DK* action theories are very much restricted in a similar way that the basic action theories implemented in Prolog are, but at the same time they essentially *extend* the situation calculus basic action theories to account for fluent-based disjunctive knowledge. This is achieved through the treatment of possible values and the motivation is that since such theories capture only fluent-based disjunctive knowledge, they allow for an efficient implementation.

5.1 Embedding $\mathcal{L}_P$ in Situation Calculus

In this section we will be showing how $\mathcal{L}_P$ can be embedded in the situation calculus. We focus on $\mathcal{L}_P$-formulas of a specific syntactic normal form which is appropriate for the implementation theorem we will be proving in the next section.

Starting from the fact that $\mathcal{L}_P$ semantics rely on the simple structures for the evaluation of atomic formulas, the next two lemmas show how to construct logical equivalent formulas which syntactically illustrate this property. This is done by "unpacking" the nested fluent-terms. What follows is essentially a generalization of two properties which we used in the fluent regression principle proof, 4.2.1.

Lemma 5.1.1 (Possibly equals)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any n-ary function symbol f, $\mathcal{L}$-terms $\vec{t}, t'$, $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$\mathcal{M} \models f(\vec{t}) \approx t'[\mu] \quad \textit{iff} \quad \mathcal{M} \models \exists \vec{x} \bigwedge x_j \approx t_j \wedge f(\vec{x}) \approx t'[\mu]$$

Proof: By the definition of truth in a structure 3.2.6, $\mathcal{M} \models f(\vec{t}) \approx t'[\mu]$ iff there exists a simple structure $\mathcal{M}'$ such that $\mathcal{M}' \preceq \mathcal{M}$ and

$$f(\vec{t})^{\mathcal{M}'}[\mu] = t'^{\mathcal{M}'}[\mu] \tag{5.1}$$

Since $\mathcal{M}'$ is a simple structure, there exist $c_{t_1}, ..., c_{t_n}$ such that

$$t_j^{\mathcal{M}'}[\mu] = \{c_{t_j}\}$$

and by corollary 3.3.8,

$$f(\vec{t})^{\mathcal{M}'}[\mu] = f(\vec{x})^{\mathcal{M}'}[\mu(c_{t_j}/x_j)], \quad \text{where} \quad t_j^{\mathcal{M}'}[\mu] = \{c_{t_j}\}$$

and variables $\vec{x}$ do not appear in $\vec{t}, t'$. Therefore, (5.1) holds iff there exists a structure $\mathcal{M}'$ such that $\mathcal{M}' \prec \mathcal{M}$ and

$$f(\vec{x})^{\mathcal{M}'}[\mu] = t'^{\mathcal{M}'}[\mu(c_{t_j}/x_j)], \quad \text{where} \quad t_j^{\mathcal{M}'}[\mu] = \{c_{t_j}\}$$

By the definition of truth in a structure 3.2.6 and lemma 3.3.3, this holds iff

$$\mathcal{M} \models f(\vec{x}) \approx t'[\mu(c_{t_j}/x_j)], \quad \text{where} \quad c_{t_j} \in t_j^{\mathcal{M}}[\mu]$$

By the definition of truth in a structure and lemma 3.3.5, this holds iff

$$\mathcal{M} \models \bigwedge x_j \approx t_j \wedge f(\vec{x}) \approx t'[\mu(c_{t_j}/x_j)]$$

and by the definition of truth in a structure once more, this holds iff

$$\mathcal{M} \models \exists \vec{x} \bigwedge x_j \approx t_j \wedge f(\vec{x}) \approx t'[\mu]$$

□

Lemma 5.1.2 (Possibly equals 2)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$. For any $\mathcal{L}$-terms t_1, t_2, $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$\mathcal{M} \models t_1 \approx t_2[\mu] \quad \textit{iff} \quad \mathcal{M} \models \exists x\ t_1 \approx x \wedge t_2 \approx x[\mu]$$

Proof: In a similar manner as the previous lemma. □

The last two lemmas suggest a way of expanding $\mathcal{L}$-formulas into logical equivalent ones, so that each atom mentions at most a desired number of function symbols. For instance, an $\mathcal{L}$-formula can be expanded into one that has atoms that mention at most one function symbol. We use these lemmas to define *$\mathcal{E}$-normal form* for $\mathcal{L}$-formulas, so that *DK* action theories in that normal form can be embedded in situation calculus and implemented in Prolog using techniques for implementing basic action theories [Rei01].

The important issue is the fact that functions have unique names in Prolog and so the theories to be implemented will have to be restricted to comply with this. Therefore, functional fluents, which take terms as values, and functions, which intend to emulate predicates using constants $\top$ and $\bot$, need to be translated into predicates appropriately. In order to do this, we will be translating such function symbols and their value into corresponding predicates with one extra argument. These issues will become clear later in this section where we define the transformation of $\mathcal{L}_P$ to situation calculus and in the next section where we formally see the restrictions needed for implementing situation calculus basic action theories in Prolog. The formal definitions follow.

Definition 5.1.1 (Predicate function symbols)**.** Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and a subset of the non-fluent function symbols in $\mathcal{L}$, $\mathcal{P}$, which are intended to capture the functionality of predicates. We call this the set of *predicate function symbols* and includes all non-fluent function symbols which are intended to be used as predicates, with the aid of $\top$ and $\bot$.

We call the non-fluent symbols in $\mathcal{L}$ which are not in $\mathcal{P}$, *pure function symbols* and in the same manner, a *pure $\mathcal{L}$-term* is a rigid $\mathcal{L}$-term that mentions no symbols in $\mathcal{P}$ and

a *pure $\mathcal{L}$-formula* is a formula which mentions only pure terms. □

In what follows, we may be talking about $\mathcal{L}$, predicate functions and pure terms, while keeping the declaration of $\mathcal{P}$ implicit. Now, we define a normal form for $\mathcal{L}$-formulas, such that all atomic formulas mention at most one fluent symbol or one predicate function symbol. We call this, *$\mathcal{E}$-normal form.*

Definition 5.1.2 ($\mathcal{E}$-normal form)**.** Consider an $\mathcal{L}_P$ language $\mathcal{L}$ and $\mathcal{P}$ the set of predicate function symbols. An $\mathcal{L}$-formula α is in *$\mathcal{E}$-normal form* iff every atomic formula in α is one of the following:

- either pure
- or of the form

$$f(\vec{t}) \approx t'$$

where f is a predicate symbol or a fluent symbol and $\vec{t}, t'$ are pure $\mathcal{L}$-terms.

We define $\mathcal{E}$ to be a recursive operator which applies lemmas 5.1.1 to replace the non-pure arguments and 5.1.2 to replace the equality atoms between fluents and predicate function symbols in the intuitive way, so that $\mathcal{E}[\alpha]$ is in $\mathcal{E}$-normal form. □

Example 5.1.1. Let α be the formula

$$\mathit{dice}(\mathit{fl}(t_1, S), S) \approx \mathit{dice}(t_2, S)$$

where dice, fl are fluent symbols and t_1, t_2, S are pure terms. Then, according to definition 5.1.2, the following is a logical equivalent formula in $\mathcal{E}$-normal form.

$$\exists x \exists y \ \mathit{fl}(t_1, S) \approx x \wedge \mathit{dice}(x, S) \approx y \wedge \mathit{dice}(t_2, S) \approx y$$

□

Corrolary 5.1.3 ($\mathcal{E}$-normal form)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ with predicate function symbols $\mathcal{P}$. For any $\mathcal{L}$-formula α, $\mathcal{L}$-structure $\mathcal{M}$ and variable assignment μ,*

$$\mathcal{M} \models \alpha[\mu] \quad \textit{iff} \quad \mathcal{M} \models \mathcal{E}[\alpha][\mu]$$

Proof: By the definition of operator $\mathcal{E}$, at every step $\mathcal{E}$ replaces formula α by a logical equivalent formula according to lemmas 5.1.1 and 5.1.2. □

Based on the intuition that non-pure function symbols are intended to be captured by predicates, we define the transformation for $\mathcal{L}_P$ syntax and semantics into situation calculus. From now on we will be focusing on formulas that are in $\mathcal{E}$-normal form only.

Definition 5.1.3 (Situation calculus language $\hat{\mathcal{L}}$)**.** For an $\mathcal{L}_P$ language $\mathcal{L}$, we define $\hat{\mathcal{L}}$ to be the situation calculus language which is the same as $\mathcal{L}$ except that

- instead of symbol $\approx$ it includes symbol $=$
- for each n-ary predicate function symbol $f \in \mathcal{P}$, it includes the $(n+1)$-ary predicate symbol P_f instead.
- for every n-ary fluent symbol fl, it includes the $(n+1)$-ary relational fluent symbol P_{fl} instead. □

Definition 5.1.4 (Situation calculus formula $\hat{\alpha}$)**.** For an $\mathcal{L}$-formula α which is in $\mathcal{E}$-normal form, we recursively define $\hat{\alpha}$ to be a formula of $\hat{\mathcal{L}}$, as follows.

- α is a pure atomic formula, $t_1 \approx t_2$. Then, $\hat{\alpha}$ is $t_1 = t_2$.
- α is an atomic formula which mentions a predicate function symbol f. Since α is in $\mathcal{E}$-normal form, it is of the form $f(\vec{t}) \approx t'$, where $\vec{t}, t'$ are pure terms of the correct sort and f is a predicate function symbol. Then, $\hat{\alpha}$ is $P_f(\vec{t}, t')$.
- α is an atomic formula which mentions a fluent symbol fl. Since α is in $\mathcal{E}$-normal form, it is of the form $fl(\vec{t}, S) \approx t'$, where $\vec{t}, S, t'$ are pure terms of the correct sort and fl is a fluent symbol. Then, $\hat{\alpha}$ is $P_{fl}(\vec{t}, t', S)$.
- α is $\neg\alpha_1$. Then $\hat{\alpha}$ is $\neg\hat{\alpha}_1$.
- α is $\alpha_1 \wedge \alpha_2$. Then $\hat{\alpha}$ is $\hat{\alpha}_1 \wedge \hat{\alpha}_2$.

- α is $\exists x \alpha_1$. Then $\hat{\alpha}$ is $\exists x \hat{\alpha}_1$.

For a set of $\mathcal{L}$-formulas Γ, $\hat{\Gamma}$ is $\{\hat{\alpha} : \alpha \in \Gamma\}$ □

Example 5.1.2 (Situation calculus formula $\hat{\alpha}$)**.** Continuing example 5.1.1, let α be

$$\exists x \exists y \ \mathit{fl}(t_1, S) \approx x \wedge \mathit{dice}(x, S) \approx y \wedge \mathit{dice}(t_2, S) \approx y$$

Then $\hat{\alpha}$ is

$$\exists x \exists y \ P_{\mathit{fl}}(t_1, x, S) \wedge P_{\mathit{dice}}(x, y, S) \wedge P_{\mathit{dice}}(t_2, y, S)$$

□

Definition 5.1.5 (Situation calculus structure $\hat{\mathcal{M}}$)**.** For an $\mathcal{L}_P$ language $\mathcal{L}$ and an $\mathcal{L}$-structure $\mathcal{M}$, we define $\hat{\mathcal{M}}$ as the following structure of situation calculus language $\hat{\mathcal{L}}$.

- For each sort i of the language $\hat{\mathcal{L}}$, $\hat{\mathcal{M}}$ includes the same universe M_i as $\mathcal{M}$.
- For each relational fluent symbol P_{fl} in $\hat{\mathcal{L}}$, there is a functional fluent symbol fl in $\mathcal{L}$ and $P_{fl}^{\hat{\mathcal{M}}}$ is the following relation defined on structure $\mathcal{M}$:

  $$\{\langle \vec{c}, c_v, c_s \rangle : c_v \in fl^{\mathcal{M}}(\vec{c}, c_s)\}$$

  Intuitively, the possible values c_v are encapsulated in the predicate relation appropriately, keeping the situation argument c_s last.
- Similarly, for each predicate symbol P_f in $\hat{\mathcal{L}}$, $P_f^{\hat{\mathcal{M}}}$ is the following relation:

  $$\{\langle \vec{c}, c_v \rangle : c_v \in f^{\mathcal{M}}(\vec{c})\}$$

  The possible values c_v are encapsulated in the predicate relation.
- For each pure function symbol f in $\hat{\mathcal{L}}$, $f^{\hat{\mathcal{M}}}(\vec{c}) = d$, where $f^{\mathcal{M}}(\vec{c}) = \{d\}$. □

Note that since $|\mathcal{M}|$ and $|\hat{\mathcal{M}}|$ are identical and since the variable symbols are the same in $\mathcal{L}$ and $\hat{\mathcal{L}}$, we can use a variable assignment μ both for reasoning about truth in an $\mathcal{L}$-structure $\mathcal{M}$ and an $\hat{\mathcal{L}}$-structure $\hat{\mathcal{M}}$. We now proceed to prove the main theorem that will lead to the result that $\mathcal{L}_P$ can be embedded in the situation calculus.

Theorem 5.1.4 (Embedding theorem)**.** *For an $\mathcal{L}_P$ language $\mathcal{L}$, $\mathcal{L}$-structure $\mathcal{M}$, variable assignment μ and $\mathcal{L}$-formula α in $\mathcal{E}$-normal form,*

$$\mathcal{M} \models \alpha[\mu] \quad \textit{iff} \quad \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

Proof: By induction on the construction of $\mathcal{L}$-formulas α in $\mathcal{E}$-normal form.

- **Base case:** α is an atomic $\mathcal{L}$-formula.

  If α is pure then it has the form $t_1 \approx t_2$ where t_1, t_2 are pure $\mathcal{L}$-terms. Since t_1, t_2 are pure, they are also rigid and therefore they are interpreted into singleton sets. Then, by the definition of truth in a structure 3.2.6,

  $$\mathcal{M} \models \alpha[\mu] \text{ iff } t_1^{\mathcal{M}}[\mu] = t_2^{\mathcal{M}}[\mu]$$

  where $t_1^{\mathcal{M}}[\mu] = \{c_1\}$ and $t_2^{\mathcal{M}}[\mu] = \{c_2\}$ for some c_1, c_2 in $|\mathcal{M}|$. By the definition of structure $\hat{\mathcal{M}}$ 5.1.5, $t_1^{\hat{\mathcal{M}}}[\mu] = c_1$ and $t_2^{\hat{\mathcal{M}}}[\mu] = c_2$, so

  $$\mathcal{M} \models \alpha[\mu] \text{ iff } t_1^{\hat{\mathcal{M}}}[\mu] = t_2^{\hat{\mathcal{M}}}[\mu]$$

  $$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models t_1 = t_2[\mu]$$

  and by the definition of $\hat{\alpha}$ 5.1.4,

  $$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

  If α mentions the $(n+1)$-ary fluent symbol fl, then since it is in $\mathcal{E}$-normal form, it has the form $fl(\vec{t}, S) \approx t'$, where $\vec{t}, t', S$ are pure. Since $\vec{t}, t', S$ are pure, they are also rigid and so they are interpreted into singleton sets. Then, by the definition of truth in a structure 3.2.6 and lemma 3.3.3,

  $$\mathcal{M} \models \alpha[\mu] \text{ iff } t'^{\mathcal{M}}[\mu] \in fl(\vec{t}, S)^{\mathcal{M}}[\mu]$$

  where

  $$t_j^{\mathcal{M}}[\mu] = \{c_j\}, \quad t'^{\mathcal{M}}[\mu] = \{c'\}, \quad S^{\mathcal{M}}[\mu] = \{c_S\} \text{ for some } c_j, c_S, c' \in |\mathcal{M}|$$

By the definition of term denotation 3.2.5,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } c' \in fl^{\mathcal{M}}(c_1, ..., c_n, c_S)[\mu]$$

Now, by the definition of structure $\hat{\mathcal{M}}$,

$$t_j^{\hat{\mathcal{M}}}[\mu] = c_j, \quad t'^{\hat{\mathcal{M}}}[\mu] = c', \quad S^{\hat{\mathcal{M}}}[\mu] = c_S$$

and

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \langle t_1^{\hat{\mathcal{M}}}[\mu], ..., t_n^{\hat{\mathcal{M}}}[\mu], t'^{\hat{\mathcal{M}}}[\mu], S^{\hat{\mathcal{M}}}[\mu]\rangle \in P_{fl}^{\hat{\mathcal{M}}}$$

or equivalently

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models P_{fl}(\vec{t}, t', S)[\mu]$$

Finally, by the definition of $\hat{\alpha}$ 5.1.4,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

If α mentions a predicate symbol f, then since it is in $\mathcal{E}$-normal form, it has the form $f(\vec{t}) \approx t'$, where $\vec{t}, t'$ are pure. The proof of the theorem for this case is very similar to the one when α has the form $fl(\vec{t}, S) \approx t'$ and we omit it.

Thus, the theorem holds for the base case.

- **Induction hypothesis:** The theorem holds for formulas α_1, α_2.
- **Induction Step:** We prove the theorem for formulas $\neg\alpha_1$, $\alpha_1 \wedge \alpha_2$, $\exists x\alpha_1$.
  - α is $\neg\alpha_1$. By the definition of truth in a structure 3.2.6,

    $$\mathcal{M} \models \alpha[\mu] \text{ iff } \mathcal{M} \not\models \alpha_1[\mu]$$

    By induction hypothesis,

    $$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \not\models \hat{\alpha}_1[\mu]$$

    and by the definition of $\hat{\alpha}$ 5.1.4,

    $$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

- α is $\alpha_1 \wedge \alpha_2$. By the definition of truth in a structure 3.2.6,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \mathcal{M} \models \alpha_1[\mu] \text{ and } \mathcal{M} \models \alpha_2[\mu]$$

By induction hypothesis,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}_1[\mu] \text{ and } \hat{\mathcal{M}} \models \hat{\alpha}_2[\mu]$$

Therefore,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}_1 \wedge \hat{\alpha}_2[\mu]$$

and by the definition of $\hat{\alpha}$ 5.1.4,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

- α is $\exists x \alpha_1$. By the definition of truth in a structure 3.2.6, $\mathcal{M} \models \alpha[\mu]$ iff there exists a $c \in |\mathcal{M}|$ of the correct sort, such that $\mathcal{M} \models \alpha_1[\mu(c/x)]$.
By induction hypothesis and since $|\mathcal{M}|$ and $|\hat{\mathcal{M}}|$ are identical, $\mathcal{M} \models \alpha[\mu]$ iff there exists a $c \in |\hat{\mathcal{M}}|$ of the correct sort, such that $\hat{\mathcal{M}} \models \hat{\alpha}_1[\mu(c/x)]$. Therefore,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \exists x \hat{\alpha}_1[\mu]$$

and by the definition of $\hat{\alpha}$ 5.1.4,

$$\mathcal{M} \models \alpha[\mu] \text{ iff } \hat{\mathcal{M}} \models \hat{\alpha}[\mu]$$

So, we have proved that the theorem holds for all non-atomic cases of α.

Thus, induction follows and the embedding theorem holds. □

We can now prove that entailment in $\mathcal{L}_P$ can be reduced to entailment in situation calculus. There is an issue regarding predicate function symbols: we prove that entailment can be reduced when considering $\mathcal{L}_P$ theories that treat the predicate functions in the intended way, that is, they are forced to be interpreted into exactly one of $\top$ or $\bot$. We state this precisely in the next corollary about reducing entailment.

Corrolary 5.1.5 (Entailment)**.** *Consider a theory $\mathcal{D}$ of an $\mathcal{L}_P$ language $\mathcal{L}$ such that it consists of formulas in $\mathcal{E}$-normal form and for every predicate function symbol f in $\mathcal{P}$, $\mathcal{D}$ includes exactly one axiom of the form*

$$\forall \vec{x} \forall y \ f(\vec{x}) \approx y \ \equiv \ y \approx \top \wedge \Theta(\vec{x}) \vee y \approx \bot \wedge \neg \Theta(\vec{x})$$

where $\Theta(\vec{x})$ is a situation independent formula whose free variables are among $\vec{x}$. Then, for any $\mathcal{L}$-formula α in $\mathcal{E}$-normal form,

$$\mathcal{D} \models \alpha \ \ \textit{iff} \ \ \hat{\mathcal{D}} \models \hat{\alpha}$$

Proof: ($\Leftarrow$) Assume that $\hat{\mathcal{D}} \models \hat{\alpha}$. For any structure $\mathcal{M}$ that satisfies $\mathcal{D}$, we get by theorem 5.1.4 that $\hat{\mathcal{M}}$ satisfies $\hat{\mathcal{D}}$. Now, since $\hat{\mathcal{D}} \models \hat{\alpha}$, $\hat{\mathcal{M}}$ also satisfies $\hat{\alpha}$. Therefore, by theorem 5.1.4 again, $\mathcal{M} \models \alpha$.
($\Rightarrow$) Assume that $\mathcal{D} \models \alpha$. The proof is the same as the other direction but in order to use theorem 5.1.4, we need to show that for any model $\mathcal{M}'$ of $\hat{\mathcal{D}}$ there is an $\mathcal{L}$-structure $\mathcal{M}$ such that $\hat{\mathcal{M}} = \mathcal{M}'$. We prove this by showing that we can construct a $\mathcal{M}$ as the reverse transformation of $\mathcal{M}'$.

Let $\mathcal{M}'$ be an arbitrary model of $\hat{\mathcal{D}}$. We construct $\mathcal{L}$-structure $\mathcal{M}$ as follows.

- For each sort i of the language $\mathcal{L}$, $\mathcal{M}$ includes the same universe M_i as $\mathcal{M}'$.

- For each fluent symbol fl in $\mathcal{L}$, there is a relational fluent symbol P_{fl} in $\hat{\mathcal{L}}$ and $fl^{\mathcal{M}}$ is defined on structure $\mathcal{M}'$ as follows:

$$fl^{\mathcal{M}}(\vec{c}, c_s) = \{c_v : \langle \vec{c}, c_v, c_s \rangle \in P_{fl}^{\mathcal{M}'}\}$$

- Similarly, for each function symbol in $\mathcal{L}$ that appears in $\mathcal{P}$, there is a predicate symbol P_f in $\hat{\mathcal{L}}$ and $f^{\mathcal{M}}$ is defined on structure $\mathcal{M}'$ as follows:

$$f^{\mathcal{M}}(\vec{c}) = \{c_v : \langle \vec{c}, c_v \rangle \in P_f^{\mathcal{M}'}\}$$

Note that in order for $\mathcal{M}$ to be an $\mathcal{L}$-structure, $f^{\mathcal{M}}(\vec{c})$ must be a singleton set. This is guaranteed by the form of $\mathcal{D}$ and the fact that $\mathcal{M}'$ is a model of $\hat{\mathcal{D}}$: For each function symbol of $\mathcal{L}$ in $\mathcal{P}$, $\mathcal{D}$ includes an axiom of the form

$$\forall \vec{x} \forall y \; f(\vec{x}) \approx y \;\equiv\; y \approx \top \wedge \Theta(\vec{x}) \vee y \approx \bot \wedge \neg\Theta(\vec{x})$$

where $\Theta(\vec{x})$ is a situation independent formula whose free variables are among $\vec{x}$. Therefore, for each predicate symbol P_f in $\hat{\mathcal{L}}$, $\hat{\mathcal{D}}$ includes the axiom

$$\forall \vec{x} \forall y \; P_f(\vec{x}, y) \;\equiv\; y = \top \wedge \hat{\Theta}(\vec{x}) \vee y = \bot \wedge \neg\hat{\Theta}(\vec{x})$$

where $\hat{\Theta}(\vec{x})$ is a situation independent formula whose free variables are among $\vec{x}$. This axiom guarantees that for each $\vec{x}$ there is only one element of the form $\langle \vec{x}, y \rangle$ in relation $P_f^{\mathcal{M}'}$, as $\mathcal{M}'$ is a model of $\hat{\mathcal{D}}$. Thus, $f^{\mathcal{M}}(\vec{c})$ is a singleton set.

- Finally, for each function symbol f in $\mathcal{L}$ that does not appear in $\mathcal{P}$, $f^{\mathcal{M}}(\vec{c}) = \{d\}$, where $f^{\mathcal{M}'}(\vec{c}) = d$.

It is easy to verify that $\hat{\mathcal{M}} = \mathcal{M}'$. Therefore, since $\mathcal{M}'$ was arbitrary, for any model $\mathcal{M}'$ of $\hat{\mathcal{D}}$ there is an $\mathcal{L}$-structure $\mathcal{M}$ such that $\hat{\mathcal{M}} = \mathcal{M}'$ and we can use theorem 5.1.4 to prove this direction in the same way as the other one. □

5.2 Implementation Theorem for *DK* Action Theories

By corollary 5.1.5 follows that under conditions, entailment in $\mathcal{L}_P$ can be reduced to entailment in situation calculus using the transformation for formulas 5.1.4. We now show how *DK* action theories can be soundly implemented in Prolog, utilizing this result and the fact that a special form of situation calculus basic action theories can be soundly implemented in Prolog.

In order to achieve this, we need to define a restricted form of *DK* action theories $\mathcal{D}$ which, when reduced to situation calculus theories $\hat{\mathcal{D}}$ using the transformation 5.1.4, they form appropriate basic action theories such that the *implementation theorem* presented in [Rei01] applies. The necessary restrictions for the "translated" basic action theories $\hat{\mathcal{D}}$ essentially ensure that the theories conform with the requirements of Clark's theorem [Cla78], [Llo87] for soundly implementing theories in Prolog.

The key notion that summarizes the restrictions needed for basic action theories in situation calculus is that they should have a *closed form initial database* [Rei01]. In the same manner and following closely this definition, we define the notion of a *closed form initial database* for *DK* action theories; the definition is only tuned to apply to the syntax of $\mathcal{L}_P$. Then, we define the *closed form DK action theories* which can be soundly implemented in Prolog.

Definition 5.2.1 (Closed form initial database for *DK* action theories)**.** Consider an $\mathcal{L}_P$ language $\mathcal{L}$ which includes only finitely many function symbols and $\mathcal{P}$ the set of predicate function symbols. An initial database $\mathcal{D}_{S_0}$ of a *DK* action theory of $\mathcal{L}$ is in *closed form* iff

- For every fluent symbol in $\mathcal{L}$, $\mathcal{D}_{S_0}$ contains exactly one initial specification axiom for that fluent,

  $$\forall \vec{x} \forall y \; fl(\vec{x}, S_0) \approx y \equiv \Psi_{fl}(\vec{x}, y)$$

  where $\Psi(\vec{x}, y)$ is a formula uniform in S_0 and whose free variables are among $\vec{x}, y$.

- For every predicate function symbol f in $\mathcal{P}$, $\mathcal{D}_{S_0}$ contains exactly one axiom of the form

  $$\forall \vec{x} \forall y \; f(\vec{x}) \approx y \;\equiv\; y \approx \top \wedge \Theta(\vec{x}) \vee y \approx \bot \wedge \neg\Theta(\vec{x})$$

  where $\Theta(\vec{x})$ is a situation independent formula whose free variables are among $\vec{x}$.

- The rest of $\mathcal{D}_{S_0}$ consists of the following (countably infinite) sentences:

  (a) For each pair of distinct pure function symbols f, g of sort object (including constant symbols),

  $$\neg f(\vec{x}) \approx g(\vec{y})$$

  (b) For each pure function symbol f of sort object,

  $$f(x_1, ..., x_n) \approx f(y_1, ..., y_n) \rightarrow x_1 \approx y_1 \wedge ... \wedge x_n \approx y_n$$

  (c) For each pure $\mathcal{L}$-term t of sort object other than the variable x of sort object,

  $$\neg t[x] \approx x$$

  (d) For each pure $\mathcal{L}$-term A of sort action other than the variable a of sort action,

  $$\neg A[a] \approx a$$

  In some restricted language which does not include the fluents and predicate functions, or in $\hat{\mathcal{L}}$: Axioms (a) and (b) are uniqueness of names axioms for objects. Axioms (c) and (d) strengthen the uniqueness of names axioms for actions and objects and, as mentioned in [Rei01], they essentially capture the occur-check requirement for the unification algorithm of proper Prolog systems. □

A closed form initial database $\mathcal{D}_{S_0}$ for a *DK* action theory is carefully defined so that when it is translated to situation calculus according to definition 5.1.4, it results to a closed form initial database $\hat{\mathcal{D}}_{S_0}$ for a situation calculus basic action theory, as presented in [Rei01]. In the same manner, closed form *DK* action theories $\mathcal{D}$ satisfy the necessary requirements so that the corresponding situation calculus basic action theory $\hat{\mathcal{D}}$ can be soundly implemented in Prolog. Essentially, these requirements are the projection of the ones needed by the implementation theorem for basic action theories in situation calculus [Rei01], in $\mathcal{L}_P$.

Definition 5.2.2 (Closed form *DK* action theory)**.** A *DK* action theory of an $\mathcal{L}_P$ language $\mathcal{L}$ is a *closed form DK action theory* iff

- $\mathcal{L}$ includes only finitely many function symbols
- the initial database $\mathcal{D}_{S_0}$ is in closed form □

So, we finally come to the intended result: By corollary 5.1.5, entailment of a closed form *DK* action theory $\mathcal{D}$ is reduced to entailment of a situation calculus theory $\hat{\mathcal{D}}$ which satisfies the requirements for the implementation theorem for basic action theories [Rei01] and can be soundly implemented in Prolog. Actually, there is a small complication in this procedure which has to do with the treatment of predicates.

$\mathcal{L}_P$ is designed to be fully functional and so, as we saw in the previous chapters, predicate functionality needs to be emulated by special functions with the aid of constants $\top$ and $\bot$. On the other hand, functions in Prolog have unique names and therefore non-pure symbols in $\mathcal{L}_P$ need to be emulated by non-standard predicates in order to get implemented appropriately. In this way, the treatment of an intended predicate symbol is a little awkward, as it is first captured by a function symbol f in $\mathcal{L}$ which is then translated into a predicate symbol P_f with an extra argument in $\hat{\mathcal{L}}$. Despite being awkward, this poses no complications for the predicate function symbols except for $Poss$. The problem with $Poss$ is that in $\hat{\mathcal{D}}$, $Poss$ is translated into P_{Poss} which is no longer the standard binary predicate symbol used in situation calculus, but a ternary predicate symbol which encapsulates its truth value in its last extra argument.

Nevertheless, this is merely a technical issue, rather than an essential problem. The definition of a closed form *DK* action theory $\mathcal{D}$ is such that for every ground instance of a $Poss$ term it is the case that the theory entails either that $Poss$ has the value $\top$ or the value $\bot$ and this holds for $\hat{\mathcal{D}}$ and P_{Poss} atoms too. We can have the standard $Poss$ syntax in $\hat{\mathcal{L}}$ as a macro based on P_{Poss}.

Definition 5.2.3 (*Poss* macro in $\hat{\mathcal{L}}$)**.** For an $\mathcal{L}_P$ language $\mathcal{L}$ and the corresponding situation calculus language $\hat{\mathcal{L}}$, we define the macro $Poss(A, S)$ to be the $\hat{\mathcal{L}}$-formula

$$P_{Poss}(A, S, \top)$$

For a closed form *DK* action theory $\mathcal{D}$, the corresponding situation calculus theory $\hat{\mathcal{D}}$ and the action precondition axiom for an action $A(\vec{t})$ in $\hat{\mathcal{D}}$,

$$\forall \vec{x} \forall s \forall y \; P_{Poss}(A(\vec{x}), s, y) \; \equiv \; \Pi_A(\vec{x}, s) \wedge y = \top \vee \neg \Pi_A(\vec{x}, s) \wedge y = \bot$$

we define an alternative action precondition axiom

$$\forall \vec{x} \forall s \; Poss(A(\vec{x}), s) \; \equiv \; \Pi_A(\vec{x}, s)$$

□

Lemma 5.2.1 (Correctness of *Poss* macro in $\hat{\mathcal{L}}$)**.** *For a closed form DK action theory $\mathcal{D}$ of an $\mathcal{L}_P$ language $\mathcal{L}$ and the corresponding $\hat{\mathcal{L}}$, $\hat{\mathcal{D}}$,*

$$\hat{\mathcal{D}} \models Poss(A, S) \quad \textit{iff} \quad \hat{\mathcal{D}} \models P_{Poss}(A, S, \top)$$

$$\hat{\mathcal{D}} \models \neg Poss(A, S) \quad \textit{iff} \quad \hat{\mathcal{D}} \models P_{Poss}(A, S, \bot)$$

and for the purpose of evaluating regressable formulas in $\hat{\mathcal{L}}$, the alternative action precondition axiom

$$\forall \vec{x} \forall s \; Poss(A(\vec{x}), s) \; \equiv \; \Pi_A(\vec{x}, s)$$

can be used instead of the action precondition axiom for A in $\hat{\mathcal{D}}$,

$$\forall \vec{x} \forall s \forall y \; P_{Poss}(A(\vec{x}), s, y) \; \equiv \; \Pi_A(\vec{x}, s) \wedge y = \top \vee \neg \Pi_A(\vec{x}, s) \wedge y = \bot$$

Proof: Directly from the definition of the action precondition axiom in $\hat{\mathcal{D}}$. Note that

$$\hat{\mathcal{D}} \models P_{Poss}(A(\vec{t}), S, \top) \;\; \text{iff} \;\; \hat{\mathcal{D}} \models \Pi_A(\vec{t}, S)$$

$$\hat{\mathcal{D}} \models P_{Poss}(A(\vec{t}), S, \bot) \;\; \text{iff} \;\; \hat{\mathcal{D}} \models \neg \Pi_A(\vec{t}, S)$$

□

We will be using the *Poss* macro and the alternative action precondition axioms so that $\hat{\mathcal{D}}$ has the standard basic action theory syntax. There is one last thing we need to make precise before we present the implementation theorem for closed form *DK* action theories. Since we will be using the *Poss* macro for the evaluation of formulas, we need to ensure that all $\mathcal{L}$-formulas which will be translated in $\hat{\mathcal{L}}$ are restricted to mention *Poss* in atoms of the form $Poss(A, S) \approx \top$ or $Poss(A, S) \approx \bot$ only and also that they are translated appropriately in $\hat{\mathcal{L}}$ using the *Poss* macro. For the next results we need to focus in formulas which are in $\mathcal{E}^+$-normal form which is essentially the syntactic $\mathcal{E}$-normal form with the needed restriction for *Poss* atoms.

Definition 5.2.4 ($\mathcal{E}^+$-normal form)**.** A formula $\mathcal{L}$-α is in $\mathcal{E}^+$-normal form iff it is in $\mathcal{E}$-normal form and for all atoms of the form

$$Poss(\vec{t}) \approx t'$$

t' is either $\top$ or $\bot$. □

In what follows we assume that all $\mathcal{L}$-formulas are in $\mathcal{E}^+$-normal form. This is not a restriction because all formulas can be transformed in some logical equivalent formula in $\mathcal{E}$-normal form (corollary 5.1.3) and furthermore, in the context of a closed form *DK* action theory, every formula of the form $f(\vec{t}) \approx t'$ is logically equivalent to either $f(\vec{t}) \approx \top$, or $f(\vec{t}) \approx \bot$, or $false$. Also, for simplicity, we do not define another transformation from $\mathcal{L}_P$ to situation calculus, but assume that $\mathcal{L}$-formulas which mention *Poss* are translated according to 5.1.4 into appropriate atoms in $\hat{\mathcal{L}}$ utilizing the *Poss* macro defined in 5.2.3.

We now copy the *implementation theorem* found in [Rei01], applied to $\hat{\mathcal{D}}$ according to the previous remarks about *Poss*.

Corrolary 5.2.2 (Implementation theorem for $\hat{\mathcal{D}}$)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ which includes only finitely many function symbols and the closed form DK action theory $\mathcal{D}$ of $\mathcal{L}$. Let $\hat{\mathcal{D}}$ be the situation calculus basic action theory which we get from $\mathcal{D}$, according*

to the definition 5.1.4. Then, $\hat{\mathcal{D}}$ has finitely many relational fluents and action function symbols and $\hat{\mathcal{D}}_{S_0}$ is in closed form. So, according to the implementation theorem, let $\mathcal{G}$ be the Prolog program obtained from the following sentences, after transforming them with the Lloyd-Topor transformations, [Llo87]:

- *For each definition of a non-fluent predicate of $\hat{\mathcal{D}}_{S_0}$ of the form $\forall \vec{x}\ P(\vec{x}) \equiv \Theta_P(\vec{x})$:*

$$\Theta(\vec{x}) \rightarrow P(\vec{x})$$

- *For each definition of a fluent predicate of $\hat{\mathcal{D}}_{S_0}$ of the form $\forall \vec{x}\ P_{fl}(\vec{x}, S_0) \equiv \hat{\Psi}_{fl}(\vec{x}, S_0)$:*

$$\hat{\Psi}_{fl}(\vec{x}, S_0) \rightarrow P_{fl}(\vec{x}, S_0)$$

- *For each action precondition axiom of $\hat{\mathcal{D}}_{APA}$ of the form $\forall \vec{x} \forall s\ Poss(A(\vec{x}), s) \equiv \hat{\Pi}_A(\vec{x}, s)$:*

$$\hat{\Pi}_A(\vec{x}, s) \rightarrow Poss(A(\vec{x}), s)$$

- *For each successor state axiom of $\hat{\mathcal{D}}_{SSA}$ of the form $\forall \vec{x} \forall a \forall s\ P_{fl}(\vec{x}, do(a, s)) \equiv \hat{\Phi}_{fl}(\vec{x}, a, s)$:*

$$\hat{\Phi}_{fl}(\vec{x}, a, s) \rightarrow P_{fl}(\vec{x}, do(a, s))$$

Then, $\mathcal{G}$ provides a sound *Prolog implementation of the basic action theory $\hat{\mathcal{D}}$ for the purposes of proving regressable sentences, in the following sense:*

- *Whenever a proper Prolog interpreter succeeds on a normal Prolog goal $\hat{G}$ with answer substitution θ, then $\hat{\mathcal{D}} \models (\forall)\hat{G}\theta$. $(\forall)\hat{G}\theta$ denotes the result of universally quantifying all the free variables (if any) of $\hat{G}\theta$.*

- *Whenever a proper Prolog interpreter returns failure (that is, whenever it finitely fails to obtain derivation) on a normal Prolog goal $\hat{G}$, then $\hat{\mathcal{D}} \models (\forall)\neg\hat{G}$.*

Note that the query $\hat{G}$ should be a regressable $\hat{\mathcal{L}}$-formula which is first transformed by the Lloyd-Topor transformation before execution by this program. □

So, closed form *DK* action theories can be soundly implemented in Prolog with the aid of the transformation defined in 5.1.4 and the implementation theorem for basic action theories in situation calculus.

Corrolary 5.2.3 (Implementation theorem for $\mathcal{D}$)**.** *Consider an $\mathcal{L}_P$ language $\mathcal{L}$ which includes only finitely many function symbols and the closed form DK action theory $\mathcal{D}$ of $\mathcal{L}$. Let $\hat{\mathcal{D}}$ be the situation calculus basic action theory which we get from $\mathcal{D}$, according to the definition 5.1.4 and $\mathcal{G}$ be the Prolog program we get according to the implementation theorem for $\hat{\mathcal{D}}$ (corollary 5.2.2).*

Then, $\mathcal{G}$ provides a sound *Prolog implementation of the closed form DK action theory $\mathcal{D}$ for the purposes of proving regressable sentences in $\mathcal{E}^+$-normal form, in the following sense:*

- *Whenever a proper Prolog interpreter succeeds on a normal Prolog goal $\hat{G}$ with answer substitution θ, then $\mathcal{D} \models (\forall)G\theta$. $(\forall)G\theta$ denotes the result of universally quantifying all the free variables (if any) of $G\theta$.*
- *Whenever a proper Prolog interpreter returns failure (that is, whenever it finitely fails to obtain derivation) on a normal Prolog goal $\hat{G}$, then $\mathcal{D} \models (\forall)\neg G$.*

Note that the query $\hat{G}$ should be a regressable $\hat{\mathcal{L}}$-formula in $\mathcal{E}^+$-normal form which is first transformed by the Lloyd-Topor transformation before execution by this program.

Proof: Directly, from corollary 5.1.5 and the implementation theorem 5.2.2. □

The next section presents a simple example of the application of the implementation theorem and discusses the implementation of closed form *DK* action theories in Indigolog.

5.3 The Dice World – A Simple Example

As we saw in this chapter, the main issue for implementing closed form *DK* action theories $\mathcal{D}$ is to reduce entailment to an appropriate basic action theory $\hat{\mathcal{D}}$. This proved to be

quite intuitive: in just a few words, closed form DK action theories are defined in such way, so that the "possibly equals" effect of equality in $\mathcal{L}_P$ semantics can be encapsulated into the definitions of appropriate relational fluents P_{fl}.

We now present an example of a closed form DK action theory which is based on definitions and axioms we used in the examples of the previous chapters. This is not a typical example of some interesting domain, rather than a simple example which summarizes the implementation issues discussed in the previous section.

The Dice World consists of a number of four-sided dice on a table. The dynamics of the domain is limited to the dynamic state of the dice which is affected by the actions of picking up, rolling or placing down a dice setting its value. Let $\mathcal{L}$ be the $\mathcal{L}_P$ language which includes the following symbols.

- Two fluent symbols: the binary function symbols *dice*, which is used the same way as all previous examples, and *holding*, which expresses that the agent is holding some object.

- Three action symbols: the unary function symbols $rollDice, pickup$ and the binary function symbol $setDiceValue$. Similar to what we saw in example 4.1.2, action $rollDice(x)$ rolls the dice x to the table and $setDiceValue(x, y)$ places the dice x on the table setting so that its value is y. Action $pickup$ captures the intuitive effect.

- One unary function symbol $isdice$ emulating predicate functionality as in example 3.3.5.

The set $\mathcal{P}$ of predicate functions consists of $Poss$ and $isdice$. Also, all function symbols are defined so that they get arguments of the intuitive sort.

Let $\mathcal{D}$ be the following DK action theory of $\mathcal{L}$:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{SSA} \cup \mathcal{D}_{APA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{S_0}$$

where

- $\mathcal{D}_{S_0}$ includes a similar initial specification axiom for *dice* as we saw in example 4.1.1 and one for *holding*. These axioms express that in the initial situation the agent is holding nothing and the dice are on the table with value 1 facing up.

$$\forall x \forall y \ \mathit{dice}(x, S_0) \approx y \equiv isdice(x) \approx \top \wedge y \approx 1 \vee isdice(x) \approx \bot$$

$$\forall x \forall y \ \mathit{holding}(x, S_0) \approx y \equiv y \approx \bot$$

  $\mathcal{D}_{S_0}$ also includes the required axiom for the function symbol *isdice* we saw in example 3.3.5, which captures the fact that there are three dice with specific names. For simplicity we chose the numbers 1,2,3 for names.

$$\forall x \ isdice(x) \approx y \equiv (x \approx 1 \vee x \approx 2 \vee x \approx 3) \wedge y \approx \top \vee \neg(x \approx 1 \vee x \approx 2 \vee x \approx 3) \wedge y \approx \bot$$

- $\mathcal{D}_{SSA}$ includes the successor state axiom for *dice* we saw in example 4.1.2 tuned for $setDiceValue, pickup$ and an intuitive successor state axiom for fluent *holding*:

$$\begin{aligned}\forall x \forall a \forall s \forall y \ \mathit{dice}(x, do(a, s)) \approx y \ \equiv \ & a \approx rollDice(x) \wedge (y \approx 1 \vee y \approx 2 \vee y \approx 3 \vee y \approx 4) \\ \vee \ & a \approx pickup(x) \wedge (y \approx 1 \vee y \approx 2 \vee y \approx 3 \vee y \approx 4) \\ \vee \ & a \approx setDiceValue(x, y) \\ \vee \ & \neg(a \approx rollDice(x) \vee a \approx pickup(x) \\ & \quad \vee \exists z \ a \approx setDiceValue(x, z)) \wedge y \approx \mathit{dice}(x, s)\end{aligned}$$

  Action *pickup* has a similar effect to the value of a dice as action *rollDice*, which is not very accurate. Nevertheless, we mention once again that this is not intended

to be a completely realistic real-world example.

$$\begin{aligned}\forall x \forall a \forall s \forall y \ \mathit{holding}(x, do(a,s)) \approx y \ \equiv \ & a \approx pickup(x) \wedge y \approx \top \\ \vee \ & a \approx rollDice(x) \wedge y \approx \bot \\ \vee \ & \exists z \ a \approx setDiceValue(x,z) \wedge y \approx \bot \\ \vee \ & \neg(a \approx pickup(x) \vee a \approx rollDice(x) \\ & \vee \exists z \ a \approx setDiceValue(x,z)) \wedge y \approx \mathit{holding}(x,s)\end{aligned}$$

Note that except for *pickup*, all the actions have the effect that the agent is no longer holding the dice.

- D_{APA} includes an action precondition axiom for each of the the action function symbols.

$$\forall s \forall x \forall y \ Poss(rollDice(x), s) \approx y \ \equiv \ \Pi_1(x,s) \wedge y \approx \top \vee \neg\Pi_1(x,s) \wedge y \approx \bot$$

$$\forall s \forall x \forall z \forall y \ Poss(setDiceValue(x,z), s) \approx y \ \equiv \ \Pi_2(x,s) \wedge y \approx \top \vee \neg\Pi_2(x,s) \wedge y \approx \bot$$

$$\forall s \forall x \forall z \forall y \ Poss(pickup(x), s) \approx y \ \equiv \ \Pi_3(x,s) \wedge y \approx \top \vee \neg\Pi_3(x,s) \wedge y \approx \bot$$

where

$$\Pi_1(x,s) \equiv \Pi_2(x,y,s) \equiv \ isdice(x) \approx \top \wedge \mathit{holding}(x,s) \approx \top$$

$$\Pi_3(x,s) \equiv \ true$$

- Σ is the foundational axioms for situations and $\mathcal{D}_{UNA}$ is the set of unique names axioms for actions function symbols.

5.3.1 The Dice World in Prolog

According to definition 5.1.4, $\hat{\mathcal{D}}$ is the following DK action theory of $\hat{\mathcal{L}}$:

$$\hat{\mathcal{D}} = \hat{\Sigma} \cup \hat{\mathcal{D}}_{SSA} \cup \hat{\mathcal{D}}_{APA} \cup \hat{\mathcal{D}}_{UNA} \cup \hat{\mathcal{D}}_{S_0}$$

where

- $\hat{\mathcal{D}}_{S_0}$ includes the following sentences:

$$\forall x \forall y \; P_{\mathit{dice}}(x, y, S_0) \equiv P_{isdice}(x) \wedge y = 1 \vee \neg P_{isdice}(x)$$

$$\forall x \forall y \; P_{\mathit{holding}}(x, y, S_0) \equiv y = \bot$$

$$\forall x \; P_{isdice}(x) \equiv (x = 1 \vee x = 2 \vee x = 3)$$

- $\hat{\mathcal{D}}_{SSA}$ includes the following sentences:

$$\begin{aligned}\forall x \forall a \forall s \forall y \; P_{\mathit{dice}}(x, y, do(a, s)) \; \equiv \; & a = rollDice(x) \wedge (y = 1 \vee y = 2 \vee y = 3 \vee y = 4) \\ \vee \; & a = pickup(x) \wedge (y = 1 \vee y = 2 \vee y = 3 \vee y = 4) \\ \vee \; & a = setDiceValue(x, y) \\ \vee \; & \neg(a = rollDice(x) \vee pickup(x) \vee \exists z \; a = setDiceValue(x, z)) \wedge P_{\mathit{dice}}(x, y, s)\end{aligned}$$

$$\begin{aligned}\forall x \forall a \forall s \forall y \; P_{\mathit{holding}}(x, y, do(a, s)) \; \equiv \; & a = pickup(x) \wedge y = \top \\ \vee \; & a = rollDice(x) \wedge y = \bot \\ \vee \; & a = setDiceValue(x, z) \wedge y = \bot \\ \vee \; & \neg(a = pickup(x) \vee a = rollDice(x) \\ & \quad \vee \exists z \; a = setDiceValue(x, z)) \wedge P_{\mathit{holding}}(x, y, s)\end{aligned}$$

- $\hat{\mathcal{D}}_{APA}$ includes the following sentences:

$$\forall s \forall x \; Poss(rollDice(x), s) \; \equiv \; \hat{\Pi}_1(x, s)$$

$$\forall s \forall x \forall z \; Poss(setDiceValue(x, z), s) \; \equiv \; \hat{\Pi}_2(x, z, s)$$

$$\forall s \forall x \; Poss(pickup(x), s) \; \equiv \; \hat{\Pi}_3(x, s)$$

where

$$\hat{\Pi}_1(x, s) \equiv \hat{\Pi}_2(x, y, s) \equiv \; P_{isdice}(x) \wedge P_{\mathit{holding}}(x, \top, s)$$

$$\hat{\Pi}_3(x, s) \equiv \; true$$

- $\hat{\Sigma}$ and $\hat{\mathcal{D}}_{UNA}$ are according to definition 5.1.4.

Now, according to the implementation theorem for $\mathcal{D}$, 5.2.3, we construct the Prolog program $\mathcal{G}_1$ which appears in table 5.3.1.

```
isdice(X)              :- (X=1; X=2; X=3).

dice(X,Y,s0)           :- (isdice(X), Y=1); \+isdice(X).
dice(X,Y,do(A,S))      :- (A=rollDice(X), (Y=1; Y=2; Y=3; Y=4));
                          (A=pickup(X), (Y=1; Y=2; Y=3; Y=4));
                          A=setDiceValue(X,Y);
                          (\+(A=rollDice(X); A=pickup(X); A=setDiceValue(X,Z)),
                          dice(X,Y,S)).

holding(X,Y,s0)        :- Y=bot.
holding(X,Y,do(A,S))   :- (A=pickup(X), Y=top);
                          (A=rollDice(X), Y=bot);
                          (A=SetDiceValue(X,Z), Y=bot);
                          (\+(A=pickup(X); A=rollDice(X);
                          A=SetDiceValue(X,Z)), holding(X,Y,S)).

poss(rollDice(X),S)       :- isdice(X), holding(X,top,S).
poss(setDiceValue(X,Y),S):- isdice(X), holding(X,top,S).
poss(pickup(X),S)         :- true.
```

Table 5.1: Prolog program G_1

We will be testing G_1 on SWI-Prolog Version 5.2.13 (Copyright (c) 1990-2003 University of Amsterdam). Before we do that, we note that the soundness result of the implementation theorem relies on a fundamental result of Keith Clark [Cla78], [Llo87]

that correlates first order logic theories with Prolog programs. We *emphasize* that this result holds about a *proper* Prolog interpreter which in particular includes the *occur-check*[1] in the unification algorithm and does not *flounder*[2].

Most of the Prolog systems, including SWI-Prolog, fail both of these conditions. The absence of the occur-check is not a problem in general and [AP94] provides a mechanism for transforming every program into one for which only the calls to the built-in unification predicate need to be resolved by a unification algorithm with the occur-check.

What is more important is that in the existing Prolog systems, soundness is granted as long as negation as failure is used *only on ground atoms*, so as to avoid floundering. This means, for example, that some negation that appears in the scope of a universal quantifier is likely to falsify the soundness result. This is very restricting, since the predicates that are used in programs such as $\mathcal{G}_1$, capture the possible values for the fluents and the universal quantifier is needed to express essential notions like the K macro.

We now see some examples on reasoning about actions in the context of theory $\mathcal{D}$, using Prolog program $\mathcal{G}_1$ and then we discuss how this issue regarding floundering can be treated. Note that the second last argument of some fluent predicate is essentially the possible value for the functional fluent that is intended to represent.

- What is a possible value for each of the dice in the initial situation?

```
?- dice(1,Y,s0).
Y = 1 ;
No
?- dice(2,Y,s0).
Y = 1 ;
No
?- dice(3,Y,s0).
```

[1]The unification algorithm fails to unify the variable x with any complex term which includes x.
[2]Application of negation as failure to non-ground terms.

```
Y = 1 ;
No
?-
```

- Is is possible to execute the sequence of actions $[pickup(1), rollDice(1)]$?

```
?- poss(pickup(1),s0), poss(rollDice(1),do(pickup(1),s0)).
Yes
?-
```

- What is a possible value for each of the dice after these actions are performed? Let this situation be S_2.

```
?- dice(1,Y,do(rollDice(1),do(pickup(1),s0))).
Y = 1 ;
Y = 2 ;
Y = 3 ;
Y = 4 ;
No
?- dice(2,Y,do(rollDice(1),do(pickup(1),s0))).
Y = 1 ;
No
?- dice(3,Y,do(rollDice(1),do(pickup(1),s0))).
Y = 1 ;
No
?-
```

- Is is possible to execute the action $rollDice(2)$ or $setDiceValue(2, 1)$ in situation S_2?

```
?- poss(setDiceValue(2,2),do(rollDice(1),do(pickup(1),s0))).
No
?- poss(rollDice(2),do(rollDice(1),do(pickup(1),s0))).
No
?-
```

- Is is possible to execute the sequence of actions $[pickup(2), setDiceValue(2,2)]$ in situation S_1?

```
?- poss(setDiceValue(2,2),do(pickup(2),do(rollDice(1),
|    do(pickup(1),s0)))).
Yes
?-
```

- What is a possible value for each of the dice after these actions are performed?

```
?- dice(1,Y,do(setDiceValue(2,2),do(pickup(2),do(rollDice(1),
|    do(pickup(1),s0))))).
Y = 1 ;
Y = 2 ;
Y = 3 ;
Y = 4 ;
No
?- dice(2,Y,do(setDiceValue(2,2),do(pickup(2),do(rollDice(1),
|    do(pickup(1),s0))))).
Y = 2 ;
No
?- dice(3,Y,do(setDiceValue(2,2),do(pickup(2),do(rollDice(1),
|    do(pickup(1),s0))))).
```

```
Y = 1 ;

No

?-
```

So, we can reason about the possible values of a fluent fl and we can actually get the completeness effect of a strong fl-formula, like $fl = [1, 2, 3, 4]$, using Prolog's functionality that can give all the solutions for a goal. For instance, we saw that in situation S_2 dice 1 has exactly the possible values 1, 2, 3, and 4. This is what the formula α

$$\mathit{dice}(1, S_2) = [1, 2, 3, 4]$$

expresses. As we noted in the beginning though, any negation applied to non-ground terms will falsify the soundness result and therefore, the formula itself cannot be soundly evaluated by the Prolog program. If we expand the $\mathit{dice}(1, S_2)$-formula macro it becomes clear why floundering is a problem.

$$\forall y \ \mathit{dice}(1, S_2) \approx y \equiv y \approx 1 \vee y \approx 2 \vee y \approx 3 \vee y \approx 4$$

Negation that is hiding in the $\equiv$ macro is applied to non-ground atoms and so, any results regarding closure on possible values is not guaranteed to be sound.

Nevertheless, we can restrict universal quantification to be bounded and force the variables in non-ground atoms to range over some finite set. In this way we lose expressive power, which one could argue that is not very crucial in real-world robotic domains, but we also avoid floundering, ensuring soundness of the results. In order to do this, we need to define a generic mechanism for doing a bounded version of the universal or existential quantifier, instead of using normal Prolog unification and the Lloyd-Topor rules.

The *Golog languages [LRL$^+$97], [DGLL00], [DGL99], which we briefly talked about in chapter 2, account for this issue and furthermore offer a concise way of representing situation calculus basic action theories. The intention is to use the existing Indigolog framework [DGL99] to represent closed form *DK* action theories, instead of augmenting

programs like $\mathcal{G}_1$ that are constructed according to the implementation theorem 5.2.3. Furthermore, in doing this we benefit all the rich functionality of Indigolog, as described in chapter 2.

5.3.2 Implementing Closed Form *DK* Action Theories in Indigolog

We now briefly present how situation calculus basic action theories are represented in Indigolog and then discuss how closed form *DK* action theories can be implemented in the Indigolog framework using the same primitives. The discussion is informal and is based on the intuitions behind the implementation of basic action theories in Indigolog and the observation that, in a sense, closed form *DK* action theories are capturing the effect of many possible basic action theories.

The implementation of a basic action theory in Indigolog is treated in a generic way. Instead of writing a Prolog program which captures the theory itself, the basic notions of basic action theories are implemented in a generic evaluation module and the specific details of the theory in question are provided by the user in a concise Prolog program. The implementation of a basic action theory in Indigolog includes:

- The definition of functional fluents using predicate `fun_fluent(Fluent)`.
- The definition of primitive actions using predicate `prim_action(Action)`.
- The definition of sensing actions using predicate `senses(Action, Fluent)`. The sensing actions have the effect that after they are executed, the value of the fluent is set to be the sensed value.
- The representation of situation independent action precondition axioms using the predicate `poss(Action, Condition)`. This expresses that when `Condition` holds, action `Action` can be performed.

- The representation of the initial situation S_0 using predicate `initially(Fluent, Value)`. This expresses that the fluent `Fluent` has value `Value` in S_0.

- The representation of the successor state axioms using predicate `causes_val(Action, Fluent, Value, Condition)`. This expresses that when `Condition` holds, the execution of action `Action` causes the fluent `Fluent` to get the value `Value`.

The representation of basic action theories is the same as in Golog, more details can be found in [LRL$^+$97]. Details about sensing in Indigolog can be found in [DGL99].

We now review the way functional fluents are treated in Indigolog. Since all function terms have unique names in Prolog, in order to implement functional fluents, their effect has to be compiled into predicates. This can be done in the same way a *DK* theory $\mathcal{D}$ is translated in a basic action theory $\hat{\mathcal{D}}$, using a predicate which encapsulates the functional fluent's value as an argument. Instead of that, in Indigolog there is a special predicate which captures this effect in a generic way. This is the predicate `has_val(Fluent, Value, History)` and is intended to express that after performing the sequence of actions `History`, the functional fluent `Fluent` has the value `Value`.

The evaluation of formulas in Indigolog relies on `has_val` and a special predicate `holds(Formula, History)` which implements the revised Lloyd-Topor transformations [Rei01]. Furthermore, `holds` provides bounded existential and universal quantification functionality. The soundness of the Indigolog programs relies then on the programmer who should ensure that the Indigolog programs are such that they will produce ground atoms whenever during the execution of the program negation as failure is used. Note that the syntax of formulas `Formula` is very intuitive and includes operators `neg(F)`, `or(F1, F2)`, `and(F1, F2)`, `all(X, F)`, `some(X, F)`, `all(X, Domain, F)`, `some(X, Domain, F)`, `impl(F1, F2)`, `equiv(F1, F2)` and the use of equality in the normal way.

The intuition is that Indigolog functionality is convenient for implementing closed form *DK* action theories too. The only difference in implementing a closed form *DK* action theory using Indigolog primitives is that the predicates `initially` and `causes_val` are not forced to have a unique solution for `Value` for each instantiation of the arguments. For example, the fluent *dice*(1) may have *two possible values* in the initial situation. This is can be captured by having *two* `initially` *clauses* in the Prolog program implementing the theory. In a similar way, the predicate `causes_val` can be defined so that to capture *many options* for the value of some fluent after an action has been performed. In this way, the effect of non-standard equality in $\mathcal{L}_P$ is captured by normal equality in Prolog with the aid of predicate `has_val`, which under these circumstances is more like `has_poss_val`, capturing the fact that a fluent has a possible value rather than a definite one. From another point of view, we could say that a closed form *DK* action theory summarizes many possible basic action theories, each of which defines a possible value for the fluents.

So, the implementation of closed form *DK* action theories can be done in a similar way as basic action theories using the Indigolog framework. The only difference is that the primitives are used so that to provide more options for the *possible* values of the fluents. We will now focus only on the evaluation mechanism of Indigolog that essentially solves the projection task.

Indigolog is built so that to be able to implement various action theories and evaluate formulas depending on the corresponding evaluation module. We provide a new evaluation module for doing projection in closed form *DK* action theories. The closed form *DK* action theory evaluation module is the same as the one for basic action theories, without the optimizations which prevent backtracking for predicate `has_val`. Indigolog functionality as a high-level programming language, which we discussed in chapter 2, is implemented in another module. In the same manner, we can provide a new indigolog incremental interpreter module so that the language constructs take into account the

possibility that there can be more than one solutions for `has_val`. Again, the issue is to remove any optimizations that are based on the assumption that fluents can be proven to have at most one value.

5.3.3 The Dice World in Indigolog

Based on the previous remarks, we construct the following Indigolog program $\mathcal{G}_2$, as the implementation of the dice world theory $\mathcal{D}$ we saw in section 5.3. We briefly explain the structure of $\mathcal{G}_2$.

We first define the finite domains which we will use for bounded quantification. These will be also used with the predicate `domain(Variable, Domain)` which assigns a user-defined domain to a variable, to make formulas more readable.

```
/* Domains, Sorts */
dice([1,2,3]).          %The domain of dice names
diceValues4([1,2,3,4]). %The domain of possible values for 4-sided dice
diceValues([1,2,3,4,5,6,7,8,9,10]).
                        %The domain of possible values for all dice
```

We then specify the functional fluents and the primitive actions of the language. Note that we also include functional fluent t which will be used to illustrate the effect of possible nesting of fluent terms, as in example 4.1.2.

```
/* Functional Fluents and Actions */
fun_fluent(dice(_)).
fun_fluent(holding(_)).
fun_fluent(t).
prim_action(pickup(_)).
prim_action(rollDice(_)).
prim_action(setDiceValue(_,_)).
```

Then, we specify the initial situation and the causal laws which capture the dynamics of the fluents. The initial situation axioms include the definition of the predicate function *isdice*.

```
/* Initial Situation */
initially(dice(X),Y):- domain(X, dice),Y=1.
initially(holding(_),Y):- Y=bot.
initially(t,Y):- Y=1;Y=2.
isdice(X):- domain(X,dice).

/* Causal Laws */
/* Fluent dice(_) */
causes_val(pickup(X), dice(X), Y, true):- domain(Y,diceValues4).
causes_val(rollDice(X), dice(X), Y, true):- domain(Y,diceValues4).
causes_val(setDiceValue(X,Y), dice(X), Y, true).
/* Fluent holding(_) */
causes_val(pickup(X), holding(X), top, true).
causes_val(rollDice(X),holding(X), bot, true).
causes_val(setDiceValue(X,_), holding(X), bot, true).
```

Finally, we specify the preconditions for the actions in the domain.

```
/* Action Preconditions */
poss(pickup(X), true).
poss(rollDice(X),(isdice(X), holding(X,top))).
poss(setDiceValue(X),(isdice(X), holding(X,top))).
```

Now, following example 4.1.2, consider situation S where the dice 1 has the value 3 and dice 2 has the value 4. The term t of the example that has the possible values 1 and

2 is now the fluent `t`. Let S be the situation which results after executing the sequence of actions $[pickup(1), setDiceValue(1,3), pickup(2), setDiceValue(2,4)]$.

- First we verify that the possible values for the fluents in S are the intended ones.

```
?- holds(dice(1)=Y,[setDiceValue(2,4),pickup(2),setDiceValue(1,3),
|    pickup(1)]).
Y = 3 ;
No
?-holds(dice(2)=Y,[setDiceValue(2,4),pickup(2),setDiceValue(1,3),
|    pickup(1)]).
Y = 4 ;
No
?- holds(dice(3)=Y,[setDiceValue(2,4),pickup(2),setDiceValue(1,3),
|    pickup(1)]).
Y = 1 ;
No
?-
```

- As also mentioned earlier, in this framework we can also express the strong *dice*-formulas, using bounded universal quantification over all possible values of all dice. $\mathit{dice}(1, S) = [3]$:

```
?- holds(all(v,diceValues,equiv(dice(1)=v,v=3)),
|    [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
Yes
?-
```

dice$(2, S) = [4]$:

```
?- holds(all(v,diceValues,equiv(dice(2)=v,v=4)),
|     [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
Yes
?-
```

dice$(3, S) = [1]$:

```
?- holds(all(v,diceValues,equiv(dice(3)=v,v=1)),
|     [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
Yes
?-
```

Note also, that we can prove that wrong strong *dice*-formulas do not hold.

dice$(3, S) = [2]$:

```
?- holds(all(v,diceValues,equiv(dice(3)=v,v=2)),
|     [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
No
?-
```

dice$(3, S) = [1, 2]$:

```
?- holds(all(v,diceValues,equiv(dice(3)=v,or(v=1,v=2))),
|     [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
No
?-
```

- Finally, we now verify that in S we get the correct results for some of the formulas we examined in example 4.1.2. Note that the formulas to be evaluated are first

transformed into $\mathcal{E}^+$-normal form.

$\mathit{dice}(t, S)) = [3, 4]$:

```
?- holds(all(y,diceValues,some(x,t,equiv(dice(x)=y,or(y=3,y=4)))),
|    [setDiceValue(2,4),pickup(2),setDiceValue(1,3),pickup(1)]).
Yes
?-
```

$\mathit{dice}(1, do(setDiceValueTo1(1), S)) = [1]$

```
?- holds(all(y,diceValues,equiv(dice(1)=y,y=1)),
|    [setDiceValue(1,1),pickup(1),setDiceValue(2,4),pickup(2),
|    setDiceValue(1,3),pickup(1)]).
Yes
?-
```

$\mathit{dice}(t, do(setDiceValueTo1(1), S)) = [1, 4]$:

```
?-
holds(all(y,diceValues,some(x,dice,
|    and(equiv(dice(x)=y,or(y=3,y=4)),t=x))),
|    [setDiceValue(1,1),pickup(1),setDiceValue(2,4),pickup(2),
|    setDiceValue(1,3),pickup(1)]).
Yes
?-
```

We close this chapter with a brief note on the complexity of doing theorem proving in the context of a closed form *DK* action theory.

In closed form *DK* theories, incomplete knowledge is captured by the possible values of fluents only and since the fluents are forced to be always independent, the intention

is that the truth evaluation of some formula breaks down to atoms, which implies the desired efficiency. This is not always true, though, because a non-ground fluent atom β that appears in the scope of some existential quantifier can have *more than one ground instances that satisfy it*, as a result of the interpretation of $\approx$. So, as the variables in β may appear in other atoms of the same formula too, atoms of the formula can in general be *correlated*, essentially forming constraints about the possible values for the fluent terms. In this way, the evaluation of a formula is a constraint satisfaction problem on the possible values of the fluents, which is in the worst case exponential in the number of fluent symbols appearing in the formula.

This worst case scenario is not appealing but it illustrates exactly the design stance we adopted which is to *allow for variable efficiency* according to the user's needs. Complex properties regarding (fluent-based) disjunctive knowledge can be expressed, but the user is warned that they do cost in efficiency. The dominating factor in the complexity of determining the truth evaluation of a regressed formula α in $\mathcal{E}$-normal form is *the maximal number of correlated fluents* in α. Note, that the correlation of the fluents is easy to identify, as it is the correlation of some argument or the value of a fluent with some argument or the value of some other fluent. It is something that *does not happen accidentally when doing knowledge representation*, like perhaps some complex syntactic property of the formula which does not correspond to some *semantic* property too. For instance, this happens in the case of nested fluents in $\mathcal{L}$-terms, which in $\mathcal{E}$-normal form are expanded appropriately. Note also, that even if it is common two fluents to be correlated, it is not common that many fluents are correlated mutually. There can be many puzzles where this is true, but in common theories about robotic domains where *DK* action theories are intended to be used, we expect the maximal number of correlated fluents to be small, allowing for efficient theorem proving.

Finally, note that this issue about complexity arises when the user is reasoning about the possible values using variables in a general way to form constraints about them, or

when this effect occurs by expanding nested fluent terms. If the user uses only the strong fl-formula and the K macros when reasoning about the possible values, then the intended efficiency is guaranteed.

Chapter 6

Conclusion

In this thesis we explored some alternative semantics for the situation calculus, which are based on the notion of possible values. $\mathcal{L}_P$ languages feature extended expressiveness regarding disjunctive knowledge, as a direct consequence of the non-standard semantics. We regard $\mathcal{L}_P$ theories to be implicitly epistemic in a limited way, as the notion of "being possibly equal" is equivalent to the notion of "not known that it is not equal", in the normal modal logic sense.

In chapter 4, we showed that the normal situation calculus syntax utilizes the features of $\mathcal{L}_P$ semantics in a natural way, resulting to epistemic action theories with a special treatment of disjunctive knowledge. The definition of DK action theories is based on the usual notions, such as the notion of the successor state axiom, keeping most of the syntax identical to situation calculus basic action theories. For instance, the standard syntax for a successor state axiom, when used in $\mathcal{L}_P$, has the effect that it actually defines how the possible values of a fluent change according to the execution of actions, instead of the value of the fluent itself. This also illustrates the intuition behind our view of $\mathcal{L}_P$ theories as being implicitly epistemic.

$\mathcal{L}_P$ expressiveness is suitable for restricting DK action theories so that any disjunctive knowledge captured refers to one fluent only, each time. In this way, the fluents

are independent and the disjunctive knowledge captured by the theory is limited to be fluent-based. This is a property that allows for variable efficiency depending on the representational needs. In chapter 5, we showed that closed form *DK* action theories are carefully defined *DK* action theories which capture only fluent-based disjunctive knowledge.

The implementation of such theories is based on their reduction to some appropriate situation calculus basic action theory. This essentially shows that we can achieve the limited disjunctive knowledge effect by directly defining the resulting non-standard basic action theories. These theories treat knowledge in a way that resembles the modal fluents of Demolombe and Pozos Parra. In particular, it is as if all fluents are modal and functional, incorporating the possible value as an extra argument of each fluent. Nevertheless, closed form *DK* action theories benefit in that they feature intuitive syntax based on the possible values for the fluents and subsume the situation calculus basic action theories. Furthermore, in chapter 5 we showed that closed form *DK* action theories can be implemented in languages like Indigolog in a natural way, using the already implemented language primitives.

A quick comparison with other approaches in the literature shows that closed form *DK* action theories are weaker than the Scherl and Levesque epistemic fluent approach, which was actually our intention so that to allow for efficient evaluation mechanisms, and that it has comparable expressiveness with the PKS system. On the other hand, even if in this thesis we focused on (closed form) *DK* action theories which are based on the standard situation calculus syntax, more general $\mathcal{L}_P$ theories of action can capture the functionality of most of the approaches in the literature, including the Scherl and Levesque epistemic fluent. An interesting question is then, whether the extended expressiveness regarding disjunctive knowledge, that $\mathcal{L}_P$ semantics provides, can reveal hidden properties or correlations between the various approaches, in a similar way that it did regarding fluent-based disjunctive knowledge.

Focusing on the structure of $\mathcal{L}_P$ semantics, we see that it is based on intuitions from many-valued logics, so that terms are interpreted into multiple objects. We saw that this functionality along with the proper use of the constant symbols $\top$ and $\bot$ is capable of emulating predicates. It would be interesting to explore the possibility of using more than two truth values along with the appropriate foundation axioms to express other modalities such as likelihood for example. From a different point of view, it seems interesting to explore whether $\mathcal{L}_P$ semantics can capture the functionality of standard many-valued approaches in the literature, such as for instance three-valued logics.

Finally, we point out some possible directions of future research regarding Indigolog framework, in the context of closed form DK action theories. Based on the notion of possible values, we can define variations of Indigolog constructs. For instance, we could define a search operator Σ_{cond} that returns a conditional plan based on the incomplete knowledge represented in the epistemic state of the agent, or an optimistic search operator Σ_{opt} which returns a linear plan that is only possible to be successful according to the epistemic state again.

We also note that in the context of possible values, the sensing functionality that Indigolog provides is very limited, as a sensing action can only result to setting the value of a fluent to be the sensed result, without any incomplete knowledge. In order to fully utilize the $\mathcal{L}_P$ epistemic setting, though, it is necessary that sensing actions can result in incomplete knowledge that depends on the situation that sensing was performed. For example, this is necessary in order to represent sensing actions which can rule out possible values for fluents, essentially shrinking the incomplete knowledge. Similarly, sensing which results to certain incomplete knowledge is useful in domains where there are noisy sensors.

Bibliography

[AP94] Krzysztof R. Apt and Alessandro Pellegrini. On the occur-check-free PROLOG programs. *ACM Transactions on Programming Languages and Systems*, 16(3):687–726, May 1994.

[Bel77] N. Belnap. A useful four-valued logic. In J. Dunn and G. Epstein, editors, *Modern uses of multiple-valued logic*, pages 8–37. Reidel Publishing Co., 1977.

[BP98] Fahiem Bacchus and Ron Petrick. Modeling an agent's incomplete knowledge during planning and execution. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 432–443, San Francisco, CA, June 1998. Morgan Kaufmann Publishers.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322, New York, 1978. Plenum Press.

[DGL99] Giuseppe De Giacomo and Hector Levesque. An incremental interpreter for high-level programs with sensing. In Hector J. Levesque and Fiora Pirri, editors, *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.

[DGLL97] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in

the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on AI (IJCAI'97)*, pages 1221–1226, Nagoya, August 1997.

[DGLL00] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[DPP00] Robert Demolombe and Maria del Pilar Pozos Parra. A simple and tractable extension of situation calculus to epistemic logic. In *Proc. of 12th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, number 1932 in LNAI, pages 515–524, Charlotte, NC, USA, October 2000. Springer.

[EGW97] Oren Etzioni, Keith Golden, and Daniel S. Weld. Sound and efficient closed-world reasoning for planning. *Artif. Intell.*, 89(1-2):113–148, 1997.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Orlando, Florida, 1972.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fun99] J. Funge. Representing knowledge within the situation calculus using intervalvalued epistemic fluents, 1999.

[Gin88] M. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[Hin62] J. Hintikka. Knowledge and belief. *Knowledge and Belief*, 1962.

[Lev96] Hector Levesque. What is planning in the presence of sensing? In *The Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI-96*, pages 1139–1146, Portland, Oregon, August 1996. American Association for Artificial Intelligence.

[Lev98] H. J. Levesque. A completeness result for reasoning with incomplete first-order knowledge bases. In *Proceedings of KR-1998, Sixth International Conference on Principles of Knowledge Representation and Reasoning*, Trento, Italy, June 1998.

[LL03] Yongmei Liu and Hector Levesque. A tractability result for reasoning with incomplete first-order knowledge bases. In *Proc. IJCAI-2003*, pages 83–88, Acapulco, Mexico, August 2003.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second edition, 1987.

[LMT97] Jorge Lobo, Gisela Mendez, and Stuart R. Taylor. Adding knowledge to the action description language $\neg$. In *Proceedings AAAI-97*, pages 454–459, 1997.

[LRL+97] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Moo85] R.C. Moore. A formal theory of knowledge and action. In J.R. Hobbs and R.C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex, Norwood, NJ., 1985.

[PL02] Ron Petrick and Hector Levesque. Knowledge equivalence in combined action theories. In *Proceedings of KR-2002*, Toulouse, France, April 2002.

[PR99] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999.

[PS92] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the first international conference on Artificial intelligence planning systems*, pages 189–197. Morgan Kaufmann Publishers Inc., 1992.

[Rei91] Ray Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Rei01] Raymond Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[SB01] Tran Cao Son and Chitta Baral. Formalizing sensing actions: A transition function based approach. *Artificial Intelligence*, 125:19–91, 2001.

[SL03] R. Scherl and H. J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, (144):1–39, 2003.

[Thi00] Michael Thielscher. Representing the knowledge of a robot. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 109–120, San Francisco, 2000. Morgan Kaufmann.

[Urq86] A. Urquhart. Many-valued logic. In D. Gabbay and F. Guenthner, editors, *Handbook of philosophical logic, vol III*. Reidel Publishing Co., 1986.